Welcome to the Onshape Developer Documentation

Welcome to the Onshape Developer Documentation. You can find resources here for developing applications that integrate with Onshape. Use the navigation bar on the left to navigate through the documentation.

- New to Onshape? See the Onshape Architecture guide.
- New to REST APIs? Continue to the API Introduction page.
- Ready to develop your first Onshape app? Start with our Quick Start section.
- Already a pro? Head right over to the Onshape API Explorer.
    - Use `https://companyName.onshape.com/glassworks/explorer` for Enterprise accounts.
- Looking to share your app with others? Check out our App Store documentation.
- Stuck? Check out our Get Help section.

API Explorer

We document all available Onshape REST API endpoints in our Glassworks API Explorer:

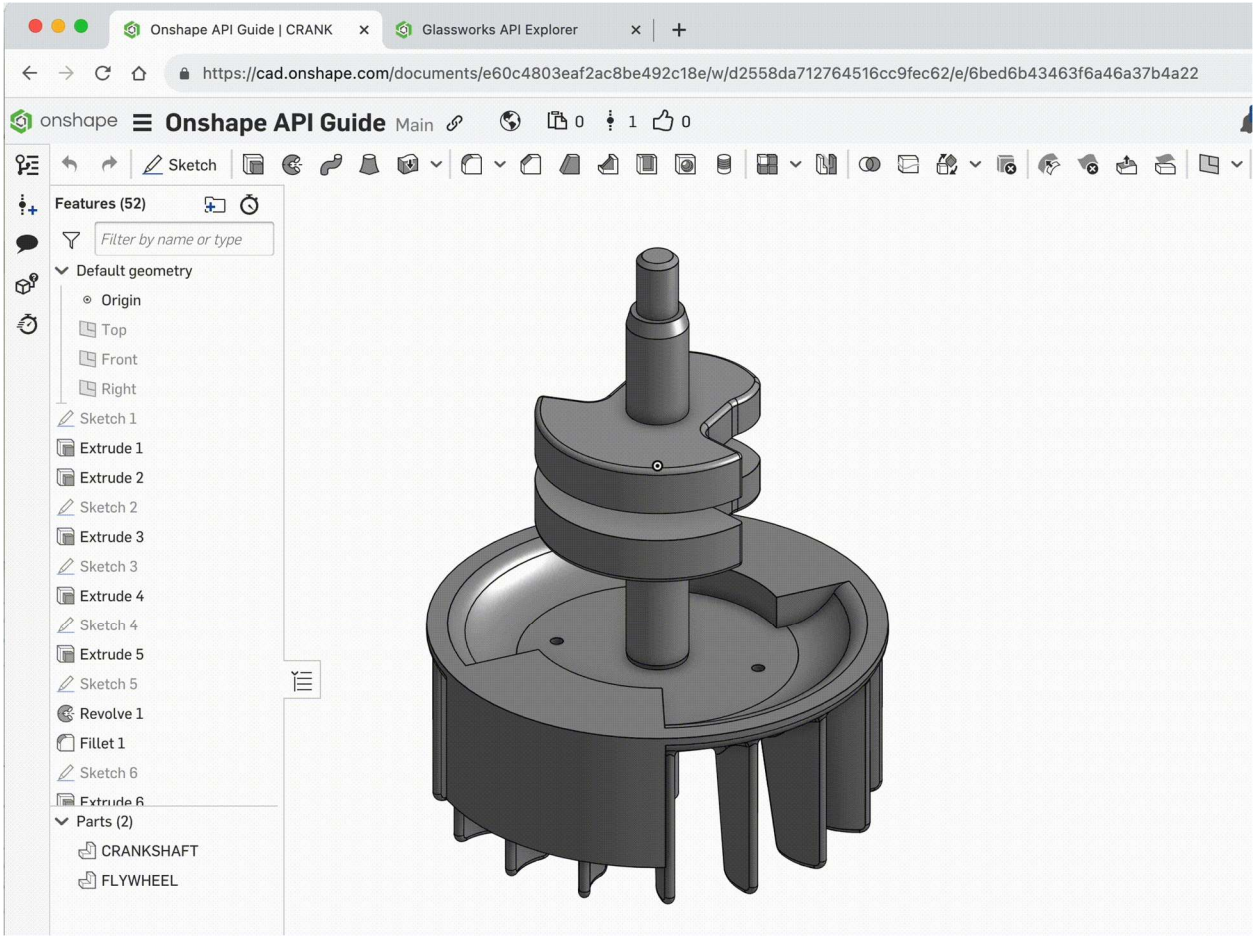
https://cad.onshape.com/glassworks/explorer/

This API Explorer site enables you to run API requests directly within its interface and provides the output from the API call. To try an endpoint in the API Explorer, follow these steps or follow along with the video below:

1. Open this public Onshape document in your browser: https://cad.onshape.com/documents/e60c4803eaf2ac8be492c18e/w/d2558da712764516cc9fec62/e/6bed6b43463f6a46a37b4a22
2. Open the API Explorer in a new browser tab: https://cad.onshape.com/glassworks/explorer/
    - Note: For Enterprise accounts, substitue `cad` in this URL with your company name.
3. Scroll down to Document.

4. Click to expand the `getDocument` endpoint. (Hint: it appears in the API Ref as `GET /documents/{did}`).
5. Go back to the public document you opened in Step 2, and copy the document ID from the Onshape URL (`e60c4803eaf2ac8be492c18e`).

6. Paste the document ID into the `did` field in the API Explorer.
    - Note: If you can't edit the `did` field, click the **Try it out** button. This will toggle to a **Cancel** button when the fields are editable.
7. Scroll down and click **Execute**.
    - Note: If you receive a `403` error, see the Authentication section for help.
8. Scroll to the bottom of the 200 response body. We have correctly returned `Onshape API Guide` as the document name.



IMPORTANT NOTE: The documentation in the API Explorer reflects the supported interface. Some API calls may, for historical reasons, return additional undocumented fields. Unless the return fields are documented in the API Explorer, you should NOT use them, as they may be removed without warning. Your application should always ignore unexpected or undocumented return data. Onshape reserves the right to add, remove or change any undocumented fields.

Authentication

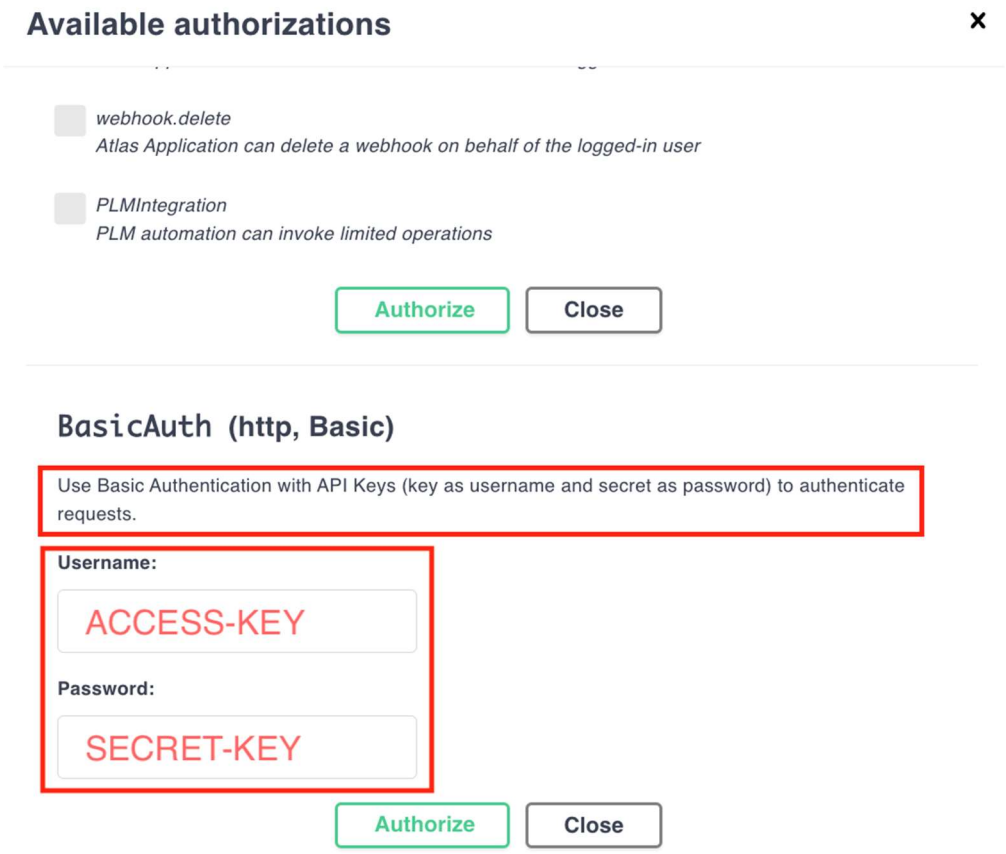
You can authenticate in the API Explorer in one of three ways:

1. **Onshape**:
    1. Open Onshape in a new tab in your browser.
    2. Sign in with your Onshape credentials. Onshape will pass your credentials to the API Explorer.
2. **API Keys**:
    1. Click **Authorize** in the top-right of the API Explorer page and scroll to the bottom of the dialog.

    2. Provide your API access key in the Username field and your secret key in the Password field. See API Keys for help creating your API Keys. Do NOT enter

your Onshape credentials.



3. Click **Authorize**, and then click **Close**.

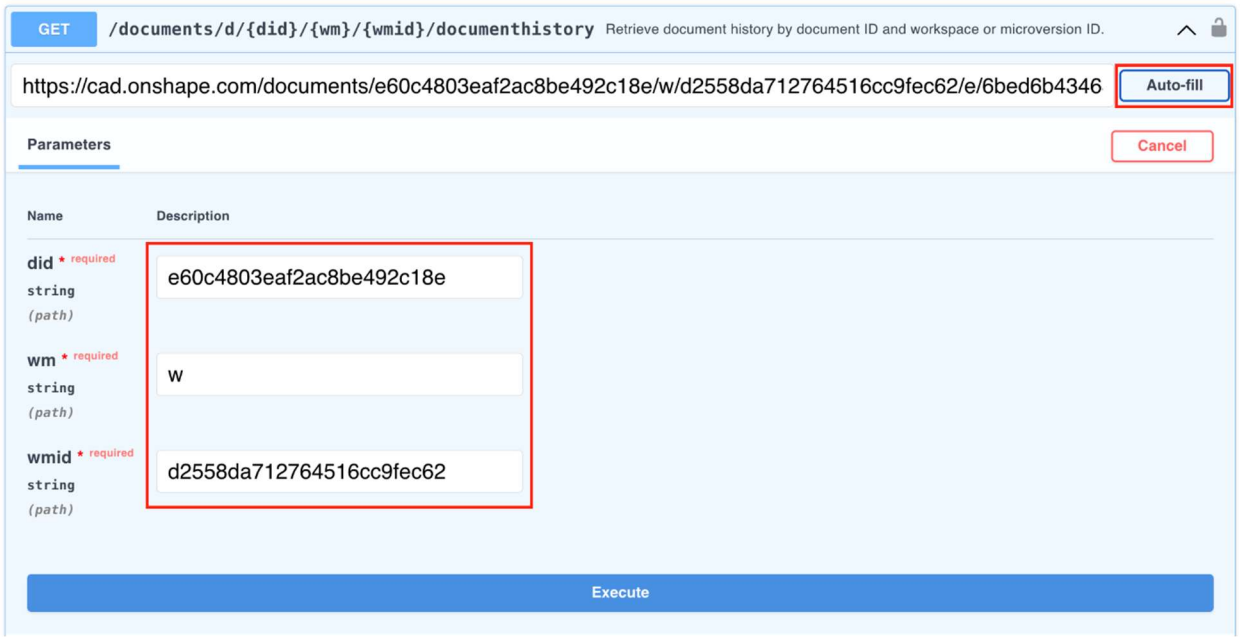
3. **Oauth**:
    1. Click **Authorize** in the top-right of the API Explorer page.

    2. Fill out the OAuth fields. See OAuth for more information on autheneticating with OAuth2.
    3. Click **Authorize**, and then click **Close**.

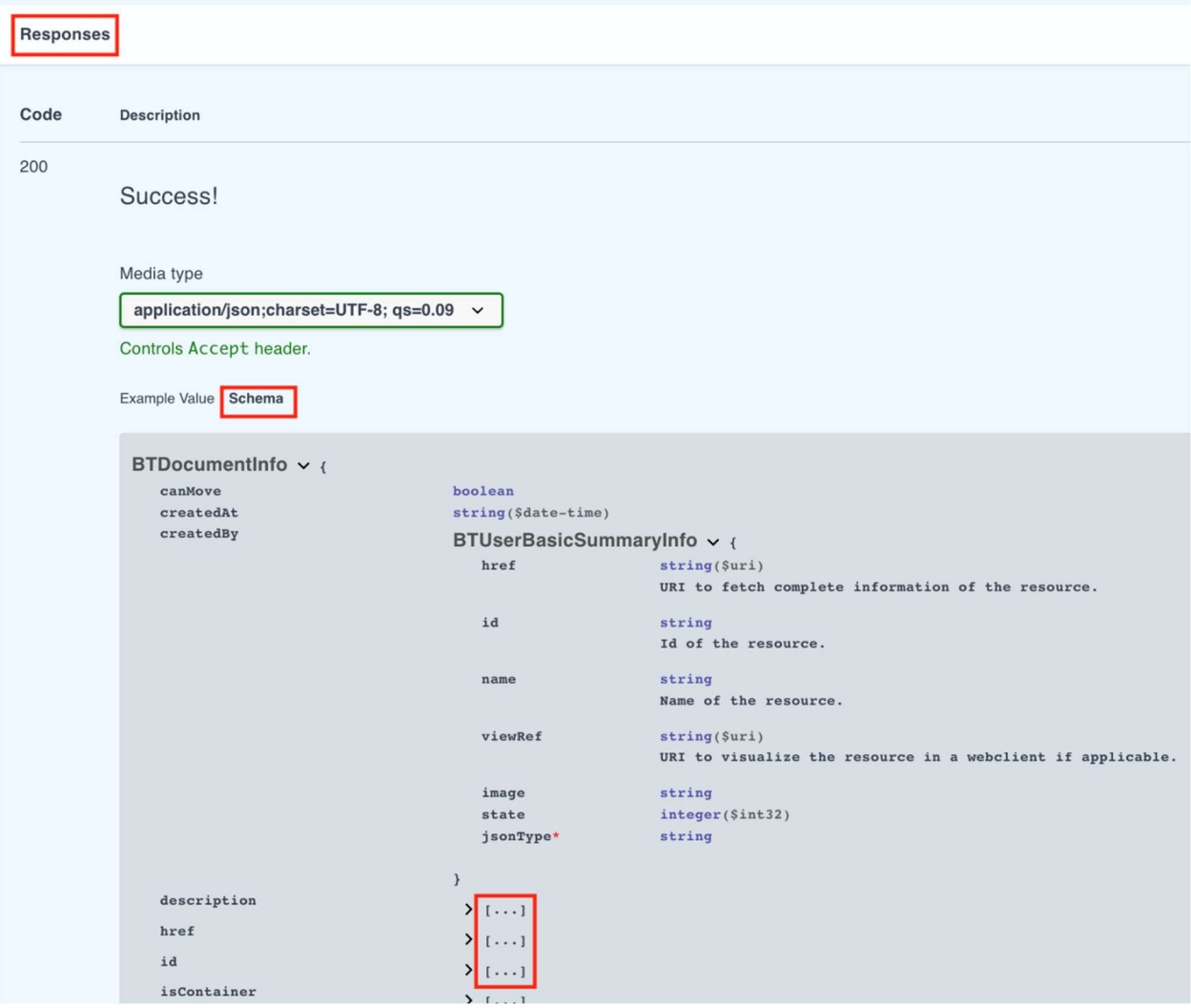
Use the Auto-fill feature

1. Expand the endpoint you want to use in the API Explorer.
2. Paste an entire Onshape URL into the top field.
3. Click **Auto-fill**. The document ID, workspace/version/microversion ID, and element ID are pushed from the URL into the correct fields.
4. Confirm all fields are filled out as expected. Not every parameter can be extracted from an Onshape URL, so there may be more fields to fill out.



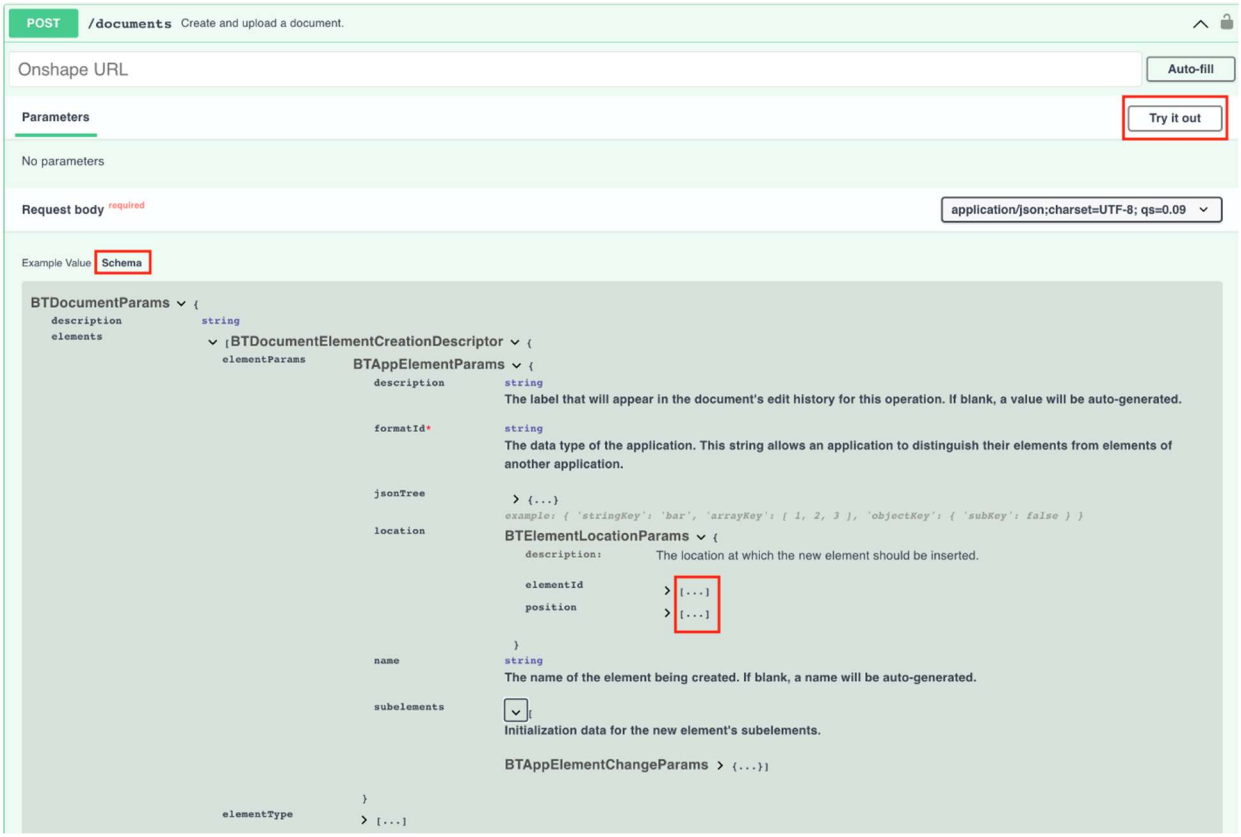
View response body docs

1. Expand the endpoint you want to use the in API Explorer.
2. Scroll down to the Responses section.
3. Click Schema.
4. Click the `[...]` symbols to expand the docs for the response JSON.



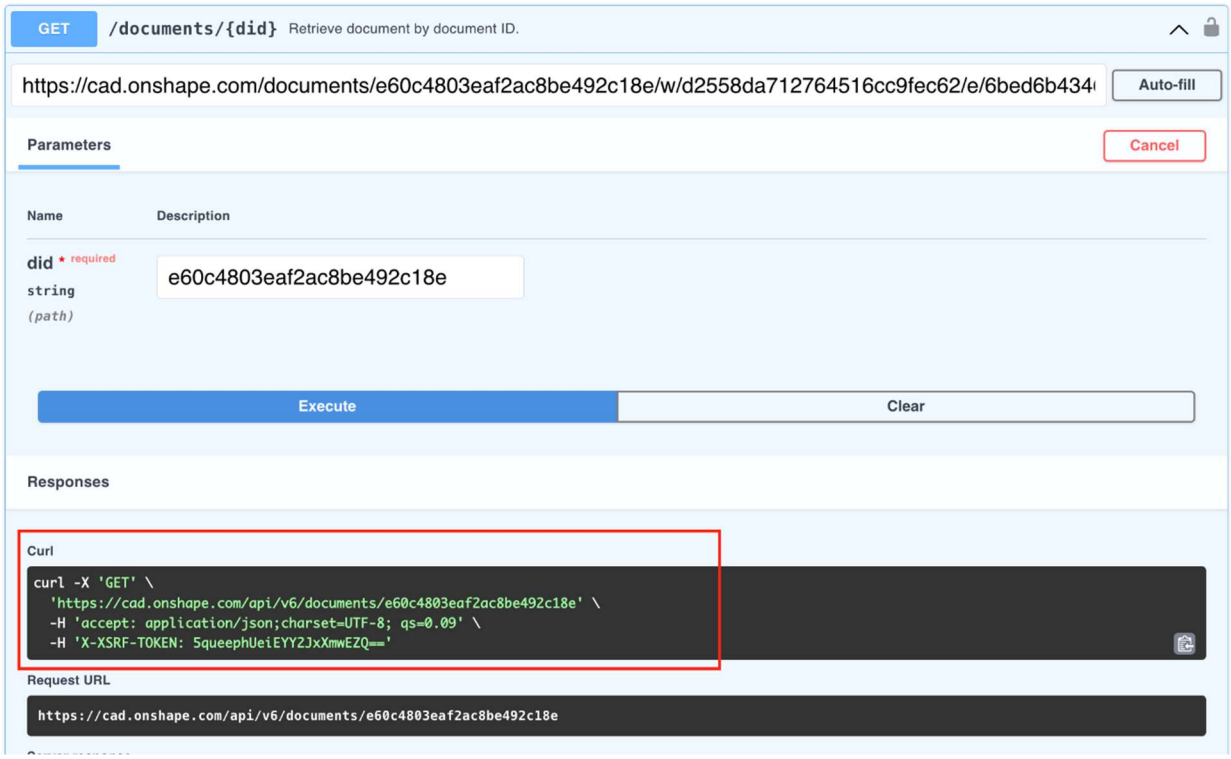
View request body docs

1. Expand the endpoint you want to use the in API Explorer.
2. **Click the *Cancel* button to make the schema viewable.**
3. Click Schema.
4. Click the `[...]` symbols to expand the docs for the response JSON.



Copy a cURL

1. Expand the endpoint you want to use in the API Explorer.
2. Fill out the parameter fields.
3. Click Execute in the API Explorer.
4. Copy the curl from the Curl field.



Troubleshooting

- If the parameter fields in the API Explorer are grayed out, click the **Try it Out!** button to toggle it to a **Cancel** button. The parameter fields should become editable.
- If you can't see the request body JSON docs, click the **Cancel** button to toggle it back to the **Try it Out!** button.
- If you see authentication issues, review the Authentication section above.

Architecture

Design in Onshape typically beings with a document, which is the container that includes all content related to a specific design. All data in an Onshape document is stored in Elements. Part Studios and Assemblies are two of the most common element types in a design. Throughout the design process, creating versions can be useful for product development management while working on the "Main" workspace. See also:

- The API Introduction page for information on how documents, workspaces, and elements are assembled into a URL.
- The Associativity page for information on how Parts, Assemblies, and Elements relate to each other.

Elements

All data in an Onshape document are stored in Elements (represented as tabs in the user interface). Onshape documents contain five kinds of elements:

- **Part Studio**: Contains zero or more parts
- **Assembly**: Contains zero or more parts or assemblies
- **Blob** (Binary Large OBject): Can be provided by a partner or by the end user. For example, the user can upload a PDF file, an image, or a text file. Partner applications can store arbitrary data, but we recommend using the structured storage available in an element for better integration.
- **Application**: Presents an iframe to the user. The user interface is managed by a server that can be provided by a third-party. Onshape Drawings are a special case of an application element.
- **Feature Studio**: Contains the definition for Onshape Features, which are defined in FeatureScript.

Workspaces, Versions, and Microversions

A document is stored in Onshape as a collection of changes.

- You can think of a **workspace** as a branch of the document, similar to a branch in a source control system. Documents can be branched to create new workspaces.
- Each individual change to the document creates a new document **microversion**. As the document is edited, changes are applied to the active workspace, creating new microversions.
- Periodically, the user may designate versions of the document. A **version** is a named snapshot of the entire document at some point in time (that is, at some microversion).

You cannot change a version or microversion of a document; all changes are applied to a workspace (and create a new microversion). Thus, while in general the GET methods of the API can read from a version, microversion, or workspace, the POST methods generally require a workspace, and create a new microversion when data is written to the document. (An exception is that it is possible to set metadata within a version; this does not create a new microversion).

The following IDs are used by many of the APIs. Each ID (except for Geometry IDs such as Part, Face and Edge) is a 24-character string that is used internally by

Onshape to uniquely identify the resource. The Geometry IDs are variable-length strings used to resolve to a specific geometric entity within a model.

ID	Description
User ID	Identifies a single user.
Document ID	Identifies a document. The logged-in user must have access to the requested document for the API to succeed.
Workspace ID	The Workspace ID identifies a workspace within the document. Workspaces are used to distinguish between different branches of the document.
Version ID	The Version ID identifies a specific named version.
Microversion ID	The Microversion ID identifies an internal revision of the document.
Element ID	The Element ID identifies an element within the document.
Part ID **Face ID** **Edge ID**	The Part ID identifies a part within a part studio. The Part ID should generally not be stored for long-term use, as it is only expected to be valid during the course of a session.

Note that a Part ID may reference a part that no longer exists if the model is changed, so it is best to specify a Version or Microversion to pick the context for the Part ID. Note that even with the Version or Microversion, internal changes to the Onshape system may also change the Part ID. Onshape provides mechanisms for maintaining persistent references. See the Associativity page for more information. Face and Edge IDs are used in similar ways.

The following table identifies Onshape concepts and the corresponding Git concepts. Note that this is not a direct mapping, and the implementation of the concepts is very different.

Onshape concept	Git concept
Document	Repository
Element	File
Workspace	Branch

Onshape concept	Git concept
Version	Tag
Microversion	Commit

Linked Documents

Although a document can contain a complex model tree involving many Part Studio and Assembly elements, it is often more efficient to split the content into multiple documents. Connections between documents always refer to a specific version of the target document. *Once a version is used as the target of a linked document, that document version is preserved as long as any document references it, even if the containing document is deleted.* Additionally, any user that has access to the referring document will have limited read access to the target document, regardless of what permissions are currently on the target document.

Configurations

Onshape Part Studios can be constructed to be configurable using Onshape Configurations. API calls that reference Part Studios (primarily within the Parts and Part Studios APIs) often accept a `configuration` parameter that identifies what specific configuration of the Part Studios is being referenced. When not specified, the API implementation typically uses the configuration that is currently selected within the Part Studio. An interactive ad-hoc API call might not behave consistently in an application, so be sure to specify the configuration parameter where applicable.

Onshape Data Model

Onshape data is stored in replicated databases in the cloud. The Onshape data model is influenced by the Git data model and similar source code repositories.

Documents contain **elements**. Elements are presented as tabs in the user interface. With some exceptions, all data in a document is stored within an element. The following table describes what data stored in each Element type:

Element Type	Description
Part Studio	Each Part Studio contains exactly one Feature list. The Feature list contains Features such as sketches, planes, extrudes, etc. Each Feature contains one or parameters. Whenever the

Element Type	Description
	Feature list changes, the parametric history is evaluated, and the model is regenerated.
Assembly	Each Assembly contains an assembly tree, which contains parts and/or other assemblies (sub-assemblies), along with mate information. Onshape provides an API call to retrieve the assembly tree definition.
Blob	Each Blob element contains an uninterpreted binary object that has been uploaded to Onshape, typically from a file. Onshape depends on the browser client to display some blob data (e.g., PDF and image data), but does not interpret the data. A blob element can be updated with new data.
Application	Each Application element contains zero or more sub-elements, providing a structured set of transactional data that is defined and managed by an application. Application data can be displayed in the Onshape tab in an iframe; the application is responsible for rendering the data in the iframe from its server.

Note that Onshape Drawing elements are Application elements managed by Onshape.

Tessellated data is not stored persistently in Onshape; it is generated on demand for display by the Onshape clients, or in response to application REST API requests. This data may be cached for performance.

Document data

All elements, including Assemblies, Part Studios, Drawings, or even apps, are history based. Each change to an element or set of elements represents a unique record in the document's history, known as a microversion. The document can be restored to that particular state any time in the future.

Part Studio data

The Part Studio element is defined by a list of features, some of which (e.g., a sketch), may have a complex internal structure composed of entities. Part Studio features and entities are referenced by unique persistent identifiers. Part Studio

features and entities can appear, disappear, and reappear depending on the current microversion of the model.

Assembly data
The Assembly element is defined as a list of assembly features and a tree of subassemblies/part instances. Occurrence ID is a unique persistent identifier of an occurrence of a part in the assembly structure.

External application data
An external application has complete control over how it manages/stores documents, however, to take advantage of the Onshape data model, there is a set of endpoints they should use to store state. These are collectively known as the AppElement API.

Model presentation data
A valid model definition usually corresponds to a real-world manufacturable topology, represented internally as a set of parts, faces, edges, and vertices and the set of relations between them. Each of these has a unique identifier in every state of the model. The identifier represents an encoded index in the model's history, and its value depends on the structure of the model's history. The value is not guaranteed to be preserved across model changes, and will almost always change if the model changes in significant ways. The model can be tessellated into a set of geometric primitives, which approximate the shape of the model. Tessellated data can be used for visual representation of the model or other processing related to the shape of the model.

The following changes in the topological representation can occur between two microversions of the model:

- New topological entities appear
- Id of existing topology change
- Topological entities disappear
- Existing topological entities are merged into a single entity
- Existing topological entity are split into multiple entities

The model microversion and topology ID can be used to identify topological entities across the model changes. Topology ID defined in a specific microversion can be translated into a set of topology IDs in the current microversion of the model. (The Topology ID is sometimes referred to as a Deterministic ID within Onshape, and is

exposed in specific API calls as partId, faceId, etc.). See the PartStudio APIs to see what topology IDs are exposed.

Quick Start

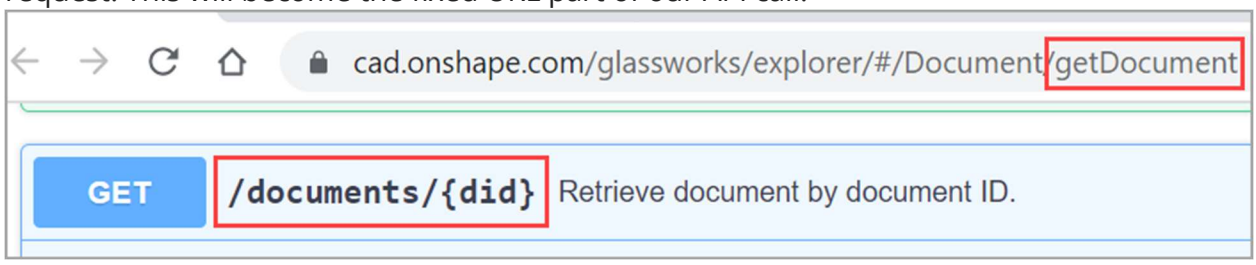
In this example, we will call an Onshape REST API endpoint to send a document name to our console. Please note that the sample shown on this page is only designed to be used as a quick start guide and does not represent a full Onshape application.

System Requirements

- You must be signed in to your Onshape account at https://cad.onshape.com (or https://companyName.onshape.com for Enterprise accounts).
- This example is coded in Python. The equivalent code is provided in other languages at the end of the example. To follow along with this tutorial, you can download and install Python here: https://www.python.org/downloads/.

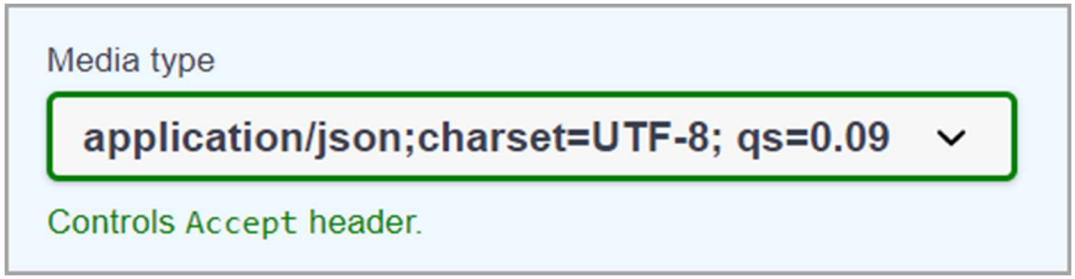
Review the API Endpoint

1. Go to the API Explorer and scroll to `Document`.
2. Expand the `GET /documents/{did}` endpoint. Note that in the URL, the name of this API is `getDocument`.
3. Make a note of the URL structure and the parameters required to make this request. This will become the fixed URL part of our API call.



For this endpoint, we only need to get the document ID from the document URL.

4. Scroll down and make a note of the `Media Type` that we'll need to include in our header.



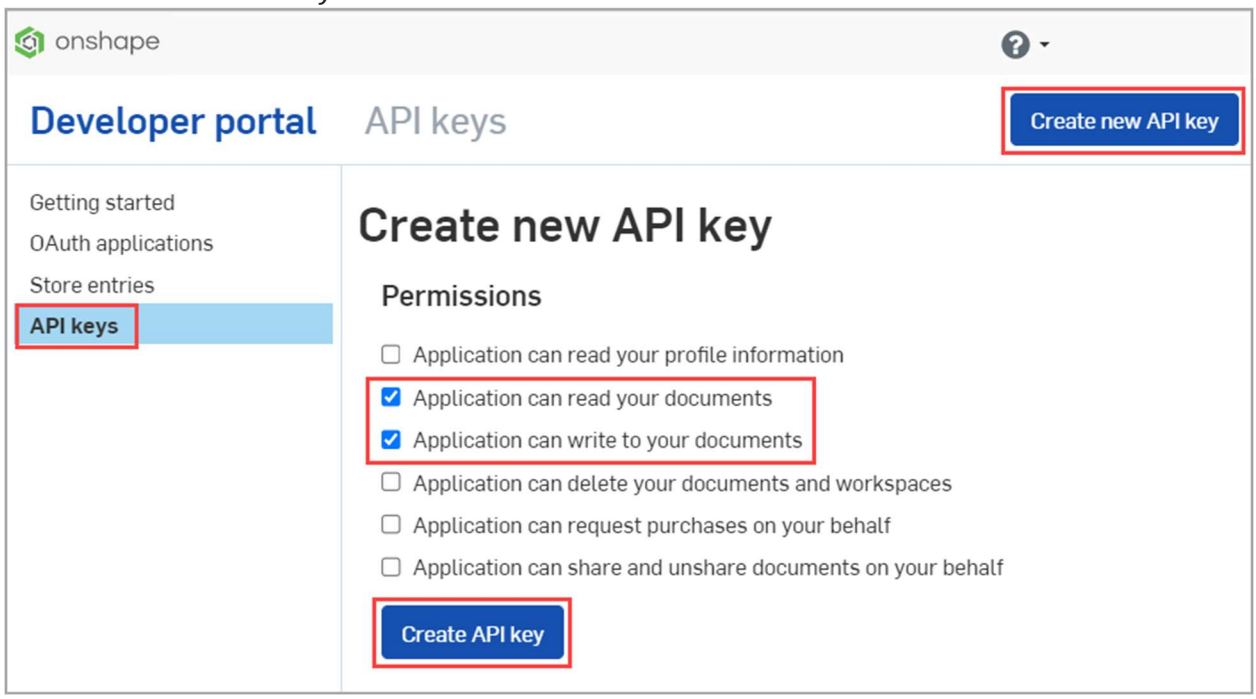
Review the Document

Navigate to [this public document](), and make a note of the document ID in the URL (`e60c4803eaf2ac8be492c18e`).

Create your API Keys

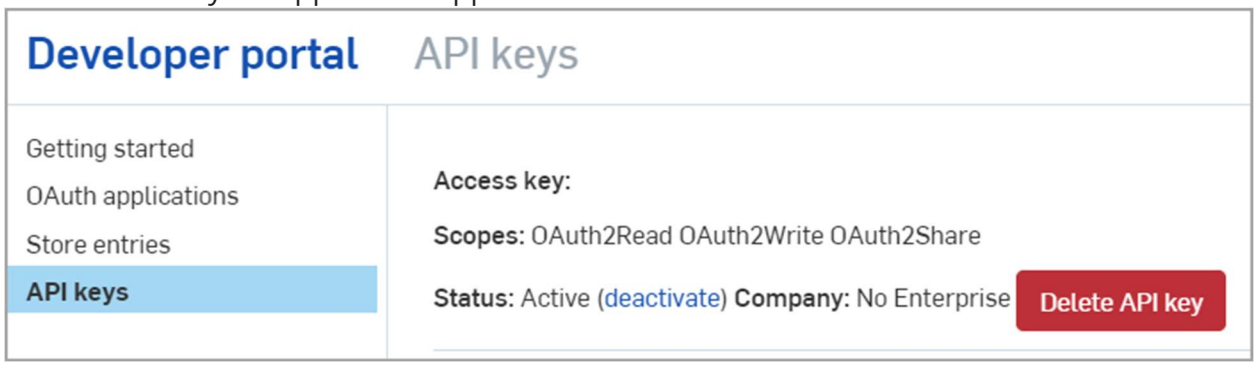
1. Go to [https://dev-portal.onshape.com]().
2. In the left pane, click `API keys`.
3. Click the `Create new API key` button.
4. Select the following permissions for your app:
    - `Application can read your documents.`
    - `Application can write to your documents.`

5. Click the `Create API key` button.



6. Copy both the **access key** and **secret key** from the pop-up window, save them somewhere, then click the `Close` button.
IMPORTANT NOTE: You will not be able to find the secret key again, so save it somewhere safe!

7. The details for your application appear.



8. Open your terminal and run the following command, replacing `ACCESS_KEY` and `SECRET_KEY` with the **access key** and **secret key** you created above. Remember to include the colon (`:`) between the keys.
    - **MacOS**:
    - `printf ACCESS_KEY:SECRET_KEY | base64`
    - **Windows**:
    - `powershell "[convert]::ToBase64String([Text.Encoding]::UTF8.GetBytes(\"ACCESS_KEY:SECRET_KEY\"))"`
9. You will receive a long, base-64-encoded string. You will need this string later, so keep it somewhere safe. We'll refer to it as our `CREDENTIALS`.

Write Your Code

1. Create a new file called `hello.py`.
2. Start your file by importing the necessary libraries.

```
import requests
import jsons
```

3. Next, define the URL for the API call:

```
# Assemble the URL for the API call
api_url = "ASSEMBLED_URL"
```

4. Replace `ASSEMBLED_URL` with the fully formed API. This is where we'll put together everything we've learned so far:
    1. The base URL:
        - `https://cad.onshape.com/api`
        - `https://companyName.onshape.com/api` for Enterprise accounts
    2. The fixed URL is specified in the `getDocument` API in Glassworks: `/documents/{did}`
    3. The document ID parameter from the public document URL to include in the fixed URL: `{did}: e60c4803eaf2ac8be492c18e`

4. Together, this makes the URL for our API request: `https://cad.onshape.com/api/documents/e60c4803eaf2ac8be492c18e`

5. We don't need to send any optional parameters with our request, so we can define them as an empty object:

```
# Optional query parameters can be assigned
params = {}
```

6. Now, define your API keys:

```
# Use the encoded authorization string you created from your API Keys.
api_keys = ("CREDENTIALS")
```

7. Replace `CREDENTIALS` with the string you created in the last section.
8. Next, define your headers:

```
# Define the header for the request
headers = {'Accept': 'MEDIA_TYPE',
           'Content-Type': 'application/json'}
```

9. Replace `MEDIA_TYPE` with the Media type we obtained from the API Explorer during the Review the API section above:

```
application/json;charset=UTF-8;qs=0.09
```

10. Put all the variables you just defined together into the request:

```
# Put everything together to make the API request
response = requests.get(api_url,
                        params=params,
                        auth=api_keys,
                        headers=headers)
```

11. And finally, print the name value from the response:

```
# Convert the response to formatted JSON and print the `name` property
print(json.dumps(response.json()["name"], indent=4))
```

12. Make sure your file matches the full example below:

```
import requests
import jsons

# Assemble the URL for the API call
api_url = "https://cad.onshape.com/api/documents/e60c4803eaf2ac8be492c18e"

# Optional query parameters can be assigned
params = {}
```

```
# Use the encoded authorization string you created from your API Keys.
api_keys = ("CREDENTIALS")

# Define the header for the request
headers = {'Accept': 'application/json;charset=UTF-8;qs=0.09',
           'Content-Type': 'application/json'}

# Putting everything together to make the API request
response = requests.get(api_url,
                        params=params,
                        auth=api_keys,
                        headers=headers)

# Convert the response to formatted JSON and print the `name` property
print(json.dumps(response.json()["name"], indent=4))
```

Run Your Code

1. Open your terminal and navigate into the folder where you saved your `hello.py` file: `cd ~/<your-file-path>`
2. Install the necessary modules:

```
python3 -m pip install requests
python3 -m pip install jsons
```

3. Run your code:
   `python3 hello.py`
4. Confirm that your console displays:
   `"Onshape API Guide"`

Other Language Examples

Remember to replace `CREDENTIALS` with your credentials.

cURL

Returns the entire response json. Scroll to the bottom to the see `name` field.

```
curl -X 'GET' \
https://cad.onshape.com/api/documents/e60c4803eaf2ac8be492c18e' \
  -H 'Content-Type: application/json' \
  -H 'Accept: application/json;charset=UTF-8; qs=0.09' \
  -H 'Authorization: Basic CREDENTIALS'
```

JavaScript

```
import fetch from 'node-fetch';

async function getDocument(url='') {
    const response = await fetch(url, {
      method: 'GET',
```

```
      headers: {
        'Content-Type': 'application/json',
        Accept: 'application/json;charset=UTF-8;qs=0.09',
        Authorization: `Basic ${btoa('CREDENTIALS')}`
      }
    });
    return response.json();
}

getDocument('https://cad.onshape.com/api/documents/e60c4803eaf2ac8be492c18e'
).then((data) => {
    console.log(data.name);
});
```

C++

Returns the entire response json. Scroll to the bottom to the see `name` field.

```
#include <iostream>
#include <string>
#include <stdio.h>
using namespace std;

int main() {
    string url = "curl ";

    url += "-X 'GET' ";
    url += "'https://cad.onshape.com/api/documents/e60c4803eaf2ac8be492c18e'";
    url += "-H 'accept: application/json;charset=UTF-8; qs=0.09' ";
    url += "-H 'Authorization: Basic CREDENTIALS'";

    system(url.c_str());
    return 0;
}
```

Python

```
import requests
import jsons

# Assemble the URL for the API call
api_url = "https://cad.onshape.com/api/documents/e60c4803eaf2ac8be492c18e"

# Optional query parameters can be assigned
params = {}

# Use the encoded authorization string you created from your API Keys.
api_keys = ("CREDENTIALS")

# Define the header for the request
headers = {'Accept': 'application/json;charset=UTF-8;qs=0.09',
           'Content-Type': 'application/json'}

# Putting everything together to make the API request
response = requests.get(api_url,
```

```
                        params=params,
                        auth=api_keys,
                        headers=headers)

# Convert the response to formatted JSON and print the `name` property
print(json.dumps(response.json()["name"], indent=4))
```

Why Onshape?

Why Onshape?

As long as there have been applications that manage organizational data into a database, there has been a need to share that data between different departments and therefore, usually, different systems. In a typical design/manufacturing organization, there could be at least four or five mission-critical databases that manage the data for different departments and for different stages in the product's lifecycle.

Initially, these systems provide the capabilities required by their consumers (i.e., the departments that use these systems). For instance, the Finance might use QuickBooks, Manufacturing might use a manufacturing planning and execution system (MES), Engineering might use a Product Data management System (PDM), and so on for each group in the organization.

This often leads to disparate silos of data and knowledge. The departments in an organization do not work in a vacuum; each is dependent on information generated by other groups. For instance, Manufacturing can't produce accurate assembly instructions without input from engineering on the designs and the bill of materials. Finance can't price the product without understanding its contents and which parts are manufactured in-house or purchased.

Therefore, the need to integrate these systems becomes critical for the organization to function optimally. Initially, connecting one system to another can be a straightforward process. This usually involves some services to get the systems to talk to each other, however it isn't too painful as long as the requirements are clearly defined.

Anyone who has implemented integrations between PLM (Product Lifecycle Management) systems or ERP (Enterprise Resource Planning) systems will tell you of the nightmare scenarios they encountered. Often this is the result of poorly

scoped and defined requirements, conflicting requirements coming from multiple departments, and the many integration points required between systems. The result is that the organization is not getting what it wants or needs, the customer is paying for services that do not provide the promised solution, and usually the project is long overdue. All this equals an unhappy customer and often the software vendor's solutions are blamed for the disaster.

Over the years, many technologies have appeared (and some of them, just as quickly disappeared) to enable integration without the need to write thousands of lines of custom code that needs to be re-written for every software upgrade. Several technologies provide "codeless" integration between SaaS products (Zapier, for example). These solutions are particularly good for generic use cases for data exchange between systems, but can be limited when it comes to custom modifications to the data being sent that might be required by a specific customer. In addition, they have the overhead of requiring a subscription to their service. Sending corporate IP through another third-party can also cause data security issues.

Therefore, we can understand that in most organizations integration between systems is a necessary evil that must be tackled, either with an out-of-the-box solution or through some custom coding.

Early on, Onshape understood that as an engineering system, it cannot exist in a vacuum; it must be able to communicate with other systems. For this reason, the REST API was developed.

An API, or *application programming interface*, is a set of rules that define how applications or devices can connect to and communicate with each other. A REST API is an API that conforms to the design principles of the REST, or *representational state transfer architectural style. For this reason, REST APIs are sometimes referred to RESTful APIs.*

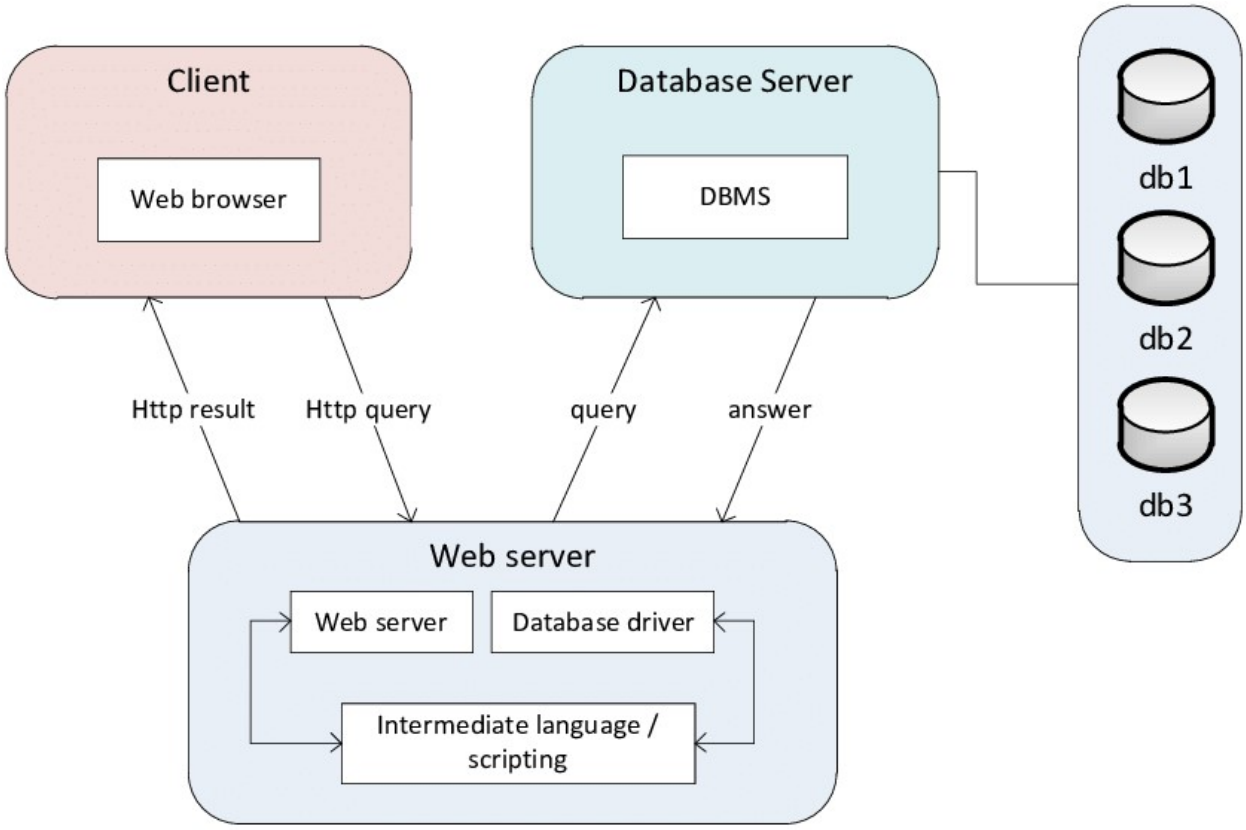
Onshape SaaS

Onshape was built from the ground up as a true SaaS-based system; Onshape had no investment in legacy code and was able to develop an application that truly runs as a multi-tenant SaaS solution from the first line of code. Many companies claim to run cloud-based solutions, but since they have such a large investment in their legacy code, that they can't just discard and start again from scratch. Instead, they tend to try and port that code to the web.

More often than not, porting existing code to the web and calling it a SaaS solution is no more than a marketing ploy; it isn't a true SaaS solution if it wasn't written as one. These are generally known as cloud-hosted solutions. This means that a typical three-tier data management solution (which could have previously been installed on a set of servers), has now been modified to be hosted on the web.

Traditional three-tier architecture

Traditional PLM systems typically use a three-tier architecture, mainly consisting of an application server, a database server, and a client (either a web client or a thick client installed on the client hardware).



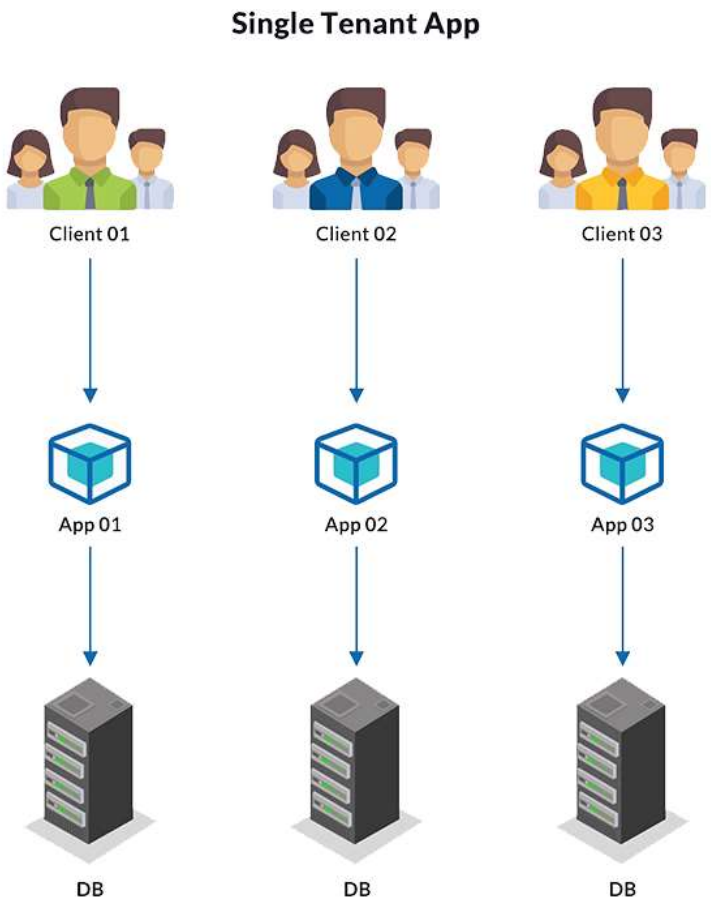
Typical three-tier architecture

To connect to and integrate with this architecture, APIs are usually exposed on the application or web server. If this architecture is ported to the web, it cannot make customizations through the API, since it would modify the behavior of the program for everyone connected to that application server.

Single- vs multiple-tenant architectures

The three-tier architecture is typical of most PLM solutions on the market today, which is fine if you want the solution to be installed on company servers and be accessible to people within the company only.

When this type of solution is ported to the web, software vendors typically must create a single-tenant application where an application server and a database server are provisioned for each new customer.

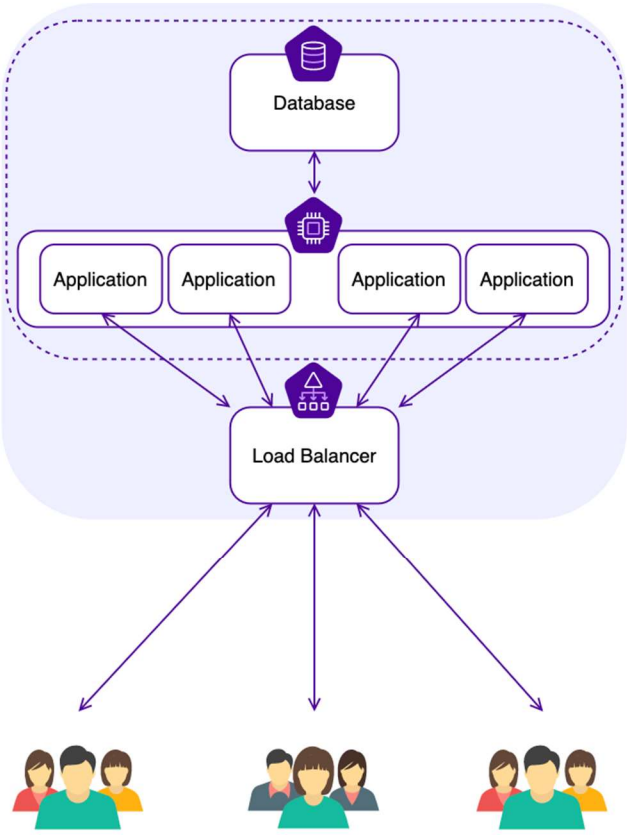


Single-tenant architecture

In this case, the vendor must use expensive hardware to host more customers, which is not a sustainable model.

Modern 21st century software solutions use multi-tenant solutions that can be hosted on services such as Amazon cloud, Azure, etc. There are many benefits to

this architecture, including that servers can be provisioned and decommissioned on the fly to provide ultimate performance whenever required. Since servers cost money, decommissioning servers when they are not required is a key benefit to a true SaaS solution.



Multi-tenant architecture

Since each application is separate in this architecture, we can enable customizations that can't be implemented in a single-tenant architecture. For example, we can provide access to the REST APIs that are required for Onshape integration. In the single-tenant architecture, if you provide API access to the application server, one customer will be modifying that application for all customers who are registered on that tenant.

The Onshape Difference

Onshape does not work like other legacy CAD systems. Onshape was built from scratch for the cloud and as a modern CAD system, so many of the failings of legacy CAD systems were excluded.

There are many differences and benefits to Onshape, which are well-documented in the Onshape Help and training materials.

The information in this section is specific to integrations, since Onshape does not behave like a traditional file-based systems. When writing an integration for Onshape, it is critical to understand the nuances in Onshape's design practices and how data is organized in Onshape.

Data-driven/fileless

Most traditional PDM/PLM systems integrated with CAD systems enable this integration on a per-file basis. This means that you have an object in the PDM/PLM system that corresponds directly to a file in the CAD system. In this way, the PDM/PLM system can manage access to the files, build assemblies from the files, view the CAD data, and much more.

Onshape, however, does not work this way.

Being data-driven means that Onshape has no files, just data, so an integration into Onshape is going to look different from any integration to a CAD system that you might have done previously.

In traditional CAD, a single file represents a snapshot of what the design looked like at a specific moment in time. Unless it's changed, it will remain in that state forever. PDM systems manage these files, and once a designer decides to make a revision or a release, the file is locked, and a new file can be created to represent any further updated versions or releases of the design. PDM/PLM systems are very good at managing this data in an up-to-date structure, but it does have the drawbacks. They generate many file copies of a specific design, and once a file is taken out from the system (for instance, to share with a supplier), it is no longer managed and tracked.

Onshape uses data instead of files. The data is always up-to-date and can be collaborated on in real-time without the need to send file copies back and forth. This means that Onshape views versions and releases differently than those traditional systems do. When integrating with Onshape, we must design for data rather than files.

Files can be generated from the Onshape data. For example, you can generate a PDF of a drawing upon release or of a STEP file that can be used by other downstream systems.

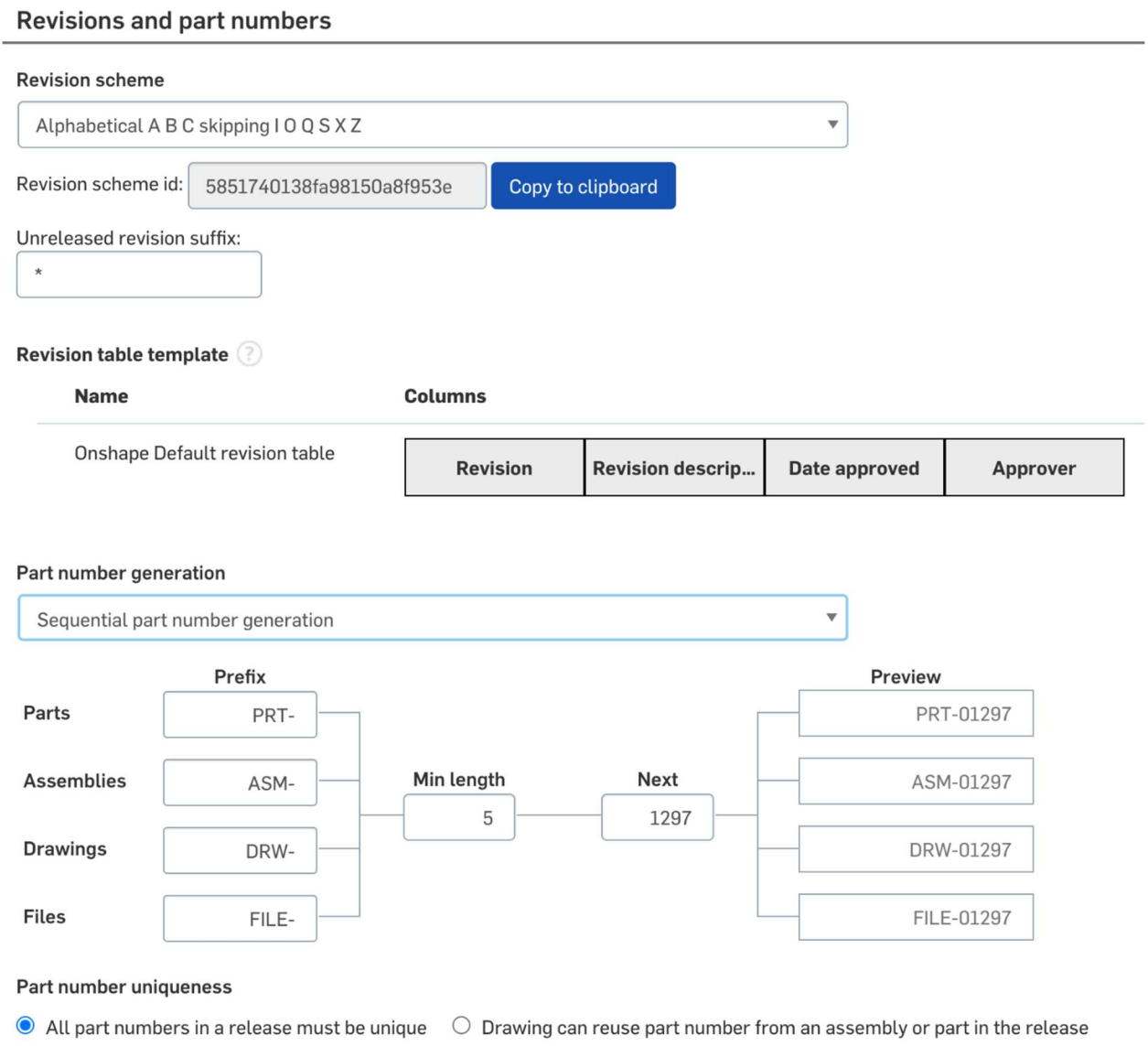
A key benefit of a data-driven system is the ability to retrieve detailed, real-time analytics. Onshape has comprehensive analytics; including who can view or edit a design, when and exactly what edits are made, which commands were used, and how long was spent modifying the design.

Built in PDM

Up until now, CAD was one software program, and PDM/PLM was another program that had to be integrated with the CAD. In many cases, both programs could be sold by the same software vendor (even though there are many PLM systems available that are sold by independent vendors who have no CAD system). Regardless, a PDM/PLM system always had to be an added solution to the CAD system.

No matter how deep the integration between a CAD system and a PLM system, there is always the need to sync data between the two. This is usually a weak point in any solution that is prone to errors.

Being data-driven, Onshape already has PDM built in as part of the CAD system. This is unique in the industry: CAD and PDM as part of the same solution with no additional piece of software required.



Onshape's revision and part number schema definition interface

For instance:

- Since the **data is always up-to-date**, the correct state of any design is always represented in real-time with no delay for syncing between systems.
- Unlike file-based systems, **the data is never locked**; it is always available and always changing.
- PDM system **data management aspects are fully integrated** into every aspect of the CAD system.
- True **real-time collaboration/co-design** on both design and data is enabled.

So, what does this mean when it comes to integrating Onshape with another PLM system? First and foremost, we must understand that there are many things that a PLM system does that Onshape's PDM capabilities can't do. Integrating Onshape to a PLM system should augment the powerful capabilities already available inside Onshape, not necessarily replace them. Similarly, Onshape does not replace PLM-native capabilities. Instead, depending on the business case, we can use the best-in-class capabilities of each system to augment the other.

The Onshape release process is an example of the augmentation of each system's capabilities. Onshape has a specific way of managing the release of data that is different from traditional PDM systems. This capability is inherently suited to a data-driven approach and provides a lot of value to the update of design data in Onshape. At the same time, PLM systems provide enterprise release processes that may include many people and different departments that extend beyond the engineering domain. Such PLM processes can be highly customized and suited to the organizations established business processes.

In this scenario, it doesn't make sense to avoid the enterprise release processes in the PLM system. However, also omitting Onshape's release capabilities could put data between Onshape and the PLM system out of sync and prevent Onshape from updating data (e.g., watermarks and title blocks on drawings, icons related the visualizing the state of data, etc.).

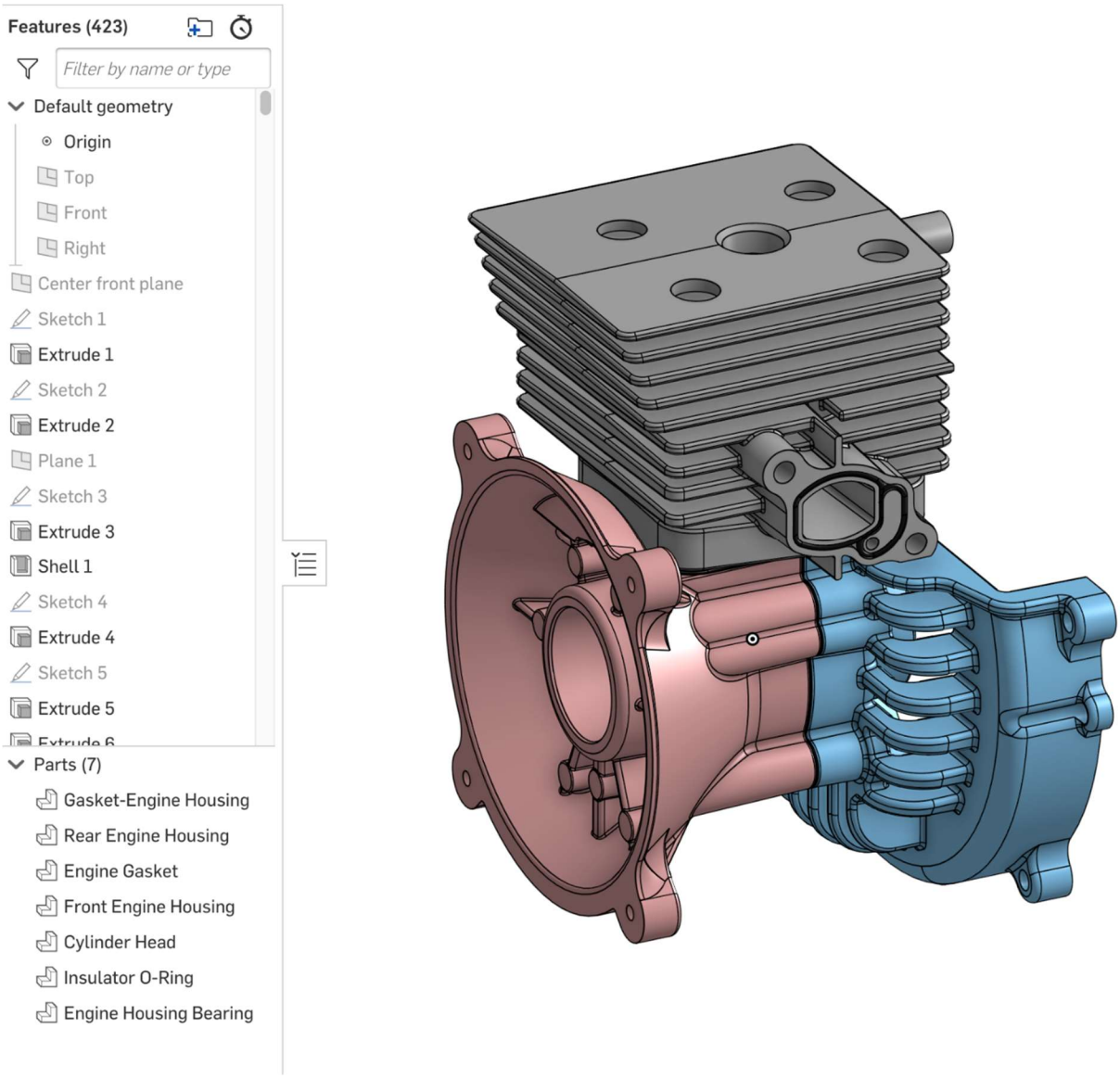
In this case, we want to use the best-in-class features of each software solution without compromising the capability provided by each solution. If we plan our integration correctly, this can be achieved by initiating the release of the data in Onshape, transferring the release data to the PLM system where the release process will be triggered, and finally automating the release in Onshape once the process has been completed in the PLM system.

Multi-part Part Studios

In traditional CAD systems, one file typically equals one part. While design-in-context is available in most CAD systems, and multiple solid bodies can be created, each part is self-contained in a separate file. For PLM systems, this makes it easy to associate an object in the PDM/PLM database with a specific CAD file. *This is not the case in Onshape.*

In Onshape, parts are designed in what's called a *Part Studio.* Within a Part Studio, the designer is free to create as many parts as they want. The general rule is that the parts should be related to each other in a system, thereby making it easier to

design one part from another, however there is a lot of flexibility in how the designer wishes to work.



An example of a multi-part Part Studio in Onshape

The structure of the Onshape document is discussed in detail in the Onshape Architecture page. The Part Studio is included in an Onshape document.

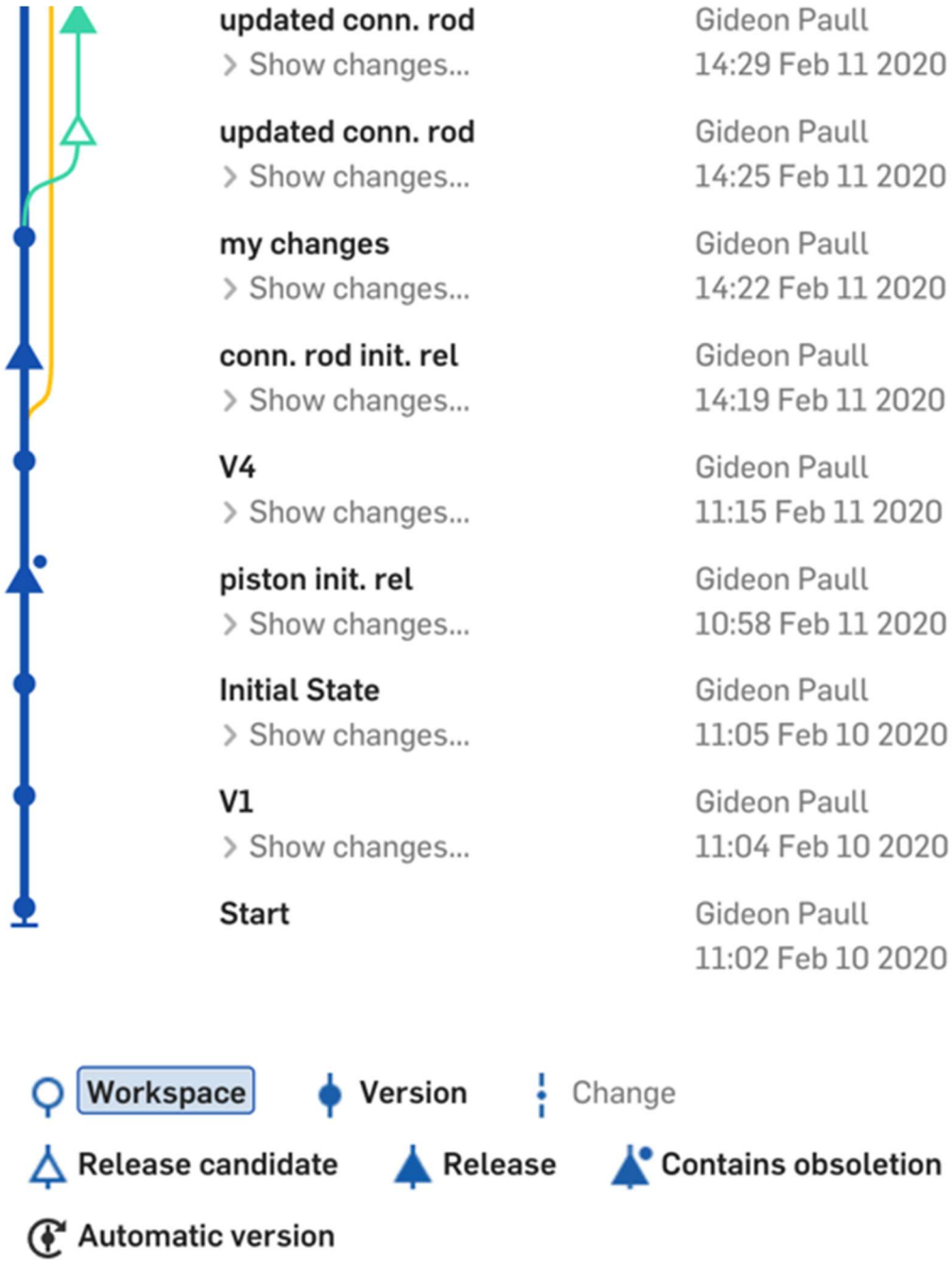
We can already begin to understand that the traditional CAD/PDM paradigm of "one file per object" will not work with Onshape; the designer would be forced by the PDM/PLM system to only create one part per Part Studio. This would therefore

limit the designer's freedom for creativity in Onshape and seriously reduce the powerful functionality available for the designer to use.

Therefore, we need to re-think how we integrate with Onshape versus how we integrate with traditional CAD systems. Fortunately, Onshape's REST API supports the multi-part Part Studio scenario. Instead of associating a file with an object in the PDM/PLM database, we now use the REST API to associate a Part with its corresponding object.

Versions and releases

Traditional PDM/PLM systems provide design release support by locking a CAD file for access. The access controls are defined in the database and the definition of a part/assembly/drawing as released is controlled by the database. When a new revision of the part is required, a file copy is made, and the database provides access to the new copy. Generally, the old copy representing the previous release persists in the file store and can be referenced by the database. *This is not how Onshape works.*



Since there are no files in Onshape (just data) no file locking or copy mechanisms are available. Instead, Onshape looks at the data as a continuous timeline that is always moving forward and always changing as the design evolves. The data is never locked; it is always available.

In place of file copies that represent versions and releases of the design, Onshape provides the ability to create *versions* as bookmarks in the timeline. When creating a version, Onshape places a bookmark in the timeline that represents the state of the

design at that specific moment in time. Releases work in a similar way, but they are defined as official, company-approved processes and have special meaning.

In addition to creating versions and releases, Onshape can create *branches*, which can be defined as alternative timelines. A designer might want to experiment with alternate design ideas without modifying the existing design that others are working on. By creating a branch from any point in the timeline, the designer is free to experiment with alternate ideas. If the ideas work, they can be merged into the current timeline at any point.

From an integration perspective, we need to take into consideration how Onshape works with versions and releases. Since a release represents a company-approved design, Onshape provides processes for the approval of a release and the change of state of a design. Onshape also provides APIs and triggers (events) that enable integration points throughout the release process. It is through the triggers and the APIs that integration of any third-party system that wishes to manage the release process is enabled.

Workflows

Release and obsoletion workflows are included with Onshape and can be customized to meet company standards.

For details on how to implement and customize Onshape's workflows, please review these online help topics:

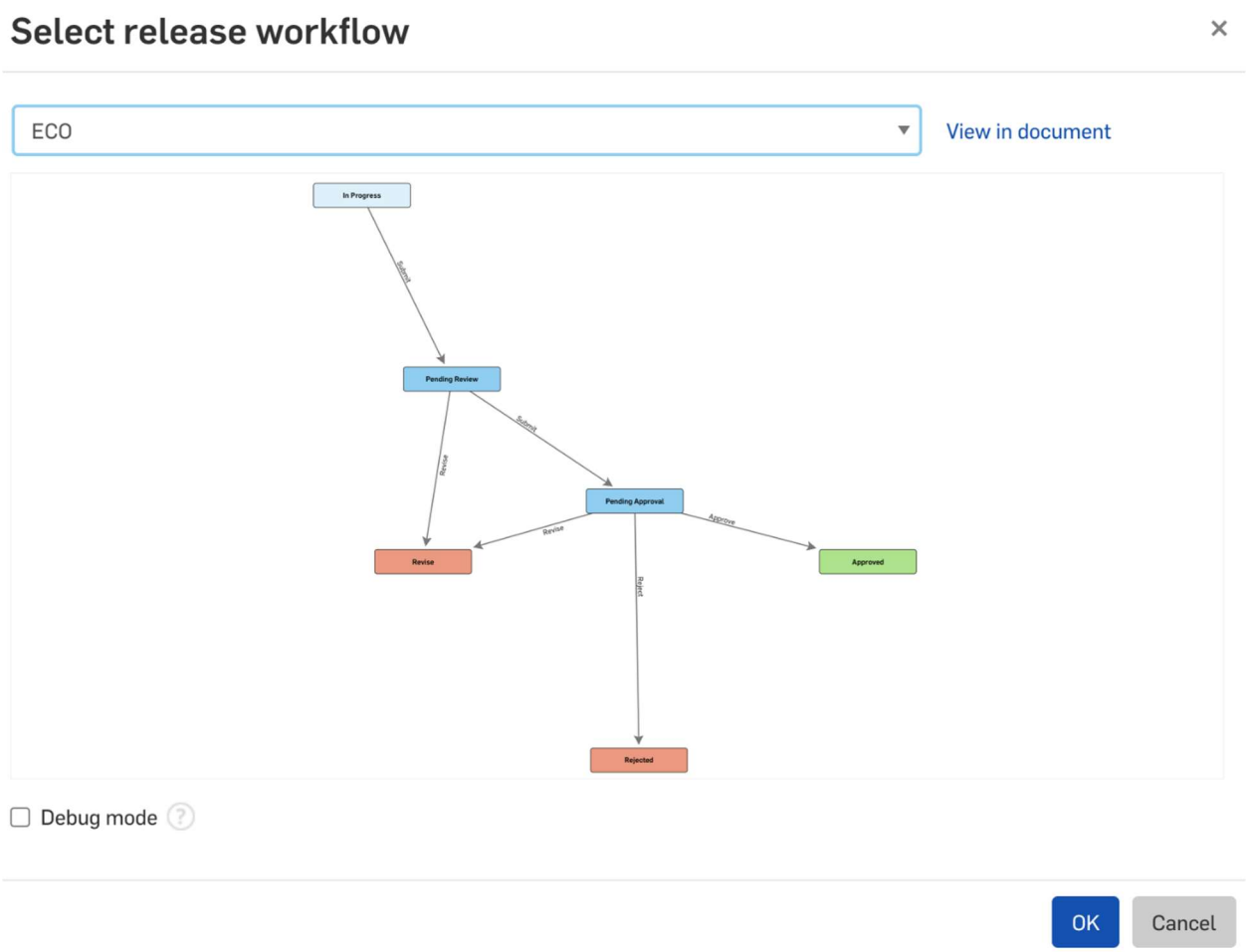
- How to design release management processes
- How to create a customized release workflow

Most PDM/PLM systems can model a company's business processes in a workflow. These can be highly automated processes that move data and file references through a process of reviews and approvals. Onshape also has this capability, which is currently used for release and obsoletion processes.

There are no files or file references in Onshape that are moved through the process. Onshape only has data. Therefore, it is the data that is referenced at each stage of the process. Traditional PDM systems might make file copies and lock files as they move through a release process. If the process is rejected at any stage, those files must be discarded, the previous version of the files unlocked and all states updated. In short, it system must rewind back to the state of the files and the

data when the workflow was initiated. This is a lot of complex actions that must occur when a process is rejected for any reason. *Onshape doesn't work this way.*

A release process can be started on data (such as assemblies, parts, drawings, etc.). For example, if the state of a referenced part is updated to "Pending," and the process is rejected at any stage, there is no rewinding of files and data; the data just reverts to the original "In Progress" state, and the workflow is discarded. Since the workflow didn't complete, nothing related to the data has actually changed. When you are used to traditional PDM systems, this feels like an anti-climax, and we often receive the question, "But where's my process? Where's the data that was attached to the process?". Well, the answer is: nothing changed. Until the process is completed, nothing actually changes, so the data is in the same state it was prior to the initialization of the release process.



A custom release process in Onshape

OAuth

See the gltf-viewer-app for a working example of OAuth2.

What is OAuth2?

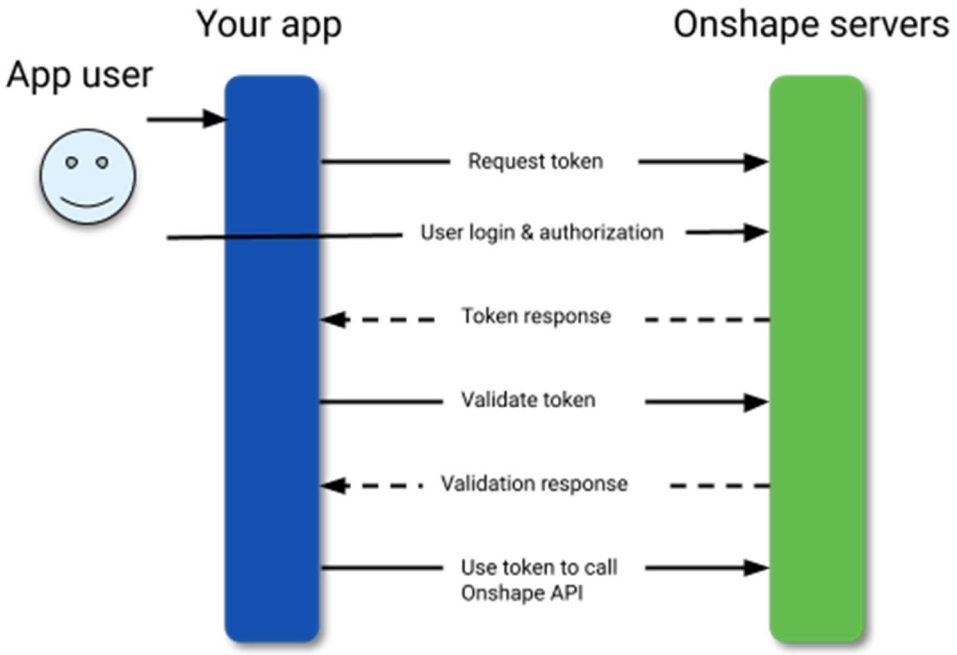
The OAuth (Open authorization) protocol was developed by the Internet Engineering Task Force, an open standards organization that develops and promotes voluntary Internet standards (particularly the technical standards that comprise the Internet protocol suite) to enable secure, delegated access to an application's resources.

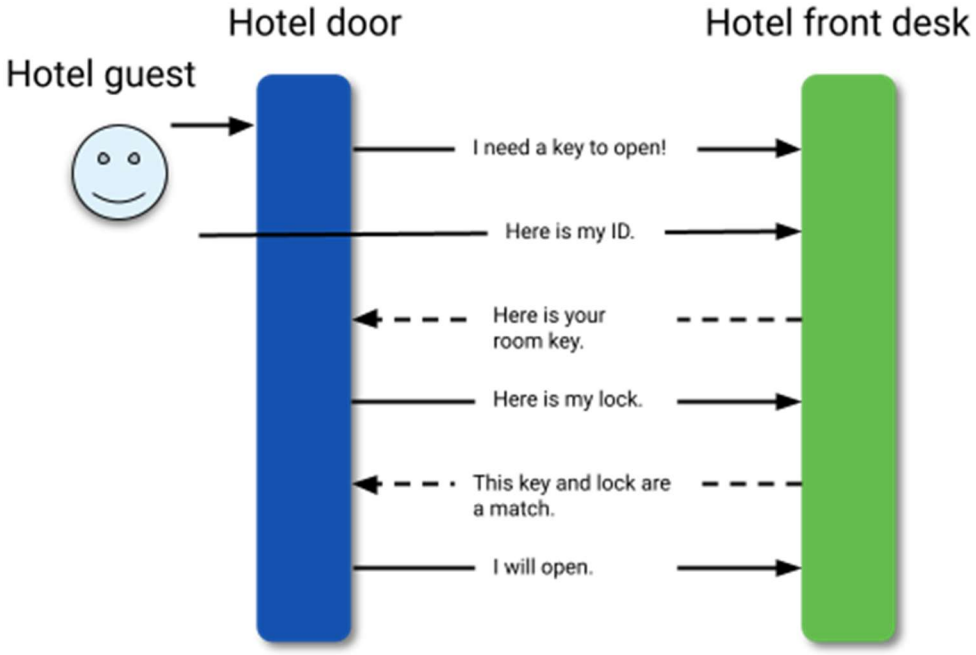
The OAuth2 protocol enables an application to access a resource that is under the control of someone else. In order to access that resource, a *token* is required. The token represent the delegated rights of access (that is, what rights this application has, such as read/write/update, scope, rights to different resources, and more).

This means the application can be accessed by a third-party system without that system impersonating the user that controls the resource.

A good analogy is the hotel check-in process. When you arrive at the front desk of a hotel, you provide an ID and a form of payment. Then, you are given a key card that opens a specific door. When you reach that door, you swipe your key card and are granted access. The door itself doesn't know who you are or anything about you, it just knows that the key card was encoded correctly, and it allows you access. At some point, the key card expires, and the door no longer lets you into the room. This is the same for access tokens in the OAuth2 flow.

With the OAuth2 protocol, you register your application with the third party, and you are given a set of keys. These keys get exchanged for an access token that grants you access to resources in the third-party application. The token expires regularly; you miust get a new token to access the application again. For this, you are provided with a *refresh token*. Sending the refresh token to the authentication server updates your access token and gives you a new refresh token.





OAuth2 & Onshape

The first step in the OAuth flow is for the Onshape user to request that Onshape let the third-party application access Onshape.

Once the user has authorized the application, they are redirected to a predefined URL (called a *redirect URL*) with a code that will requests an access token from Onshape. Therefore, the redirect URL should contain a script that can capture the authorization code.

You will use the access token to authenticate requests to the Onshape API. The token expires after preset amount of time. To get a new valid access token after one has expired, you must use the refresh token to request a new access token. Refreshing the access token also provides you with an updated refresh token to use in the next refresh access token request. Make sure to store both the the access token and the refresh token, and update them with each refresh of the token. The authorization token must accompany any call to the API, this is done by adding the token to an Authorization field in the header of each request:

```
Authorization: Bearer <accessToken>
```

If correctly authenticated, most responses from the REST API call return JSON data (though some return binary data), with an HTTP response code of `200 Success`, `204 - No Content`, or `301 - Permanent Redirect`. `301` responses will include a redirect for you to follow.

In the event that the authorization code is incorrect (for instance, if it expired), you will receive an **HTTP 401** response. This response means that the client request has not been completed, since it lacks valid authentication credentials for the requested resource. In this event, your code for each call to the REST API should include a catch clause for a 401 exception. Once caught, you can refresh the token and make the request again. Pay close attention to the `Content-Type` header for what data to parse and expect.

When integrating with Onshape, OAuth tokens give third-party applications (such as desktop applications or web services) access to users' data as defined by the permissions scope (such as users' documents or profile information). Using OAuth terminology, Onshape acts as both the authorization and resource server, while the desktop or web-based application is the client. Resource owners have the option of granting or denying access to applications.

Once obtained, an OAuth token will work for third-party APIs under `/api`. Do NOT attempt to use an OAuth token to fetch the URLs typically displayed in a web browsers location bar.

More resources

- Digital Ocean - A good resource for learning more about OAuth2.
- RFC 6749 - The reference for the OAuth framework as a whole. Most of this document describes how to implement the OAuth exchanges described by the reference within the context of Onshape and client applications.
- RFC 6750 - Describes the exchange of OAuth access tokens between clients and OAuth servers.

Implement OAuth2

This OAuth tutorial demonstrates how to recreate the authentication process in Node.js found in the gltf-viewer-app sample code. The final code in Node.js and other languages can be found at the end of this page.

1: Register the app

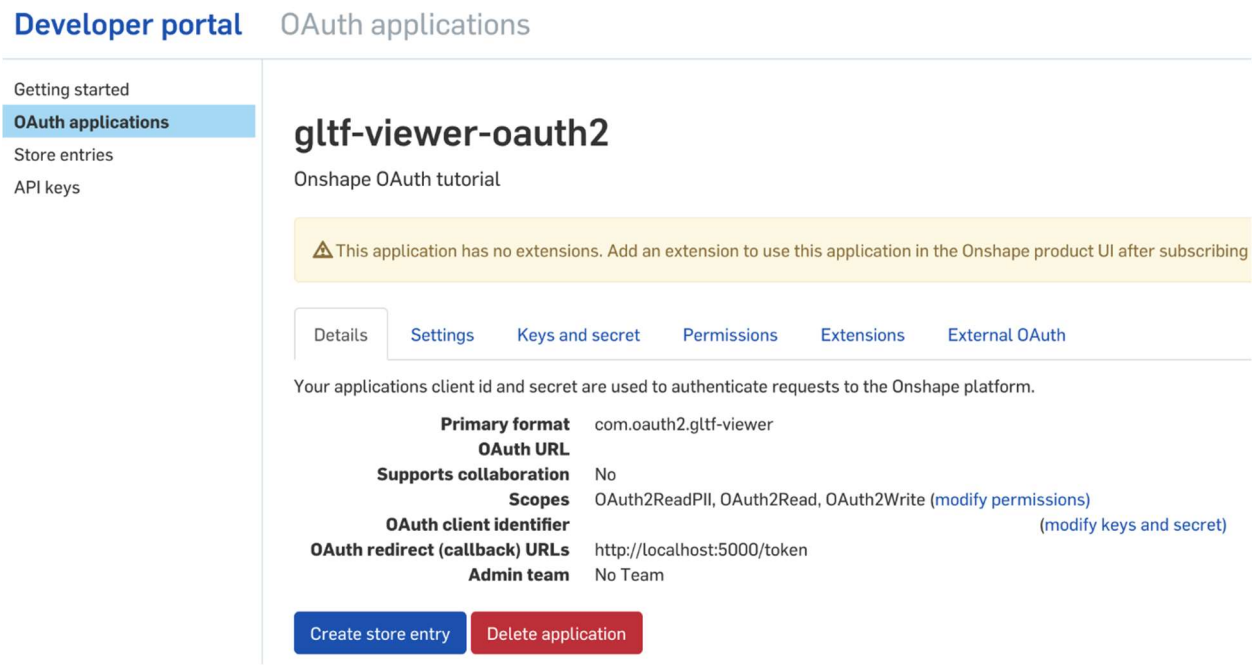
1. Navigate to https://dev-portal.onshape.com/signin and sign in.
2. In the left sidebar, click **OAuth applications**.

3. Click the **Create new OAuth application** button.
4. Fill out the form as follows:
    - Name: `gltf-viewer-yourname`
        - The application name to display to users.
        - Should include the name of your company to differentiate it from other possibly similar applications.
    - Primary format: `com.yourname.gltf-viewer`
        - String that uniquely identifies your application and is a marker for the data it might store on Onshape servers.
        - Cannot be changed after the application is registered.
    - Summary: `Onshape OAuth tutorial`
        - Description of your application.
        - Displayed to the user when they're asked to grant the application permission to access their data.
    - Redirect URLs: `http://localhost:5000/token`
        - Your application must specify at least one URL used in the OAuth protocol exchanges.
        - This URL must also use SSL (a URL that begins with https), with two exceptions applicable for installed desktop applications: `http://localhost:<port>` and `urn:ietf:wg:oauth:2.0:oob`.
        - e.g., `https://app-gltf-viewer-yourname-c11f263794bc.herokuapp.com/oauthRedirect`
    - Admin team: `No Team`
        - Optional.
        - If defined, members of the team can make changes to the definition of this OAuth application.
        - See the Help Docs: Teams page for more information on creating teams in Onshape.
    - OAuth URL: none
        - Should contain the URL of your deployed application.
        - This is the first URL called from the Onshape Applications page.
        - The page hosted at this URL should handle the OAuth authentication. Once your application's server has been authenticated on behalf of the user, that user should be redirected to your applications content.
        - If you have not deployed your app yet, you can leave this field blank (as shown in this example) for local work and update it later.
        - e.g., `https://app-gltf-viewer-yourname-c11f263794bc.herokuapp.com/oauthSignin`
    - Permissions:
        - This is also called application scope, and it defines what access rights your application has to the user's data.
        - **Application can read your profile information** - Enable your application to access the Onshape user profile. Check this option.
        - **Application can read your documents** - Onshape documents created by this user can be accessed with read privileges only. Check this option.
        - **Application can write to your documents** - The user-owned Onshape documents can be modified by this application. Check this option.

- **Application can delete documents and workspaces** - Your application will be able to delete a workspace within a document or the complete Onshape document. Do not check this option for this example.
- **Application can request Purchases on Your behalf** - The application will have access to make purchases if required. Do not check this option for this example.
- **Application can share and unshare documents on your behalf** - Onshape's document sharing capabilities are very powerful; they enable other parties to access your shared documents with predefined rights. If this option is checked, the application can automatically share a document with other people. Do not check this option for this example.

5. Click **Create application**.
6. **COPY THE OAUTH SECRET FROM THE POP-UP WINDOW.**
    - **You will not be able to access this secret again.**
    - **This secret is unique to you and your app and should be protected like any sensitive password. For example, it should *NOT* be checked in to source code control systems.**
7. **Copy the OAuth client identifier from the app Details page that opens.**
    - These OAuth secret and client ID keys will be used in your code for requesting a one-time user authorization code from Onshape.

Your application is now registered with Onshape and you have options to modify the application definition through this portal.



2: Get the user authorization code

We'll start by loading the basic libraries required to run this sample. We'll use Passport to authenticate requests through plugins known as strategies. In this example, we'll use an Onshape-developed plugin called `passport-onshape`, but you can define your own strategy to use with Passport, if you prefer. You can find more information on Passport here.

1. Create a directory for your app, and then install Passport and passport-onshape:

```
npm install passport
npm install passport-onshape
```

2. Next, create a file calls `app.js` and add the following definitions to the top of the file:

```
// App definitions
const path = require('path');
const uuid = require('uuid');

const express = require('express');
const session = require('express-session');
const bodyParser = require('body-parser');

const passport = require('passport');
const OnshapeStrategy = require('passport-onshape');

const config = require('./config');
```

3. Next, tell Express to use Passport and initialize it. Note: you can replace the Express code with code for the web server of your choice.

```
// Tell Express to use Passport, and initialize it.
const app = express();

app.use(express.static(path.join(__dirname, 'public')));
app.use(express.static(path.join(__dirname, 'dist')));
app.use(bodyParser.json());

app.set('trust proxy', 1); // To allow to run correctly behind Heroku when
deployed

app.use(session({
    secret: config.sessionSecret,
    saveUninitialized: false,
    resave: false,
    cookie: {
        name: 'app-gltf-viewer',
        sameSite: 'none',
        secure: true,
        httpOnly: true,
        path: '/',
        maxAge: 1000 * 60 * 60 * 24 // 1 day
    }
}));
app.use(passport.initialize());
app.use(passport.session());
```

4. Next, we'll store the Onshape user information so it can be retrieved from `req.user` in each call. Passport uses the `serializeUser` function to persist user data (after successful authentication) into the session. The function `deserializeUser` is used to retrieve user data from session.

```
//Store the Onshape user information
passport.serializeUser((user, done) => done(null, user));
passport.deserializeUser((obj, done) => done(null, obj));
```

6. Initialize Passport with the Onshape Strategy:

```
//Initialize Passport with the Onshape Strategy
passport.use(new OnshapeStrategy({
        clientID: config.oauthClientId,
        clientSecret: config.oauthClientSecret,
        callbackURL: config.oauthCallbackUrl,
        authorizationURL: `${config.oauthUrl}/oauth/authorize`,
        tokenURL: `${config.oauthUrl}/oauth/token`,
        userProfileURL: `${config.oauthUrl}/api/users/sessioninfo`
    },
    (accessToken, refreshToken, profile, done) => {
        profile.accessToken = accessToken;
        profile.refreshToken = refreshToken;
        return done(null, profile);
    }
));
```

7. Open your environment variables file (e.g., `.env`, `.bashrc`, `.bash_profile`, `.zshrc`, etc.) and add the following environment variables, then save and close the file.

```
authorizationURL : https://oauth.onshape.com/oauth/authorize
tokenURL : https://oauth.onshape.com/oauth/token
userProfileURL : https://cad.onshape.com/api/users/sessioninfo
```

The callback function will provide us with the `accessToken`, the `refreshToken`, and the user's Onshape profile once authentication has been successfully passed. We can now use this to update our database with user-specific information.

Note that if you store the `accessToken` and `refreshToken` in the database along with the user record, you must update it each time that the access codes are refreshed.

8. Next, we define our endpoint where the authorization flow starts (in this case, `/oauthSignin`). This is the endpoint that we previously defined in the Onshape application setup. This will redirect to an Onshape page in order for the user to confirm (or deny) the applications access to theOnshape resources.

```
//Define the Onshape API endpoint
app.use('/oauthSignin', (req, res) => {
    /* These 5 lines are specific to the glTF Viewer sample app. You can replace
them with the input for whatever Onshape endpoints you are using in your app */
    const state = {
        docId: req.query.documentId,
```

```
        workId: req.query.workspaceId,
        elId: req.query.elementId
    };
    req.session.state = state;
    return passport.authenticate('onshape', { state: uuid.v4(state) })(req, res);
}, (req, res) => {

 });
```

3: Exchange the code for an access token

Fortunately, if you are using Passport, there isn't much to do once the user grants authorization. The return URL will contain the one-time authorization token, which Passport will extract and exchange for an access token and a refresh token, which are available in Passport callback function.

1. Add the following code to `app.js`:

```
//Exchange the code for an access token
app.use('/oauthRedirect', passport.authenticate('onshape', { failureRedirect:
'/grantDenied' }), (req, res) => {
    /* This code is specific to the glTF Viewer sample app. You can replace it
with the input for whatever Onshape endpoints you are using in your app. */
    res.redirect(`/?documentId=${req.session.state.docId}
&workspaceId=${req.session.state.workId}&elementId=
    ${req.session.state.elId}`);
});
```

2. If the user clicks **Deny** instead of **Authorize Application**, they are taken to a page that notifies them that access to the application was denied. We can see that in the `failureRedirect` argument. Add the following to `app.js`:

```
//Handle denied access
app.get('/grantDenied', (req, res) => {
    res.sendFile(path.join(__dirname, 'public', 'html', 'grantDenied.html'));
})
```

Now we have received the access token, and it can be accessed from `res.user.accessToken` on this page or from `req.user.accessToken` from any other page you redirect to from here.

4: Use the access token

1. Add the following to the bottom of `app.js`. You can see that the access token is used as an Authorization header:

```
//Use the access token as an Authorization header
makeOnshapeAPICall: async (req, res) => {
        try {
            const apiUrl =
"https://cad.onshape.com/api/documents?ownerType=1&sortColumn=createdAt&sortOrder
=desc&offset=0&limit=20"; //You can replace this with any Onshape API endpoint
URL.
```

```
            const resp = await fetch(normalizedUrl, { headers: { Authorization:
`Bearer ${req.user.accessToken}` }});
            const data = await resp.text();
            const contentType = resp.headers.get('Content-Type');
            res.status(resp.status).contentType(contentType).send(data);
        } catch (err) {
            res.status(500).json({ error: err });
        }
    }
```

Note: in the glTF Viewer sample app, this code appears in `utils.js` instead of `app.js`.

5: Refresh the token

When the access token expires, it must be refreshed by making another `POST` request to `https://oauth.onshape.com/oauth/token` with the following URL-encoded form body (with `Content-Type application/x-www-form-urlencoded`):

```
grant_type=refresh_token&refresh_token=\<refresh_token\>&client_id=\<client_id\>&
client_secret=\<client_secret\>
```

As with the authorization code data, the parameters in the form body must be URL-encoded. The response to this `POST` request will be a JSON-encoded structure with a new `access_token` value that can be used for the next 60 minutes.

Refresh tokens are valid for the lifetime of the user's grant. If a user who previously granted access to your application decides to revoke the grant, the refresh token is invalidated. If the user decides to re-grant application access, a new refresh token is generated and returned along with the access token.

1. Add the following to `app.js`:

```
/** After landing on the home page, we check if a user had already signed in. If
no user has signed in, we redirect the request to the OAuth sign-in page. If a
user had signed in previously, we will attempt to refresh the access token of the
user. After successfully refreshing the access token, we will simply take the
user to the landing page of the app. If the refresh token request fails, we will
redirect the user to the OAuth sign-in page again. */
app.get('/', (req, res) => {
    if (!req.user) {
        return res.redirect(`/oauthSignin${req._parsedUrl.search ?
req._parsedUrl.search : ""}`);
    } else {
        refreshAccessToken(req.user).then((tokenJson) => {
            // Dereference the user object and update the access token and
refresh token in the in-memory object.
            let usrObj = JSON.parse(JSON.stringify(req.user));
            usrObj.accessToken = tokenJson.access_token;
            usrObj.refreshToken = tokenJson.refresh_token;
            // Update the user object in PassportJS. No redirections will happen
here, this is a purely internal operation.
            req.login(usrObj, () => {
```

```
                return res.sendFile(path.join(__dirname, 'public', 'html',
'index.html'));
            });
        }).catch(() => {
            // Refresh token failed, take the user to OAuth sign in page.
            return res.redirect(`/oauthSignin${req._parsedUrl.search ?
req._parsedUrl.search : ""}`);
        });
    }
});

//Refresh the access token
const refreshAccessToken = async (user) => {
    const body = 'grant_type=refresh_token&refresh_token=' + user.refreshToken +
'&client_id=' + config.oauthClientId + '&client_secret=' +
config.oauthClientSecret;
    let res = await fetch(config.oauthUrl + "/oauth/token", {
        method: 'POST',
        headers: {
            'Content-Type': 'application/x-www-form-urlencoded'
        },
        body: body
    });
    if (res.ok) {
        return await res.json();
    } else {
        throw new Error("Could not refresh access token, please sign in again.");
    }
}

app.use('/api', require('./api'));
module.exports = app;
```

2. Save the file.
3. To see the authentication working in practice, you can follow the instructions in the glTF Viewer README to deploy the glTF Viewer app.

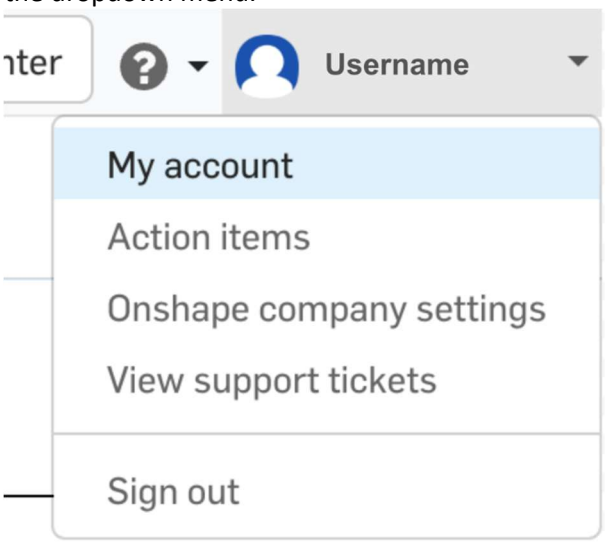
6: Grant authorization

For apps published in the Onshape App Store, the Onshape user must grant authorization to your application to access the Onshape data. This must be done by each user of your app.

To grant the application access to a user's data in Onshape, the *Onshape user* must follow the steps below:

1. Sign in to `cad.onshape.com` (or `https://companyName.onshape.com` for Enterprise acccounts).

2. Click their name in the top-right corner of the Onshape window, and then click **My account** in the dropdown menu.



3. Click **Applications** in the left sidebar.
    - Note that the gltf-viewer app will not appear in this list until it has been deployed and subscribed to as described in the glTF Viewer README.
4. Click **Grant** next to your app name to grant it access to their Onshape data. The Onshape user can click **Revoke** at any time to prevent your app from accessing their Onshape data.
5. The user will see the Authorize application screen shown below and will need to confirm their authorization grant by clicking **Authorize application**. The user is then redirected to the Redirect URL you specified in your code. Your app can now access the user's Onshape resources and profile.

Notes

Installed desktop applications

OAuth is designed for interactions between two servers using a browser. However, it can also be used by an installed desktop (or mobile) application. The application must perform a similar role to that of a third party server: it must exchange the code for an access token structure.

To enable this, Onshape allows two special forms of redirect URI to be registered:

- `http://localhost:<port>` Causes the browser to attempt to load a page from the host upon which it is running. The code parameter will be supplied exactly the same as outlined above. If the application can listen on the registered port and behave as a simple web server for the redirect URL, it can retrieve the code in the same way as a deployed web server.
- `urn:ietf:wg:oauth:2.0:oob` Causes the browser to display a simple page after a request has been granted instead of going to a new URL. The page contains simple instructions to copy and paste code into an application field. The browser will also update the title of the window to contain the code. An application could also look for browsers with window titles containing the string `Success code=<code>` and automatically grab the code from the browser window title. If an error occurs (e.g., the grant is denied), the browser window title will contain `Error description=<error string>`.

Debugging

Debugging OAuth can be a little tricky. Some tips are below:

1. Make sure you are correctly URL encoding the values supplied to the oauth/authorize and oauth/token endpoints.
2. Use a `GET /oauth/authorize` but a `POST /oauth/token` and make sure that the GET uses query parameters but that the `POST` uses a URL-encoded form body.
3. If you supply a `redirect_uri` to `/oauth/authorize`, you must also supply it as an additional parameter in the `POST` to `/oauth/token`
4. Use a tool such as Burp or Charles to deliberately 'man-in-the-middle' the connection requests between your server and Onshape, and verify that you are performing the correct REST operations (GET vs. POST) and correctly URL-encoding the parameter values.

Final Code

The above example uses Node.js to authenticate an Onshape app. This section includes the code for using OAuth2 with other coding languages.

Node.js

Prerequisites

```
npm install passport
npm install passport-onshape
```

Environment variables

```
authorizationURL : <https://oauth.onshape.com/oauth/authorize>
tokenURL : <https://oauth.onshape.com/oauth/token>
userProfileURL : <https://cad.onshape.com/api/users/sessioninfo>
```

app.js

```
//App definitions
const path = require('path');
const uuid = require('uuid');

const express = require('express');
const session = require('express-session');
const bodyParser = require('body-parser');

const passport = require('passport');
const OnshapeStrategy = require('passport-onshape');

const config = require('./config');

//Tell Express to use Passport, and initialize it.
const app = express();

app.use(express.static(path.join(__dirname, 'public')));
app.use(express.static(path.join(__dirname, 'dist')));
app.use(bodyParser.json());
```

```
app.set('trust proxy', 1); // To allow to run correctly behind Heroku when
deployed.

app.use(session({
    secret: config.sessionSecret,
    saveUninitialized: false,
    resave: false,
    cookie: {
        name: 'app-gltf-viewer',
        sameSite: 'none',
        secure: true,
        httpOnly: true,
        path: '/',
        maxAge: 1000 * 60 * 60 * 24 // 1 day
    }
}));
app.use(passport.initialize());
app.use(passport.session());

//Store the Onshape user information
passport.serializeUser((user, done) => done(null, user));
passport.deserializeUser((obj, done) => done(null, obj));

//Initialize Passport with the Onshape Strategy
passport.use(new OnshapeStrategy({
        clientID: config.oauthClientId,
        clientSecret: config.oauthClientSecret,
        callbackURL: config.oauthCallbackUrl,
        authorizationURL: `${config.oauthUrl}/oauth/authorize`,
        tokenURL: `${config.oauthUrl}/oauth/token`,
        userProfileURL: `${config.oauthUrl}/api/users/sessioninfo`
    },
    (accessToken, refreshToken, profile, done) => {
        profile.accessToken = accessToken;
        profile.refreshToken = refreshToken;
        return done(null, profile);
    }
));

//Define the Onshape API endpoint
app.use('/oauthSignin', (req, res) => {
    /* These 5 lines are specific to the glTF Viewer sample app. You can replace
them with the input for whatever Onshape endpoints you are using in your app. */
    const state = {
        docId: req.query.documentId,
        workId: req.query.workspaceId,
        elId: req.query.elementId
    };
    req.session.state = state;
    return passport.authenticate('onshape', { state: uuid.v4(state) })(req, res);
}, (req, res) => {
```

```
});

app.use('/oauthRedirect', passport.authenticate('onshape', { failureRedirect:
'/grantDenied' }), (req, res) => {
    /* This code is specific to the glTF Viewer sample app. You can replace it
with the input for whatever Onshape endpoints you are using in your app. */

res.redirect(`/?documentId=${req.session.state.docId}&workspaceId=${req.session.s
tate.workId}&elementId=${req.session.state.elId}`);
});

//Handle denied access
app.get('/grantDenied', (req, res) => {
    res.sendFile(path.join(__dirname, 'public', 'html', 'grantDenied.html'));
})

/** After landing on the home page, we check if a user had already signed in. If
no user has signed in, we redirect the request to the OAuth sign-in page. If a
user had signed in previously, we will attempt to refresh the access token of the
user. After successfully refreshing the access token, we will simply take the
user to the landing page of the app. If the refresh token request fails, we will
redirect the user to the OAuth sign-in page again. */
app.get('/', (req, res) => {
    if (!req.user) {
        return res.redirect(`/oauthSignin${req._parsedUrl.search ?
req._parsedUrl.search : ""}`);
    } else {
        refreshAccessToken(req.user).then((tokenJson) => {
            // Dereference the user object, and update the access token and
refresh token in the in-memory object.
            let usrObj = JSON.parse(JSON.stringify(req.user));
            usrObj.accessToken = tokenJson.access_token;
            usrObj.refreshToken = tokenJson.refresh_token;
            // Update the user object in PassportJS. No redirections will happen
here, this is a purely internal operation.
            req.login(usrObj, () => {
                return res.sendFile(path.join(__dirname, 'public', 'html',
'index.html'));
            });
        }).catch(() => {
            // Refresh token failed, take the user to OAuth sign in page.
            return res.redirect(`/oauthSignin${req._parsedUrl.search ?
req._parsedUrl.search : ""}`);
        });
    }
});

//Refresh the access token
const refreshAccessToken = async (user) => {
    const body = 'grant_type=refresh_token&refresh_token=' + user.refreshToken +
'&client_id=' + config.oauthClientId + '&client_secret=' +
config.oauthClientSecret;
```

```
    let res = await fetch(config.oauthUrl + "/oauth/token", {
        method: 'POST',
        headers: {
            'Content-Type': 'application/x-www-form-urlencoded'
        },
        body: body
    });
    if (res.ok) {
        return await res.json();
    } else {
        throw new Error("Could not refresh access token, please sign in again.");
    }
}

app.use('/api', require('./api'));
module.exports = app;

//Use the access token in an Authorization header.
makeOnshapeAPICall: async (req, res) => {
        try {
            const apiUrl =
"https://cad.onshape.com/glassworks/explorer/#/Document/getDocuments"; //You can
replace this with any Onshape API endpoint URL.
            const resp = await fetch(normalizedUrl, { headers: { Authorization:
`Bearer ${req.user.accessToken}` }});
            const data = await resp.text();
            const contentType = resp.headers.get('Content-Type');
            res.status(resp.status).contentType(contentType).send(data);
        } catch (err) {
            res.status(500).json({ error: err });
        }
    }
```

Python

This Python code only works on a local machine. To deploy the code, you can replace the Flask code with the web server of your choice.

Prerequisites

```
pip3 install flask
pip3 install requests_oauthlib
```

app.py

```
from flask import Flask, request, redirect, session, url_for
from flask.json import jsonify
from requests_oauthlib import OAuth2Session
import os

app = Flask(__name__)
app.secret_key =
b'F\xf5\xe5\xc0\xbe\tg\x7f\xac\x89\x87e\xc24\xe8m\x1c\xd9\xda\x96G,\x90i'
```

```
os.environ['OAUTHLIB_INSECURE_TRANSPORT'] = "1"

client_id = <Client ID of your application>
client_secret = <Client Secret of your application>
authorization_base_url = "https://oauth.onshape.com/oauth/authorize"
token_url = "https://oauth.onshape.com/oauth/token"
redirect_url = "http://localhost:5000"

@app.route('/')
def home():
  onshape = OAuth2Session(client_id, redirect_uri=redirect_url)
  auth_url, state = onshape.authorization_url(authorization_base_url)
  session['oauth_state'] = state
  return redirect(auth_url)

@app.route('/token', methods=["GET"])
def token():
  onshape = OAuth2Session(client_id, state=session['oauth_state'],
redirect_uri=redirect_url)
  token = onshape.fetch_token(token_url, client_secret=client_secret,
authorization_response=request.url)
  session['oauth_token'] = token
  return redirect(url_for('.documents'))

@app.route('/documents', methods=["GET"])
def documents():
  extra = {
    'client_id': client_id,
    'client_secret': client_secret,
  }
  onshape = OAuth2Session(client_id, token=session['oauth_token'],
redirect_uri=redirect_url)
  session['oauth_token'] = onshape.refresh_token(token_url, **extra)
  return
jsonify(onshape.get('https://cad.onshape.com/api/v6/documents?q=Untitled&ownerTyp
e=1&sortColumn=createdAt&sortOrder=desc&offset=0&limit=20').json())

if __name__ == "__main__":
  app.run()
```

API Keys

NOTE: The authentication processes outlined on this page are for local and internal testing only. All applications submitted to the Onshape App Store must follow the Onshape OAuth2 protocols.

Why API Keys?

API keys are useful for small applications meant for personal use, allowing developers to avoid the overhead of the OAuth workflow. Creating an app is very easy with API keys: create an API key with the Developer Portal, set up a function to build your API key header as in the samples, and make your API calls! There's no need to deal with OAuth redirects or things like that.

We've moved over to using API keys for authenticating requests instead of using cookies for several reasons.

1. Security: Each request is signed with unique headers so that we can be sure it's coming from the right place.
2. OAuth: The API key system we're now using for HTTP requests is the same process developers follow when building full-blown OAuth applications; there's no longer a disconnect between the two.

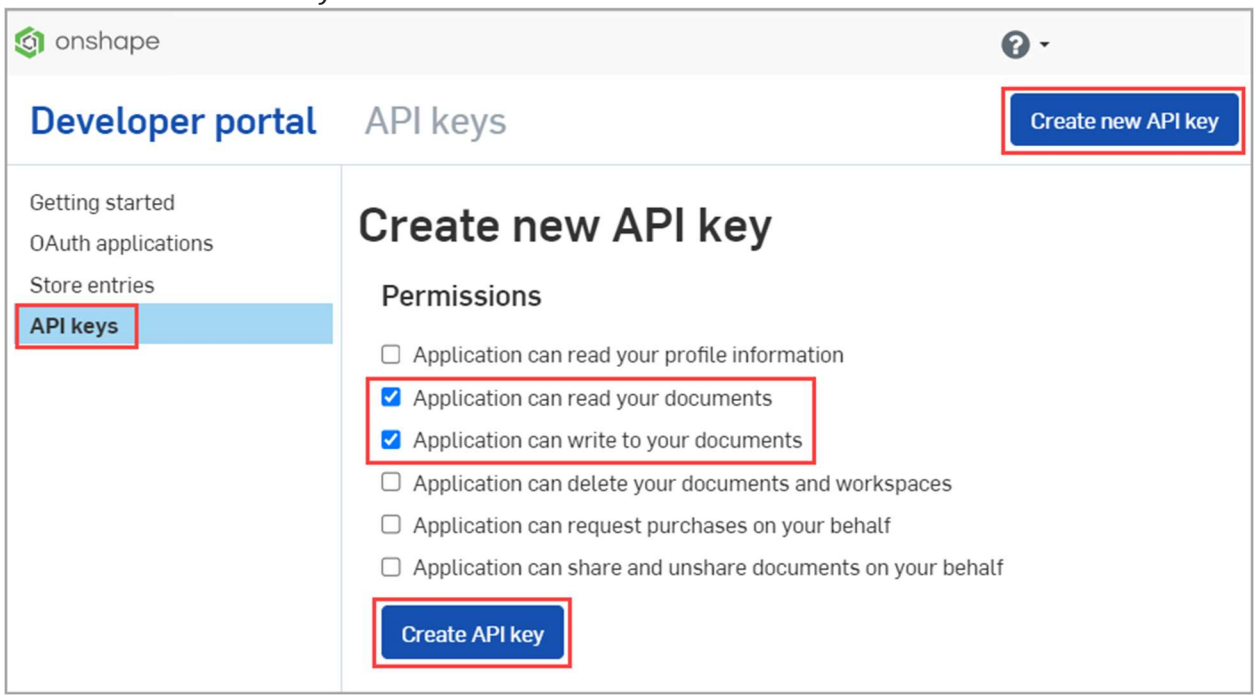
Once you create an API key, it will only be valid in the stack on which it was created. An API key created on the partner stack, for example, will not function on the production stack.

If you need information or have a question unanswered in this documentation, feel free to chat with us by sending an email to api-support@onshape.com or by checking out the forums. If you are a member of the DevPartners group (see the Development help page for information) more detailed instructions and code examples are in the apikey sample repo.

1. Create API Keys

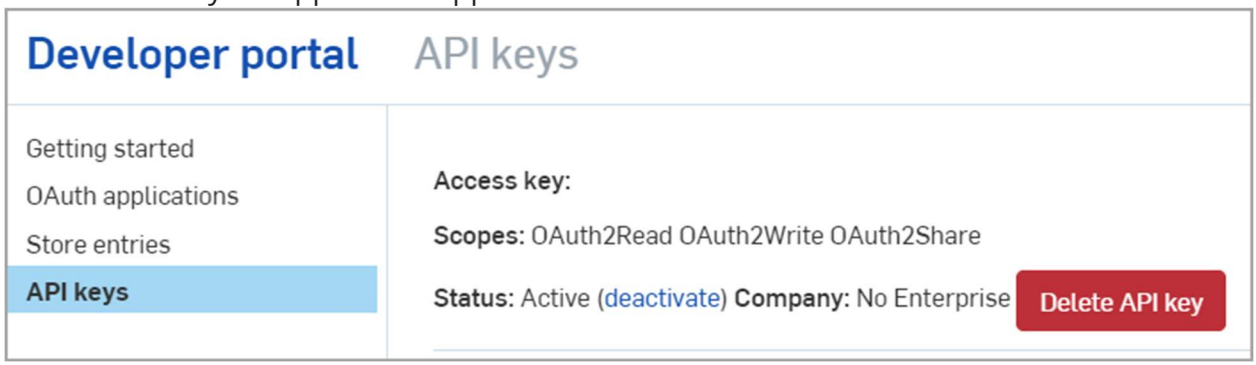
1. Go to https://dev-portal.onshape.com.
2. In the left pane, click `API keys`.
3. Click the `Create new API key` button.
4. Select the desired permissions for your app.

5. Click the `Create API key` button.



6. Copy both the **access key** and **secret key** from the pop-up window, save them somewhere, then click the `Close` button.
IMPORTANT NOTE: You will not be able to find the secret key again, so save it somewhere safe!

7. The details for your application appear.



8. Now that you have a key pair, see Generate a Request Signature for information on signing your requests to use our API.

Once you have your access key and secret, you will want to avoid giving others access to them, since they're tied directly to your personal Onshape account. Think of your API key as a username and password pair. Do not place them directly in the code for your application, especially if others might see it. The samples we provide here use a separate configuration file to contain this information, but there are other ways to keep the access key and secret safe, like setting them as environment variables.

Scopes

There are several scopes available for API keys (equivalent to OAuth scopes):

- `OAuth2Read` - Read non-personal information (documents, parts, etc.)
- `OAuth2ReadPII` - Read personal information (name, email, etc.)
- `OAuth2Write` - Create and edit documents, etc.
- `OAuth2Delete` - Delete documents, etc.
- `OAuth2Purchase` - Authorize purchases from account

2. Select an Authentication Option

Please select an option for authentication:

- Basic Authorization: Lowest security. For local testing only.
- Request Signature: Medium security. For testing and internal use.
- OAuth2: Highest security. Required for all Onshape Apps.

Basic Authorization

For local testing, you can provide a basic authentication via your API Keys.

1. Open your terminal and run the following command,
   replacing `ACCESS_KEY` and `SECRET_KEY` with the **access key** and **secret key** you

created earlier. Remember to include the colon (`:`) between the keys. *You will receive a long, base-64-encoded string as output.*
    - **MacOS**:
    - `printf ACCESS_KEY:SECRET_KEY | base64`
    - **Windows**:
    - `powershell "[convert]::ToBase64String([Text.Encoding]::UTF8.GetBytes(\"ACCESS_KEY:SECRET_KEY\"))"`
2. Add the authorization header to your code, replacing `CREDENTIALS` with the string you receieved in Step 1:

```
3. -H 'Authorization: Basic CREDENTIALS' \
```

See our Quick Start Guide for an example of using Basic Authorization in an app.

Request Signature

For additional security, you can include your API Keys as part of a request signature. This provides more security than the Basic Authorization above, but less security than OAuth2.

To ensure that a request is coming from you, we have a process for signing requests that you must follow for API calls to work. Everything is done via HTTP headers that you'll need to set:

1. *Date*: A standard date header giving the time of the request; must be accurate within **5 minutes** of request. Example: `Mon, 11 Apr 2016 20:08:56 GMT`
2. *On-Nonce*: A string that satisfies the following requirements (see the code for one possible way to generate it):
    - At least 16 characters
    - Alphanumeric
    - Unique for each request
3. *Authorization*: This is where the API keys come into play. You'll sign the request by implementing this algorithm:
    - **Input**: Method, URL, On-Nonce, Date, Content-Type, AccessKey, SecretKey
    - **Output**: String of the form: `On <AccessKey>:HmacSHA256:<Signature>`
    - **Steps to generate the signature portion**:
        1. Parse the URL and get the following:
            1. The path, e.g. `/api/documents` (no query params!)
            2. The query string, e.g. `a=1&b=2`
                - NOTE: If no query paramaters are present, use an empty string
        2. Create a string by appending the following information in order. Each field should be separated by a newline (`\n`) character, and the string must be converted to lowercase:
            1. HTTP method

    2. On-Nonce header value
    3. Date header value
    4. Content-Type header value
    5. URL pathname
    6. URL query string
3. Using SHA-256, generate an HMAC digest, using the API secret key first and then the above string, then encode it in Base64.
4. Create the `On <AccessKey>:HmacSHA256:<Signature>` string and use that in the Authorization header in your request.

Below is an example function to generate the authorization header, using Node.js's standard `crypto` and `url` libraries:

```
// ...at top of file
var u = require('url');
var crypto = require('crypto');

/**
* Generates the "Authorization" HTTP header for using the Onshape API
*
* @param {string} method - Request method; GET, POST, etc.
* @param {string} url - The full request URL
* @param {string} nonce - 25-character nonce (generated by you)
* @param {string} authDate - UTC-formatted date string (generated by you)
* @param {string} contentType - Value of the "Content-Type" header; generally
"application/json"
* @param {string} accessKey - API access key
* @param {string} secretKey - API secret key
*
* @return {string} Value for the "Authorization" header
*/
function createSignature(method, url, nonce, authDate, contentType, accessKey,
secretKey) {
    var urlObj = u.parse(url);
    var urlPath = urlObj.pathname;
    var urlQuery = urlObj.query ? urlObj.query : ''; // if no query, use empty
string

    var str = (method + '\n' + nonce + '\n' + authDate + '\n' + contentType +
'\n' +
        urlPath + '\n' + urlQuery + '\n').toLowerCase();

    var hmac = crypto.createHmac('sha256', secretKey)
        .update(str)
        .digest('base64');

    var signature = 'On ' + accessKey + ':HmacSHA256:' + hmac;
    return signature;
}
```

Redirects

Some API endpoints return 307 redirects. You must generate an Authorization header for the redirect as well, but please note that the server portion of the URL might be different, the redirect URL may contain query parameters that must be encoded in the Authorization header, etc.

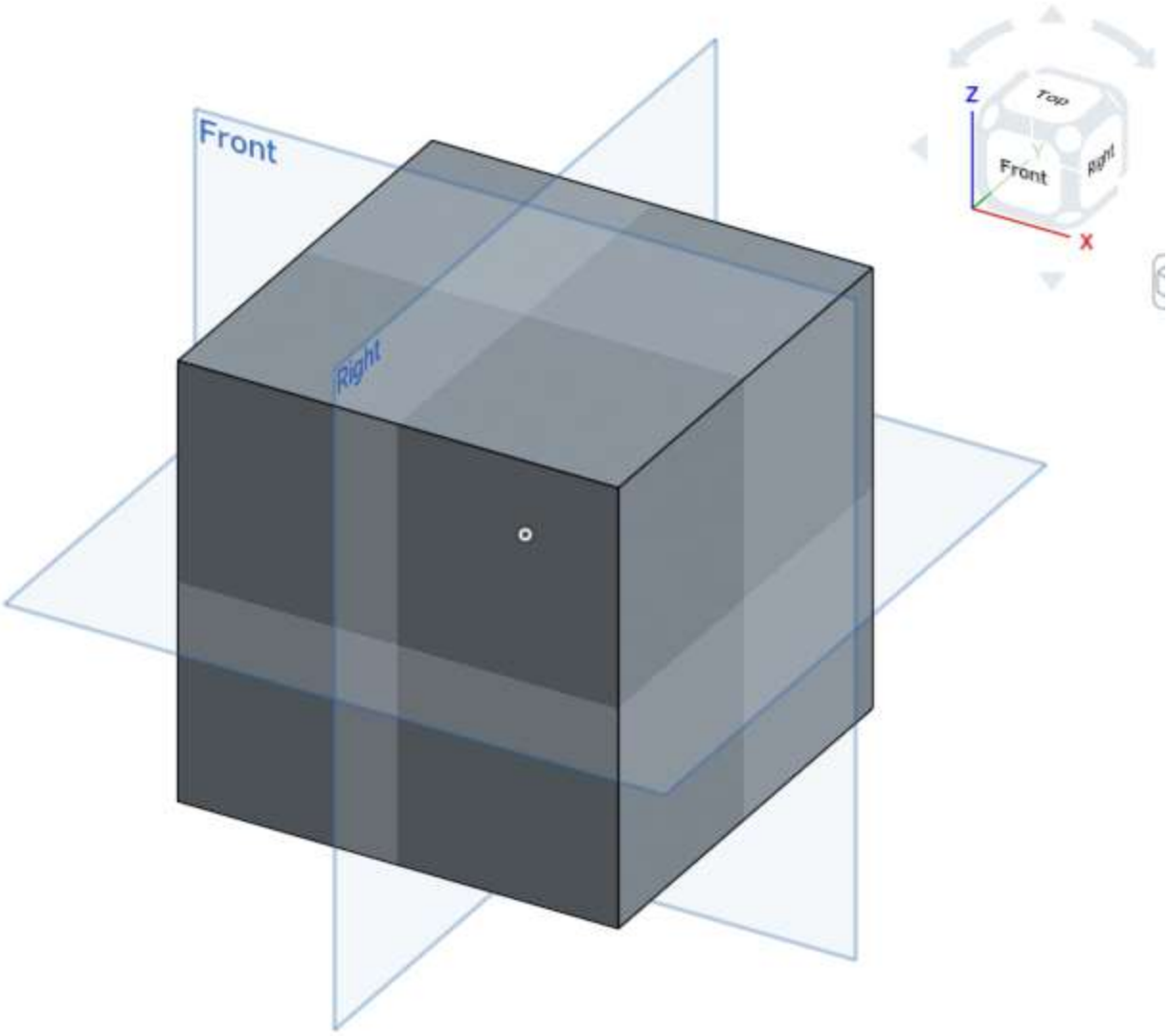
Associativity

Onshape does not expose a persistent ID for any of these entities. When the model changes, the ID may change. Therefore, Onshape provides an API to enable mapping IDs from a previous microversion to the current microversion. Assuming a simple case of maintaining associativity for a face, an abstract workflow might be:

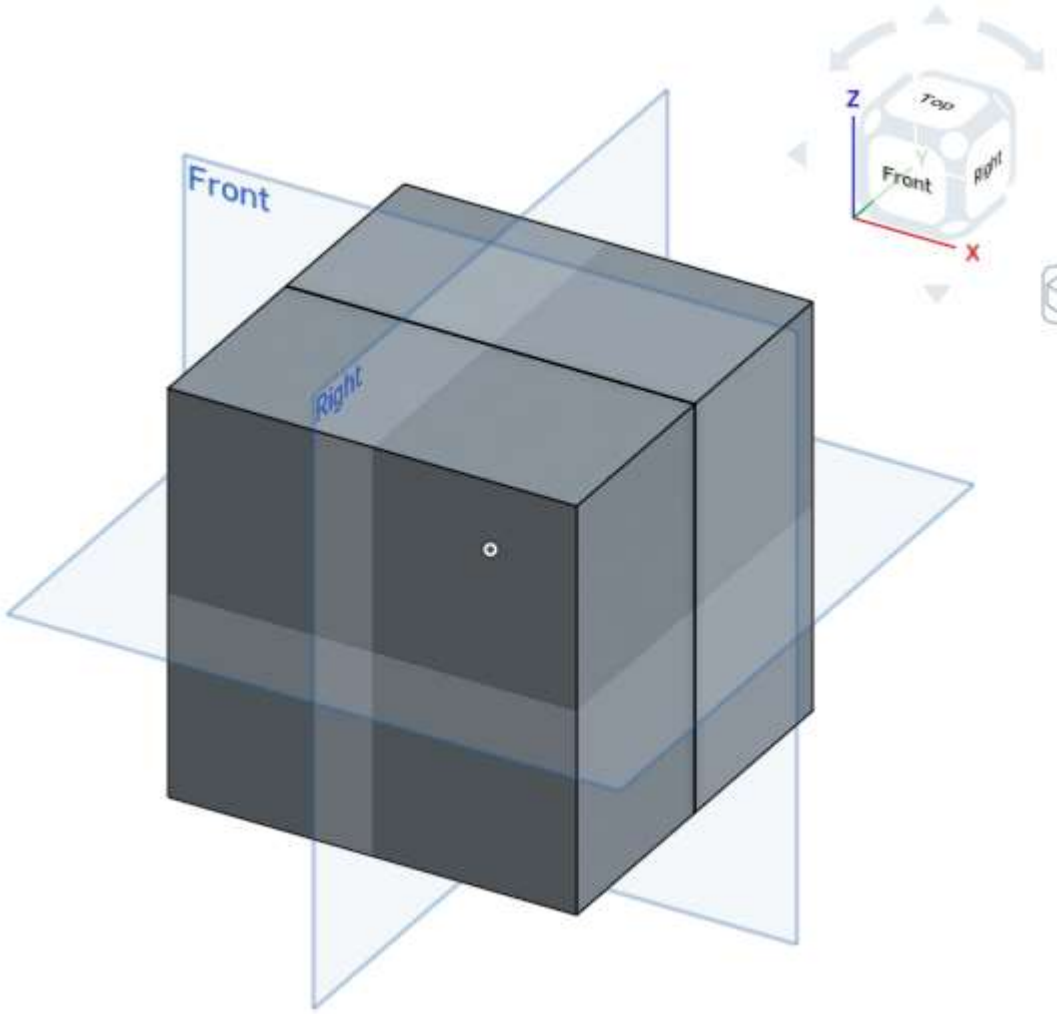
1. Read the tessellated model data.
2. Select the face of interest.
3. Store the Face ID and Document Microversion ID for the face.
4. [user changes model]
5. Call the REST API to translate from the known Face ID to an ID in the new model.
6. Re-apply application-specific data to the face(s) in the new model. Note that a face may become zero, one or multiple faces in the new model, depending on what changes the user made.

Associativity Example

1. Create a cube in Onshape:



2. Get the document microversion ID from the URL: `https://cad.onshape.com/api/d/<docid>/w/<wid>/microversionId`.
3. Use the appropriate REST API to get the tessellated faces (`getPartStudioFaces`) and edges (`getPartStudioEdges`}. Note the ids:
    - Part ID: `JHD`
    - Front face ID: `JHO`
    - Top edge of the front face ID: `JHd`
    - Right edge of the top face ID: `JHt`
4. Split cube with the Front plane and translate the IDs:



POST

`https://cad.onshape.com/api/partstudios/d/<docid>/w/<wid>/e/<eid>/idtranslations`

Body:

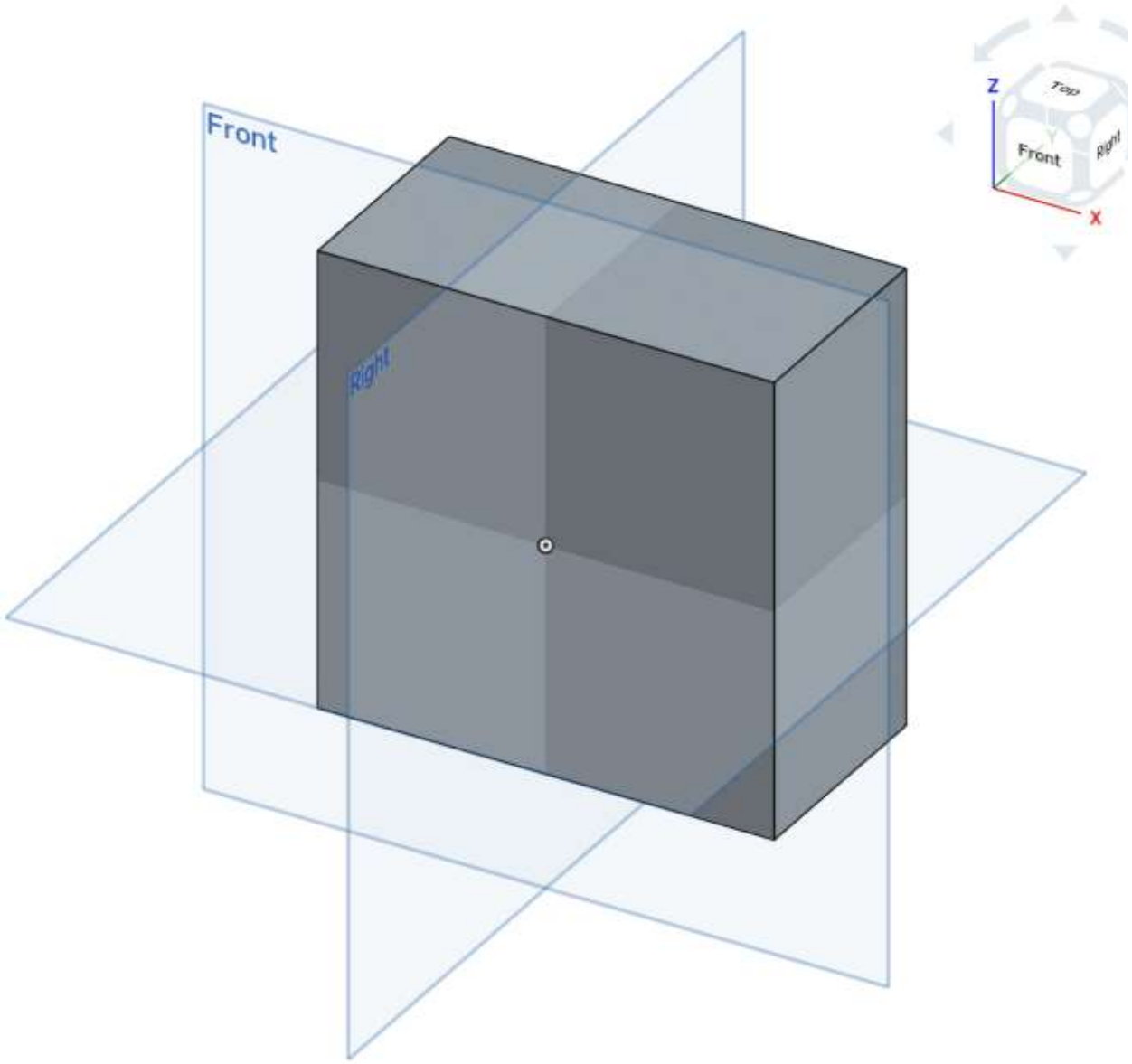
```
{
  "sourceDocumentMicroversion" : "47e75ab2ee8b4356a76ebd47",
  "ids" : ["JHD", "JHO", "JHd", "JHt"  ]
}
```

Response:

```
{
    "documentId": "748d6e850c9248328189922b",
    "elementId": "042a6fa54e79451e8076463d",
    "sourceDocumentMicroversion": "47e75ab2ee8b4356a76ebd47",
    "ids": [
        { "source": "JHD", "status": "SPLIT", "target": ["JID", "JIH"] },
        { "source": "JHO", "status": "OK", "target": ["JHO"] },
        { "source": "JHd", "status": "OK", "target": ["JHd"] },
```

```
        { "source": "JHt", "status": "SPLIT", "target": ["JI5", "JI9"] }
    ],
    "targetDocumentMicroversion": "78bc7f3fcf82475085c2f3ab"
}
```

4. Delete one of the parts, and translate the IDs:



POST

`https://cad.onshape.com/api/partstudios/d/<docid>/w/<wid>/e/<eid>/idtranslations`

Body:

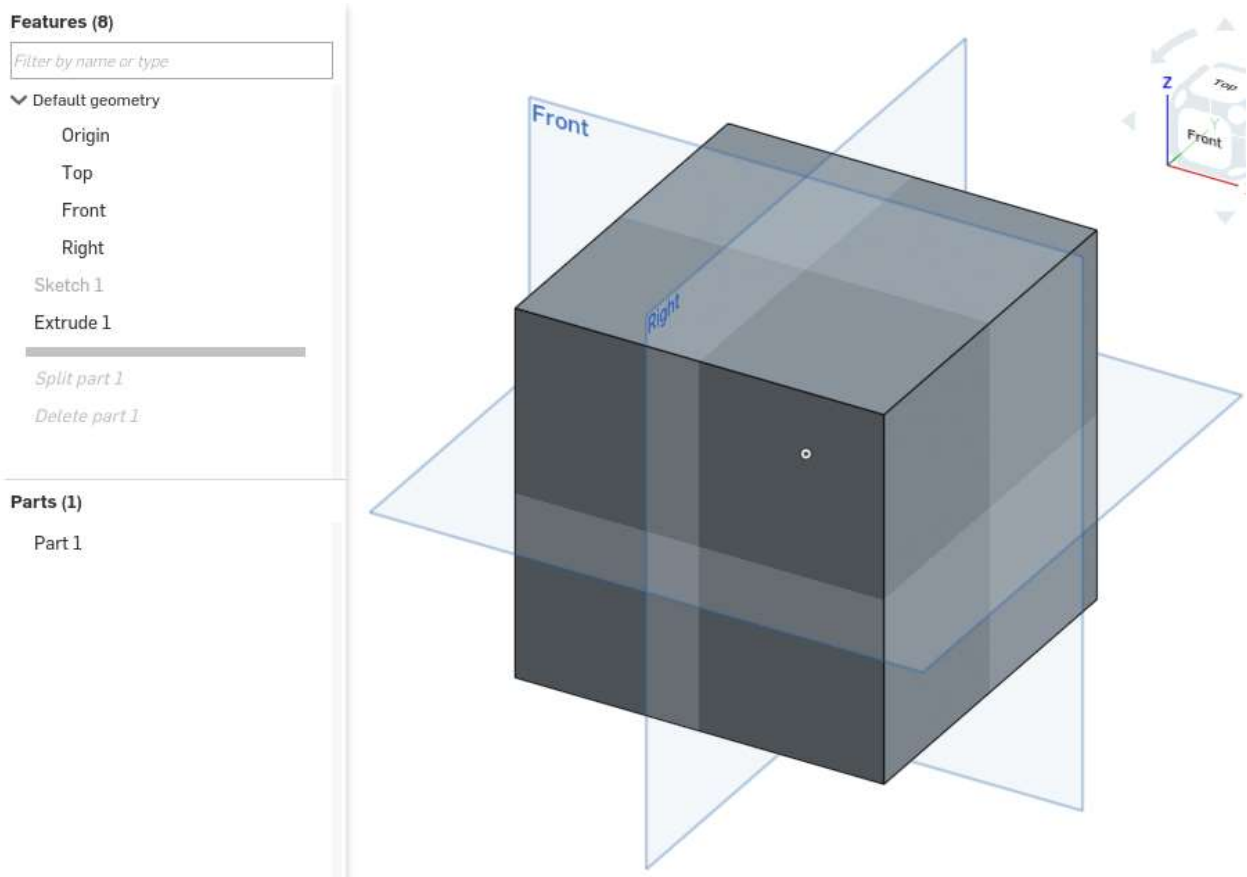
```
{
```

```
  "sourceDocumentMicroversion" : "47e75ab2ee8b4356a76ebd47",
  "ids" : ["JHD", "JHO", "JHd", "JHt"]
}
```

Response:


```
{
    "documentId": "748d6e850c9248328189922b",
    "elementId": "042a6fa54e79451e8076463d",
    "sourceDocumentMicroversion": "47e75ab2ee8b4356a76ebd47",
    "ids": [
        { "source": "JHD", "status": "OK", "target": ["JID"] },
        { "source": "JHO", "status": "FAILED_TO_RESOLVE", "target": [] },
        { "source": "JHd", "status": "FAILED_TO_RESOLVE", "target": [] },
        { "source": "JHt", "status": "OK", "target": ["JI5"] }
    ],
    "targetDocumentMicroversion": "52aa74d34b624f3aaef33204"
}
```


5. Roll back the delete and the split, and translate the IDs:



POST

`https://cad.onshape.com/api/partstudios/d/<docid>/w/<wid>/e/<eid>/idtranslations`

Body:


```
{
  "sourceDocumentMicroversion" : "47e75ab2ee8b4356a76ebd47",
  "ids" : ["JHD", "JHO", "JHd", "JHt"]
}
```


Response:


```
{
    "documentId": "748d6e850c9248328189922b",
    "elementId": "042a6fa54e79451e8076463d",
    "sourceDocumentMicroversion": "47e75ab2ee8b4356a76ebd47",
    "ids": [
        { "source": "JHD", "status": "OK", "target": ["JID"] },
        { "source": "JHO", "status": "OK", "target": ["JHO"] },
        { "source": "JHd", "status": "OK", "target": ["JHd"] },
        { "source": "JHt", "status": "OK", "target": ["JHt"] }
    ],
    "targetDocumentMicroversion": "52aa74d34b624f3aaef33204"
}
```


Billing

This document describes APIs that will allow partners to interact with the Onshape billing system.

Please address questions to "api-support@onshape.com" for the fastest response.

Overview

All billing is done through "plans" that are created in the Developer Portal. A "plan" has the following attributes:

Name (also called SKU)	A unique (within your company) plan name
Description	A user-visible description of the plan
Amount	The cost of the plan (may be one-time or recurring, depending on the type)
Type	Monthly, One-time or Consumable

Onshape defines three kinds of plans:

Plan type	Description

Recurring (Monthly Subscription)	A plan that is renewed monthly at a fixed cost. All Apps in the app store must have a Free monthly plan (which is created by default), and may have additional paid plans.
One-time	A plan that is purchased once (not renewed monthly). A user may purchase these multiple times.
Consumable	A plan that represents a consumable unit, such as "hours of rendering" or “simulation runs”. Consumable plans are not fully implemented at this time, but the consumable functionality can be implemented using One-time Purchase plans as described below.

Users may purchase plans through the App Store interface. In addition, if your application has the OAuth Purchase Scope, your application can initiate “in-app” purchases by calling Onshape to request a purchase.

The basic steps for interacting with Onshape Billing:

1. Define one or more plans using the Developer Portal interface
2. Use the Onshape API to determine the current user’s plan
3. Provide features and/or limits based on the current plan

Using the Onshape Billing API

```
GET /api/accounts/purchases
```

Returns a list of purchase made by the current user for plans owned by the current application. Use this information to determine what capabilities or features the user is entitled to use.

```
DELETE /api/accounts/purchases/<purchase id>
```

Cancel a recurring purchase.

```
POST /api/accounts/purchases/<purchase id>/consume
```

Indicate the use of a consumable. (Not fully implemented at this time)

```
GET /api/billing/plans/client/<client id>
```

Get a list of the billing plans defined for this client.

Initiating a purchase from an application (in-app purchases)

To initiate a purchase of a subscription or one time item you must set the browser’s location to particular URL within the Onshape stack:

```
https://cad.onshape.com/billing/purchase?redirectUri=RRRR&clientId=CCCC&sku=SSSS&user
Id=UUUU
```

Each of the query parameters should be URL encoded. The clientId is your application's OAuth Client ID, the sku is the name/sku field for an item (you can find this in the developer portal or it's retrievable through the /api/billing/plans REST endpoints). The user Id should be the Onshape user Id for the current user and is available through the /api/users/session REST endpoint. The redirectUri is the URI the user will be returned to within your website when the purchase is finished.

When the browser's location is changed to this pattern the Onshape stack will serve content to confirm the users identity, confirm the details of what is being purchased (or obtained if the item is free) and then after the user agrees to the purchase will confirm the transaction (with our payment processor if the item is not free) and then redirect the user back to the supplied redirectUri (the browser location will be changed to the redirectUri). Additionally Onshape will add a `success=true` or `success=false` query parameter to the redirectURI indicating whether the user completed successfully (payment was taken if required etc.) or failed, either due to cancelling the purchase or an issue with payment.

When the browser fetches the redirectUri your application must call back through the `/api/account/purchases` API to get confirmation of the purchase - do NOT assume that a fetch of the redirectUri with a `success=true` query parameter actually indicates a purchase has occurred. Query the Onshape stack with the `/api/account/purchases` API to ensure that the required item has actually been bought.

Consumable Items

A detailed description of the interface for managing consumable purchases will be provided shortly. You can use one-time plans to achieve similar results:

1. Define a one-time purchase plan with a description indicating the nature of the purchase, for example:

   RENDER-10 Ten rendering hours $100

2. Keep track of the number of hours that the user has consumed. You can store and retrieve this information in Onshape using the following APIs. These APIs allow you to store and retrieve arbitrary information on a per-user basis.

   POST /applications/clients/:cid/settings/users/:uid GET /applications/clients/:cid/settings/users/:uid

3. Check the number of available "units" by getting the purchases and the record of consumables. Be sure to include UI in your application that the user can use to see their remaining quantity.
4. Alternately, you can store the consumption data in your own system; you do not need to use the Onshape API to manage that data.

Onshape intends to provide a richer set of APIs that help track the purchase and consumption of consumables in the near future.

Other billing models

You can use these mechanisms to implement other models. For example, a time-limited trial could be implemented by scanning purchases for the first "purchase" and denying service if it is more than a defined number of days in the past. A "fixed number of uses per month" could be implemented as a monthly subscription, string usage data with the settings API, and denying service after a fixed number of uses.

Samples

Onshape will provide sample code for both desktop and integrated applications demonstrating the use of the billing APIs and workflow. If you are subscribed to the Onshape Github Partner group, you will have access to those samples as soon as they are posted.

Testing

Please contact api-support@onshape.com to discuss details of testing billing & subscriptions.

Data Import & Export

This page describes the APIs Onshape provides for importing files to Onshape and exporting files from Onshape into different formats. We refer to the process of importing and exporting files from one format to another as *translating* the files.

Onshape provides several APIs to support this format translation. These fall into three categories:

- Synchronous exports - Export Onshape content to glTF, STL, or Parasolid format.
- Asynchronous exports - Export Onshape content into a variety of other formats.
- Import to Onshape - Import a translatable file by uploading it to an Onshape blob element.

Note

This page provides sample code as curls. See the curl documentation for more information.

Note

All Onshape API calls must be properly authenticated by replacing the `CREDENTIALS` variable in the curls below. See the API Keys page for instructions and the Quick Start for an example. All applications submitted to the Onshape App Store *must* authenticate with OAuth2.

Note

This documentation refers to Onshape IDs in the following format: `{did}, {wid}, {eid}, {pid}, {otherId}`. These represent document, workspace, element, part, and other IDs (respectively) that are needed make the API calls. Please see API Guide: API Intro for information on what these IDs mean and how to obtain them from your documents. Never include the curly braces (`{}`) in your API calls.

Note

For Enterprise accounts, replace **cad** in all Onshape URLs with your company domain.
https://**cad**.onshape.com > https://**companyName**.onshape.com

Synchronous exports

Onshape provides a simple way to export content to common formats (glTF, Parasolid, and STL). Most of the interfaces defined here operate by requesting an HTTP redirect to a different URL where the request is fulfilled. Applications must explicitly handle the redirect and attachment authentication headers to the follow-up request, or it will fail.

The following endpoints are available. We've included an example curl with each one.

- Export Part to glTF
- curl -X 'GET' \
- 'https://cad.onshape.com/api/v6/parts/d/{did}/w/{wid}/e/{eid}/partid/{partid}/gltf?rollbackBarIndex=-1&outputSeparateFaceNodes=false&outputFaceAppearances=false' \
- -H 'accept: model/gltf-binary;qs=0.08'
- Export Part to Parasolid
- curl -X 'GET' \
- 'https://cad.onshape.com/api/v6/parts/d/{did}/w/{wid}?elementId={eid}&withThumbnails=false&includePropertyDefaults=false' \
- -H 'accept: application/json;charset=UTF-8; qs=0.09'
- Export Part to STL
- curl -X 'GET' \
- 'https://cad.onshape.com/api/v6/parts/d/{did}/w/{wid}/e/{eid}/partid/{partid}/stl?mode=text&grouping=true&scale=1&units=inch' \
- -H 'accept: application/octet-stream'
- Export PartStudio to glTF
- curl -X 'GET' \
- 'https://cad.onshape.com/api/v6/partstudios/d/{did}/w/{wid}/e/{eid}/gltf?rollbackBarIndex=-1&outputSeparateFaceNodes=false&outputFaceAppearances=false' \
- -H 'accept: model/gltf-binary;qs=0.08'
- Export PartStudio to Parasolid
- curl -X 'GET' \
- 'https://cad.onshape.com/api/v6/partstudios/d/{did}/w/{wid}/e/{eid}/parasolid?version=0&includeExportIds=false&binaryExport=false' \
- -H 'accept: */*'
- Export PartStudio to STL
- curl -X 'GET' \

```
'https://cad.onshape.com/api/v6/partstudios/d/{did}/w/{wid}/e/{eid}/stl?mode=text&grouping=true&scale=1&units=inch' \
  -H 'accept: */*'
```

Export Document to JSON

```
curl -X 'POST' \
'https://cad.onshape.com/api/v6/documents/d/{did}/w/{wid}/e/{eid}/export' \
  -H 'accept: application/octet-stream' \
```

Asynchronous exports

The exports in the last section perform the format translation synchronously, returning the output immediately after some processing delay. Other format conversions are more complex and time-consuming, and in many cases, cannot be completed quickly enough to prevent connection errors. Note that the source format for an export is currently always automatically detected by Onshape. Part Studios and Assemblies are known to be `ONSHAPE` format. File uploads have their type determined by the filename suffix. For example, a file named *part7.step* is assumed to be in `STEP` format.

The following asynchronous translation APIs are available:

- BlobElement/createBlobTranslation: Export a Blob Element to the specified `formatName`.
- PartStudios/createPartStudioTranslation: Export a Part Studio to the specified `formatName`.
- Assembly/translateFormat: Export an Assembly to the specified `formatName`.
- Drawing/createDrawingTranslation: Export a Drawing to the specified `formatName`.

These asynchronous exports include a few additional steps, which are explained in more detail in the next section:

1. See what formats are available for exporting your content with **Translation/getAllTranslatorFormats**.

   - **Note:** Parts with mesh data cannot be exported to ACIS, IGES, or OBJ format.

2. Call the desired **translation API**.
   - Specify the target `formatName` in the request body JSON.
   - Specify `storeInDocument=false` (default) to export the content to new file.
   - Specify `storeInDocument=true` to export the content to a blob element in the source document.
3. Poll the `requestState` in the translation response and wait for a result of `DONE`.
4. To retrieve the exported results:
   - External files: call **Document/downloadExternalData** on the `resultExternalDataIds` from the translation response.
   - Blob elements: call **BlobElement/downloadFileWorkspace** on the `resultElementIds` from the translation response.

Async export details

To export your Onshape content to another format:

1. Determine what export format file types are available for your content by calling: Translations/getAllTranslatorFormats.
    - Note that Drawings have their own API for this call: Drawing/getDrawingTranslatorFormats
2. Next, initiate the export by calling one of the asynchronous translation APIs.
    - Note that each of these APIs takes a JSON for specifying options for the export as part of the request body. Refer to the API Explorer page for help viewing these JSON docs.
    - The target file format **must be specified in the `formatName` field** in the request body, and must match a valid format found in Step 1.
    - By default, `storeInDocument` is set to `false` in the request body to export to a single data file (or a zip of multiple files). Set to `true` to export as blob elements.
3. Wait for the translation to complete. You can either register a webhook and wait to receive a notifcation that the translation is complete (see Webhook Notifications), or you can poll the translation's `requestState`:
    - You can poll the `requestState` from the initial translation's response, or you can call Translation/getTranslation on the `translationId` from the initial translation's response.
    - When a translation is complete, `requestState` will change from `ACTIVE` to either `DONE` or `FAILED`.
    - When `requestState=DONE`, results are available to be used.
4. Retrieve the exported results:
    - If you exported to an external file, call Documents/downloadExternalData to retrieve the exported result.
        - Note that this API takes the source document ID and a "foreign ID" as required parameters.
        - Use the `resultExternalDataIds` from the translation response as the foreign ID (`fid`).
        - External data is associated with, but external to, the document used as translation context. This data is not versioned like with in-document data.
    - If your translation request body specified `storeInDocument=true`, retrieve the blob element data with BlobElement/downloadFileWorkspace.
        - The element IDs for the new blob elements can be found in the `resultElementIds` field in the translation response.

Imports

Files can be imported to Onshape as blob elements. When uploading a file to a blob element, either as a new element or an update to an existing element, if the file is a recognized format for import, it will be translated into `ONSHAPE` format by default. This behavior can be overridden by the application, if desired.

- Translation/createTranslation

```
curl -X 'POST' \
  'https://cad.onshape.com/api/v6/translations/d/{did}/w/{wid}' \
  -H 'accept: application/json;charset=UTF-8; qs=0.09' \
```

```
    -H 'Content-Type: multipart/form-data' \
    -F 'storeInDocument=true' \
    -F 'flattenAssemblies=true
    -F 'file=@/path/filename.ext'
    -F 'formatName=' \
    ...
```

BlobElement/uploadFileCreateElement

```
curl -X 'POST' \
  'https://cad.onshape.com/api/v6/blobelements/d/{did}/w/{wid}' \
    -H 'accept: application/json;charset=UTF-8; qs=0.09' \
    -H 'Content-Type: multipart/form-data' \
    -F 'storeInDocument=true' \
    -F 'file=@/path/filename.ext'
    -F 'formatName=' \
    ...
```

BlobElement/uploadFileUpdateElement

```
  curl -X 'POST' \
  'https://cad.onshape.com/api/v6/blobelements/d/{did}/w/{wid}/e/{eid}' \
    -H 'accept: application/json;charset=UTF-8; qs=0.09' \
    -H 'Content-Type: multipart/form-data' \
    -F 'storeInDocument=true' \
    -F 'locationElementId=' \
    -F 'file=@/path/filename.ext'
    ...
```

Note that these endpoints require you to specify the target document ID and workspace ID. You must also include the file to import. These APIs also includes a request body JSON for specifying options for the import.

- Override the translation to `ONSHAPE` format by specifying a valid format in the `formatName` field. Get a list of valid import formats by calling Translation/getAllTranslatorFormats.
- Specify `storeInDocument=true` to import the data as a blob element into the target document. Change to `false` to only create an external data file.
- If the source file contains an assembly and `flattenAssemblies=true`, the assembly structure is removed and a single part studio is created.
- Note that when using cURL, you must begin the path to the file with an `@` symbol.

Sample Workflows

Export a PartStudio to STL

We will export the `CRANK` PartStudio from this public document to an STL file.

1. Call the Part Studios/exportPartStudioStl endpoint on the document:

```
curl -X 'GET' \
'https://cad.onshape.com/api/v6/partstudios/d/e60c4803eaf2ac8be492c18e/w/d2558da712764516cc9fec62/e/6bed6b43463f6a46a37b4a22/stl?mode=text&grouping=true&scale=1&units=inch' \
  -H 'accept: */*'
```

5. Navigate to the request URL to download the resulting STL file:

6. https://cad.onshape.com/api/v6/partstudios/d/e60c4803eaf2ac8be492c18e/w/d2558da712764516cc9fec62/e/6bed6b43463f6a46a37b4a22/stl?mode=text&grouping=true&scale=1&units=inch
7. Open the *CRANK.stl* file from wherever your downloads are saved.

Export a Part to Parasolid

We will export the `FLYWHEEL` part from this public document to an STL file.

1. Call the Part/getPartsWMV endpoint on your document and get all the part IDs:

```
curl -X 'GET' \
'https://cad.onshape.com/api/v6/parts/d/e60c4803eaf2ac8be492c18e/w/d2558da712764516cc9fec62?elementId=6bed6b43463f6a46a37b4a22&withThumbnails=false&includePropertyDefaults=false' \
  -H 'accept: application/json;charset=UTF-8; qs=0.09'
```

5. Locate the part to export (hint: look for `name = yourPartName`) in the response body. Get the part ID from the `partId` field. In the example below, `partId = JiD` for `name=FLYWHEEL`:

```
[
  ...
  {
    "name" : "FLYWHEEL",
    "state" : "IN_PROGRESS",
    "propertySourceTypes" : {
      "57f3fb8efa3416c06701d60f" : 3,
      "57f3fb8efa3416c06701d60d" : 3,
      "57f3fb8efa3416c06701d61e" : 3,
      "57f3fb8efa3416c06701d60e" : 3,
      "57f3fb8efa3416c06701d60c" : 3
    },
    "defaultColorHash" : "FzHLKqGeuTBFjmY_2_0",
    "ordinal" : 1,
    "isMesh" : false,
    "description" : "Flywheel",
    "microversionId" : "bdb504d2d4c948493a87ccf3",
    "partNumber" : "PRT-10241",
    "elementId" : "6bed6b43463f6a46a37b4a22",
    "partId" : "JiD",
    "bodyType" : "solid",
    "customProperties" : {
      "57f3fb8efa3416c06701d61e" : "false"
    }
  ...
]
```

32. Call the Part/exportPS endpoint on the FLYWHEEL part:

```
curl -X 'GET' \
'https://cad.onshape.com/api/v6/parts/d/e60c4803eaf2ac8be492c18e/w/d2558da712764516cc9fec62/e/6bed6b43463f6a46a37b4a22/partid/JiD/parasolid?version=0' \
  -H 'accept: application/octet-stream' \
```

36. Navigate to the request URL to download the resulting file:

```
https://cad.onshape.com/api/v6/parts/d/e60c4803eaf2ac8be492c18e/w/d2558da712764516cc9fec62/e/6bed6b43463f6a46a37b4a22/partid/JiD/parasolid?version=0
```

38. Open the *CRANK.x_t* file from your downloads. Note that the file is automatically named after the PartStudio to which the part belongs.

Export a PartStudio to SOLIDWORKS

We will export the `CRANK` PartStudio from this public document to a SOLIDWORKS file.

1. Validate that SOLIDWORKS is a supported export file type by calling Translation/getAllTranslatorFormats and confirming that `validDestinationFormat=true` for `translatorName=SOLIDWORKS`.

```
curl -X 'GET' \
'https://cad.onshape.com/api/v6/translations/translationformats' \
  -H 'accept: application/json;charset=UTF-8; qs=0.09'
[
  {
    "validSourceFormat": true,
    "validDestinationFormat": true,
    "name": "SOLIDWORKS",
    "translatorName": "solidworks",
    "couldBeAssembly": true
  }
]
```

14. Initialize the export by calling PartStudio/createPartStudioTranslation.

```
curl -X 'POST' \
'https://cad.onshape.com/api/v6/partstudios/d/e60c4803eaf2ac8be492c18e/w/d2558da712764516cc9fec62/e/6bed6b43463f6a46a37b4a22/translations' \
 -H 'accept: application/json;charset=UTF-8; qs=0.09' \
 -H 'Content-Type: application/json;charset=UTF-8; qs=0.09' \
 -d '{
      "formatName": "SOLIDWORKS",
      "storeInDocument": false,
      "translate": true
 }'
```

    - Note that the API takes a JSON as part of the request body, in which you can specify options for the export.
    - In this example, we've just shown a snippet of the entire JSON.
    - A `formatName` string must be specified that matches one of the valid formats you found in the last step. In this example, we set `formatName` to `SOLIDWORKS`.
    - We want to export this to a new file, so we'll leave `storeInDocument` set to `false`.
24. Next, we poll the PartStudio/createPartStudioTranslation response until `requestState` changes from `ACTIVE` to `DONE` or `FAILED`.

```
{
  "documentId": "e60c4803eaf2ac8be492c18e",
  "requestElementId": "6bed6b43463f6a46a37b4a22",
  "requestState": "DONE",
  "resultExternalDataIds": "[{resultId}]",
  ...
}
```

32. Once `requestState=DONE`, we can call Documents/downloadExternalData to retrieve the exported result.

```
curl -X 'GET' \
```

```
'https://cad.onshape.com/api/v6/documents/d/e60c4803eaf2ac8be492c18e/externaldata/{fid}'
```

- Use the `resultExternalDataIds` value from the translation response as the foreign ID (`fid`).
- The new SOLIDWORKS file is returned as the response and will be downloaded to wherever the API call is made.

Export an Assembly to STEP

In this example, we'll export an assembly from this public document to a STEP file.

1. Make a copy of this public document so you can export the assembly. Make a note of the documentId, workspaceId, and elementId of the assembly in your new document.
2. Validate that STEP is a supported export file type for assemblies by calling Translation/getAllTranslatorFormats and confirming that `validDestinationFormat=true` and `couldBeAssembly=true` for `translatorName=STEP`.

```
curl -X 'GET' \
'https://cad.onshape.com/api/v6/translations/translationformats' \
  -H 'accept: application/json;charset=UTF-8; qs=0.09'
[
  {
    "validSourceFormat": true,
    "validDestinationFormat": true,
    "name": "STEP",
    "translatorName": "step",
    "couldBeAssembly": true
  },
  ...
]
```

16. Initialize the export by calling Assembly/translateFormat. Replace `{did}`, `{wid}`, and `{eid}` with the document, workspace, and element IDs from your copied document. Do NOT include the curly braces ({}) in the final call.

```
curl -X 'POST' \
'https://cad.onshape.com/api/v6/assemblies/d/{did}/w/{wid}/e/{eid}/translations' \
  -H 'accept: application/json;charset=UTF-8; qs=0.09' \
  -H 'Content-Type: application/json;charset=UTF-8; qs=0.09' \
  -d '{
      "allowFaultyParts": true,
      "angularTolerance": 0.001,
      "formatName": "STEP",
      "storeInDocument": true
      }'
```

- Note that the API takes a JSON as part of the request body, in which you can specify options for the export.
- In the example above, we've just shown a snippet of the entire JSON where we allow faulty parts to be exported and set the angular tolerance to 0.001.
- A `formatName` string must be specified that matches one of the valid formats you found in the last step. In this example, we set `formatName` to `STEP`.
- Set `storeInDocument` to `true` to upload the STEP file as a blob element in your document.

27. Next, we poll the Assembly/translateFormat response until `requestState` changes from `ACTIVE` to `DONE` or `FAILED`.

```
{
  "resultDocumentId" : "{did}",
  "resultWorkspaceId" : "{wid}",
  "requestState" : "DONE",
  "requestElementId" : "{eid}",
  "resultExternalDataIds" : [ "{resultExternalId}" ],
  "documentId" : "{did}",
  "workspaceId" : "{wid}",
  "resultElementIds" : {resulteid},
  "name" : "GEARBOX_CHUCK",
  "id" : "{translationId}",
  "href" : "https://cad.onshape.com/api/v6/translations/{translationId}"
}
```

41. Once `requestState=DONE`, we make a note of the `resultElementId` in the response. This is the elementId of the STEP blob.
42. Now, we can call BlobElement/downloadFileWorkspace to retrieve the exported results.

```
curl -X 'GET' \
'https://cad.onshape.com/api/v6/blobelements/d/{did}/w/{wid}/e/{resulteid}
' \
  -H 'accept: application/octet-stream'
```

    - Use the `resultElementIds` value from the translation response as the element ID (`{resulteid}`).
    - Note that you can also open your document, click the `GEARBOX_CHUCK.STEP` tab, and download the file from there.

Export a Drawing as a JSON

In this example, we'll export a Drawing from this public document to a JSON file. Exporting a Drawing to JSON is useful when you need to gather information about that drawing (for example, finding valid coordinates on which to place an inspection symbol).

1. Make a copy of this public document so you can export the assembly. Make a note of the documentId, workspaceId, and elementId of the assembly in your new document.
2. Validate that JSON is a supported export file type for Drawings by calling Drawing/getDrawingTranslatorFormats and confirming that `"name": "DRAWING_JSON"` appears in the response for the drawing element in your copied document.

```
curl -X 'GET' \
'https://cad.onshape.com/api/v6/drawings/d/{did}/w/{wid}/e/{eid}/translati
onformats' \
  -H 'accept: application/json;charset=UTF-8; qs=0.09'
[
  {
    "name": "DRAWING_JSON",
    "translatorName": "drawing_json",
    "couldBeAssembly": false
  },
  ...
]
```

14. Initialize the export by calling Drawing/createDrawingTranslation. Replace {did}, {wid}, and {eid} with the document, workspace, and element IDs from your copied document. Do NOT include the curly braces ({}) in the final call.

```
curl -X 'POST' \
'https://cad.onshape.com/api/v6/drawings/d/{did}/w/{wid}/e/{eid}/translations' \
  -H 'accept: application/json;charset=UTF-8; qs=0.09' \
  -H 'Content-Type: application/json;charset=UTF-8; qs=0.09' \
  -d '{
      "formatName": "DRAWING_JSON"
      }'
```

    - Note that the API takes a JSON as part of the request body, in which you can specify options for the export.
    - The only required JSON field is formatName, in which we've specified the format as found in the getDrawingTranslatorFormats response body.

22. Next, we poll the Drawing/createDrawingTranslation response until requestState changes from ACTIVE to DONE or FAILED.

```
{
  "resultDocumentId" : "{did}",
  "resultWorkspaceId" : "{wid}",
  "requestState" : "DONE",
  "requestElementId" : "{eid}",
  "resultExternalDataIds" : [ "{resultExternalId}" ],
  "documentId" : "{did}",
  "workspaceId" : "{wid}",
  "resultElementIds" : {eid},
  "name" : "GEARBOX_CHUCK",
  "id" : "{translationId}",
  "href" : "https://cad.onshape.com/api/v6/translations/{translationId}"
}
```

36. Once requestState=DONE, we make a note of the "resultElementIds" : {resulteid}, in the response. This is the element ID of the JSON blob.
37. Now, we can call BlobElement/downloadFileWorkspace to retrieve the exported results.

```
curl -X 'GET' \
'https://cad.onshape.com/api/v6/blobelements/d/{did}/w/{wid}/e/{resulteid}' \
  -H 'accept: application/octet-stream'
```

    - Use the {resultExternalId} value from the translation response as the element ID ({resulteid}). Do not include the curly braces in your call.
    - Note that you can also open your document, click the GEARBOX_CHUCK.JSON tab, and download the file from there.

Import a Parasolid file as a Part

In this example, we'll import the *FLYWHEEL* part from the *CRANK.x_t* file we created in the Export a Part to Parasolid example.

1. Open or create a new Onshape document in which to import the Part. Make a note of the documentId and workspaceId of your document.

2. Validate that Parasolid is a supported export file type for imports by calling Translation/getAllTranslatorFormats and confirming that `validSourceFormat=true` for `translatorName=parasolid`.

```
curl -X 'GET' \
'https://cad.onshape.com/api/v6/translations/translationformats' \
  -H 'accept: application/json;charset=UTF-8; qs=0.09'
[
  {
    "validSourceFormat": true,
    "validDestinationFormat": true,
    "name": "PARASOLID",
    "translatorName": "parasolid",
    "couldBeAssembly": true
  }
]s
```

15. Initialize the import by calling Translation/createTranslation. In this example, the filename is `CRANK.x_t`.

```
curl -X 'POST' \
'https://cad.onshape.com/api/v6/translations/d/{did}/w/{wid}' \
  -H 'accept: application/json;charset=UTF-8; qs=0.09's \
  -H 'Content-Type: multipart/form-data' \
  -F 'formatName=' \
  -F 'flattenAssemblies=true' \
  -F 'translate=true' \
  -F 'file=@/pathToFile/CRANK.x_t' \
```

    - Replace `{did}` and `{wid}` in the URL with the document and workspace IDs for the document you want to import the part to.
    - Note that when using cURL, you must begin the path to the file with an `@` symbol.
    - Note that the API takes a JSON as part of the request body, in which you can specify options for the import.
    - When importing files, the API assumes we are importing to the `ONSHAPE` file type. You can override this and import to a different file type using the `formatName` field. In this case, we can leave the `formatName` field blank to import to the `ONSHAPE` file type.

24. Next, we poll the Translation/getDocumentTranslations response until `requestState` changes from `ACTIVE` to `DONE` or `FAILED`.

```
{
  "requestState" : "DONE",
  "documentId" : "{did}",
  "workspaceId" : "{wid}",
  "resultElementIds" : [ "{resulteid}" ],
  "name" : "FLYWHEEL",
  "id" : "{id}",
  "href" : "https://cad.onshape.com/api/v1/translations/{tid}"
}
```

34. Once `requestState=DONE`, we can view the imported file as a Part in our Onshape document. The `FLYWHEEL` part appears in a new PartStudio named `CRANK` in our document.

Additional Resources

- Onshape Help: Translation

- Onshape Help: Webhooks
- API Guide: Webhook Notifications
- API Explorer

Drawings

This page describes the APIs Onshape provides for creating and manipulating Onshape drawings.

Note

This page provides sample code as curls. See the curl documentation for more information.

Note

All Onshape API calls must be properly authenticated by replacing the `CREDENTIALS` variable in the curls below. See the API Keys page for instructions and the Quick Start for an example. All applications submitted to the Onshape App Store *must* authenticate with OAuth2.

Note

This documentation refers to Onshape IDs in the following format: `{did}`, `{wid}`, `{eid}`, `{pid}`, `{otherId}`. These represent document, workspace, element, part, and other IDs (respectively) that are needed make the API calls. Please see API Guide: API Intro for information on what these IDs mean and how to obtain them from your documents. Never include the curly braces (`{}`) in your API calls.

Note

For Enterprise accounts, replace **cad** in all Onshape URLs with your company domain. https://**cad**.onshape.com > https://**companyName**.onshape.com

Endpoints

To create drawings, Onshape allows you to send all drawing data points and information through the API as part of the request body JSON.

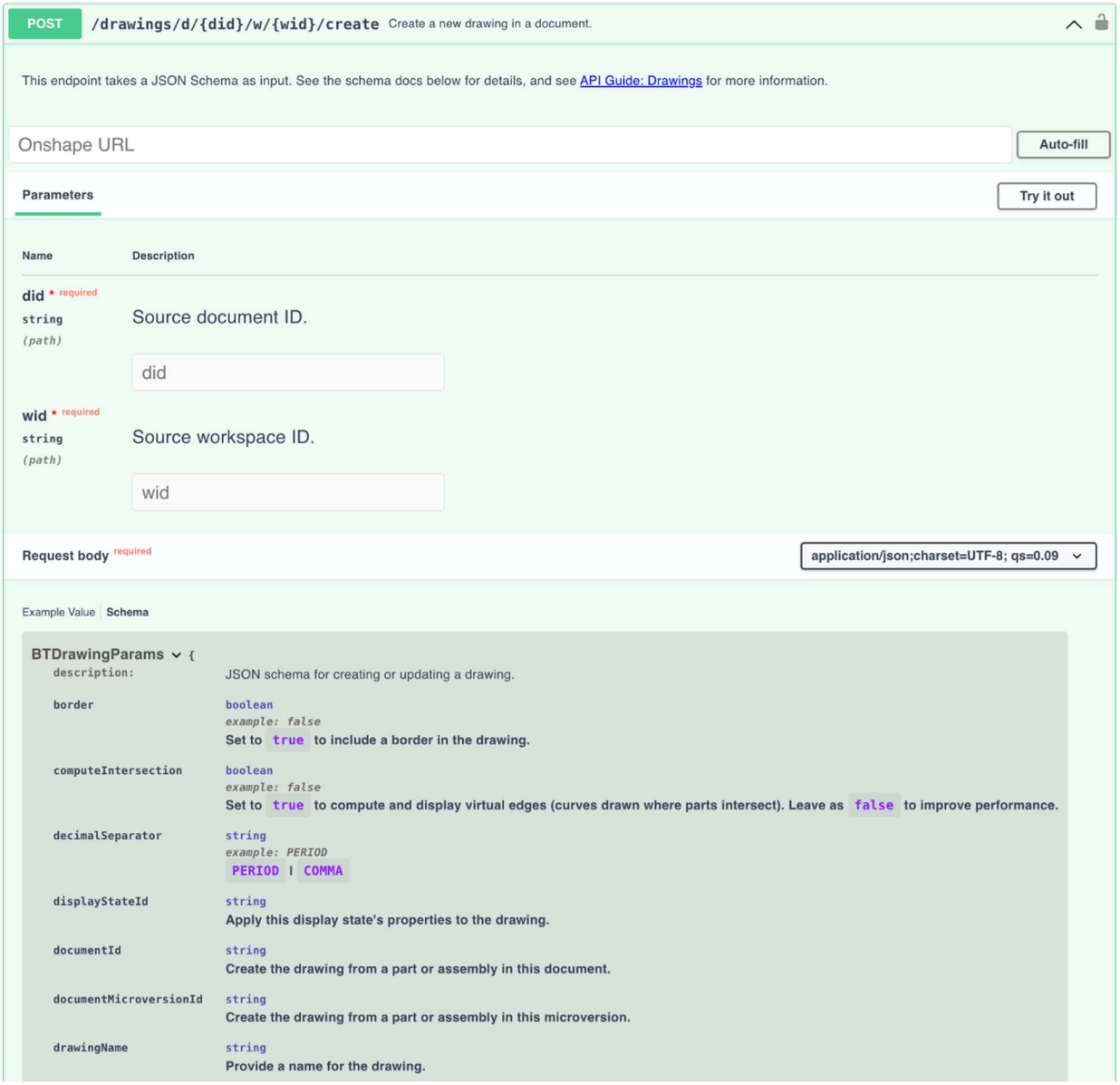
The following endpoints are available:

- Create a drawing

```
curl -X 'POST' \
  'https://cad.onshape.com/api/v6/drawings/d/{did}/w/{wid}/create' \
  -H 'Authorization: Basic CREDENTIALS' \
  -H 'Accept: application/json;charset=UTF-8; qs=0.09' \
  -H 'Content-Type: application/json;charset=UTF-8; qs=0.09' \
  -d '{
    <JSON request body options from the BTDrawingParams schema>
}'
```

Specify the document in which to create the drawing in the URL, and pass any additional options as part of the request body. You can provide a name for the drawing, manipulate the drawing graphics area, specify a part or template to create the drawing from, and more.

- See documentation for all available options in the API Explorer.
- For instructions on viewing the documentation for the request body schemas, see our API Explorer page. Check out the Sample Workflows section below for some practical examples.



- Modify a drawing

```
curl -X 'POST \
  'https://cad.onshape.doc/api/v6/drawings/d/{did}/w/{wid}/e/{eid}/modify' \
  -H 'Authorization: Basic CREDENTIALS' \
  -H 'Accept: application/json;charset=UTF-8; qs=0.09' \
  -H 'Content-Type: application/json;charset=UTF-8; qs=0.09' \
  -d '{
      "description": "Description of the modification.",
      "jsonRequests": [
        {
          <JSON request body options from the jsonRequests schema>
        }
```

```
        ]
      }'
```

Specify the drawing to modify in the URL, and pass the information on the modification in the request body. Note that the `jsonRequests` schema is not defined in the Glassworks API Explorer; see the OnshapeDrawingJson repository for this information, and check out the Sample Workflows section below for some practical examples.

- Get the drawing modification status

```
curl -X 'GET' \
  'https://cad.onshape.com/api/v6/drawings/modify/status/{mrid}
  -H 'Authorization: Basic CREDENTIALS' \
  -H 'Accept: application/json;charset=UTF-8; qs=0.09' \
  -H 'Content-Type: application/json;charset=UTF-8; qs=0.09' \
```

Provide the modification ID (from the `modifyDrawing` response body) to get the status of the modification.

- Translate a drawing: See the Translations API Guide.
- Get drawing translation formats: See the Translations API Guide.

Sample Workflows

Create a drawing from a part

In this example, we'll create a drawing from the **FLYWHEEL** part in this public document.

1. Create or open an Onshape document in which to create your drawing.
2. Start to form the Drawings/createDrawingAppElement call.
   Replace `{did}` and `{wid}` in the URL below with the document ID and workspace ID of your document (i.e., the *target* document), and replace `CREDENTIAL` with your authorization credentials.

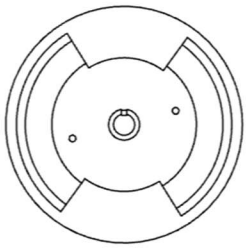
```
curl -X 'POST' \
  'https://cad.onshape.com/api/v6/drawings/d/{did}/w/{wid}/create' \
  -H 'Authorization: Basic CREDENTIALS' \
  -H 'Accept: application/json;charset=UTF-8; qs=0.09' \
  -H 'Content-Type: application/json;charset=UTF-8; qs=0.09' \
```

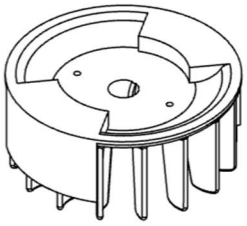
8. Add the request body information:
   - Add `flywheelDrawing` as the `drawingName` field.
   - We must specify the *source* document's document and version IDs. Note that since our target document and source document are different, we use the `external` document and version ID fields.
   - We must also provide the ID of the part to create the drawing from, and the ID of the element (i.e., tab) in which the part lives.

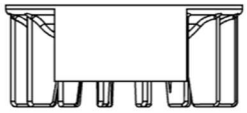
(Hint: You can call Part/getPartsWMVE to get a list of part IDs in an element.)

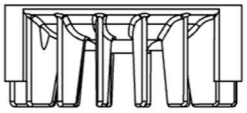
```
curl -X 'POST' \
  'https://cad.onshape.com/api/v6/drawings/d/{did}/w/{wid}/create' \
  -H 'Authorization: Basic CREDENTIALS' \
  -H 'Accept: application/json;charset=UTF-8; qs=0.09' \
  -H 'Content-Type: application/json;charset=UTF-8; qs=0.09' \
  -d '{
        "drawingName": "flywheelDrawing",
        "externalDocumentId": "e60c4803eaf2ac8be492c18e",
        "externalDocumentVersionId": "405ba186c3a70e0227ab2941",
        "elementId": "6bed6b43463f6a46a37b4a22",
        "partId": "JiD"
      }'
```

9. Call the endpoint and open your document. Confirm that you see the new `flywheelDrawing` element containing the drawing:









Create a drawing from a template

In this example, we'll create a drawing from the standard ANSI template in [this public document].

1. Open any Onshape document in which to create your drawing.
2. Start to form the [Drawings/createDrawingAppElement] call.
   Replace `{did}` and `{wid}` in the URL below with the document ID and workspace ID of your document (i.e., the *target* document), and replace `CREDENTIAL` with your authorization credentials.

```
curl -X 'POST' \
  'https://cad.onshape.com/api/v6/drawings/d/{did}/w/{wid}/create' \
  -H 'Authorization: Basic CREDENTIALS' \
  -H 'Accept: application/json;charset=UTF-8; qs=0.09' \
  -H 'Content-Type: application/json;charset=UTF-8; qs=0.09' \
```

8. Add the request body information:
   - Add `templateAnsiDrawing` as the `drawingName` field.
   - We must specify the *source* document's document ID and workspace ID.
   - We must also provide the ID of the element (i.e., tab) in which the template lives.
   - Note that we use the `template` document, workspace, and element ID fields when referring to a specific template for drawing creation.

```
curl 'https://cad.onshape.com/api/drawings/d/{did}/w/{wid}/create' \
  -H 'Authorization: Basic CREDENTIALS' \
  -H 'Accept: application/json, text/plain, */*' \
  -H 'Content-Type: application/json;charset=UTF-8' \
  -d '{
    "drawingName": "templateAnsiDrawing",
    "templateDocumentId":"cbe6e776694549b5ba1a3e88",
    "templateWorkspaceId":"24d08acf10234dbc8d3ab585",
    "templateElementId":"17eef7862b224f6fb12cbc46"
    }'
```

9. Call the endpoint and open your document. Confirm that you see the new `templateAnsiDrawing` element containing the emtpy drawing template.

Create a drawing in a custom graphics area

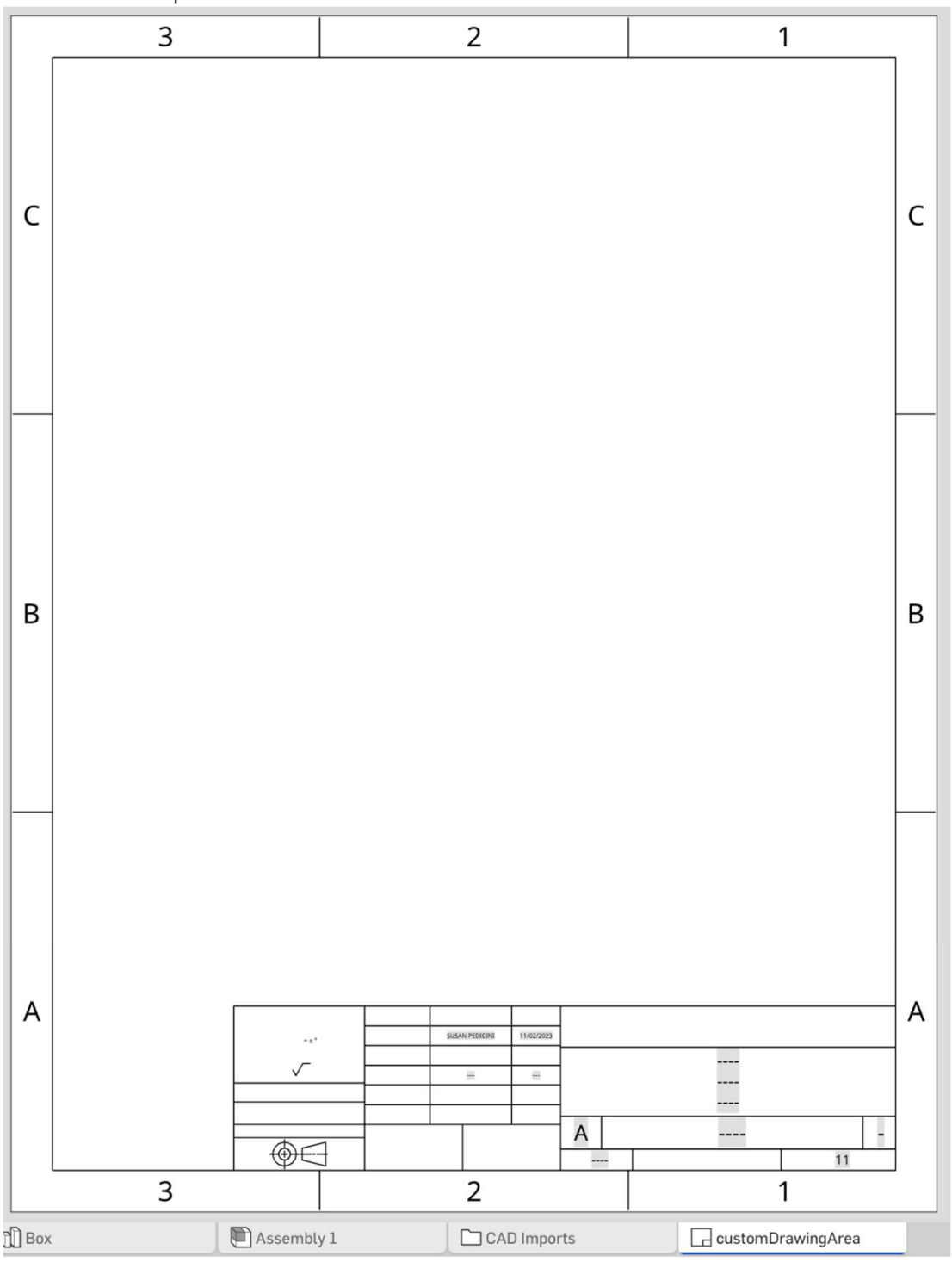
1. Open any Onshape document in which to create your drawing.
2. Start to form the [Drawings/createDrawingAppElement] call.
   Replace `{did}` and `{wid}` in the URL below with your document, and replace `CREDENTIAL` with your authorization credentials.

```
curl -X 'POST' \
  'https://cad.onshape.com/api/v6/drawings/d/{did}/w/{wid}/create' \
  -H 'Authorization: Basic CREDENTIALS' \
  -H 'Accept: application/json;charset=UTF-8; qs=0.09' \
  -H 'Content-Type: application/json;charset=UTF-8; qs=0.09' \
```

8. Add information about the drawings area. In this example, we'll add an additional column and row to the drawings area, a title block, and add a border around it.

```
curl -X 'POST' \
  'https://cad.onshape.com/api/v6/drawings/d/{did}/w/{wid}/create' \
  -H 'Authorization: Basic CREDENTIALS' \
  -H 'accept: application/json;charset=UTF-8; qs=0.09' \
  -H 'Content-Type: application/json;charset=UTF-8; qs=0.09' \
  -d '{
        "drawingName": "customGraphicsArea",
        "border": "true",
        "numberHorizontalZones": "3",
        "numberVerticalZones": "3"
        "titleblock": true
      }'
```

21. Call the endpoint and open your document. Confirm that you see the new `customGraphicsArea` element:



Add a note to a drawing

1. Open an existing (or create a new) Onshape document with a drawing.

2. Start to form the Drawings/modifyDrawing call. Replace the URL parameters with the values from your document, and replace `CREDENTIALS` with your authorization credentials.

```
curl -X 'POST \
  'https://cad.onshape.doc/api/v6/drawings/d/{did}/w/{wid}/e/{eid}/modify' \
  -H 'Authorization: Basic CREDENTIALS' \
  -H 'Accept: application/json;charset=UTF-8; qs=0.09' \
  -H 'Content-Type: application/json;charset=UTF-8; qs=0.09' \
```

8. Add information about the modification. In this example, we'll create an `Onshape::Note` on the drawing. We must specify the `messageName` and `formatVersion` for the modification, and then provide the contents and size of the annotation.

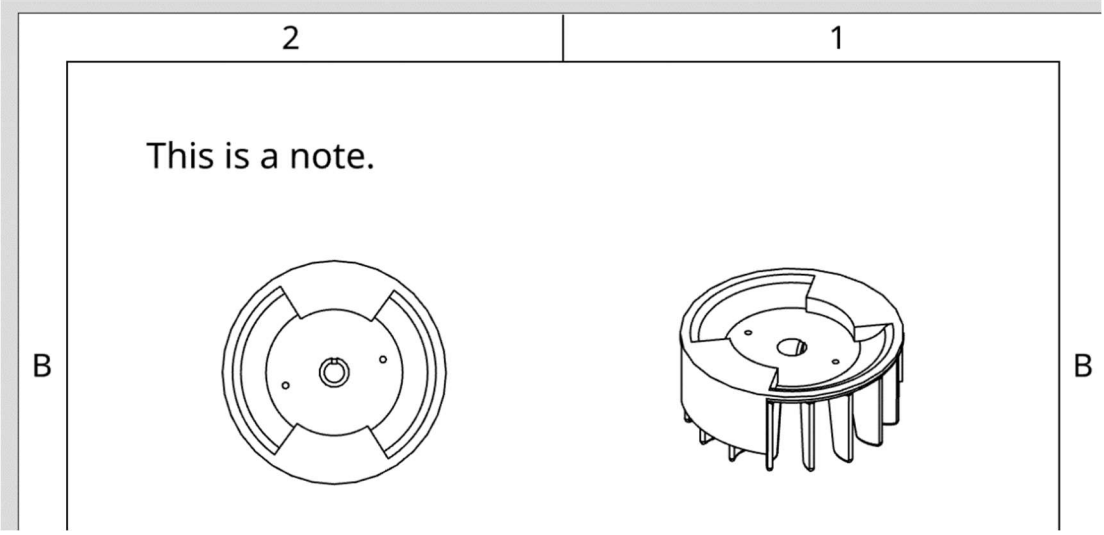
```
curl -X 'POST \
  'https://cad.onshape.doc/api/v6/drawings/d/{did}/w/{wid}/e/{eid}/modify' \
  -H 'Authorization: Basic CREDENTIALS' \
  -H 'Accept: application/json;charset=UTF-8; qs=0.09' \
  -H 'Content-Type: application/json;charset=UTF-8; qs=0.09' \
  -d '{
        "description": "Add a note to the drawing.",
        "jsonRequests": [ {
          "messageName": "onshapeCreateAnnotations",
          "formatVersion": "2021-01-01",
          "annotations": [
            {
              "type": "Onshape::Note",
              "note": {
                "position": {
                  "type": "Onshape::Reference::Point",
                  "coordinate": [
                    1,
                    10,
                    0
                  ]
                },
                "contents": "This is a note",
                "textHeight": 0.2,
                "logicalId": "note1"
              }
            }
          ]
        }]
      }'
```

39. Make the call, and then get `id` from the response body. You'll need this to poll the modification status to figure out when the modification has completed.
40. Set up the Drawings/getModificationStatus call. Replace `{mrid}` with the `id` from the last step, and replace `CREDENTIALS` with your credentials. Poll the modification status until the response returns `"requestState": "DONE"`.

```
curl -X 'GET \
```

```
  'https://cad.onshape.doc/api/v6/drawings/modify/status/{mrid}' \
  -H 'Authorization: Basic CREDENTIALS' \
  -H 'Accept: application/json;charset=UTF-8; qs=0.09' \
  -H 'Content-Type: application/json;charset=UTF-8; qs=0.09' \
```

46. Open your drawing and confirm that you see the new note. Note that your drawing may not match this image exactly, depending on your drawing and document properties. This sample document uses Inches for units.



Add a callout to a drawing

1. Open an existing (or create a new) Onshape document with a drawing.
2. Start to form the Drawings/modifyDrawing call. Replace the URL parameters with the values from your document, and replace `CREDENTIALS` with your authorization credentials.

```
curl -X 'POST \
  'https://cad.onshape.doc/api/v6/drawings/d/{did}/w/{wid}/e/{eid}/modify' \
  -H 'Authorization: Basic CREDENTIALS' \
  -H 'Accept: application/json;charset=UTF-8; qs=0.09' \
  -H 'Content-Type: application/json;charset=UTF-8; qs=0.09' \
```

8. Add information about the modification. In this example, we'll add an `Onshape::Callout` to the drawing. We must specify the `messageName` and `formatVersion` for the modification, and then provide the contents and size of the annotation:

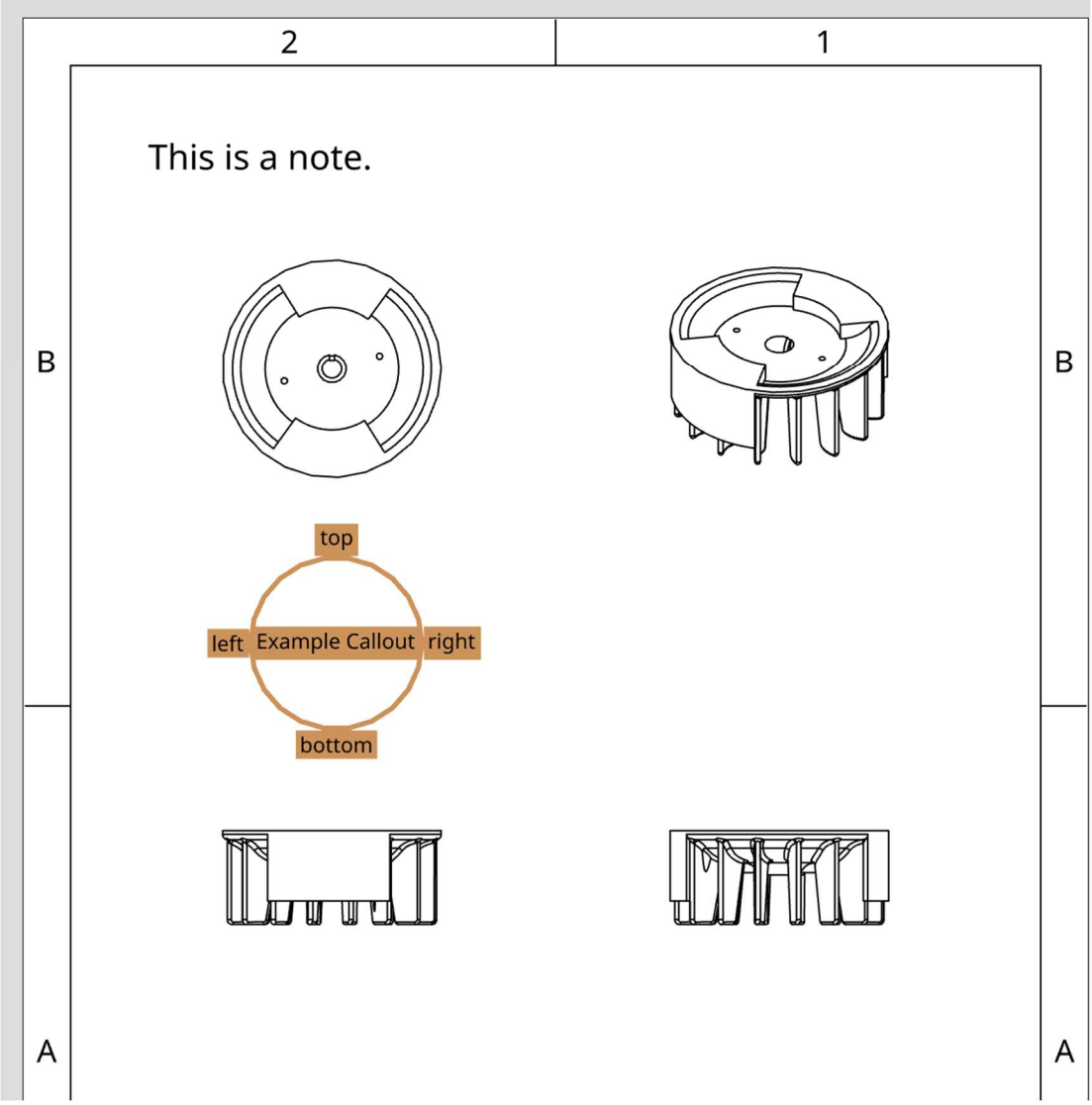
```
curl -X 'POST \
  'https://cad.onshape.doc/api/v6/drawings/d/{did}/w/{wid}/e/{eid}/modify' \
  -H 'Authorization: Basic CREDENTIALS' \
  -H 'Accept: application/json;charset=UTF-8; qs=0.09' \
  -H 'Content-Type: application/json;charset=UTF-8; qs=0.09' \
  -d '{
        "description": "Add a callout to the drawing.",
        "jsonRequests": [ {
```

```
          "messageName": "onshapeCreateAnnotations",
          "formatVersion": "2021-01-01",
          "annotations": [
            {
              "callout":
              {
                "borderShape": "Circle",
                "borderSize": 0,
                "contents": "Example Callout",
                "contentsBottom": "bottom",
                "contentsLeft": "left",
                "contentsRight": "right",
                "contentsTop": "top",
                "position": {
                  "coordinate": [
                    2.5,
                    6,
                    0
                  ],
                  "type": "Onshape::Reference::Point"
                },
                "textHeight": 0.12,
                "logicalId": "callout1"
              },
              "type": "Onshape::Callout"
            }
          ]
        }]
      }'
```

46. Make the call, and then get `id` from the response body. You'll need this to poll the modification status to figure out when the modification has completed.
47. Set up the Drawings/getModificationStatus call. Replace `{mrid}` with the `id` from the last step, and replace `CREDENTIALS` with your credentials. Poll the modification status until the response returns `"requestState": "DONE"`.

```
curl -X 'GET \
  'https://cad.onshape.doc/api/v6/drawings/modify/status/{mrid}' \
  -H 'Authorization: Basic CREDENTIALS' \
  -H 'Accept: application/json;charset=UTF-8; qs=0.09' \
  -H 'Content-Type: application/json;charset=UTF-8; qs=0.09' \
```

53. Open your drawing and confirm that you see the new callout.



Add a centerline to a drawing

1. Open an existing (or create a new) Onshape document with a drawing.
2. Start to form the Drawings/modifyDrawing call. Replace the URL parameters with the values from your document, and replace `CREDENTIALS` with your authorization credentials.

```
curl -X 'POST \
  'https://cad.onshape.doc/api/v6/drawings/d/{did}/w/{wid}/e/{eid}/modify' \
  -H 'Authorization: Basic CREDENTIALS' \
  -H 'Accept: application/json;charset=UTF-8; qs=0.09' \
  -H 'Content-Type: application/json;charset=UTF-8; qs=0.09' \
```

8. Add information about the modification. In this example, we'll add an `Onshape::Centerline` to the drawing. We must specify

the `messageName` and `formatVersion` for the modification, and then provide the coordinates of the centerline ends.

- Note: you can Export a Drawing to JSON to get a list of valid coordinates and handles.
- Note: if you have access, you can refer to the ODA documentation for more detailed formatting information.

```
curl -X 'POST \
  'https://cad.onshape.doc/api/v6/drawings/d/{did}/w/{wid}/e/{eid}/modify' \
  -H 'Authorization: Basic CREDENTIALS' \
  -H 'Accept: application/json;charset=UTF-8; qs=0.09' \
  -H 'Content-Type: application/json;charset=UTF-8; qs=0.09' \
  -d '{
        "description": "Add a centerline to the drawing",
        "jsonRequests": [ {
          "messageName": "onshapeCreateAnnotations",
          "formatVersion": "2021-01-01",
          "annotations": [
            {
              "pointToPointCenterline": {
                  "point1": {
                    "coordinate": [
                        2,
                        4,
                        0
                    ],
                    "type": "Onshape::Reference::Point",
                    "uniqueId": "point1",
                    "viewId": "51fa8b6040e411dfd17a4cda"
                  },
                  "point2": {
                    "coordinate": [
                        7,
                        6,
                        1
                    ],
                    "type": "Onshape::Reference::Point",
                    "uniqueId": "point2",
                    "viewId": "ay6a8b6020e4h7dfdnn1499i"
                  }
              },
              "type": "Onshape::Centerline::PointToPoint"
            }
          ]
        }]
      }'
```

4. Make the call, and then get `id` from the response body. You'll need this to poll the modification status to figure out when the modification has completed.
5. Set up the Drawings/getModificationStatus call. Replace `{mrid}` with the `id` from the last step, and replace `CREDENTIALS` with your credentials. Poll the modification status until the response returns `"requestState": "DONE"`.

```
curl -X 'GET \
  'https://cad.onshape.doc/api/v6/drawings/modify/status/{mrid}' \
  -H 'Authorization: Basic CREDENTIALS' \
  -H 'Accept: application/json;charset=UTF-8; qs=0.09' \
  -H 'Content-Type: application/json;charset=UTF-8; qs=0.09' \
```

11. Open your drawing and confirm that you see the new centerline.

Add a dimension to a drawing

1. Open an existing (or create a new) Onshape document with a drawing.
2. Start to form the Drawings/modifyDrawing call. Replace the URL parameters with the values from your document, and replace `CREDENTIALS` with your authorization credentials.

```
curl -X 'POST \
  'https://cad.onshape.doc/api/v6/drawings/d/{did}/w/{wid}/e/{eid}/modify'
\
  -H 'Authorization: Basic CREDENTIALS' \
  -H 'Accept: application/json;charset=UTF-8; qs=0.09' \
  -H 'Content-Type: application/json;charset=UTF-8; qs=0.09' \
```

8. Add information about the modification. In this example, we'll add an `Onshape::Dimension` to the drawing. We must specify the `messageName` and `formatVersion` for the modification, and then provide the coordinates and formatting options for the dimension. (Hint: you can Export a Drawing to JSON to get a list of valid coordinates and handles.)

```
curl -X 'POST \
  'https://cad.onshape.doc/api/v6/drawings/d/{did}/w/{wid}/e/{eid}/modify'
\
  -H 'Authorization: Basic CREDENTIALS' \
  -H 'Accept: application/json;charset=UTF-8; qs=0.09' \
  -H 'Content-Type: application/json;charset=UTF-8; qs=0.09' \
  -d '{
        "description": "Add a dimension to the drawing",
        "jsonRequests": [ {
          "messageName": "onshapeCreateAnnotations",
          "formatVersion": "2021-01-01",
          "annotations": [
            {
              "radialDimension": {
                "centerPoint": {
                  "coordinate": [
                    0.2800021171569824,
                    0.014964947476983043,
                    0.079502
```

```
                  ],
                  "type": "Onshape::Reference::Point",
                  "uniqueId": "point1",
                  "viewId": "e11c38795c04ca55047f7ea7"
                },
                "chordPoint": {
                  "coordinate": [
                    0.2920149764955524,
                    0.010030535983985095,
                    0.079502
                  ],
                  "type": "Onshape::Reference::Point",
                  "uniqueId": "point2",
                  "viewId": "e11c38795c04ca55047f7ea7"
                },
                "formatting": {
                  "dimdec": 2,
                  "dimlim": false,
                  "dimpost": "R<>",
                  "dimtm": 0,
                  "dimtol": false,
                  "dimtp": 0,
                  "type": "Onshape::Formatting::Dimension"
                },
                "logicalId": "dimension1",
                "textOverride": "",
                "textPosition": {
                  "coordinate": [
                    191.80537349378181,
                    89.76274130852224,
                    0
                  ],
                  "type": "Onshape::Reference::Point"
                }
              },
              "type": "Onshape::Dimension::Radial"
            }
          ]
        }]
      }'
```

67. Make the call, and then get `id` from the response body. You'll need this to poll the modification status to figure out when the modification has completed.
68. Set up the Drawings/getModificationStatus call. Replace `{mrid}` with the `id` from the last step, and replace `CREDENTIALS` with your credentials. Poll the modification status until the response returns `"requestState": "DONE"`.

```
curl -X 'GET \
  'https://cad.onshape.doc/api/v6/drawings/modify/status/{mrid}' \
  -H 'Authorization: Basic CREDENTIALS' \
  -H 'Accept: application/json;charset=UTF-8; qs=0.09' \
  -H 'Content-Type: application/json;charset=UTF-8; qs=0.09' \
```

74. Open your drawing and confirm that you see the new dimension.

Add a geometric tolerance to a drawing

1. Open an existing (or create a new) Onshape document with a drawing.
2. Start to form the Drawings/modifyDrawing call. Replace the URL parameters with the values from your document, and replace `CREDENTIALS` with your authorization credentials.

```
curl -X 'POST \
  'https://cad.onshape.doc/api/v6/drawings/d/{did}/w/{wid}/e/{eid}/modify' \
  -H 'Authorization: Basic CREDENTIALS' \
  -H 'Accept: application/json;charset=UTF-8; qs=0.09' \
  -H 'Content-Type: application/json;charset=UTF-8; qs=0.09' \
```

8. Add information about the modification. In this example, we'll add an `Onshape::GeometricTolerance` to the drawing. We must specify the `messageName` and `formatVersion` for the modification, and then provide the frames and position of the annotation:

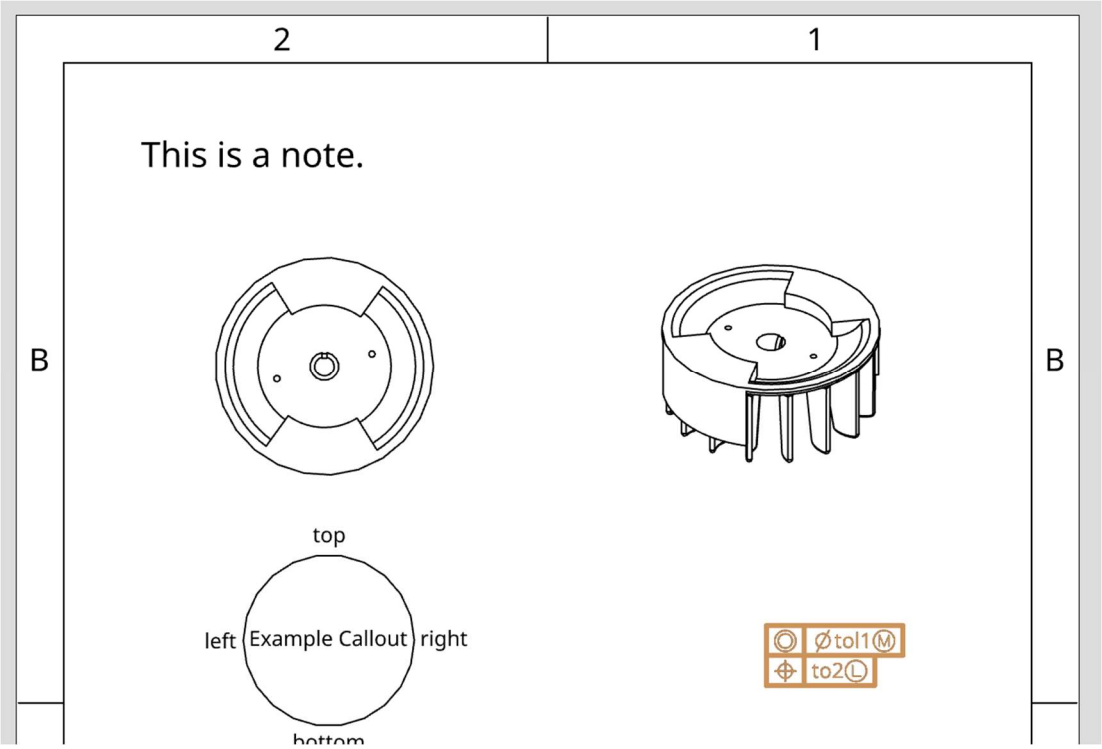
```
curl -X 'POST \
  'https://cad.onshape.doc/api/v6/drawings/d/{did}/w/{wid}/e/{eid}/modify' \
  -H 'Authorization: Basic CREDENTIALS' \
  -H 'Accept: application/json;charset=UTF-8; qs=0.09' \
  -H 'Content-Type: application/json;charset=UTF-8; qs=0.09' \
  -d '{
        "description": "Add a geometric tolerance to the drawing",
        "jsonRequests": [ {
          "messageName": "onshapeCreateAnnotations",
          "formatVersion": "2021-01-01",
          "annotations": [
            {
              "geometricTolerance": {
                "frames": [
                  "{\\fDrawing Symbols Sans;◎}%%v{\\fDrawing Symbols Sans;Ø}tol1{\\fDrawing Symbols Sans;Ⓜ}%%v%%v%%v%%v%%v\n",
                  "{\\fDrawing Symbols Sans;⌖}%%vto2{\\fDrawing Symbols Sans;Ⓛ}%%v%%v%%v%%v%%v\n"
                ],
                "logicalId": "geometricTolerance1",
                "position": {
                  "coordinate": [
                    6,
                    6,
                    0
                  ],
                  "type": "Onshape::Reference::Point"
                }
              },
              "type": "Onshape::GeometricTolerance"
            }
          ]
        }]
```

```
40.       }'
```

41. Make the call, and then get `id` from the response body. You'll need this to poll the modification status to figure out when the modification has completed.
42. Set up the Drawings/getModificationStatus call. Replace `{mrid}` with the `id` from the last step, and replace `CREDENTIALS` with your credentials. Poll the modification status until the response returns `"requestState": "DONE"`.

```
43. curl -X 'GET \
44.   'https://cad.onshape.doc/api/v6/drawings/modify/status/{mrid}' \
45.   -H 'Authorization: Basic CREDENTIALS' \
46.   -H 'Accept: application/json;charset=UTF-8; qs=0.09' \
47.   -H 'Content-Type: application/json;charset=UTF-8; qs=0.09' \
```

48. Open your drawing and confirm that you see the new annotation.



Add an inspection symbol to a drawing

1. Open an existing (or create a new) Onshape document with a drawing.
2. Start to form the Drawings/modifyDrawing call. Replace the URL parameters with the values from your document, and replace `CREDENTIALS` with your authorization credentials.

```
curl -X 'POST \
  'https://cad.onshape.doc/api/v6/drawings/d/{did}/w/{wid}/e/{eid}/modify' \
  -H 'Authorization: Basic CREDENTIALS' \
  -H 'Accept: application/json;charset=UTF-8; qs=0.09' \
  -H 'Content-Type: application/json;charset=UTF-8; qs=0.09' \
```

3. Add information about the modification. In this example, we'll add an `Onshape::InspectionSymbol` to the drawing. We must specify the `messageName` and `formatVersion` for the modification, and then provide the shape and position of the inspection symbol. (Hint: you can Export a Drawing to JSON to get a list of valid coordinates and handles.)

```
curl -X 'POST \
  'https://cad.onshape.doc/api/v6/drawings/d/{did}/w/{wid}/e/{eid}/modify' \
  -H 'Authorization: Basic CREDENTIALS' \
  -H 'Accept: application/json;charset=UTF-8; qs=0.09' \
  -H 'Content-Type: application/json;charset=UTF-8; qs=0.09' \
  -d '{
        "description": "Add an inspection symbol to the drawing",
        "jsonRequests": [ {
          "messageName": "onshapeCreateAnnotations",
          "formatVersion": "2021-01-01",
          "annotations": [
            {
              "inspectionSymbol": {
                "borderShape": "Circle",
                "borderSize": 2,
                "logicalId": "inspection1",
                "parentAnnotation": "h:10000577",
                "parentLineIndex": 0.0,
                "position": {
                  "coordinate": [
                    2.6,
                    6,
                    0
                  ],
                  "type": "Onshape::Reference::Point"
                },
                "textHeight": 2
              },
              "type": "Onshape::InspectionSymbol"
            }
          ]
        }]
      }'
```

4. Make the call, and then get `id` from the response body. You'll need this to poll the modification status to figure out when the modification has completed.
5. Set up the Drawings/getModificationStatus call. Replace `{mrid}` with the `id` from the last step, and replace `CREDENTIALS` with your credentials. Poll the modification status until the response returns `"requestState": "DONE"`.

```
curl -X 'GET \
  'https://cad.onshape.doc/api/v6/drawings/modify/status/{mrid}' \
  -H 'Authorization: Basic CREDENTIALS' \
  -H 'Accept: application/json;charset=UTF-8; qs=0.09' \
```

```
  -H 'Content-Type: application/json;charset=UTF-8; qs=0.09' \
```

6. Open your drawing and confirm that you see the new inspection symbol

Feature List API

Onshape Part Studio Feature Access API

Summary

The Onshape Part Studio tab tracks edits in terms of a feature list. Creation and modification of geometry is performed by manipulating that list. This document describes capabilities that are intended to allow partners and customers to manipulate the feature list from software.

The feature API comprises the following API methods:

- GET /api/partstudios/DWMVE/features - get feature list
- GET /api/partstudios/DWMVE/featurespecs - get feature specs
- POST /api/partstudios/DWE/features - add feature to feature list
- POST /api/partstudios/DWE/features/featureid/:fid - update an existing feature in feature list
- DELETE /api/partstudios/DWE/features/featureid/:fid - delete an existing feature from feature list
- POST /api/partstudios/DWE/features/features/updates - selectively update features in feature list
- POST /api/partstudios/DWE/features/rollback - move the rollback bar
- POST /api/partstudios/DWMVE/featurescript - evaluate featurescript

As well as related configuration API methods:

- GET /api/partstudios/DWMVE/configuration - get part studio configuration
- POST /api/partstudios/DWE/configuration - update part studio configuration

Note that the use of “DWE” occurrences within this document are a shorthand notation for “d/:did/w/:wid/e/:eid” and DWMVE occurrences within this document are a shorthand notation for any of “d/:did/w/:wid/e/:eid”, “d/:did/v/:vid/e/:eid”, “d/:did/m/:mid/e/:eid”. It is assumed that the reader is familiar with these URL path patterns from the API Explorer documentation.

API Methods

The methods described here are also documented at an overview level in the API Explorer within the Part Studios group.

Get Feature List

```
GET /api/partstudios/DWMVE/features - get feature list
```

If you have a part studio element, you can call the get feature list api to find the features that are instantiated within the part studio. The return structure contains the following fields:

- features - A list of user-define features in the part studio.
- defaultFeatures - A list of the default feature (pre-defined) in the part studio
- imports - A list of capabilities that may be referenced by the features. We currently only support a specific predefined set of geometry capabilities.
- featureStates - A list of feature states, one per feature, which describe whether the feature is valid. If a feature has been added to the feature list with an incorrect definition it remains in the feature list.
- isComplete - A boolean indicating whether the features represents the entire part studio (true) or is only a subset (false). The result is a subset if the call to the api specifies a filter on the feature ids
- rollbackIndex - The ordinal position of the rollback bar w.r.t. the list of features. Onshape only executes features that are prior to the rollback bar.
- serializationVersion - A string identifying the version of the structure serialization rules used to encode the output. This is included so that if the output is fed back in and the software has changed incompatibilities can be detected
- sourceMicroversion - The microversion identifier for the document that describes the state from which the result was extracted. This is critical when attempting to use geometry ids that are included in the output, since the interpretation of a geometry id is dependent on the document microversion.
- libraryVersion - An integer indicating the version number for FeatureScript in the Part Studio

Get Feature Specs

```
GET /api/partstudios/DWMVE/featurespecs - get feature specs
```

Returns a list of feature specs that are available within the part studio. A feature spec provides a data description of the interface to a feature. This can, in theory, allow an application to use introspection to allow dynamically generated features. In practice, we expect that the application developer understands the features ahead of time and might utilize the feature spec to understand the options available and the required format for feature addition/modification.

Add Feature

```
POST /api/partstudios/DWE/features
```

A feature can be added to the feature list by calling the add feature API. The API accepts as input a JSON structure containing the fields:

- feature - A single feature definition, in the same format that is output by the get feature list API, except that there is no need to provide feature ids, node ids, or typeNames.
- sourceMicroversion - The microversion of the document that is assumed. Any geometry ids included in the feature are interpreted in the context of this microversion.
- rejectMicroversionSkew (optional) - If set to true, the call will refuse to make the addition if the current microversion for the document does not match the source Microversion. Otherwise, a best-effort attempt is made to re-interpret the feature addition in the context of a newer document microversion.

The call returns a structure with the following fields:

- feature - The input feature, echoed back with id value assignments in place
- featureState - The state of the feature
- serializationVersion - As described previously
- sourceMicroversion - The microversion of the document in which the returned feature is defined
- microversionSkew - If rejectMicroversionSkew was not set to true on input and the document microversion had changed since the input sourceMicroversion, this is set to true to indicate that a re-interpretation was made.
- libraryVersion - An integer indicating the version number for FeatureScript in the Part Studio

The feature is added immediately before the rollback bar. Any geometry ids specified in the feature must be valid at that point in the feature tree. For example, if applying a fillet to an edge, that edge must exist in the feature tree. Filleting the edge will normally make it invalid at future states of the feature tree, since the fillet feature removes the edge.

Update Feature

```
POST /api/partstudios/DWE/features/featureid/:fid
```

An existing feature can be modified by calling the update feature API. This API accepts the same input body format and returns the same output format as the Add Feature API. However, instead of adding a new feature prior to the rollback bar location, it replaces an existing feature in the location of the existing feature.

Update Features

```
POST /api/partstudios/DWE/features/updates
```

Multiple existing features can be modified by calling the update features API. This API accepts a list of features and to update, which must already exist in the part studio. This call does not fully re-define the features but instead, updates only the parameters supplied in the top-level feature structure, and optionally will update feature suppression attributes.

Applications that need to update parameters for multiple features can typically achieve faster model rebuild time by updating multiple features in a single call. It also has the benefit that it allows features to be suppressed or unsuppressed without specifying parameters. This is particularly useful in the unsuppress case because parameters containing query values are not populated when reading from the feature list.

Delete Feature

```
DELETE /api/partstudios/DWE/features/featureid/:fid
```

An existing feature can be removed from the feature list by calling the delete feature API. The API accepts only URL path parameters and return only the following fields:

- serializationVersion - As described earlier
- sourceMicroversion - As described earlier

Move Rollback Bar

```
POST /api/partstudios/DWE/features/rollback
```

The rollback bar can be moved using this API. This is useful if a feature needs to be added at a location other than the current rollback bar location. As input, it accepts the following fields:

- rollbackIndex - The index at which the rollback index should be placed. Features with entry index (0-based) higher than or equal to this value are rolled back. The value must be in the range 0 to the number of entries in the feature list
- serializationVersion
- sourceMicroversion

  The result returned by the API includes the fields

- rollbackIndex - The rollback index in the updated state
- serializationVersion - As described previously
- sourceMicroversion - As described previously
- microversionSkew - As described previously

Evaluate FeatureScript

```
POST api/partstudios/DWMVE/featurescript
```

This API allows the caller to execute a FeatureScript function to query information about the existing feature tree that is not exposed through the other methods described here. As input, it accepts the following fields:

- script - The definition of a FeatureScript function that takes a Context argument and a map argument
- queries - A list of key, value pairs, where the key is a FeatureScript identifier that will appear as a key in the map supplied to the script function and the valid is a list of geometry Id strings, where the list is converted to “query” form form use within the script function.

See the example below to better understand this usage.

Configuration API Methods

Part Studios may be unconfigured or configured. A configured part studio has a list of configuration parameters that define the knobs that allow adjustment of the part studio part content. Configuration parameters are referred to as “inputs” in the UI and may be one of Enum (or List), Boolean, Quantity or String. Each parameter includes a default value setting that is used if the parameter is not otherwise set.

Configured part studios also have a “current” configuration. This is a list of configuration parameter settings for the current representation of the part studio. Alternate configurations of the part studio may be used concurrently, but the current configuration defines the parts seen by the user within in the part studio.

Get Configuration

```
GET /api/partstudios/DWMVE/configuration - get part studio configuration
```

This method reads the current configuration information for a Part Studio. The return structure contains the following fields:

- configurationParameters - A list of the parameters that allow configuration of the part studio.
- currentConfiguration - A list of configuration parameter settings for the current representation of the part studio.
- serializationVersion - As described previously
- sourceMicroversion - As described previously
- microversionSkew - As described previously
- libraryVersion - As described previously

Update Configuration

`POST /api/partstudios/DWE/configuration - update part studio configuration`

This method modifies the current configuration information for a Part Studio. When changing configuration parameters it is important that parameterId values be maintained consistently across changes so that features that reference the parameter do not get broken. The API accepts as input a JSON structure containing the fields:

- configurationParameters - A list of the parameters that allow configuration of the part studio.
- currentConfiguration - A list of configuration parameter settings for the current representation of the part studio.
- sourceMicroversion (optional) - The microversion of the document that is assumed.
- rejectMicroversionSkew (optional) - If set to true, the call will refuse to make the addition if the current microversion for the document does not match the source Microversion.

The call returns a structure with the following fields:

- configurationParameters - A list of the parameters that allow configuration of the part studio.
- currentConfiguration - A list of configuration parameter settings for the current representation of the part studio.
- serializationVersion - As described previously
- sourceMicroversion - As described previously
- microversionSkew - As described previously
- libraryVersion - As described previously

API Usage Details

The feature access and modification API in Onshape presents the internal form of feature definitions rather than providing a translation layer between external form and internal form. We suggest that you familiarize yourself with the formats involved by calling the Get Feature List API on existing part studios in order to understand the encoding of features.

Some important things to know about the JSON encoding are:

- Default values are omitted in the encoded output. For string fields the default value is “”, for boolean fields it is false, and for numeric fields it is 0.
- It uses a special tagging system in order to manage polymorphic data structures. Specifically, objects generally are encoded by enclosing them within another object that declares the type information for the enclosed object.

As an example of the type tagging mechanism, the GET features api might return a structure that looks like this:


```
{
  "features": [
    {
      "type": 134,
      "typeName": "BTMFeature",
      "message": {
        "featureType": "cube",
        "featureId": "FLqo5rpNof3IXgh_0",
        "name": "Cube 1",
        "parameters": [
          {
            "type": 147,
            "typeName": "BTMParameterQuantity",
            "message": {
              "expression": "1.0*in",
              "parameterId": "sideLength",
              "nodeId": "TyUNOSxJ/f9z5t1T"
            }
          }
        ],
        "nodeId": "Mr60Xw6RGWdr0MfYV"
      }
    }
  ],
  "imports": [
    {
      "type": 136,
      "typeName": "BTMImport",
      "message": {
        "path": "onshape/std/geometry.fs",
        "version": "268.0",
        "nodeId": "M2/0RrOwK0Q+hWr9z"
      }
    }
  ],
  "featureStates": [
    {
      "key": "FLqo5rpNof3IXgh_0",
      "value": {
        "type": 1688,
        "typeName": "BTFeatureState",
        "message": {
```

```
        "featureStatus": "OK"
       }
     }
   }
  ],
  "isComplete": true,
  "rollbackIndex": 1,
  "serializationVersion": "1.1.6",
  "sourceMicroversion": "b5b4834bd0674b4489b2b2b7"
}
```


We see that in this case, the features array contains a single feature. The “type” and “typeName” fields describe the type that is encoded within the “message” field. The “type” field provides definitive type information and is the type number assigned to the structure. This type number is a permanent assignment. The “typeName” field is a convenience field that is generated on output and ignored on input. It is intended to allow developers to associate meaningful names with the type numbers.

General Features

In the example above, we see that the single feature is of type 134, which is the general feature type. A general feature is an instantiation of the feature template that is describe by its corresponding feature spec. The BTMFeature structure includes the following fields:

- featureType - Specifies the name of the feature spec that this instantiates
- featureId - The internal identifier of the feature instance within this part studio. It is internally generated when a feature is added.
- nodeId - An internal identifier for the feature node. This is also internally generated when the feature is added.
- name - The user visible name of the feature.
- namespace - An indicator of where the feature definition can be found. Features in the FeatureScript standard library have a namespace value of "" whereas custom features identify the featurestudio that contains the definition. See the Custom Features section below for more information.
- parameters - A list of parameter values for instantiation of the feature spec. Parameters are present for all defined parameters, even if not used in a specific instantiation.

All parameters have the following fields in common:

- parameterId - The name of the parameter spec that this applies to
- nodeId - An internal identifier for the parameter node

Parameters are typically one of the following types:

- BTMParameterQuantity (type 147) - specifies the value for a parameter defined in the feature spec as a BTParameterSpecQuantity. It has the field:
    - expression - An expression defining the value for the parameter.

  An example of its usage is in the depth parameter in the extrude feature.

- BTMParameterQueryList (type 148) - specifies the value for a parameter defined in the feature spec as a BTParameterSpecQuery. It has the field:
    - queries - a list of query objects. The query objects could be either
        - SBTMIndividualQuery objects (type 138) with the field:
            - geometryIds - A list of geometry id values indentifying geometry that the feature applies to.
        - SBTMIndividualSketchRegionQuery objects (type 140) with the field:
            - featureId - The featureId of a sketch, with the query identifying all regions of the sketch

  An example of its usage is in the entities parameter in the extrude feature.

- BTMParameterBoolean (type 144) - specifies the value for a parameter defined in the feature spec as a BTParameterSpecBoolean. It has the field:
    - value - The boolean value

  An example of its usage is in the oppositeDirection parameter in the extrude feature.

- BTMParameterEnum (type 145) - specifies the value for a parameter defined in the feature spec as a BTParameterSpecEnum. It has the fields:
    - enumName - The name of the enum type that value is a member of
    - value - The name of the enum member

  An example of its usage is in the bodyType parameter in the extrude feature.

Other parameter types exist for special cases and are not described here.

Sketches

Although most features are of type BTMFeature, there is also a BTMSketch type, which defines a sketch. The structure of a sketch feature extends the BTMFeature and is relatively complex. It is suggested that the developer use a manually edited sketch as a template for any sketches that they want to create programmatically. However, we will give a high-level overview of the content for the sketch.

Sketches have top-level fields:

- entities - the sketch geometry
- constraints - the geometric relationship constraints and dimensions for the sketch
- parameters - the sketch parameters. The only parameter is the parameter that identifies the sketch plane
- featureType - set to “newSketch”
- featureId - the feature id
- name - the sketch name

The entities describe curves in terms of unbounded curves plus parameterized ranges. The geometry is always specified in meters, regardless of the user’s choice of units. This geometry

provides an initial guess for the sketch. The actual solve state may differ, depending on whether the input constraints are satisfied.

The constraints describe the requirements for sketch solution. These typically include constraints such as COINCIDENT, HORIZONTAL, VERTICAL, PARALLEL, PERPENDICULAR, TANGENT that control geometric positioning and constraints such as DISTANCE, RADIUS, DIAMETER, ANGLE that provide dimensional constraints. The constraints typically have one or two objects that are constrained. These are referenced as localFirst for the first constrained entity, which is within the sketch, and either localSecond or externalSection, depending on whether the second constrained entity is local to the sketch. When entities are local to the sketch they are identitified by entity ids (long unique names) and when external they are identified by a BTMIndiviualQuery.

Feature Specs

The feature spec for a feature provides a description of the inputs that it accepts. As an example, the feature spec for a cube is shown below. It defines a single parameter named “sideLength”. The parameter spec includes a list of ranges that specify valid ranges for the parameter value. Each range is specific to a particular unit and has a defaultValue for that unit. Some parameters described by the feature spec may be optional. These normally have a visibilityCondition that describes a logical test as to whether the parameter should be exposed for editing, based on other parameter values. This provides a fairly reliable way to determine whether the parameter is required for a particular feature instance, but it is safe to include default values for a feature even if it has a visibilityCondition that indicates it is not visible.


```
{
  "type": 129,
  "typeName": "BTFeatureSpec",
  "message": {
    "featureType": "cube",
    "featureTypeName": "Cube",
    "parameters": [
      {
        "type": 173,
        "typeName": "BTParameterSpecQuantity",
        "message": {
          "quantityType": "LENGTH",
          "ranges": [
            {
              "type": 181,
              "typeName": "BTQuantityRange",
              "message": {
                "units": "meter",
                "minValue": 0.00001,
                "maxValue": 500,
                "defaultValue": 0.025,
                "location": {
                  "type": 226,
                  "typeName": "BTLocationInfo",
                  "message": {
```

```
            "document": "onshape/std/primitives.fs",
            "parseNodeId": "PX2DsNKne8oO7ilPS",
            "languageVersion": 268,
            "nodeId": "t8iQqgzAr/bCB2AZ"
          }
        }
      }
    },
    {
      "type": 181,
      "typeName": "BTQuantityRange",
      "message": {
        "units": "centimeter",
        "minValue": 0.001,
        "maxValue": 50000,
        "defaultValue": 2.5,
        "location": {
          "type": 226,
          "typeName": "BTLocationInfo",
          "message": {
            "document": "onshape/std/primitives.fs",
            "parseNodeId": "PKXPCya7aeoNITDFW",
            "languageVersion": 268,
            "nodeId": "69sE3wOlv3FLiJxn"
          }
        }
      }
    },
    {
      "type": 181,
      "typeName": "BTQuantityRange",
      "message": {
        "units": "millimeter",
        "minValue": 0.01,
        "maxValue": 500000,
        "defaultValue": 25,
        "location": {
          "type": 226,
          "typeName": "BTLocationInfo",
          "message": {
            "document": "onshape/std/primitives.fs",
            "parseNodeId": "P8uyYds3YIBBpzlVN",
            "languageVersion": 268,
            "nodeId": "8huKnMQb9mH9B+ef"
          }
        }
      }
    },
    {
      "type": 181,
      "typeName": "BTQuantityRange",
      "message": {
```

```
        "units": "inch",
        "minValue": 0.0003937007874015748,
        "maxValue": 19685.03937007874,
        "defaultValue": 1,
        "location": {
          "type": 226,
          "typeName": "BTLocationInfo",
          "message": {
            "document": "onshape/std/primitives.fs",
            "parseNodeId": "P4MfX8crrr+qc9vFS",
            "languageVersion": 268,
            "nodeId": "BMU3SmZpR83uoKkd"
          }
        }
      }
    },
    {
      "type": 181,
      "typeName": "BTQuantityRange",
      "message": {
        "units": "foot",
        "minValue": 0.00003280839895013123,
        "maxValue": 1640.4199475065616,
        "defaultValue": 0.1,
        "location": {
          "type": 226,
          "typeName": "BTLocationInfo",
          "message": {
            "document": "onshape/std/primitives.fs",
            "parseNodeId": "P60BmwPj7LnRT0CTh",
            "languageVersion": 268,
            "nodeId": "ywVSbQADr8dRv9+s"
          }
        }
      }
    },
    {
      "type": 181,
      "typeName": "BTQuantityRange",
      "message": {
        "units": "yard",
        "minValue": 0.00001093613298337707,
        "maxValue": 546.8066491688538,
        "defaultValue": 0.025,
        "location": {
          "type": 226,
          "typeName": "BTLocationInfo",
          "message": {
            "document": "onshape/std/primitives.fs",
            "parseNodeId": "PsE6sGD66MJVH2xQh",
            "languageVersion": 268,
            "nodeId": "zAy0tYs5aC0clPLL"
```

```
                }
              }
            }
          }
        ],
        "parameterId": "sideLength",
        "parameterName": "Side length",
        "defaultValue": {
          "type": 147,
          "typeName": "BTMParameterQuantity",
          "message": {
            "units": "meter",
            "value": 0.025,
            "parameterId": "sideLength",
            "nodeId": "MC6QUqafapZdxtrDy"
          }
        }
      }
    }
  ]
 }
}
```


Visibility conditions come in 3 variants:

- BTParameterVisibilityAlwaysHidden - Special-case parameters that are not directly shown to the user
- BTParameterVisibilityLogical - Allows a logical condition to express when a parameter is visible. Allows nested AND/OR/NOT expressions. The leaves of the expression tree are BTParameterVisibilityOnEqual
- BTParameterVisibilityOnEqual - A simple test that a parameter has been assigned a specific value.

A ParameterSpec can be one of numerous types. The most commonly used types are:

- BTParameterSpecQuery - indicates that a query parameter should be supplied. The parameter spec has an allowable number of selections and query filter that describes the allowable types that can be selected. See below for additional information about query filters. An example usage is identifying the entities to chamfer in a “Chamfer” feature.
- BTParameterSpecString - indicates that a string value should be supplied. For example, this is used for the name of a variable in a “Variable” feature.
- BTParameterSpecQuantity - indicates that a number value should be supplied. In allows a quantityType, which describes the type of number (length, angle, mass, count, etc.) and range limits on the value. An example usage is the depth parameter in an “Extrude” feature.
- BTParameterSpecEnum - indicates that an enumerated value should should be provided. An example usage is the operationType parameter in an “Extrude” feature.
- BTParameterSpecBoolean - indicates that a boolean true/false value should be specified. An example usage is the offset parameter in the “Boolean” feature

Some less commonly used parameter spec types are:

- BTParameterSpecDerived - indicates that an import of a part from another Part Studio should be specified.
- BTParameterSpecLookupTablePath - provides a list of string values that can be chosen.
- BTParameterSpecForeignId - indicates that a “foreign id” value should be supplied. Foreign ids are currently not generally accessible through the API, but identify a file that is made available throughout the system.

Parameter Spec query filters:

Query Filters are even more numerous than parameter spec types.

Basic selections:

- BTBodyTypeFilter - a specific body type (solid, sheet, wire, acorn, mate_connector)
- BTClosedCurveFilter - a curve that is either closed or not closed, depending on the isClosed value. (NO)
- BTConstructionObjectFilter - an object that either is or is not identified as “construction”, depending on the isConstruction value.
- BTEdgeTopologyFilter - an edge that either is or is not an internal edge, depending on the isInternalEdge value.
- BTEntityTypeFilter - selects a specific type of entity based on the entityType value (vertex, edge, face, body, degenerate_edge).
- BTFeatureTypeFilter - selects a specific type of feature base on the featureType value. (NO)
- BTGeometryFilter - select specific geometry types based on the geometryType value (line, circle, arg, plane, cylinder,cone sphere torus, spline, ellipse).
- BTImageFilter - an entity that either is or is not an image, based on the isImage value. (no)
- BTMateConnectorFilter - a mate connector (NO)
- BTMateFilter - a mate object
- BTPlaneOrientationFilter - excludes planes that are aligned with another plane, based on the normal value.
- BTSketchObjectFilter - an object that either is or is not a sketch object, or is a sketch object created by the user, depending on the objectType value (not_sketch_object, any_sketch_object, user_entity).
- BTTextObjectFilter - an object that either is or is not a text object, depending on the isText value.
- BTTextStrokeFilter - an object that either is or is not a text stroke, depending on the isStroke value.

Logical operations:

- SBTAndFilter - combines query filters with “AND”
- SBTOrFilter - combines query filters with “OR”

- SBTNotFilter - negates query filters

Feature Script Evaluation

For certain tasks when creating and updating features, there may be information needed that is embedded within the existing model. You can often access it by running a Feature Script function. Feature Script is a language that is used to define the behavior of features.

As an example of its usage, consider the case of using a face of an existing part to define new geometry. A plane is defined in FeatureScript as an origin, x-direction vector and normal vector. These can be queried from FeatureScript by using the evPlane function, but it wants a "query" that identifies the face. Here is an example of how you might do this with a call to the Evaluate FeatureScript API call. Suppose you know that "JCC" is the geometry Id of a particular face. The following body can be passed to the evaluate function.

```
{
  "script" : "function (context is Context, queries is map) {
              return evPlane(context, {face:queries.id});
              }",
  "queries" : [ { "key" : "id", "value" : [ "JCC" ] } ]
}
```

Assuming that we have done everything right, the output of this might look something like this:


```
{
  "result": {
    "type": 2062,
    "typeName": "BTFSValueMap",
    "message": {
      "value": [
        {
          "type": 2077,
          "typeName": "BTFSValueMapEntry",
          "message": {
            "key": {
              "type": 1422,
              "typeName": "BTFSValueString",
              "message": {
                "value": "normal"
              }
            },
            "value": {
              "type": 1499,
              "typeName": "BTFSValueArray",
              "message": {
                "value": [
                  {
                    "type": 772,
                    "typeName": "BTFSValueNumber",
                    "message": {
                      "value": -1
                    }
```

```
              },
              {
                "type": 772,
                "typeName": "BTFSValueNumber",
                "message": {}
              },
              {
                "type": 772,
                "typeName": "BTFSValueNumber",
                "message": {}
              }
            ],
            "typeTag": "Vector"
          }
        }
      }
    },
    {
      "type": 2077,
      "typeName": "BTFSValueMapEntry",
      "message": {
        "key": {
          "type": 1422,
          "typeName": "BTFSValueString",
          "message": {
            "value": "origin"
          }
        },
        "value": {
          "type": 1499,
          "typeName": "BTFSValueArray",
          "message": {
            "value": [
              {
                "type": 1817,
                "typeName": "BTFSValueWithUnits",
                "message": {
                  "unitToPower": [
                    {
                      "key": "METER",
                      "value": 1
                    }
                  ],
                  "typeTag": "ValueWithUnits"
                }
              },
              {
                "type": 1817,
                "typeName": "BTFSValueWithUnits",
                "message": {
                  "value": 0.012700000000000001,
                  "unitToPower": [
```

```
                    {
                      "key": "METER",
                      "value": 1
                    }
                  ],
                  "typeTag": "ValueWithUnits"
                }
              },
              {
                "type": 1817,
                "typeName": "BTFSValueWithUnits",
                "message": {
                  "value": 0.012700000000000001,
                  "unitToPower": [
                    {
                      "key": "METER",
                      "value": 1
                    }
                  ],
                  "typeTag": "ValueWithUnits"
                }
              }
            ],
            "typeTag": "Vector"
          }
        }
      }
    },
    {
      "type": 2077,
      "typeName": "BTFSValueMapEntry",
      "message": {
        "key": {
          "type": 1422,
          "typeName": "BTFSValueString",
          "message": {
            "value": "x"
          }
        },
        "value": {
          "type": 1499,
          "typeName": "BTFSValueArray",
          "message": {
            "value": [
              {
                "type": 772,
                "typeName": "BTFSValueNumber",
                "message": {}
              },
              {
                "type": 772,
                "typeName": "BTFSValueNumber",
```

```
                    "message": {
                      "value": -1
                    }
                  },
                  {
                    "type": 772,
                    "typeName": "BTFSValueNumber",
                    "message": {}
                  }
                ],
                "typeTag": "Vector"
              }
            }
          }
        }
      ],
      "typeTag": "Plane"
    }
  },
  "serializationVersion": "1.1.6",
  "sourceMicroversion": "27db48fb62bf6ac3b3ddaeaa",
  "libraryVersion": 296
}
```


You can see from the output that the FeatureScript object representation uses the same typing rules that is used for other output. In addition, FeatureScript objects may have a typeTag field associated with them, which indicates that type-specific behavior should be applied. For instance, we see cases where a BTFSValueArray has a typeTag of “Vector”, which allows the array to be accepted where a Vector is declared to be required.

In order to help describe the interpretation of the structure above, here is a possible JSON representation of the result from the output shown above, but with weaker typing:


```
{
  "normal" : [ -1, 0, 0 ],
  "origin" : { "value" : [ 0,
                           0.012700000000000001,
                           0.012700000000000001 ],
               "units" : "meter" },
  "x" : [ 0, -1, 0 ]
}
```


Custom Features

Custom features can be used in the feature apis with a little additional work. The key to using custom features is that you must set a namespace field in the feature to tell Onshape where to look for the feature defintion. The namespace field identifies a specific version of a feature studio. There are two standard forms for the namespace field to consider:

- Intra-workspace - In this case, the namespace field has the form “e::m” where elementId is the elementId of the FeatureStudio that defines the feature and microversionId is the element microversionId of the FeatureStudio. It is important to note that the element

microversionId is different from the document microversionId that is more typically encountered in API usage.

- External - When the definition to be used lives in a different document or in a specific version of the current document it is referenced using the form “d::v::e::m” This is the same form as for the intra-workspace case but with a documentId and versionId additionally specified. The documentId is the id of the document containing the FeatureStudio and the versionId is the version of the document to be used.

In both of the forms listed, it is necessary to determine the FeatureStudio element microversionId. One way to do this is to call the GET /api/documents/DWMV/elements API, which reports the element microversion for each of the elements. There is also a GET /api/featurestudios/DWMV/featurespecs API that provides a featurespec for each of the features defined in the FeatureStudio. These featurespecs also have a namespace field that reports the namespace of the FeatureStudio in the intra-workspace form.

Examples

Below are several examples of how the API can be used in order to help you get started. The examples are quite trivial, but should provide a basic demonstration of how to use these API methods. You should be able to execute the calls against a part studio of yours and see results immediately. The calls could be executed using your preferred software environment but interactive use in a REST-aware tool is likely the easiest way to try the examples.

Example 1

In our first example we will create a cube using the cube feature. The feature accepts only a single parameter, which is the length of a side, and creates a cube with a corner at the origin and aligned with the three default planes. In running the example, be sure to replace DWE with the the d/:did/w/:wid/e/:eid that is appropriate for the part studio that you are operating against.

POST /api/partstudios/DWE/features

```
{
  "feature" : {
    "type": 134,
    "typeName": "BTMFeature",
      "message": {
        "featureType": "cube",
        "name": "Cube 1",
        "parameters": [
          {
             "type": 147,
             "typeName": "BTMParameterQuantity",
             "message": {
               "expression": "1*in",
               "parameterId": "sideLength"
             }
           }
        ]
      }
```

```
    }
}
```
This returns output similar to the following:


```
{
    "feature": {
        "type": 134,
        "typeName": "BTMFeature",
        "message": {
            "featureType": "cube",
            "featureId": "FuJu9c8PvO5oyTgaV",
            "name": "Cube 1",
            "parameters": [
                {
                    "type": 147,
                    "typeName": "BTMParameterQuantity",
                    "message": {
                        "expression": "1*in",
                        "parameterId": "sideLength",
                        "nodeId": "M+iZRdlIZjnuz8DSS"
                    }
                }
            ],
            "nodeId": "MUdiYYWlCa3arVy8q"
        }
    },
    "featureState": {
        "type": 1688,
        "typeName": "BTFeatureState",
        "message": {
            "featureStatus": "OK"
        }
    },
    "serializationVersion": "1.1.6",
    "sourceMicroversion": "2d31ccc170551a83995b89c8"
}
```

The output returns us the feature definition that we provided as input with nodeIds and a featureId, plus information telling us that that the feature executed correctly and information about the serialization version and microversion of the document that resulted from our feature addition.

Example 2

In our second example we will create a cube where the sideLength parameter is defined by a variable and will then update the variable to have a new value.

Step 1) Create a variable - we create a feature of type assignVariable, which defines a variable and assigns it a value. Here, the variables name is “size” and is displayed in the feature list as “Cube size”, and it is assigned to have the value “1*in” with a variableType of ANY. Once again, be sure to replace DWE with the the d/:did/w/:wid/e/:eid that is appropriate for the part studio that you are operating against.

A variable can be created with variableType set to “ANY”, “LENGTH”, “ANGLE”, or “NUMBER”. This supplies a constraint on the supplied value and each type requires setting the appropriate corresponding parameter, which would be anyValue, lengthValue, angleValue or numberValue, respectively in addition to setting the value parameter. For the intended purpose of using it as a length value it might be more appropriate to use “LENGTH” as the type, which would require setting the lengthValue and value parameters to a length value. We have chosen to demonstrate the use of the “ANY” type here because it can be used in a wider range of applications than a “LENGTH” variable, but does not provide the value type checking that is provided by “LENGTH”, “ANGLE” or “NUMBER” types.

POST /api/partstudios/DWE/features


```
{
  "feature" : {
    "type": 134,
    "typeName": "BTMFeature",
    "message": {
        "featureType": "assignVariable",
        "name": "Cube size",
        "parameters": [
            {
                "type": 145,
                "typeName": "BTMParameterEnum",
                "message": {
                    "enumName": "VariableType",
                    "value": "ANY",
                    "parameterId": "variableType"
                }
            },
            {
                "type": 149,
                "typeName": "BTMParameterString",
                "message": {
                    "value": "size",
                    "parameterId": "name"
                }
            },
            {
                "type": 147,
                "typeName": "BTMParameterQuantity",
                "message": {
                    "expression": "1*in",
                    "parameterId": "anyValue"
                }
            }
        ]
    }
  }
}
```

The returned structure will include a featureId value for the variable. Make note of this value and we will use it in step 3 below.

2. Create the cube feature, referencing the variable created

POST /api/partstudios/DWE/features

```
{
  "feature" : {
    "type": 134,
    "typeName": "BTMFeature",
      "message": {
        "featureType": "cube",
        "name": "Cube 1",
        "parameters": [
          {
            "type": 147,
            "typeName": "BTMParameterQuantity",
            "message": {
              "expression": "#size",
              "parameterId": "sideLength"
            }
          }
        ]
      }
    }
}
```

3. Update the variable to have a new value. We assume here that the featureId value returned in step 1 is “FuJu9c8PvO5oyTgaV” and we will change the cube size from 1 inch to 10 centimeters.

POST /api/partstudios/DWE/features/featureid/FuJu9c8PvO5oyTgaV

```
{
  "feature" : {
    "type": 134,
    "typeName": "BTMFeature",
    "message": {
        "featureType": "assignVariable",
        "name": "Cube size",
        "parameters": [
            {
                "type": 145,
                "typeName": "BTMParameterEnum",
                "message": {
                    "enumName": "VariableType",
                    "value": "ANY",
                    "parameterId": "variableType"
                }
            },
            {
```

```
                "type": 149,
                "typeName": "BTMParameterString",
                "message": {
                    "value": "size",
                    "parameterId": "name"
                }
            },
            {
                "type": 147,
                "typeName": "BTMParameterQuantity",
                "message": {
                    "expression": "10*cm",
                    "parameterId": "anyValue"
                }
            }
        ]
    }
  }
}
```

Take a look at the part studio and you will see the cube at its new size. You can edit the Cube size variable and change the value again.

Example 3

In this example, we create a Sketch and extrude it. This demonstrates relationships betweeen features. First, the sketch must be created on a plane, and for convenience, we will use the pre-defined front plane. Then, the extrude needs to describe what is to be extruded, and for convenience, we use the BTMIndividualSketchRegionQuery.

1. Determine the geometryId for the front plane.

POST /api/features/DWE/featurescript

```
{
  "script" :
    "function(context is Context, queries)
      {
        return transientQueriesToStrings(evaluateQuery(context,
qCreatedBy(makeId(\"Front\"), EntityType.FACE)));
      }",
  "queries" : []
}
```

This will likely return something like this:

```
{
  "result": {
    "type": 1499,
    "typeName": "BTFSValueArray",
    "message": {
      "value": [
        {
```

```
          "type": 1422,
          "typeName": "BTFSValueString",
          "message": {
            "value": "JCC"
          }
        }
      ]
    }
  },
  "serializationVersion": "1.1.6",
  "sourceMicroversion": "a53cabe7d36e30ee100b1d2a",
  "libraryVersion": 298
}
```

This tells us that the geometryId for the face created by the “Front” feature is “JCC”.

2. Determine the geometryId for the Origin

POST /api/features/DWE/featurescript

```
{
  "script" :
    "function(context is Context, queries)
      {
        return transientQueriesToStrings(evaluateQuery(context,
qCreatedBy(makeId(\"Origin\"), EntityType.VERTEX)));
      }",
  "queries" : []
}
```

This will report the geometryId for the origin (probably with the value “IB”) which we will use in the sketch.

3. Create the sketch. This will be just a circle on the front plane with its center at the origin and diameter of 1 inch.

POST /api/partstudios/DWE/features

```
{
  "feature" : {
    "type": 151,
    "typeName": "BTMSketch",
    "message": {
      "entities": [
        {
          "type": 4,
          "typeName": "BTMSketchCurve",
          "message": {
            "geometry": {
              "type": 115,
              "typeName": "BTCurveGeometryCircle",
              "message": {
```

```
              "radius": 0.025400000000000002,
              "xDir": 1,
              "yDir": 0
            }
          },
          "centerId": "clAhDfZz-Dgmb-d0AJ-01Cv-JINJdWZLbVj1.center",
          "entityId": "clAhDfZz-Dgmb-d0AJ-01Cv-JINJdWZLbVj1"
        }
      }
    ],
    "constraints": [
      {
        "type": 2,
        "typeName": "BTMSketchConstraint",
        "message": {
          "constraintType": "COINCIDENT",
          "parameters": [
            {
              "type": 149,
              "typeName": "BTMParameterString",
              "message": {
                "value": "clAhDfZz-Dgmb-d0AJ-01Cv-JINJdWZLbVj1.center",
                "parameterId": "localFirst"
              }
            },
            {
              "type": 148,
              "typeName": "BTMParameterQueryList",
              "message": {
                "queries": [
                  {
                    "type": 138,
                    "typeName": "BTMIndividualQuery",
                    "message": {
                      "geometryIds": [
                        "IB"
                      ]
                    }
                  }
                ],
                "parameterId": "externalSecond"
              }
            }
          ],
          "entityId": "clAhDfZz-Dgmb-d0AJ-01Cv-JINJdWZLbVj1.centerSnap0"
        }
      },
      {
        "type": 2,
        "typeName": "BTMSketchConstraint",
        "message": {
          "constraintType": "DIAMETER",
```

```
          "parameters": [
            {
              "type": 149,
              "typeName": "BTMParameterString",
              "message": {
                "value": "clAhDfZz-Dgmb-d0AJ-01Cv-JINJdWZLbVj1",
                "parameterId": "localFirst"
              }
            },
            {
              "type": 147,
              "typeName": "BTMParameterQuantity",
              "message": {
                "expression": "1*in",
                "parameterId": "length"
              }
            },
            {
              "type": 147,
              "typeName": "BTMParameterQuantity",
              "message": {
                "expression": "0.637419526959446*rad",
                "parameterId": "labelAngle"
              }
            },
            {
              "type": 147,
              "typeName": "BTMParameterQuantity",
              "message": {
                "expression": "1.76201395706607",
                "parameterId": "labelRatio"
              }
            }
          ],
          "entityId": "22b5a2c0-d3ea-4376-969d-8b81944035b2"
        }
      }
    ],
    "featureType": "newSketch",
    "name": "Sketch 1",
    "parameters": [
      {
        "type": 148,
        "typeName": "BTMParameterQueryList",
        "message": {
          "queries": [
            {
              "type": 138,
              "typeName": "BTMIndividualQuery",
              "message": {
                "geometryIds": [
                  "JCC"
```

```
                ]
              }
            }
          ],
          "parameterId": "sketchPlane"
        }
      }
    ]
  }
 }
}
```

4. Create the extrude. Here we assume that the previous call returned the sketch with a feature id of “Fj1THqyY7u36ktGSr”. The extrude uses the BTMIndividualSketchRegionQuery, identifying the sketch feature in order to indicate what is to be extruded. The depth of extrude is 1 inch.

POST /api/partstudios/DWE/features

```
{
  "feature" : {
    "type": 134,
    "typeName": "BTMFeature",
    "message": {
      "featureType": "extrude",
      "name": "Extrude 1",
      "parameters": [
        {
          "type": 145,
          "typeName": "BTMParameterEnum",
          "message": {
            "enumName": "ToolBodyType",
            "value": "SOLID",
            "parameterId": "bodyType"
          }
        },
        {
          "type": 145,
          "typeName": "BTMParameterEnum",
          "message": {
            "enumName": "NewBodyOperationType",
            "value": "NEW",
            "parameterId": "operationType"
          }
        },
        {
          "type": 148,
          "typeName": "BTMParameterQueryList",
          "message": {
            "queries": [
              {
                "type": 140,
```

```
            "typeName": "BTMIndividualSketchRegionQuery",
            "message": {
              "featureId": "Fj1THqyY7u36ktGSr"
            }
          }
        ],
        "parameterId": "entities"
      }
    },
    {
      "type": 145,
      "typeName": "BTMParameterEnum",
      "message": {
        "enumName": "BoundingType",
        "value": "BLIND",
        "parameterId": "endBound"
      }
    },
    {
      "type": 147,
      "typeName": "BTMParameterQuantity",
      "message": {
        "expression": "1*in",
        "parameterId": "depth"
      }
    },
    {
      "type": 148,
      "typeName": "BTMParameterQueryList",
      "message": {
        "parameterId": "surfaceEntities"
      }
    },
    {
      "type": 144,
      "typeName": "BTMParameterBoolean",
      "message": {
        "parameterId": "oppositeDirection"
      }
    },
    {
      "type": 148,
      "typeName": "BTMParameterQueryList",
      "message": {
        "parameterId": "endBoundEntityFace"
      }
    },
    {
      "type": 148,
      "typeName": "BTMParameterQueryList",
      "message": {
        "parameterId": "endBoundEntityBody"
```

```
      }
    },
    {
      "type": 144,
      "typeName": "BTMParameterBoolean",
      "message": {
        "parameterId": "hasDraft"
      }
    },
    {
      "type": 147,
      "typeName": "BTMParameterQuantity",
      "message": {
        "expression": "3.0*deg",
        "parameterId": "draftAngle"
      }
    },
    {
      "type": 144,
      "typeName": "BTMParameterBoolean",
      "message": {
        "parameterId": "draftPullDirection"
      }
    },
    {
      "type": 144,
      "typeName": "BTMParameterBoolean",
      "message": {
        "parameterId": "hasSecondDirection"
      }
    },
    {
      "type": 145,
      "typeName": "BTMParameterEnum",
      "message": {
        "enumName": "SecondDirectionBoundingType",
        "value": "BLIND",
        "parameterId": "secondDirectionBound"
      }
    },
    {
      "type": 144,
      "typeName": "BTMParameterBoolean",
      "message": {
        "value": true,
        "parameterId": "secondDirectionOppositeDirection"
      }
    },
    {
      "type": 148,
      "typeName": "BTMParameterQueryList",
      "message": {
```

```
            "parameterId": "secondDirectionBoundEntityFace"
          }
        },
        {
          "type": 148,
          "typeName": "BTMParameterQueryList",
          "message": {
            "parameterId": "secondDirectionBoundEntityBody"
          }
        },
        {
          "type": 147,
          "typeName": "BTMParameterQuantity",
          "message": {
            "expression": "1.0*in",
            "parameterId": "secondDirectionDepth"
          }
        },
        {
          "type": 144,
          "typeName": "BTMParameterBoolean",
          "message": {
            "parameterId": "hasSecondDirectionDraft"
          }
        },
        {
          "type": 147,
          "typeName": "BTMParameterQuantity",
          "message": {
            "expression": "3.0*deg",
            "parameterId": "secondDirectionDraftAngle"
          }
        },
        {
          "type": 144,
          "typeName": "BTMParameterBoolean",
          "message": {
            "parameterId": "secondDirectionDraftPullDirection"
          }
        },
        {
          "type": 144,
          "typeName": "BTMParameterBoolean",
          "message": {
            "parameterId": "defaultScope"
          }
        },
        {
          "type": 148,
          "typeName": "BTMParameterQueryList",
          "message": {
            "parameterId": "booleanScope"
```

```
          }
        }
      ]
    }
  }
}
```

We see here that there are a lot of parameters that have no values set in them. This is the way features are created by the Onshape web client, buty the unneeded ones could be left out. All of the parameters after “depth” could be omitted without changing the result. However, leaving out a required parameter will normally result in the result reporting a featureStatus of “ERROR”.

Example 4

This example shows an example of configuring a Part Studio with a single Enum (List) input, and a cube feature is created that has its size vary based on the configuration of the Part studio.

1. Create a configuration for the part studio. This example assumes that you are starting with an unconfigured Part Studio. The Enum input is named “Size”, which 2 options available, “Small” and “Large”.

POST /api/partstudios/DWE/configuration

```
{
    "configurationParameters": [
        {
            "type": 105,
            "typeName": "BTMConfigurationParameterEnum",
            "message": {
                "enumName": "Size_conf",
                "options": [
                    {
                        "type": 592,
                        "typeName": "BTMEnumOption",
                        "message": {
                            "option": "Small",
                            "optionName": "Small"
                        }
                    },
                    {
                        "type": 592,
                        "typeName": "BTMEnumOption",
                        "message": {
                            "option": "Large",
                            "optionName": "Large"
                        }
                    }
                ],
                "namespace": "",
                "defaultValue": "Small",
                "parameterId": "Size",
                "parameterName": "Size",
```

```
                "hasUserCode": false,
                "nodeId": "MgfIjLtd/DvuaT/P/"
            }
        }
    ],
    "currentConfiguration": [
        {
            "type": 145,
            "typeName": "BTMParameterEnum",
            "message": {
                "enumName": "Size_conf",
                "value": "Large",
                "namespace": "",
                "parameterId": "Size"
            }
        }
    ]
}
```

2. Create a cube feature in the Part Studio. When the Size configuration input is set to “Small”, the cube will have a sideLength value of “1 in” and when the Size input is set to “Large” it will have a sideLength value of “2 in”.

POST /api/partstudios/DWE/features

```
{
    "feature": {
        "type": 134,
        "typeName": "BTMFeature",
        "message": {
            "featureType": "cube",
            "name": "Cube 1",
            "parameters": [
                {
                    "type": 2222,
                    "typeName": "BTMParameterConfigured",
                    "message": {
                        "configurationParameterId": "Size",
                        "values": [
                            {
                                "type": 1923,
                                "typeName": "BTMConfiguredValueByEnum",
                                "message": {
                                    "namespace": "",
                                    "enumName": "Size_conf",
                                    "enumValue": "Default",
                                    "value": {
                                        "type": 147,
                                        "typeName": "BTMParameterQuantity",
                                        "message": {
                                            "expression": "1 in"
```

```
                                    }
                                }
                            }
                        },
                        {
                            "type": 1923,
                            "typeName": "BTMConfiguredValueByEnum",
                            "message": {
                                "namespace": "",
                                "enumName": "Size_conf",
                                "enumValue": "Large",
                                "value": {
                                    "type": 147,
                                    "typeName": "BTMParameterQuantity",
                                    "message": {
                                        "expression": "2 in"
                                    }
                                }
                            }
                        }
                    ],
                    "parameterId": "sideLength"
                }
            }
        ]
    }
}
```


Metadata

This page describes the APIs Onshape provides for working with document metadata.

Note

This page provides sample code as curls. See the curl documentation for more information.

Note

All Onshape API calls must be properly authenticated by replacing the `CREDENTIALS` variable in the curls below. See the API Keys page for instructions and the Quick Start for an example. All applications submitted to the Onshape App Store *must* authenticate with OAuth2.

Note

For Enterprise accounts, replace **cad** in all Onshape URLs with your company domain.
https://**cad**.onshape.com > https://**companyName**.onshape.com

Endpoints

Get Metadata

getWVMetadata: Get metadata for a workspace or version.

```
curl -X 'GET' \
'https://cad.onshape.com/api/v6/metadata/d/{did}/wv/{wvid}?inferMetadataOwner
=false&depth=1&includeComputedProperties=true&includeComputedAssemblyProperti
es=false&thumbnail=false' \
      -H 'accept: application/json;charset=UTF-8; qs=0.09' \
      -H 'Authorization: Basic CREDENTIALS'\
      -H 'Content-Type: application/json;charset=UTF-8; qs=0.09'
```

getWMVEsMetadata: Get metadata for all elements in a document.

```
curl -X 'GET' \
'https://cad.onshape.com/api/v6/metadata/d/{did}/wv/{wvid}/e?inferMetadataOwn
er=false&depth=1&includeComputedProperties=true&includeComputedAssemblyProper
ties=false&thumbnail=false' \
      -H 'accept: application/json;charset=UTF-8; qs=0.09' \
      -H 'Authorization: Basic CREDENTIALS' \
      -H 'Content-Type: application/json;charset=UTF-8; qs=0.09'
```

getWMVEMetadata: Get metadata for an element.

```
curl -X 'GET' \
'https://cad.onshape.com/api/v6/metadata/d/{did}/wv/{wvid}/e/{eid}?inferMetad
ataOwner=false&depth=1&includeComputedProperties=true&includeComputedAssembly
Properties=false&thumbnail=false' \
      -H 'accept: application/json;charset=UTF-8; qs=0.09' \
      -H 'Authorization: CREDENTIALS' \
      -H 'Content-Type: application/json;charset=UTF-8; qs=0.09'
```

getWMVEPMetadata: Get metadata for a part.

```
curl -X 'GET' \
'https://cad.onshape.com/api/v6/metadata/d/{did}/wv/{wvid}/e/{eid}/p/{pid}?ro
llbackBarIndex=-
1&inferMetadataOwner=false&includeComputedProperties=true&includeComputedAsse
mblyProperties=false&thumbnail=false' \
      -H 'accept: application/json;charset=UTF-8; qs=0.09' \
      -H 'Authorization: Basic CREDENTIALS'\
      -H 'Content-Type: application/json;charset=UTF-8; qs=0.09'
```

getWMVEPsMetadata: Get metadata for all parts in a document.

```
curl -X 'GET' \
'https://cad.onshape.com/api/v6/metadata/d/{did}/wv/{wvid}/e/{eid}/p?inferMet
adataOwner=false&includeComputedProperties=true&includeComputedAssemblyProper
ties=false&thumbnail=false' \
      -H 'accept: application/json;charset=UTF-8; qs=0.09' \
      -H 'Authorization: Basic CREDENTIALS'\
      -H 'Content-Type: application/json;charset=UTF-8; qs=0.09'
```

getVEOPStandardContentMetadata: Get metadata for a standard content part.

`{linkedDocumentId}`: ID of the document into which the standard content part is inserted.

You can call getAssemblyDefinition to get the other values needed for the call:

`{did}`: ID of the document in which the standard content part lives.

`{vid}`: ID of the version in which the standard content part lives.

`{eid}`: ID of the element tab in which the standard content part lives.

`{pid}`: Part ID of the standard content part.

`{config}`: Encoded configuration string.

```
curl -X 'GET' \
'https://cad.onshape.com/api/metadata/standardcontent/d/{did}/v/{vid}/e/{eid}
/p/{pid}?configuration={config}&linkDocumentId={linkDocument}' \
      -H 'accept: application/json;charset=UTF-8; qs=0.09' \
      -H 'Authorization: Basic CREDENTIALS' \
      -H 'Content-Type: application/json;charset=UTF-8; qs=0.09'
```

Update Metadata

To update metadata, you send a JSON in the API request body. This JSON block must include a `jsonType` value and a `properties` object array. Each object in the `properties` array includes a `propertyId` and the metadata key/value pairs.

updateWVMetadata: Update workspace or version metadata.

```
curl -X 'POST' \
      'https://cad.onshape.com/api/v6/metadata/d/{did}/wv/{wvid}' \
      -H 'accept: application/json;charset=UTF-8; qs=0.09' \
      -H 'Authorization: Basic CREDENTIALS' \
      -H 'Content-Type: application/json;charset=UTF-8; qs=0.09' \
      -d '{
            "jsonType": ("metadata-workspace" | "metadata-version") ,
            "properties": [
            {
            "propertyId": "propertyId1",
            "key1": "value",
            "key2": "value
```

```
},
{
"propertyId": "propertyId2",
"key1": "value",
"key2": "value
}
]
}'
```

updateWVEMetadata: Update element metadata.

```
curl -X 'POST' \
      'https://cad.onshape.com/api/v6/metadata/d/{did}/wv/{wvid}/e/{eid}' \
      -H 'accept: application/json;charset=UTF-8; qs=0.09' \
      -H 'Authorization: Basic CREDENTIALS' \
      -H 'Content-Type: application/json;charset=UTF-8; qs=0.09' \
      -d '{
            "jsonType": "metadata-element",
            "properties": [
            {
            "propertyId": "propertyId",
            "key1": "value",
            "key2": "value
            }
            ]
            }'
```

updateWVEPMetadata: Update part metadata

```
curl -X 'POST' \
'https://cad.onshape.com/api/v6/metadata/d/{did}/wv/{wvid}/e/{eid}/{iden}/{pi
d}' \
      -H 'accept: application/json;charset=UTF-8; qs=0.09' \
      -H 'Authorization: Basic CREDENTIALS' \
      -H 'Content-Type: application/json;charset=UTF-8; qs=0.09' \
      -d '{
            "jsonType": "metadata-part",
            "properties": [
            {
                 "propertyId": "propertyId",
                 "key1": "value",
                 "key2": "value
            }
            ]
            }'
```

updateVEOPStandardContentPartMetadata: Update standard content part metadata.

`{linkedDocumentId}`: ID of the document into which the standard content part is inserted.

You can call getAssemblyDefinition to get the other values needed for the call:

`{did}`, `{vid}`, `{eid}`: IDs of the document, version, and element in which the standard content part lives.

`{companyId}`: ID of the company that owns the standard content part. All metadata changes to this standard content part will populate for the entire company.

`{pid}`: Part ID of the standard content part.

`{config}`: Encoded configuration string.

For each `items.properties` object, include a unique `propertyId` and at least one key/value metadata pair.

Updates made to standard content are global for all users and documents within the company.

```
curl -X 'POST' \
'https://cad.onshape.com/api/v6/metadata/standardcontent/d/{did}?linkDocument
Id={linkDocumentId}' \
      -H 'accept: application/json;charset=UTF-8; qs=0.09' \
      -H 'Authorization: Basic CREDENTIALS' \
      -H 'Content-Type: application/json;charset=UTF-8; qs=0.09' \
      -d '{
            "items": [
            {
            "href":
            "https://cad.onshape.com/api/metadata/standardcontent/d/did/v/v
            id/e/eid/c/companyId/p/pid?configuration=config&linkDocumentId=
            linkDocumentId&applyToAllConfigs=true",
            "properties": [
                {
                      "key": "value",
                      "propertyId": "propertyId1"
                },
                {
                      "key": "value",
                      "propertyId": "propertyId2"
                }
            ]
            }
            ]
            }'
```

Sample Workflows

Get metadata for an element

We will get the metadata from the `DRILL_BIT` element in this public document.

Call the getWMVEMetadata endpoint. Replace `CREDENTIALS` with your authentication credentials:

```
curl -X 'GET' \
'https://cad.onshape.com/api/v6/metadata/d/e60c4803eaf2ac8be492c18e/w/d2558da
712764516cc9fec62/e/958bceb5a2511b572dbbe851?inferMetadataOwner=false&depth=1
&includeComputedProperties=true&includeComputedAssemblyProperties=false&thumb
nail=false' \
      -H 'accept: application/json;charset=UTF-8; qs=0.09' \
      -H 'Authorization: Basic CREDENTIALS' \
      -H 'Content-Type: application/json;charset=UTF-8; qs=0.09'
```

In the response body, confirm that for `properties.name=Name`, `properties.value=DRILL_BIT`.

```
{
      "jsonType": "metadata-element",
      "elementType": 0,
      "mimeType": "onshape/partstudio",
      "elementId": "958bceb5a2511b572dbbe851",
      "properties": [
            {
            "name": "Name",
            "value": "DRILL_BIT",
            "defaultValue": null,
            "computedPropertyError": null,
            "propertySource": 0,
            ...
```

Update metadata for a part

Make a copy of this public document. Make a note of the new document's document ID, workspace ID, and element ID.

Call the Part/getPartsWMVE API on your copied document. to get a list of part IDs in the element. Only one part exists in the document, with a part ID of `JHD`.

```
{
      "name": "Main",
      ...
      "microversionId": "{mid}",
      "partNumber": null,
      "elementId": "{eid}",
      "partId": "JHD",
      "bodyType": "sheet",
      ...
}
```

Next, we need to get the ID of the property we want to update. We'll call the getWMVEPMetadata endpoint to get the current metadata JSON for the part. Don't forget to replace the URL parameters with the IDs from your copied document, and replace `CREDENTIALS` with your authorization credentials.

```
curl -X 'GET' \
```

```
'https://cad.onshape.com/api/v6/metadata/d/{did}/w/{wid}/e/{eid}/p/JHD?rollbackBarIndex=-1&inferMetadataOwner=false&includeComputedProperties=true&includeComputedAssemblyProperties=false&thumbnail=false' \
      -H 'accept: application/json;charset=UTF-8; qs=0.09' \
      -H 'Authorization: Basic CREDENTIALS' \
      -H 'Content-Type: application/json;charset=UTF-8; qs=0.09'
```

The call returns a response body in JSON format. Scroll to the `Description` properties block of the JSON response, and notice that the `value` field is an empty string.

```
...
{
      "name": "Description",
      "value": "",
      "defaultValue": null,
      "computedPropertyError": null,
      "propertySource": 3,
      "validator": {
            "min": null,
            "max": null,
            "minLength": null,
            "maxLength": 10000,
            "pattern": null,
            "quantityType": null,
            "maxCount": null,
            "minCount": null,
            "minDate": null,
            "maxDate": null
      },
      "required": false,
      "editable": true,
      "propertyId": "57f3fb8efa3416c06701d60e",
      "editableInUi": true,
      "dateFormat": null,
      "valueType": "STRING",
      "enumValues": null,
      "schemaId": "5877a03ebe4c21163b49dce0",
      "uiHints": {
            "multiline": true
      },
      "multivalued": false,
      "computedAssemblyProperty": false,
      "computedProperty": false,
      "propertyOverrideStatus": 0
},
...
```

Copy Description's `propertyId`. We'll need this ID to update the metadata.

Set up the updateWVEPMetadata call.

```
curl -X 'POST' \
'https://cad.onshape.com/api/v6/metadata/d/{did}/w/{wid}/e/{eid}/p/JHD?rollba
ckBarIndex=-1' \
      -H 'accept: application/json;charset=UTF-8; qs=0.09' \
      -H 'Authorization: Basic CREDENTIALS' \
      -H 'Content-Type: application/json;charset=UTF-8; qs=0.09' \
```

Add the request JSON to the call. Note that we need to include the `jsonType`, the `partId`, the `propertyId`, and the value to update.

```
-d '{
      "jsonType": "metadata-part",
      "partId": "JHD",
      "properties": [
             {
             "value": "",
             "propertyId": "57f3fb8efa3416c06701d60e"
             }
             ]
      }'
```

In the request JSON, change the empty Description `value` string to `"Drill bit"`:

```
-d '{
      "jsonType": "metadata-part",
      "partId": "JHD",
      "properties": [
             {
             "value": "Drill bit",
             "propertyId": "57f3fb8efa3416c06701d60e"
             }
             ]
      }'
```

Make the final call. Don't forget to replace the URL parameters and `CREDENTIALS` with your information.

```
curl -X 'POST' \
'https://cad.onshape.com/api/v6/metadata/d/{did}/w/{wid}/e/{wid}/p/JHD?rollba
ckBarIndex=-1' \
-H 'accept: application/json;charset=UTF-8; qs=0.09' \
-H 'Authorization: Basic CREDENTIALS' \
-H 'Content-Type: application/json;charset=UTF-8; qs=0.09' \
-d '{
      "jsonType": "metadata-part",
      "partId": "JHD",
      "properties": [
             {
             "value": "Drill bit",
             "propertyId": "57f3fb8efa3416c06701d60e"
             }
```

```
            ]
}'
```

Repeat steps 3 and 4 to confirm that the Description `value` for the part is now `Drill bit`.

Additional Resources

API Explorer: Metadata

API Guide: API Explorer

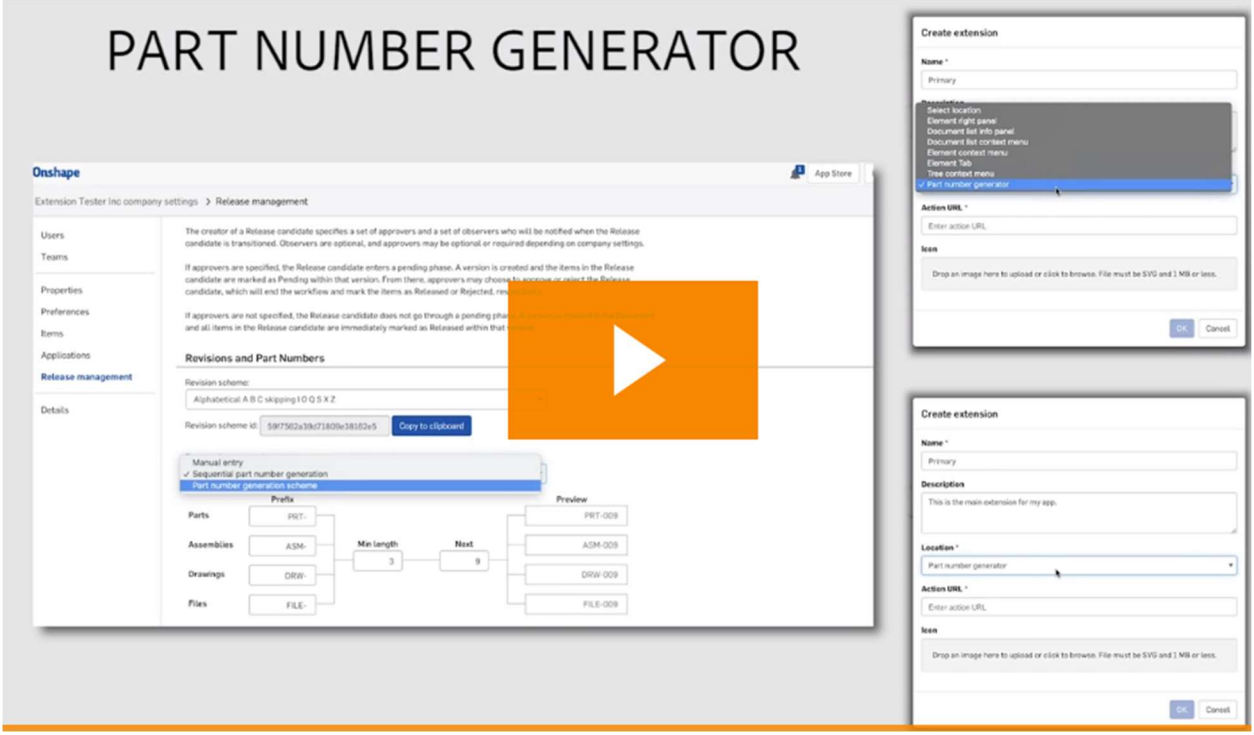
Extensions

This page provides information for some of the more common options for embedding a third-party application into the Onshape interface. Onshape provides many options for embedding commands in various menus, fly-out panels, and elements. In this example, you will embed a custom web page inside a document's right side fly-out panel. This interface will receive information from Onshape and push information from the panel back to Onshape, providing a complete, bi-directional integration scenario.

Please see also:

- Create an Extension tutorial: A step-by-step walkthrough of creating an extension.

- Create an Extension Video
  Sample:



Extension Types

We can classify extensions into two high-level types. The first type embeds a UI *from* the application *into* the Onshape UI. The embedded UI is an HTTPS page displayed in an iFrame in the Onshape UI. The UI is served from the application, and can choose to make API calls to Onshape for additional information. This is exactly like the traditional tab-based applications in Onshape, except that such extensions exist at different UI locations.

The second type of extension embeds an action that calls a REST API *exposed by* the application *from* the Onshape UI (e.g., context menu actions and toolbar actions). These types of extensions rely on External OAuth information to authenticate and make a call where Onshape acts as a client, and the application acts as a server.

Extension Attributes

Each extension exists at a specific place in the Onshape UI and works with a specific context or selection. The attributes of an extension are:

1. **Name**: This should be short and explicit. It will appear in the Onshape UI as a menu item, a toolbar tooltip, a collapsed panel icon tooltip, or a panel icon. It might be truncated in the UI if it is too long.
2. **Description**: This is where the developer can record a detailed description of the extension. It does not appear in the Onshape UI, but could appear in the grant process.

3. **Location**: This describes where the extension exists in the Onshape UI. Over time, this will cover various panels in the UI, context menus, toolbars, actions in dialogs, etc. You can see the list of currently supported locations here.
    - Please note that you can create only one element tab extension per application.
4. **Context** (selection): Some locations will work in the context of a selection. Let's say the application developer wants to show some information from a third-party system, pertinent only to parts (not assemblies or drawings). In this scenario, the developer would choose a location like 'Document list info panel', and the context as 'Selected part'. If the user searches for something in the document list, some documents, Part Studios, parts, and Assemblies would be returned. This extension will show up in the Info panel only if the selected entity is a part. Using context enables application developers to control when the extension is displayed. Check the list of contexts available for different locations here.
5. **Action URL**: Locations that embed a UI use the action URL to define the address of the page to display. The action URL is used to specify the REST endpoint if the location is an action (context menu, toolbar item, action in dialog etc.) and the action type is GET or POST. If the location is an action and the action is 'Open in new window', the action URL is the URL to open in the new window.
   The action URL can be parameterized to pass information from Onshape to the application. The action URL replaces attributes in the format {$attribute} with the appropriate value. These attributes can be used to identify the selected entity and/or make calls back to Onshape via the API. The currently supported attributes are:
    - `{$documentId}` - The Onshape ID for the current or selected document.
    - `{$workspaceOrVersion}` - This will be either `w` or `v` for workspace or version respectively depending on current opened document state or selection.
    - `{$workspaceOrVersionId}` - The Onshape ID for the current or selected workspace or version.
    - `{$workspaceId}` - Use `{$workspaceOrVersionId}` instead.
    - `{$versionId}` - Use `{$workspaceOrVersionId}` instead.
    - `{$microversionId}` - The Onshape ID for the current or select document microversion.
    - `{$elementId}` - The Onshape ID for the current or selected element (part studio, assembly, drawing).
    - `{$partId}` - The Onshape ID for the current or selected part.
    - `{$partNumber}` - The Part number property for the current or selected part, assembly or drawing.
    - `{$revision}` - The Revision property for the current or selected part, assembly or drawing.
    - `{$companyId}` - The ID for the company that owns the document.
    - `{$mimeType}` - The mime type if the current or selected element is a blob.
    - `{$featureId}` - In case of feature selected in the Feature list in a Part Studio.
    - `{$nodeId}` - In case of mate or mate feature selected in the Assembly list.
    - `{$occurrencePath}` - In case of part instances, mates, mate connectors and sub assemblies.
    - `{$configuration}` - In case of extensions inside the document, this attribute will be replaced by current element active configuration.

The attributes can exist as path parameters or query parameters or attributes in the POST body. For example:

```
[https://whispering-sea-
42267.herokuapp.com/oauthSignin?documentId={$documentId}&workspaceOrVersion={work
spaceOrVersion}&workspaceOrVersionId={$workspaceOrVersionId}&elementId={$elementI
d}&partId={$partId}&server=https://cad.onshape.com&companyId=cad&userId=5f1eba76c
14a434817d9c588&locale=en-US](https://whispering-sea-
42267.herokuapp.com/oauthSignin?documentId=%7B$documentId%7D&workspaceId=%7B$work
spaceId%7D&elementId=%7B$elementId%7D&partId=%7B$partId)
```
or

```
[https://cad.onshape.com/api/partstudios/d/{$documentId}/{$workspaceOrVersion}/{$
workspaceOrVersionId}/e/{$elementId}/stl?server=https://cad.onshape.com&companyId
=cad&userId=5f1eba76c14a434817d9c588&locale=en-
US](https://cad.onshape.com/api/partstudios/d/%7B$documentId%7D/w/%7B$workspaceId
%7D/e/%7B$elementId%7D/stl)
```
The attributes available for replacement differ by location and context selection. You can see the available attributes for each location here.

The **timeout** for `action_url` of type `GET` or `POST` is **180 seconds**.

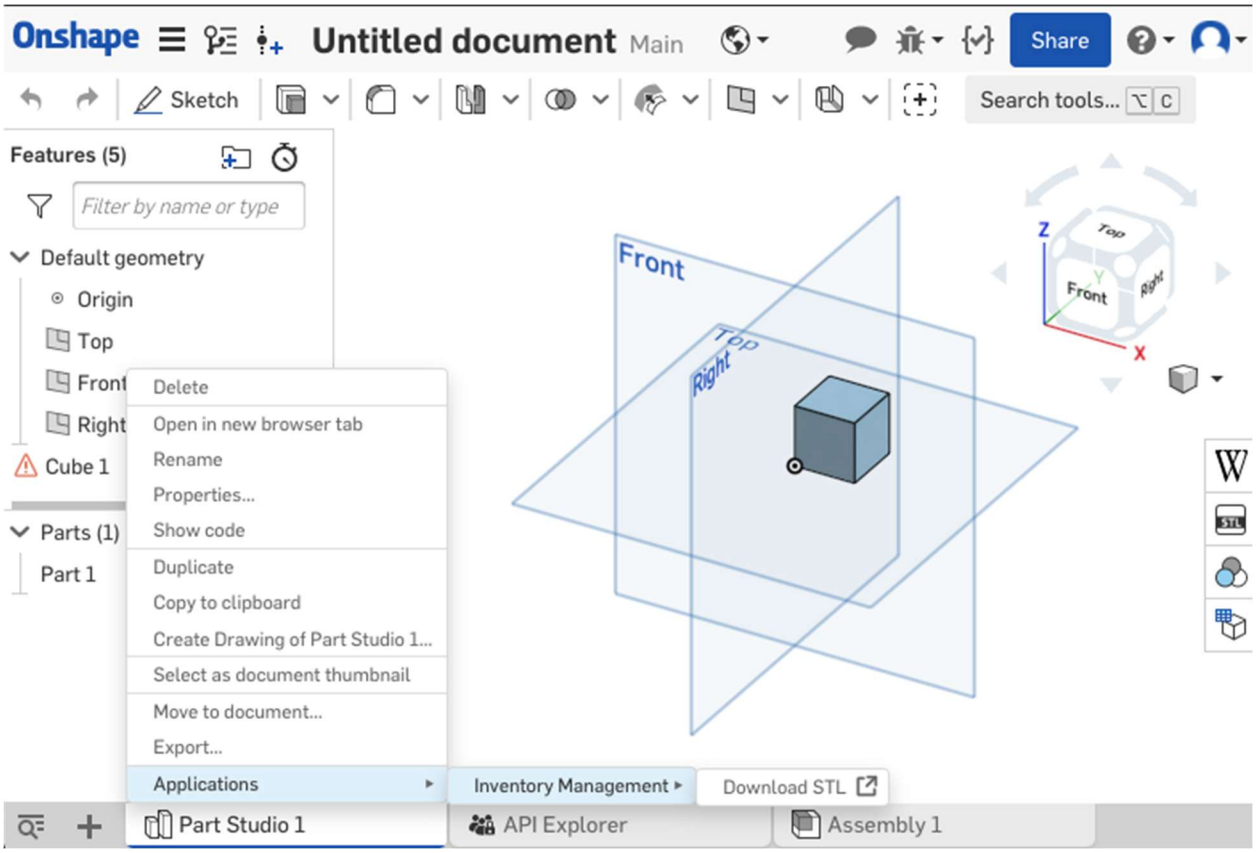
6. **Action type**: The action type is only applicable for locations that act as actions and not for locations that embed UIs. Check if action type is valid for a location here. The supported action types are:
    - `GET` - This makes a GET API call using the action URL. Parameter replacement is done on the action URL.
    - `POST` - This makes a POST API call using the action URL and the action body as the post body. Parameter replacement is done on both the action URL and the action body.
    - `Open in new window` - This opens the action URL in a new browser window. Parameter replacement is done on the action URL.
7. **Action body**: This is only applicable if the action type is POST. The action body is passed in a POST API call and must be in a valid json format.
8. **Show response**: This is only applicable if the action type is GET or POST. If this is checked, the UI will wait for a response and show the response in a dialog in the UI. The response must be in a valid json format.
9. **Icon**: The icon will be shown where the extension exists. This can be an icon in an Info panel, context menu action, toolbar button, action button in a dialog, or other supported locations.

Supported Locations and Contexts

This is the list of supported locations, their valid contexts, and whether they support action types.

Element context menu

This is the context menu for elements.



Supported contexts:

- `Part Studio`
- `Assembly`
- `Drawing`
- `Blob element`

Supported parameters for replacements:

- `{$documentId}`
- `{$workspaceOrVersion}`
- `{$workspaceOrVersionId}`
- `{$workspaceId}` DEPRECATED
- `{$versionId}` DEPRECATED
- `{$elementId}`
- `{$partNumber}`
- `{$mimeType}`
- `{$configuration}`

Default parameters as query string:

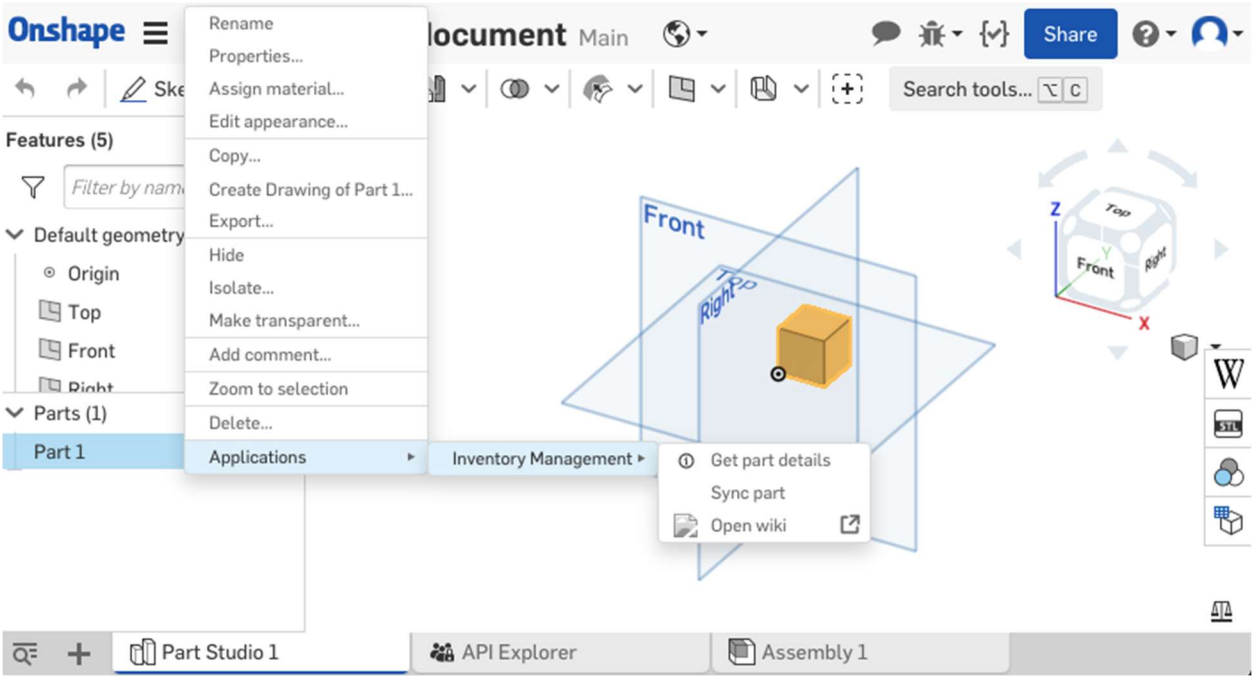
- `server`
- `companyId` - Default value is ‘cad’. If the document owner is company/enterprise, then the value is company/enterprise ID.

- `userId`
- `locale`
- `clientId`

This location supports action types.

Tree context menu

This is the context menu for the part tree, assembly tree and feature tree in part studios.



Supported contexts:

- `Part`
- `Sub assembly`
- `Feature`
- `Mate`
- `Mate feature`
- `Instance`

Supported parameters for replacement:

- `{$documentId}`
- `{$workspaceOrVersion}`
- `{$workspaceOrVersionId}`
- `{$workspaceId}` DEPRECATED
- `{$versionId}` DEPRECATED
- `{$elementId}`
- `{$partNumber}`
- `{$revision}`

- `{$featureId}`
- `{$nodeId}`
- `{$occurrencePath}`
- `{$configuration}`

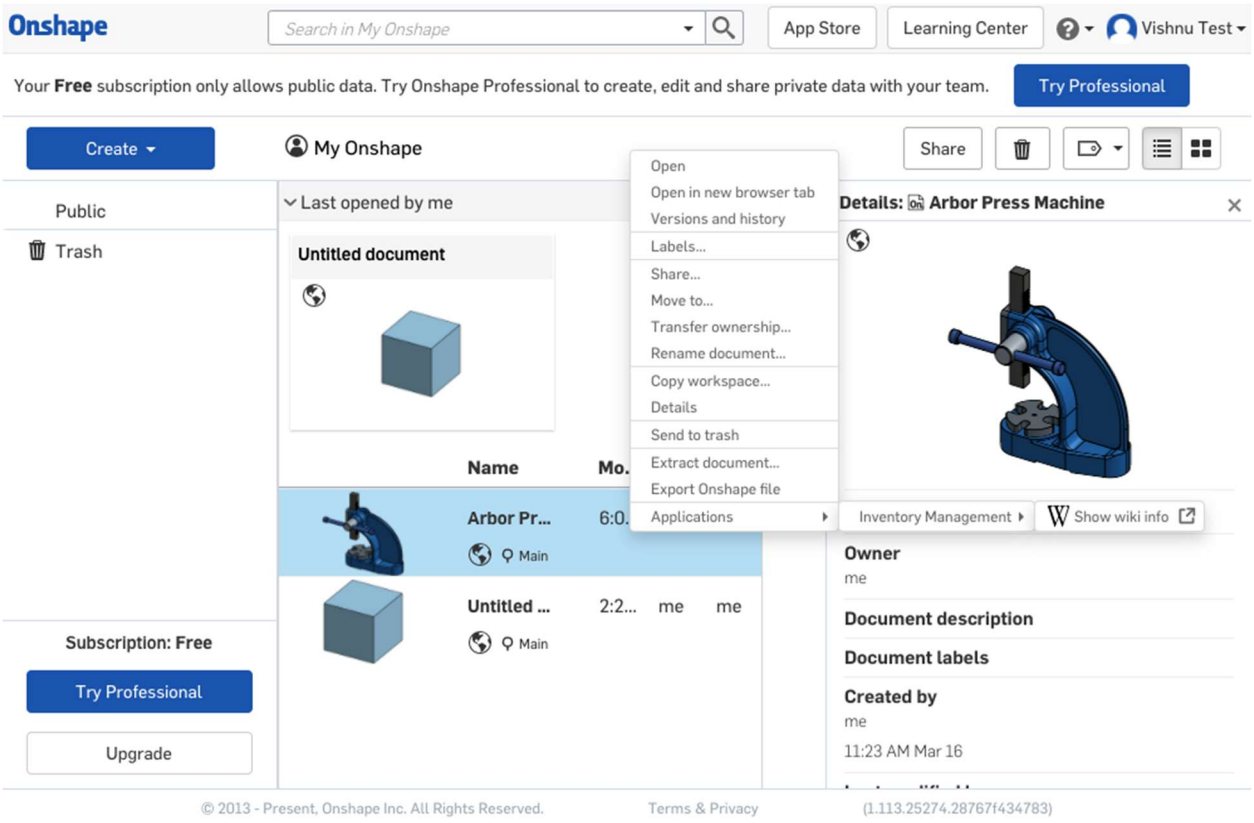
Default parameters as query string:

- `server`
- `companyId` - Default value is ‘cad’. If the document owner is company/enterprise, then the value is company/enterprise ID.
- `userId`
- `locale`
- `clientId`

This location supports action types.

Document list context menu

This the context menu available on items in the document list. This is normally documents but can be multiple types based on search results.



Supported contexts:

- `Part`
- `Document`

- Part Studio
- Assembly
- Drawing
- Blob element

Supported parameters for replacement:

- {$documentId}
- {$workspaceOrVersion}
- {$workspaceOrVersionId}
- {$workspaceId} DEPRECATED
- {$versionId} DEPRECATED
- {$elementId}
- {$partNumber}
- {$revision}
- {$configuration}

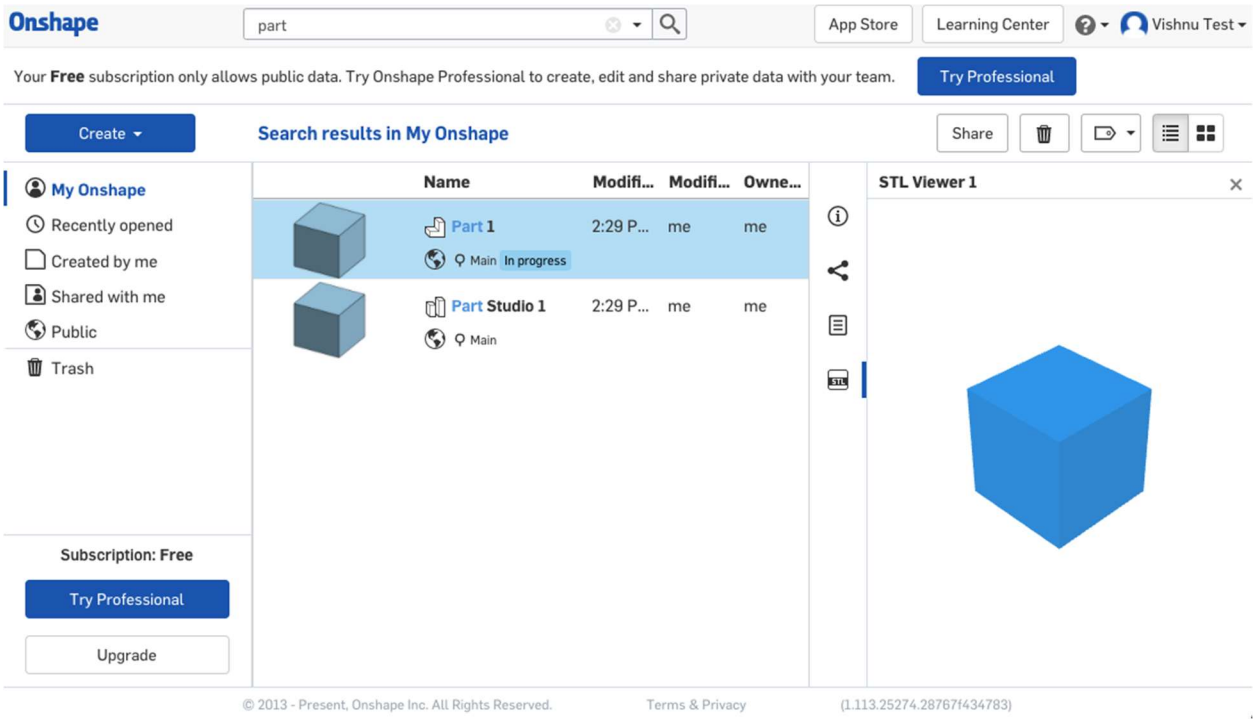
Default parameters as query string:

- server
- companyId - Default value is 'cad'. If the document owner is company/enterprise, then the value is company/enterprise ID.
- userId
- locale
- clientId

This location supports action types.

Document list info panel

This is the Info panel to the right in the document list. The document list normally contains documents, but can contain other entities as the result of a search.



Supported contexts:

- `Part`
- `Document`
- `Part Studio`
- `Assembly`
- `Drawing`
- `Blob element`

Supported parameters for replacement:

- `{$documentId}`
- `{$workspaceOrVersion}`
- `{$workspaceOrVersionId}`
- `{$workspaceId}` DEPRECATED
- `{$versionId}` DEPRECATED
- `{$elementId}`
- `{$partId}`

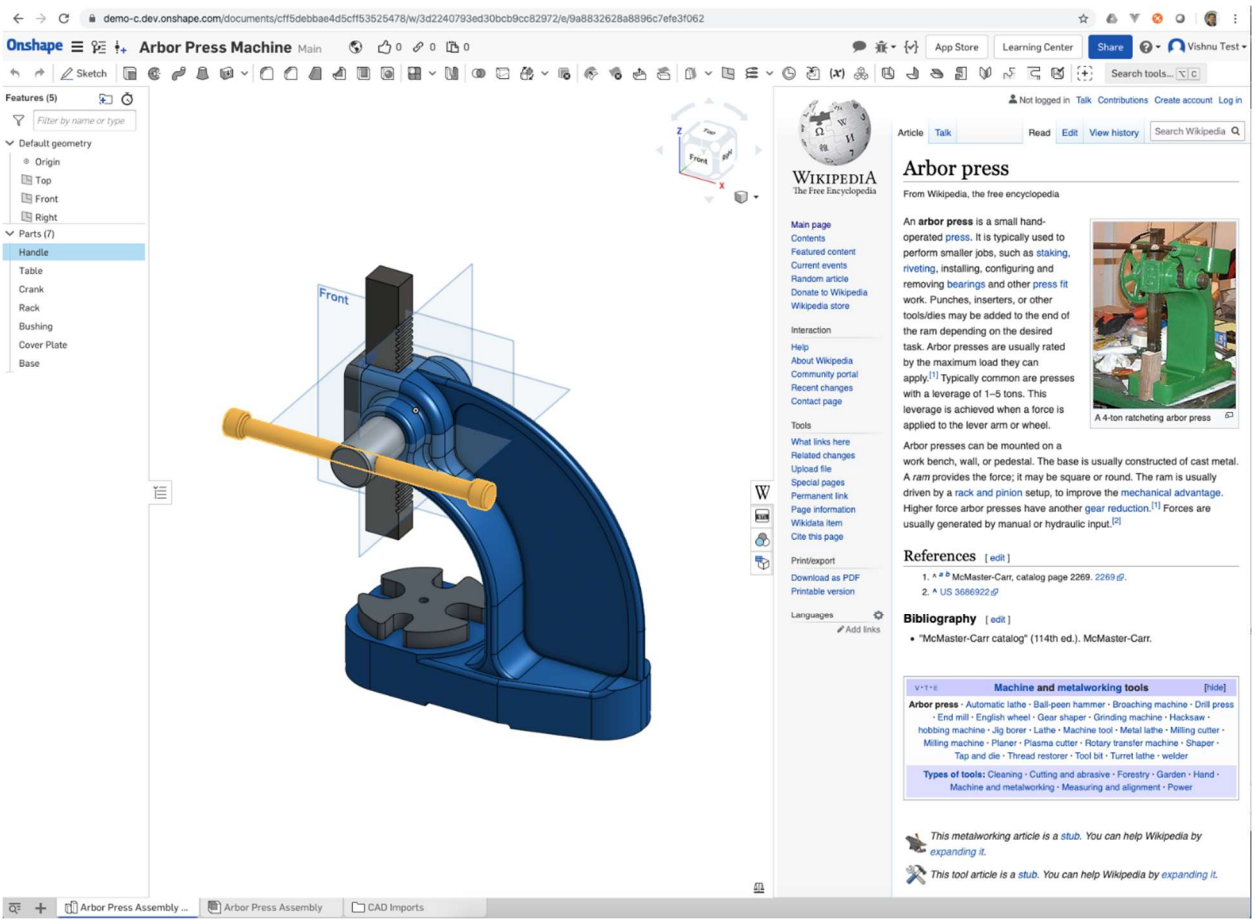
Default parameters as query string:

- `server`
- `companyId` - Default value is ‘cad’. If the document owner is company/enterprise, then the value is company/enterprise ID.
- `userId`
- `locale`
- `clientId`

This location does NOT support action types.

Element right panel

This is the panel inside a document. It currently houses the BOM, configurations, etc. Applications can use this extension location to add items in this panel.



Supported contexts:

- `Part`
- `Document`
- `Part Studio`
- `Assembly`
- `Sub assembly`
- `Feature`
- `Mate`
- `Mate feature`

Supported parameters for replacement:

- `{$documentId}`
- `{$workspaceOrVersion}`

- {$workspaceOrVersionId}
- {$workspaceId} DEPRECATED
- {$versionId} DEPRECATED
- {$elementId}
- {$partNumber}
- {$revision}
- {$featureId}
- {$nodeId}
- {$occurrencePath}
- {$configuration}

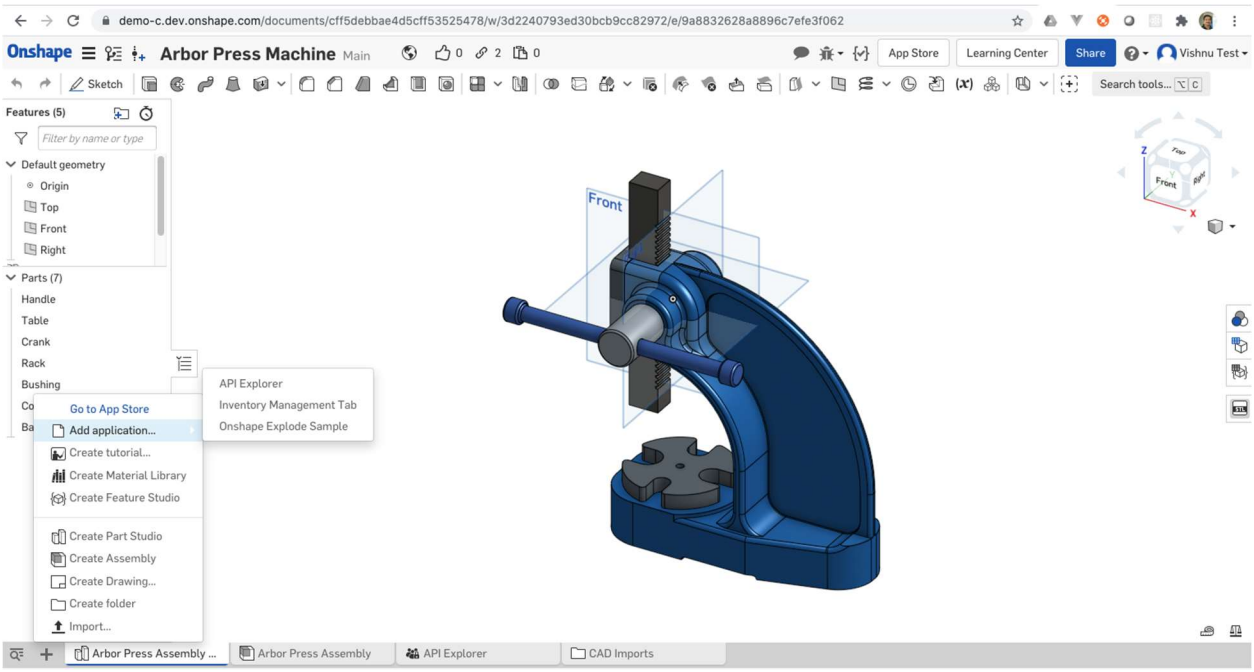
Default parameters as query string:

- server
- companyId - Default value is 'cad'. If the document owner is company/enterprise, then the value is company/enterprise ID.
- userId
- locale
- clientId

This location does NOT support action types.

New Element tab

This is the menu option for + menu -> Add application inside a document . After menu click, a new tab will be created with the action url associated with this extension.



Supported contexts:

- There are no supported contexts.

Supported parameters for replacement:

- Parameter replacement not supported.

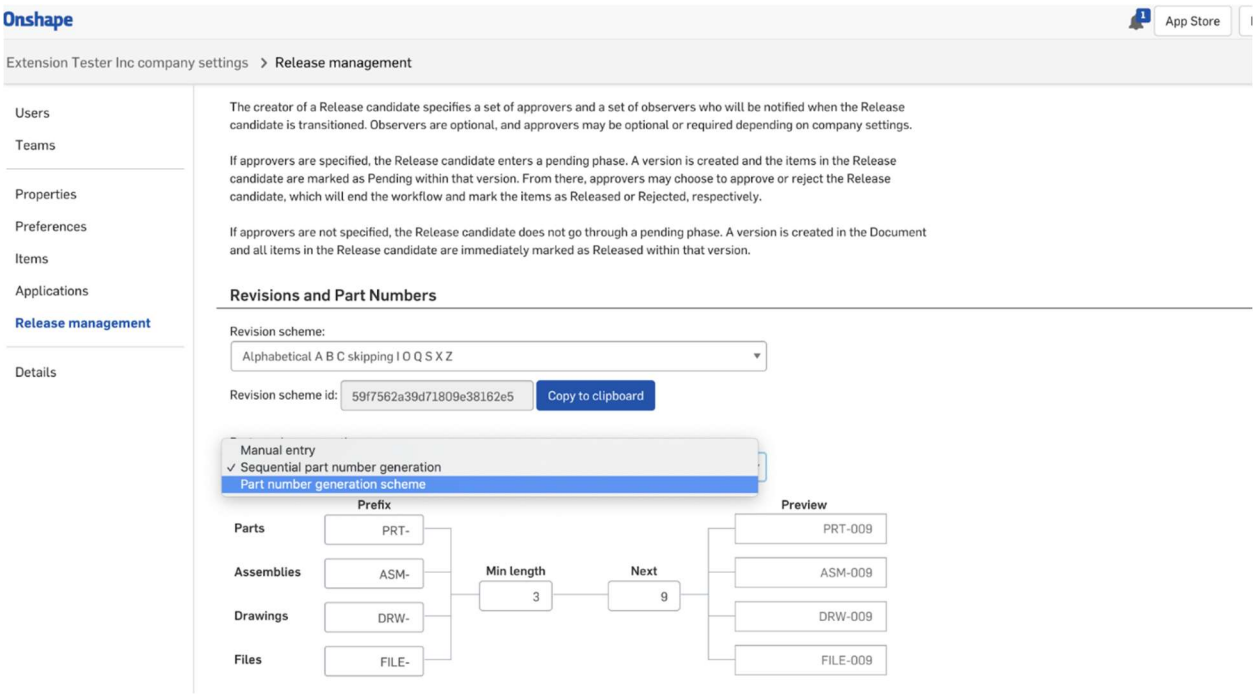
Default parameters as query string:

- `documentId`
- `workspaceId`
- `versionId`
- `elementId`
- `server`
- `companyId` - Default value is ‘cad’. If the document owner is company/enterprise, then the value is company/enterprise ID.
- `userId`
- `locale`
- `clientId`

This location supports action types.

Part number generator

This extension helps partners to embed their own custom part number generation scheme in Onshape. Each application can have only one extension of this type. Once defined, these extensions are listed as one of the part numbering schemes in the release management configuration in Company settings.



In the above screen shot, ‘Part number generation scheme’ is the user-defined name of the extension.

Supported contexts:

- There are no supported contexts.

Supported parameters for replacement:

- Parameter replacement is not supported.

Default parameters as query string:

- No default query parameters

Action URL defined by the user is assumed to be a POST API. This API should consume a predefined request body as shown below. This definition may have additional attributes in future.

```
 [
   {
            "id" : <internal part number id>,
            "documentId" : <documentId>,
            "elementId" : <elementId>,
            "workspaceId" : <workspaceId>,
            "elementType" : <elementType>,
            "partId" : <partId>,
            "companyId" : <companyId>, // Id of the company that owns the
document, else the text “cad”
            "partNumber" : <current part number>,
            "configuration" : <configuration string>,
            "categories" : <array of category ids and names> // [ { "id":
<String>, "name": <string> } ]
   }
 ]
```

Note: Categories are only passed from the Release dialog and properties dialogs for now. They are empty when part number generation is called from the BOM table or configuration table.

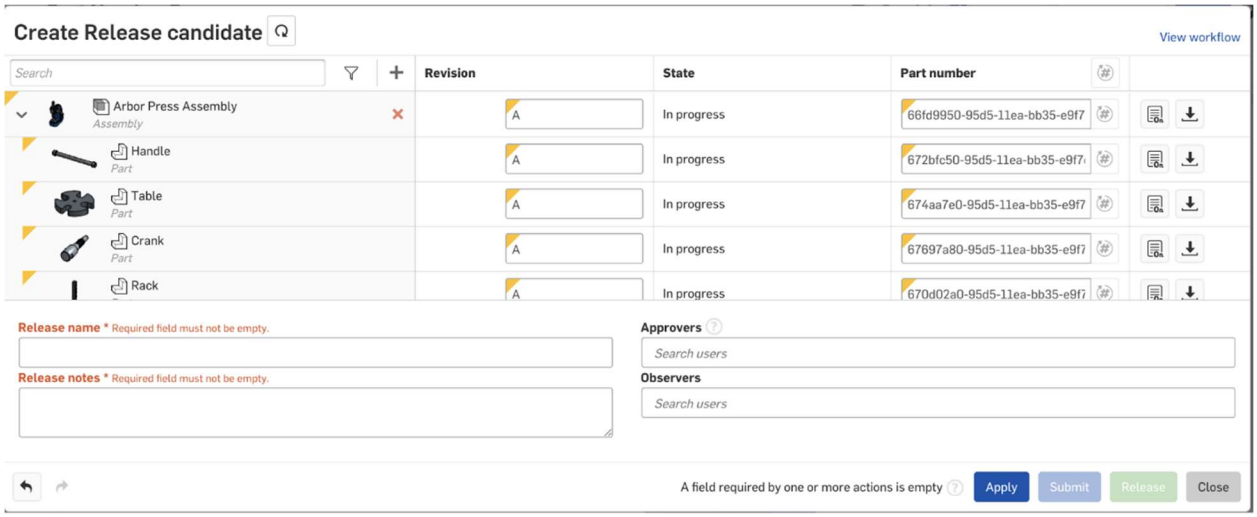
Expected response sent to Onshape is as follows:

```
 [
   {
            "id" : <internal part number id>,
            "documentId" : <documentId>,
            "elementId" : <elementId>,
            "workspaceId" : <workspaceId>,
            "elementType" : <elementType>,
            "partId" : <partId>,
            "partNumber" : <next part number generated by third party numbering
scheme>
```

```
    }
]
```

Third-party applications can simply fill the `"partNumber"` attribute with the part number generated by the custom numbering scheme and send it as a response. However, the response should at least contain `"id"` and `"partNumber"` as highlighted above; other attributes are optional.

Custom numbering schemes for part generation, once set in the Release management page, can be invoked from all the places where we set part numbers, including the Release candidate dialog shown below:



Sample code

We have provided a sample application that supports the features described in this document.

The source code for this `Inventory management` application can be found [in our public GitHub repository](#).

The instructions to install and the application are available in the `README.md` file in the repository.

The application is built on the Passport node module. It is based on this [article](#). Please read the article before proceeding with this section.

Some structural information about the application:

- The dependencies are defined in `package.json`
- The routing for inbound calls is defined in `server.js`. This includes routing for OAuth2 calls as well as calls for the rest APIs we expose that Onshape can call via the extensions.
- The OAuth2 calls are routed to `controllers/oauth2.js`. These include calls to authenticate as calls to get the bearer token.
- `controllers/oauth2.js` uses `controllers/auth.js` to interact with Passport to manage the authentication and storage.
- Other API calls to get part number, etc, route to the appropriate controller in the controllers directory.

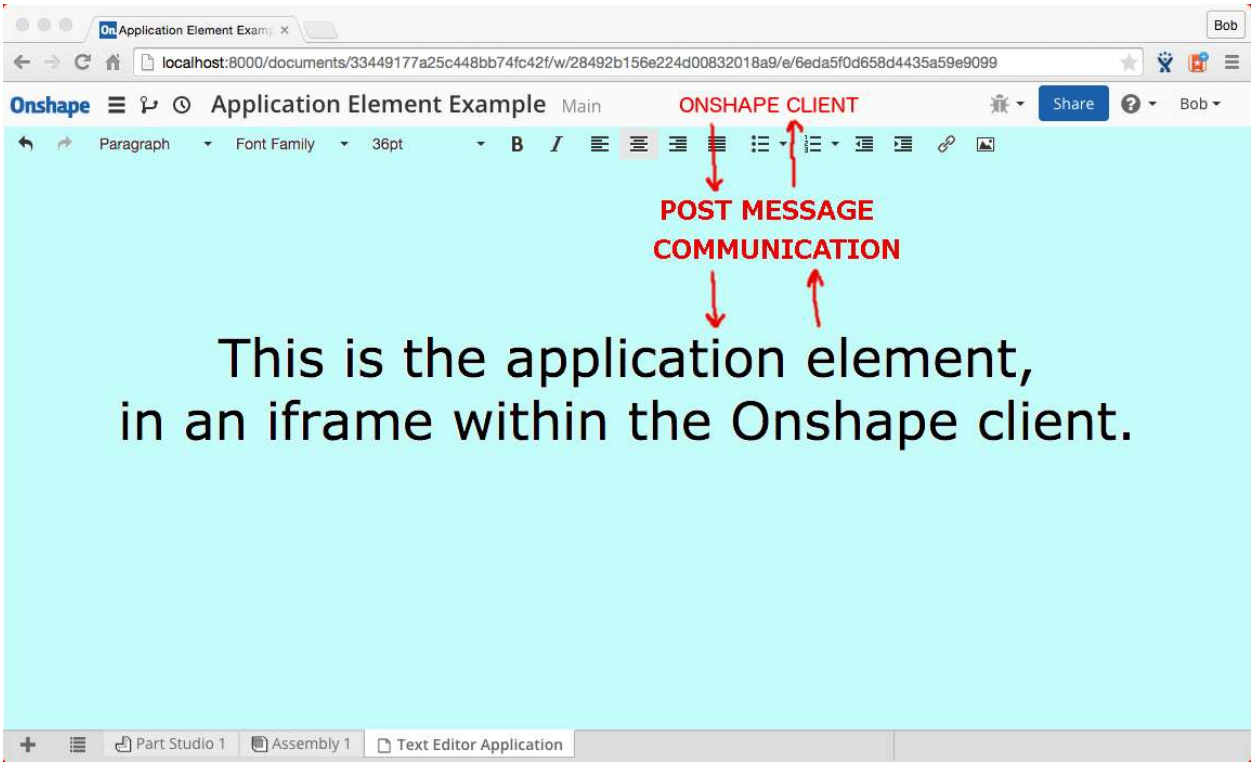
- The controllers use the model defined in the model directory.

The application is defined in the Developer Portal with extensions that use the exposed APIs.

The following screenshots define the base configuration of the application and some of the sample extensions.

Client Messaging

Application extensions and the Onshape JavaScript web client need to communicate directly, calling across the iframe containing the application extension using post message.



Onshape Client Messaging examples can be split into those that are initiated from the *application extension* and those that are initiated from the *Onshape client*.

Messages from the Extension

These Client Messaging examples can be initiated from the application extension:

- **Click/close flyouts events**: Notify the Onshape client that the user has clicked in the application extension, which should cause Onshape flyouts (versions, history, uploads, etc.) and dropdown menus (profile dropdown menu, document menu) to close. Without this, flyouts and menus might remain open over the application extension.
- **Shortcut keyboard events**: Shortcut keys (such as ?, which opens the Onshape Help dialog), can be handled by the application extension by posting a message to the Onshape client to open the dialog.
- **keepAlive**: Notify the Onshape client that the user is actively working in the application extension, which triggers the Onshape client to send a message to the server to keep the browser session alive. Without this, the Onshape browser session will timeout and ask the user to sign in again.
- **Standard Onshape dialogs**: Request from the application extension to the Onshape client to open one of the Onshape standard dialogs and send the user's choices back to the application extension. For example, if the application extension needs the user to choose a part or assembly to be operated on, the application extension can post a message to the Onshape client requesting that dialog be opened and the selected part or assembly information sent back to the application extension.
- **UI customization**": Request from the application extension to the Onshape client to customize the Onshape UI (e.g., add commands to menus, add buttons to the toolbars, etc). When these commands or toolbar buttons are clicked, the Onshape client posts a message to the application extension with the available context.
    - **Note**: This is limited to cases where the application extension is made active by the user; application extensions are not automatically loaded when a document is opened. Most UI customizations should be done when you register the application with Onshape, as those change the Onshape client automatically without needing to load the application extension first.
- **Content/material insertion**: Request from the application extension to insert content into the Onshape document. For example, insert a part into a new or existing Part Studio, apply a material to a part, add a material to a material library, etc.

Messages from Onshape

The following examples can be initiated from the Onshape client:

- **User action notification**: The Onshape client can notify an application extension when various user actions occur. For example, the Onshape client might notify when the user has made the application extension active or inactive (when the user clicks on document tabs). When an application extension is made inactive, it is moved off the edges of the browser, so it cannot be seen, but is still active, preserving its state.

- **Printing**: The Onshape client can notify an application extension when the user has chosen the **Print** command from the main Onshape document menu, enabling the application extension to perform a print operation.

Security Considerations

To ensure security, an application extension must:

- **Parse for document, workspace, and element IDs**: Parse for the `documentId`, `workspaceId`, and `elementId` that were passed as query parameters within the application extension's iframe `src` URL. You must post these back in each `POST` message.
- **Parse for the server**: Parse for the `server` that was passed as a query parameter within the application extension's iframe `src` URL. You must use this to validate messages received.
    - If the application extension uses a JavaScript library or framework (e.g., BackboneJS or AngularJS), it can parse the query parameters and maintain state in other ways.
- **Not redirect to another base URL**: The browser tells the Onshape client the origin base URL from which a `POST` message is received. The Onshape client ignores messages posted from an origin URL that doesn't match the original iframe `src` URL. It is *extremely important* to the security of your application that you verify that the origin of all messages you receive is the same as the original server query parameter in the iframe `src` (i.e., `if (server === e.origin)`). In production operation especially, the message IS NOT SAFE if the message origin does not match the iframe `src` server query parameter. Application extensions should not redirect to another base URL after the iframe has been opened, or the messages will be ignored.
- **Post a message on startup**: Onshape will not post messages until a newly started application extension has first posted a valid message to Onshape. This constraint is in effect anytime an application extension is (re)started and exists to avoid posting messages to application extensions that are not ready to handle them, are not fully loaded, etc. After your application extension is fully loaded and ready to receive messages, post a message to Onshape. A `keepAlive` message is a great first message to send to Onshape. Once Onshape receives a valid message, Onshape will start posting messages to the application extension. If the application extension later sends an invalid message Onshape will stop sending messages until a valid message is posted to Onshape.

`POST` messages submitted by application extensions to Onshape will be ignored if any of the following are true:

- The documentId, workspaceId, or elementId are missing or not valid.

- The message name is missing or not recognized.
- The origin of the `POST` message does not match the original iframe `src` URL.

Element Tab

Messages may be sent and received by element tab application extensions.

The following messages can be **sent** by Element tab application extensions:

messageName (case sensitive)	other message properties?	comment
applicationInit	yes `notifyWhenSaveRequired: whether Onshape should send a notification to save pending changes during certain operations (default is false)`	Send once on application startup.
closeFlyoutsAndMenus	no	Send when a mouse click or other event happens in the application extension. Closes Onshape flyouts and dropdown menus.
closeSelectItemDialog	no	Closes the select item dialog.
connectionLost	no	Displays the standard Onshape connection lost message in a message bubble, forcing the user to either reload the document or return to the documents page.

errorReload	yes `message: your message`	Similar to the connectionLost message, but enables an application to specify the first part of the message, which will be used instead of "Onshape is not connected." The user must reload the document or return to the documents page.
finishedSaving	yes `messageId: the id sent in the corresponding 'saveChanges' message`	Response to a 'saveChanges' message sent from Onshape. Should be sent after application has cleaned up any pending edits.
keepAlive	no	Send periodically while while the user is actively working to avoid the session from timing out.
saveAVersion	no	Send when the user types “Shift-S” in the application extension, the keyboard

		shortcut for save a version.
showKeyboardShortcutsHelp	no	Send when the user types “?” (Shift-? on most keyboards) in the application extension, the keyboard shortcut for the keyboard shortcuts help dialog.
showMessageBubble	yes `message: your message`	Send when you want to show a string in the blue message bubble at the top of the Onshape app.
startLoadingSpinner	yes `message: your message`	Send to start a large spinner in the middle of the browser window with your message underneath it.
stopLoadingSpinner	no	Send to stop the large spinner.
startWorkingSpinner	no	Send to start a small spinner in the middle bottom of the browser window.
stopWorkingSpinner	no	Send to stop the small spinner.

openSelectItemDialog	yes `dialogTitle: your dialog title (default is no title), selectBlobs: true or false (default is false), selectParts: true or false (default is false), selectPartStudios: true or false (default is false), selectAssemblies: true or false (default is false), selectMultiple: true or false (default is false), selectBlobMimeTypes: 'comma-delimited string of blob mime types to show in dialog (e.g. "application/dwt,application/dwg")' (default is an empty string) showBrowseDocuments: true or false - controls whether 'Other documents' choice should be available (default is true) showStandardContent: true or false - controls whether 'Standard content' choice should be available (default is false)`	Send when your application wants to open a dialog in which the user will select one or multiple items - blobs, parts, part studios or assemblies.
requestCameraProperties	yes `graphicsElementId: string , Element ID of the part studio or assembly`	Send to request camera properties of a specific part studio or assembly element. Note: The element should have been opened at least once in the current session. The messageName of the response is cameraProperties

The following messages can be **received** by Element tab application extensions:

messageName (case sensitive)	other message properties?	comment
show	no	Sent when an element tab application extension is shown (made active) within the Onshape client. This message is NOT sent when the element tab application extension is created.
hide	no	Sent when an element tab application extension is made inactive within the Onshape client. This message is NOT sent when an element tab application extension is deleted.
itemSelectedInSelectItemDialog	yes `documentId: id of selected item's document, workspaceId: id of selected item's workspace, empty if versionId not empty, versionId: id of selected item's version, empty if workspaceId not empty, elementId: id of element selected or containing the selected part, elementName: name of element selected or containing the selected part, elementType: type of element selected or containing the selected part -`	Sent when the user selects an item (blob, part, part studio or assembly) in the select item dialog that was opened due to an openSelectItemDialog message sent earlier. When a part is not selected, the partXxx message properties will be empty strings.

	`'partstudio', 'assembly' or 'blob', elementMicroversionId: microversion id of the element, itemType: type of item selected: 'part','partStudio' or 'assembly', partName: name of part selected, empty if itemType is not 'part', idTag: id of part, empty if no part selected`	
print	no	Sent when the user chooses the Print command while the application is the active element. The application can choose to handle this as either a print or an export to a PDF or other format.
selectItemDialogClosed	no	Sent when the select item dialog closes, either because the user selected an item and selectMultiple is false, or the user changed the active element or the user closed the dialog with the "X" button.
startFirstViewCommand	yes `documentId: id of selected item's document, workspaceId: id of selected item's workspace, empty if versionId not empty, versionId: id of selected item's`	Sent to a drawings application extension when the drawing is created with zero views. If other types of applications need a message posted to

	version, empty if workspaceId not empty, elementId: id of element selected or containing the selected part, elementName: name of element selected or containing the selected part, elementType: type of element selected or containing the selected part - ‘partstudio’, ‘assembly’ or ‘blob’, elementMicroversionId: microversion id of the element, itemType: type of item selected: ‘part’,‘partstudio’ or ‘assembly’, partName: name of part selected, empty if itemType is not ‘part’, idTag: id of part	them with creation context, contact Onshape and we can discuss using this sort of message for your application also.
export	yes fileExtension: the file extension of the export type the user chose - “.dwg”, “.dxf” are the types currently supported. baseFileName: the base portion of the expected output file. This is currently set to “<document name> - <element name>”	Sent when the user chooses a command to export the contents of the application to a file.
cameraProperties	yes graphicsElementId: string , Element ID of the part studio or assembly isValid: boolean, Indicates if the properties are valid or not. false if	Sent when application posts a requestCameraProperties message

	element ID is invalid or element has not been open in the current session projectionType: string, Denotes the projection method. Values are ‘orthographic’, ‘perspective’ . Empty string ‘’ if isValid is false viewMatrix: 16 element numeric matrix with elements at index 13, 14, 15 corresponding to position of the camera projectionMatrix: 16 element numeric matrix verticalFieldOfView: number, 0 in case of orthographic projection viewportHeight: number, eight of the viewport viewportWidth: number, width of the viewport	
takeFocus	no	Sent when the Onshape client sets focus on the content window of the element tab application extension.
saveChanges	yes messageId: a unique identifier for this message. Should be passed back in the 'finishedSaving' message.	Sent if the application specified 'notifyWhenSaveRequired' in the 'applicationInit' message. Indicates that the application should cleanup any pending edits before an Onshape process continues (i.e. version save).

Element Right Panel

Most client messaging functionality had been limited to that occurring between the Onshape client and application elements (the **Element tab** location). Limited functionality in now also available for client messaging to work with application extensions in the **Element right panel** location.

Enabled messaging to Element right panel extensions includes the communication of selections that the user makes for the following application extension contexts:

- Part Studio
- Assembly
- Document

All Security Considerations above apply to both Element tab and Element right panel extensions, with the following notes:

- Initial message from the application extension to the Onshape client, in the form of an `applicationInit` message (or one of any other messages supported by the element right panel extensions), is required to ensure the Onshape client does not send messages to the extension until it is ready.
- Once a valid `applicationInit` message is received by the Onshape client, it will start sending messages with the `messageName` value `SELECTION` upon user selection interactions.
- Prior to accepting *any* message from the Onshape client as secure, the `origin` attribute value included in incoming messages must be validated as equal to the original `server` query parameter value used to load the application extension.

Code Snippets

Parse query parameters

This JavaScript code parses the iframe `src` query parameters and uses them to post a message:

```
var documentId;
var workspaceId;
var elementId;
var server;

// Parse query parameters
var queryParameters = decodeURIComponent(window.location.search.substr(1));
var queryParametersArray = queryParameters.split('&');
for (var i = 0; i < queryParametersArray.length; i++) {
  var parameterArray = queryParametersArray[i].split('=');
```

```
    if (parameterArray.length === 2) {
      switch (parameterArray[0]) {
        case 'documentId':
          documentId = parameterArray[1];
          break;
        case 'workspaceId':
          workspaceId = parameterArray[1];
          break;
        case 'elementId':
          elementId = parameterArray[1];
          break;
        case 'server':
          server = parameterArray[1];
          break;
      }
    }
  }

  // Listen for clicks and post a message to the Onshape client
  document.getElementById('<id of your topmost element>').
    addEventListener('click', function() {
      var message = {documentId: documentId,
                     workspaceId: workspaceId,
                     elementId: elementId,
                     messageName: 'closeFlyoutsAndMenus'};
      window.parent.postMessage(message, '*');
    }, true);
```

Create a message object

The message object posted to the Onshape client is of the form:

```
{
    documentId: documentId,
     workspaceId: workspaceId,
     elementId: elementId,
     messageName: '<message name>',
    // … other properties as needed for other message types …
}
```

The message data object posted to the application extension is of the form:

```
{
    messageName: '<message name>',
 //    … other properties as needed for other message types …
}
```

The message will always have a `messageName` property.

Listen for messages

To listen for messages from the Onshape client:

```
  // server is one of the iframe src query parameters - see above
```

```
  var handlePostMessage = function(e) {
    console.log("Post message received in application extension.");
    console.log("e.origin = " + e.origin);

    // Verify the origin matches the server iframe src query parameter
    if (server === e.origin) {
      console.log("Message safe and can be handled as it is from origin '"
                  + e.origin +
                  "', which matches server query parameter '"
                  + server + "'.");
      if (e.data && e.data.messageName) {
        console.log("Message name = '" + e.data.messageName + "'");
      } else {
        console.log("Message name not found. Ignoring message.");
      }
    } else {
      console.log("Message NOT safe and should be ignored.");
    }
  };

  window.addEventListener('message', handlePostMessage, false);
```

Send and handle messages

The following is an example of how one might send an initialization message to, and handle post messages from, the Onshape client.

Note: Proper clean-up of event listeners is not included in the snippet

```
function handlePostMessage(event) {
  // ensure that the event data is from a legit source:
  if(theServerStringFromActionUrl !== event.origin) {
    console.error('origin of message is not legitimate');
    return;
  }

  // branch based on messageName attribute
  switch(event.data.messageName) {
    case 'SELECTION':
      console.debug('SELECTION event data: %o', event.data);
      break;
    default:
      console.debug(`${event.data.messageName} not handled`);
    }
}

window.addEventListener('message', handlePostMessage);

const initMessage = {
  documentId:  theDocumentId,    // required - parsed from action url
  workspaceId: theWorkspaceId,   // required - parsed from action url
```

```
  elementId:   theElementId,     // required - parsed from action url
  messageName: 'applicationInit' // required
};

window.parent.postMessage(initMessage, '*');
```

Right panel interaction sequence

The sequence diagram below illustrates the interaction between an Element right panel application extension and the Onshape client:

```
%%{
  init: {
    "theme": "default",
    "sequence": {
      "mirrorActors": false,
      "showSequenceNumbers": false,
      "width": 75,
      "height": 60,
      "actorMargin": 25,
      "messageFontSize": 13,
      "messageFontFamily": "monospace",
      "messageFontWeight": 2
    }
  }
}%%
sequenceDiagram
  actor user
  participant OSC AS Onshape Client
  participant AE AS Application Extension
  user->>+OSC: start element right panel extension
  Note right of user: via configured button
  OSC->>+AE: invoke action url (with query params)
  AE->>OSC: postMessage(messageName: 'applicationInit')
  loop selection interactions
    user->>OSC: select
    OSC->>AE: postMessage(messageName: 'SELECTION')
  end
  user->>OSC: stop element right panel extension
    Note right of user: via configured button
  deactivate AE
  OSC-XAE: destroy
  deactivate OSC
```

Right panel message exchange

The following messages are exchanged for application extensions located in the element right panel and configured for Part Studio, Assembly, or Document contexts.

The first message with `messageName` attribute set to `applicationInit` is sent to the Onshape client by an application extension once it is loaded and ready to receive and process incoming messages:

```
    {
        documentId: '<document id>',
        workspaceId: '<workspace id>',
        elementId: '<element id>',
        messageName: 'applicationInit'
    }
```

The values `<document id>`, `<workspace id>`, `<element id>`, and `<server id>`:

- Are originally included as query parameters in the action URL used to request the content of the application extension
- Must be included in messages sent to the Onshape client

While initialization is the specific intent of the `applicationInit` message, other supported `messageName` attributes have the same initialization effect upon their first receipt by the Onshape client.

Next, as the user interacts with Onshape by selecting various parts of the model, messages with the `messageName` attribute set to `SELECTION` are sent to the application extension:

```
{
    messageName: 'SELECTION',
    selections: [
        {
            selectionType: 'ENTITY',
            selectionId: 'KRiB',
            entityType: 'FACE',
            occurrencePath: [
                'MfOieM8xKIDGHe37c'
            ],
            workspaceMicroversionId: 'a781c53fbd1095e3462d2b70'
        },
        {
            selectionType: 'ENTITY',
            selectionId: 'KRdC',
            entityType: 'EDGE',
            occurrencePath: [
                'MfOieM8xKIDGHe37c'
            ],
            workspaceMicroversionId: 'a781c53fbd1095e3462d2b70'
        }
    ]
}
```

Notes

Keyboard focus

Keyboard focus will not be transferred to an application until the user clicks in the application or the application programmatically takes focus. An application should programmatically take focus when it is first loaded and when it receives a `show` message from Onshape. Shortcut keys will work immediately when the application is shown.

Future work

- New message types will be added as needed. If your application extension needs a message not listed in this document, please notify us, and we'll work with you on it.
- Mobile client support is unclear at this time.
- Onshape is considering using promises to wrap `POST` messages, which would make the application extension's JavaScript simpler and enable chaining `POST`s with other operations. Promises would make some interactions with multiple responses difficult (e.g., when you open a dialog, like the Select Item dialog, and want to receive multiple `POST` messages back due to the user clicking on multiple items in the dialog)

Structured Storage

Sub Elements

Onshape provides application elements storage that is controlled by applications through the API. These elements allow a set of named sub-elements.

The application can make changes to sub-elements independently or in arbitrary groupings. Changes may be wholesale replacements, or may be deltas. When performing a delta update, the application may post a full version as well, which allows the api to return a smaller number of deltas for subsequent queries.

An application may need to perform multiple versionable actions in the course of performing a user-level action, and we want to allow the individual actions to be collected into a single action from the perspective of document history. We do this by providing support for creation of a private transaction and support for atomically committing the transaction to the document workspace as a single user-visible action.

Onshape does not assume any knowledge about the semantics of application deltas. All merging of deltas into a consolidated form is done by the application. Applications should typically send checkpoint state for a sub-element if many delta changes have been made since the last checkpoint.

Document content and changes are logically an array of bytes, but since they are transmitted through JSON, then are expected to be presented a Base-64 encoding of the array into string form.

We use some terminology in this document that is new.

- **changeId** - an opaque identifier for the state of an application element. Each change to the application element results in a new changeId
- **transaction** - a private workspace within a document workspace for composing modifications to an application element. These changes are not visible to the user until committed.
- **transaction commit** - an operation that moves the changes performed within a transaction to the application element workspace as a single user-visible action.

Concurrent access by multiple users

If the element is being concurrently accessed by multiple sessions, updates may encounter conflicts during update. If the application has a mechanism that ensures that all accesses to the element are mediated by a single process, as is done with our part studio and assemblies, this can be addressed directly by the application. However, if the application is not able to mediate access in this way, updates by one session may invalidate state held by another session. We address this by notifying updaters when an update cannot be directly applied because their state is out of date and allowing them to refresh their state before re-applying the change.

This policy of requiring the application have current state when posting updates could be overly conservative in some cases. Detecting conflict at the sub-element level might provide for better concurrent access performance, but there probably are cases where this fails, so it probably would need some level of application control.

JSON Tree

In contrast with sub elements, JSON tree storage is a more managed data storage mechanism that Onshape itself can merge and diff. At the root of it, the data

structure is a single JSON object per Application Element. The user submits incremental changes that are then applied by Onshape to the JSON tree. Onshape stores these 'diffs' in a new microversion created as a result of the update request, or during a subsequent transaction commit request. When the user then performs a merge or restore operation, Onshape can sum and apply the requisite incremental changes. By storing diffs, Onshape provides to the user a storage mechanism that is more robust to race conditions, since multiple simultaneous edits can be optionally merged by Onshape. All of these qualities make JSON tree a preferred way to store application element data in an Onshape-native manner.

JSON Tree Edit Semantics

BTJEdit Encoding

A JSON tree edit represents an incremental change to an application element's JSON tree. The edit is a `BTJEdit` class, which is encoded as one of the following:

- Deletion:

```
{ "btType" : "BTJEditDelete-1992", "path" : "path" }
```

- Insertion:

```
{ "btType" : "BTJEditInsert-2523", "path" : "path", "value" : "newValue" }
```

- Change:

```
{ "btType" : "BTJEditChange-2636", "path" : "path", "value" : "newValue" }
```

- Move:

```
{ "btType" : "BTJEditMove-3245", "sourcePath" : "path", "destinationPath" :
"path" }
```

- List (where edit1, edit2, etc. are zero or more edits.):

```
{ "btType" : "BTJEditList-2707", "edits" : [ "edit1", "edit2", "..."] }
```

Within the above encoding, `newValue` is a stand in for any valid JSON, and `path` is a stand in for an object representing a path to the node at which to perform the edit.

BTJPath Encoding

The BTJPath object describes a path through the JSON tree to a particular node, and is encoded as follows:

```
{ "btType" : "BTJPath-3073", "startNode" : "startNode", "path" : [
"pathElement1", "pathElement2", "..."] }
```
where startNode is a string that is either empty to specify the root node or a nodeId of a node in the tree. The pathElement is one of:

- Key:

```
{ "btType" : "BTJPathKey-3221", "key" : "string" }
```

- Index:

```
{ "btType" : "BTJPathIndex-1871", "index" : "integer" }
```
In the insertion and move type edits the path elements can describe a path that doesn't currently exist. Onshape will generate the proper keys and values as needed to place the node value in the proper location.

JSON Tree Examples

Below are some examples that show the body required to perform the particular edit on a JSON tree.

Deletion Example

If the pre-existing JSON tree looks like:

```
{"myKey":  "myValue"}
```
and a delete edit looks like:

```
{"btType": "BTJEditDelete-1992",
     "path": {"btType": "BTJPath-3073", "startNode": "", "path": [{"btType":
"BTJPathKey-3221", "key": "myKey"}]}}
```
then the resulting JSON is the result of deleting the node specified by `path`:

```
{}
```

Insert Example

If the pre-existing JSON tree looks like:

```
{}
```
and the insertion edit looks like:

```
{"btType": "BTJEditInsert-2523",
     "path": {"btType": "BTJPath-3073", "startNode": "", "path": [{"btType":
"BTJPathKey-3221", "key": "insertedKey"}]},
     "value": "myValue"}
```
then the resulting JSON is the result of inserting the node described by `value` at the node specified by `path`:

```
{"insertedKey": "myValue"}
```

Change Example

If the pre-existing JSON tree looks like:

```
{"myKey":  "myValue"}
```

and the change edit looks like:

```
{"btType": "BTJEditChange-2636",
     "path": {"btType": "BTJPath-3073", "startNode": "", "path": [{"btType":
"BTJPathKey-3221", "key": "myKey"}]},
     "value": "myOtherValue"}
```

then the resulting JSON is the result of changing the node specified by `path` to the node described by `value`:

```
{"myKey":  "myOtherValue"}
```

Move Example

If the pre-existing JSON tree looks like:

```
{"myKey":  "myValue", "myOtherKey":  "myOtherValue"}
```

and the move edit looks like:

```
{"btType": "BTJEditMove-3245", "sourcePath": {"btType": "BTJPath-3073",
"startNode": "",
                                                "path": [{"btType":
"BTJPathKey-3221", "key": "myKey"}]},
     "destinationPath": {"btType": "BTJPath-3073", "startNode": "",
                         "path": [{"btType": "BTJPathKey-3221", "key":
"keyCreatedFromMove"}]}}
```

then the resulting JSON is the result of moving the node from the specified `sourcePath` to the `destinationPath`:

```
{"keyCreatedFromMove": "myValue"}
```

List Example

If the pre-existing JSON tree looks like:

```
{}
```

and the list edit looks like:

```
{"btType": "BTJEditList-2707", "edits": [
        {"btType": "BTJEditInsert-2523", "path": {"btType": "BTJPath-3073",
"startNode": "", "path": [{"btType": "BTJPathKey-3221", "key": "myKey"}]},
         "value": "myValue"},
        {"btType": "BTJEditChange-2636", "path": {"btType": "BTJPath-3073",
"startNode": "",
                                                "path": [{"btType":
"BTJPathKey-3221", "key": "myKey"}]},
```

```
         "value": ["firstValue", "secondValue"]},
        {"btType": "BTJEditInsert-2523", "path": {"btType": "BTJPath-3073",
"startNode": "",
                                                  "path": [{"btType":
"BTJPathKey-3221", "key": "myKey"},
                                                           {"btType":
"BTJPathIndex-1871", "index": 1}]},
         "value": "myBetterSecondValue"},
        {"btType": "BTJEditDelete-1992", "path": {"btType": "BTJPath-3073",
"startNode": "",
                                                  "path": [{"btType":
"BTJPathKey-3221", "key": "myKey"},
                                                           {"btType":
"BTJPathIndex-1871", "index": 2}]}}
    ]}
```

then the resulting JSON is the result of applying all the given edits in order:

```
{"myKey": ["firstValue", "myBetterSecondValue"]}
```

The intermediate steps were:

1. Insertion:
2. `{"myKey": "myValue"}`
3. Change:
4. `{"myKey": ["firstValue", "secondValue"]}`
5. List insertion:
6. `{"myKey": ["firstValue", "myBetterSecondValue", "secondValue"]}`
7. List deletion:
8. `{"myKey": ["firstValue", "myBetterSecondValue"]}`

All the examples above were tested and validated using the Python client here.

Webhook Notifications

Summary

Webhook notifications allow an application to register to receive notifications of certain events that occur within the Onshape environment. To receive a notification, an application must expose an endpoint that Onshape can call.

Example Code

- Python

Events

Each type of event that an application may receive notifications for has a unique identifier known as the event type. Event types are grouped into Event Groups. Each group shares specification requirements, as described below.

Event Groups

Event types are categorized into several different groups based on the dominant user resource of the event. The group that a given event is part of defines the required parameters needed in the registration process to identify the resource or group of resources to watch. For instance, if registering for an event in the `document` event group, the application must identify either a specific document's id or a specific company's id. If registered for a company, the event will be registered for all present and future documents owned by the company.

Application Group

Monitor changes to applications.

Registration Requirements

- `clientId`

Supported Event Types

- `onshape.user.lifecycle.updateappsettings` - occurs when user application settings are modified

Document Group

Monitor various aspects of document changes.

Registration Requirements

- `documentId` OR `companyId` must be specified in the registration body.

Note: only `documentId` is valid for the `onshape.document.lifecycle.statechange`.

Supported Event Types

- `onshape.model.lifecycle.changed` - occurs when a change to a model has been made
- `onshape.model.translation.complete` - occurs when a translation request is completed
- `onshape.model.lifecycle.metadata` - occurs when part or element metadata is modified
- `onshape.model.lifecycle.createversion` - occurs when a new version of a document is created
- `onshape.model.lifecycle.createworkspace` - occurs when a new workspace is created
- `onshape.model.lifecycle.createelement` - occurs when a new element is created
- `onshape.model.lifecycle.deleteelement` - occurs when an element is deleted
- `onshape.document.lifecycle.statechange` - occurs when an document changes state
- `onshape.model.lifecycle.changed.externalreferences` - occurs when an external reference changes
- `onshape.document.lifecycle.created` - occurs when a document is created
- `onshape.revision.created` - occurs when a revision is created
- `onshape.comment.create` - occurs when a comment is created in a document
- `onshape.comment.update` - occurs when a comment is updated in a document

- `onshape.comment.delete` - occurs when a deleted is deleted in a document

Workflow Group

Monitor release management actions.

Registration Requirements

- `companyId`

Supported Event Types

- `onshape.workflow.transition` - occurs when a revision or release package transitions through workflow states.

Lifecycle Group

Monitor webhook changes.

Registration Requirements

No requirements. This event type responds to any and all webhooks registered by the same application.

Supported Event Types

- `webhook.ping` - occurs either in response to a request by an application to call a registered webhook, or as a post-registration validation initiated by Onshape
- `webhook.register` - occurs in response to a notification registration API call
- `webhook.unregister` - occurs in response to a notification deregistation API call

Webhook Registration

An application registers for event notification by making a REST call to the Onshape web service, providing a URL to notify, and the required parameters for the event types to be registered, as mentioned in Event Groups. If the registration request is well-formed, the registration API call returns information about the registration, including a unique ID string. This ID string identifies the webhook registration. No de-duplication of notification registrations is performed by the API. Each registration call will yield a new registration ID, even if the parameters are identical to those passed in a prior call. Shortly after an application calls the notification registration API, Onshape will make make an asynchronous trial notification call to the URL generated from the URL template with an event type of webhook.register in order to test whether the application notification server is accessible. If the trial notification delivery fails to return an HTTP 200 status code, the notification registration is cancelled. The trial notification is usually delivered after the notification registration has been received by the application. However, variations in network delays may result in the trial notification occurring before response is received and processed by the application, so the notification handler should be ready to process notifications before the registration call is made.

Notifications

Notifications are delivered to an application as an HTTP POST with a JSON body. The body includes information about the identity of the registration request plus information specific to the event and notification message. An application may register for notifications to a URL that uses either HTTP or HTTPS. If HTTPS is specified by the URL template, the notification server must supply a certificate that is signed by a certificate authority (CA) recognized by Onshape. Self-signed certificates as well as certificates signed by unrecognized CAs will be rejected, causing notification delivery to fail.

Notification Deregistration

When an application no longer needs to be notified of changes specified by a particular notification registration, it should normally de-register the notification request. De-registration is performed by making an HTTP that specifies the hook to deregister. Onshape will attempt to call the deregistered hook with an event type of webhook.unregister as validation for the application that the de-registration is complete. If the application does not de-register the webhook, Onshape will continue delivering notifications until either the the application returns an error in response to a notification for the webhook, or fails to respond at all for an extended period of time.

Notification API example messages

registration


```
{
  "timestamp": "2014-12-16T23:45:10.611-0500",
  "event": "webhook.register",
  "workspaceId": "000000000000000000000000",
  "elementId": "000000000000000000000000",
  "webhookId": "544e91f7fb88ed44f5de1508",
  "messageId": "34795d2e5f5f44eeb61fb7b1",
  "data": "Some data",
  "documentId": "000000000000000000000000",
  "versionId":  "000000000000000000000000"
}
```


ping


```
{
  "timestamp": "2014-12-16T23:46:24.368-0500",
  "event": "webhook.ping",
  "workspaceId": "000000000000000000000000",
  "elementId": "000000000000000000000000",
  "webhookId": "544e91f7fb88ed44f5de1508",
  "messageId": "6808d9a622644330b6cd95f5",
  "data": "Some data",
  "documentId": "000000000000000000000000",
  "versionId":  "000000000000000000000000"
}
```


model change


```
{
```

```
  "timestamp": "2014-12-16T23:46:29.284-0500",
  "event": "onshape.model.lifecycle.changed",
  "workspaceId": "f925722bee1c43fc80fb5bb2",
  "elementId": "0f931a1ceba842299192823f",
  "webhookId": "544e91f7fb88ed44f5de1508",
  "messageId": "60f54ac1cbc04179a6642d9a",
  "data": "Some data",
  "documentId": "0f9c4392e5934f30b48ab645",
  "versionId":  "000000000000000000000000"
}
```

document state change

```
{
  "timestamp": "2014-12-16T23:46:29.284-0500",
  "event": "onshape.document.lifecycle.statechange",
  "workspaceId": "000000000000000000000000",
  "elementId": "000000000000000000000000",
  "webhookId": "544e91f7fb88ed44f5de1508",
  "messageId": "60f54ac1cbc04179a6642d9a",
  "data": "Some data",
  "documentId": "0f9c4392e5934f30b48ab645",
  "versionId":  "000000000000000000000000",
  "documentState": "TRASH"
}
```

Possible values of documentState are:

- “ACTIVE” - document is in a normal, usable state
- “TRASH” - document has been moved to the trash (User can move out of trash to ACTIVE state)
- “DELETED” - document has been deleted

user application settings change

```
{
  "timestamp": "2014-12-16T23:46:29.284-0500",
  "event": "onshape.user.lifecycle.updateappsettings",
  "workspaceId": "f925722bee1c43fc80fb5bb2",
  "elementId": "0f931a1ceba842299192823f",
  "webhookId": "544e91f7fb88ed44f5de1508",
  "messageId": "60f54ac1cbc04179a6642d9a",
  "data": "Some data",
  "userId": "567953d60a1a5fbb95940333",
  "clientId":"ABCDEFGHIJKLMNOPQRSTUVWXZY234567ABCDEFG="
}
```

translation completion

```
{
  "timestamp": "2014-12-16T23:46:29.284-0500",
  "event": "onshape.model.translation.complete",
  "workspaceId": "f925722bee1c43fc80fb5bb2",
  "elementId": "0f931a1ceba842299192823f",
  "webhookId": "544e91f7fb88ed44f5de1508",
```

```
  "messageId": "60f54ac1cbc04179a6642d9a",
  "data": "Some data",
  "documentId": "0f9c4392e5934f30b48ab645",
  "userId": "567953d60a1a5fbb95940333",
  "translationId": "4f5de10f9c4392e5934f30b4"
}
```

comment create


```
{
  "timestamp": "2014-12-16T23:46:29.284-0500",
  "event": "onshape.comment.create",
  "workspaceId": "f925722bee1c43fc80fb5bb2",
  "elementId": "0f931a1ceba842299192823f",
  "webhookId": "544e91f7fb88ed44f5de1508",
  "messageId": "60f54ac1cbc04179a6642d9a",
  "documentId": "0f9c4392e5934f30b48ab645",
  "commentId": "567953d60a1a5fbb95940333"
}
```


Launch Checklist

This checklist brings together the processes you should follow to ensure your app is launched successfully. While all tasks must be completed to submit your app to the Onshape App Store, the task sequence provided here is a suggestion.

1. Understand quality expectations

These are to ensure that the Onshape App Store remains a trusted resource and that quality is maintained Review the Quality Considerations page, and reach out to the Developer Relations team with any questions.

2. Sign in to your developer account

Sign in to your developer account at dev-portal.onshape.com, and ensure your developer account details are accurate. Contact our API Support team if you need assistance.

3. Authenticate your app

Please refer to the OAuth documentation for information on authenticating your app with OAuth2.

4. Build your app

While building your app, use the resources in our Onshape Developer Documentation, including this API Guide and our API Explorer. We recommend familiarizing yourself with the following pages:

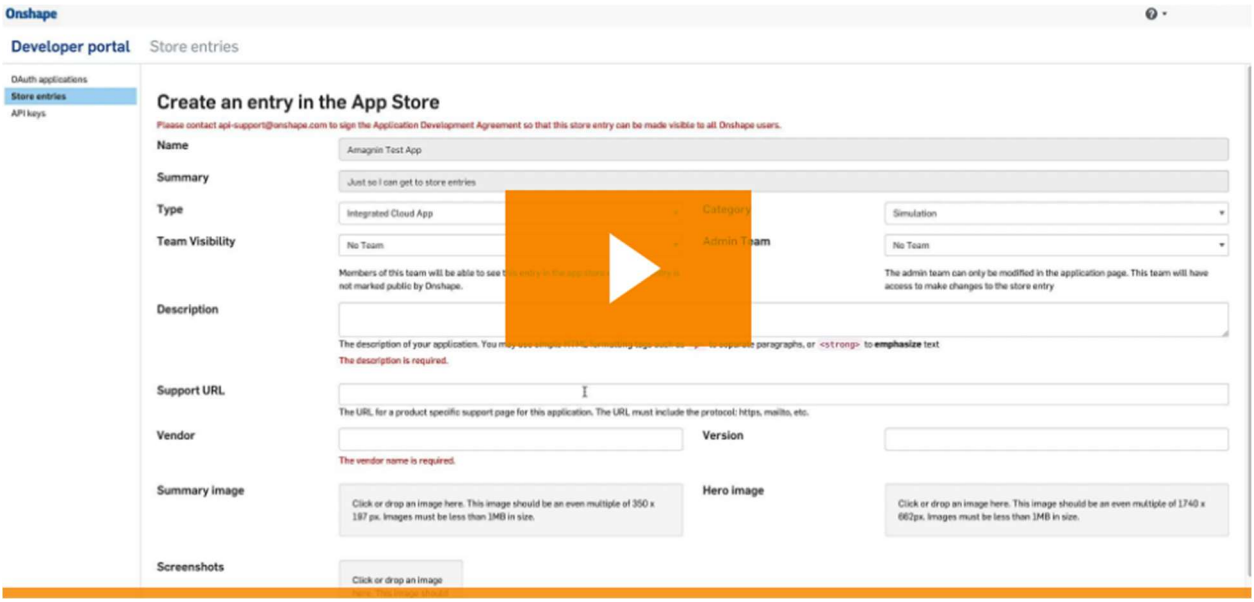
- Introduction to the Onshape REST API
- Onshape Architecture
- Onshape App Development

- Extensions
- Client Messaging
- Onshape App Store

5. Prepare your store entry

Prepare the descriptions, promotional graphics, screenshots, and videos you'll add to your store entry. Make sure you include a link to if required. Watch this video for more details.

See the video below for a walkthrough:



6. Run beta tests

During the beta period, try to enlist at least 5 active testers to get feedback before making your app available to the general public.

1. To find beta testers, contact our Developer Relations team, and recruit via the Onshape forum.
2. To give beta users early visibility, first create a team, then ensure your app is shared with that team in the Developer Portal. See the Create a Team in Onshape video for a walkthrough.

7. Determine your app's price

Once you've determined your monetization model, set up your price, billing, and other details. Billing should be tested, and to do so a staging environment is available. See the Billing API page for more details.

8. Sign and return the developer agreement

Email to the Developer Relations team to obtain yours.

9. Submit your app for final testing

Once you've returned the developer agreement, you can submit your app to the Developer Relations team for final testing. This is a comprehensive functionality and design test. During

this testing period (expect up to a week, depending on complexity), changes to code are prohibited unless requested. At the conclusion of the test, you will receive one of the following notifications from our Developer Relations team:

- Approved for release
- Approved for release with feedback
- Changes required before another round of testing

10. Integrate your support systems

Whether you use Zendesk, Jira, or email support, we'll help you determine and set up this integration. Contact the Developer Relations team to explore these options. This is the channel we will use to test your app and provide feedback.

11. Connect via Slack

Connecting directly with our Support, Tech, and Sales teams has proven to be valuable to app developers. This dedicated channel is simple to implement if you already have a paid Slack account. If not (or if you want to use the free version of Slack), we can add members of your team as guests to our account. Please contact Aaron Magnin to establish this connection.

12. Final check and publish

First, double-check you've done everything on this list. Now you're ready to publish your app to the production channel! Send an email detailing **when** you'd like this to happen to the Developer Relations team.

13. Promote your app

Start promoting your app with the Onshape badge and by presenting it to our internal teams in one of the following ways:

- To the Onshape Tech and Sales teams in a small meeting (~1 hour, Wednesdays at noon Eastern Time)
- To the entire Onshape organization in a company-wide meeting (~5-10 minutes, Thursdays at 11am Eastern Time).

14. Encourage reviews from users

The value of reviews is not to be underestimated. Reviews give users an opportunity to provide feedback, and can also signal to others that your app is worth investigating.

15. Maintain your app

Continually fix stability and performance issues. Improving the user experience will result in more engaged users, higher ratings, and in turn, more success.

Failure to respond to customer tickets in a reasonable time will lead to the removal of your app from the Onshape App Store.

16. Increase engagement and retention

Aim to increase user engagement, retain and grow your audience, and earn more revenue by:

- Encouraging repeat visits with a nurture stream and training materials
- Integrating more features from user requests
- Interacting with and understanding your audience via the Onshape forum, social media, etc.

17. Address app security

At some point, a prospect or user will enquire about your app's security controls. To address this, we recommend that you understand SOC 2 Compliance requirements, and consider filling out the Consensus Assessment Initiative Questionnaire (CAIQ). Onshape/PTC cannot and will not attest to your compliance. More on SOC Compliance can be found at the following links:

- AICPA.org
- Wikipedia: System and Organization Controls

Testing Guidelines

The purpose of this document is to help you get your application and App Store entry ready for QA testing.

Application Release Workflow (ARW)

Each application submitted to the Onshape App Store goes through a series of stage-gates:

1. Starting state: `Ok to deploy to limited visibility on Production` (Beta testing)
2. `Ok to make Public`
3. Goal state: `Application is Public`

To advance to the next stage, your application must pass testing, and your App Store entry must pass review.

Kick Off Testing

While completing the Launch Checklist, you will need to use Jira to request testing and release for your application. You can initiate the following tasks from Jira:

- `Request application testing`: This puts your application in the testing queue. We will note when testing has started (`in progress`) and when concluded, the ticket will be closed. The outcome will include notes and links to any issues generated (`tickets`). This phase may include as many iterations as needed to get your application ready.
- `Request public (general) authorization`: We will note when testing has started (`in progress`), and this request will trigger a review of your app store entry and any outstanding bugs. Note there is no implied testing of your application, simply a review of outstanding issues (`tickets`) and of the App Store entry. Success at this stage will advance the Application Release Workflow (ARW), and you can request public release.

- `Request public (general) release`: This request states that you have coordinated with the Developer Relations team and the Onshape Marketing team, and agreement has been reached that the app is ready for launch. Congratulations!

Testing Protocol

Applications are first tested against the checklist in Addendum A. *Production App Store* entries are then performed against the checklist in Addendum B.

Results will be viewable in your Onshape support system (i.e., Zendesk, Jira). The result of each test will be one of:

- `Pass`:
  - No action needed.
  - No notification issued.
- `Enhancement`: Suggestions we believe would make the application better.
  - Will NOT prevent the application from being turned on for public access.
- `Bug (low priority)`: Slight deviations from the criteria that have low end user impact.
  - Will NOT prevent the application being turned on for public access.
  - No stipulated time-frame for resolution.
- `Bug (medium priority)`: Material deviations from the criteria that are noticeable to the end-user. Represents a minor problem that requires a work-around.
  - Will NOT prevent the application from being turned on for public access.
  - Must be fixed within 30 days.
- `Bug (high priority)`: Significant deviation from the criteria.
  - WILL prevent the application from being turned on for public access.
- `Bug (MUST FIX)`: Significant deviation from protocol or security violation.
  - WILL prevent the application from being turned on for public access.
  - If the application is already public, it may be temporarily suspended from the App Store.

Testing Notes

- Testing may be requested at any time.
- Testing is done on a first-come basis.
- When testing is complete (pass or fail), you go to the back of the queue.

Addendum A

Application Test Criteria

- The application must use the Onshape OAUTH mechanism
- The OAUTH must be against the correct stack
- To be promoted to the Production stack, and hosted service must be on a monitored production server with worldwide 24/7 availability.
- The application should not generate any avoidable console (browser) errors
- The application should should provide one or more of the following options. The user should not have to leave the registration workflow to complete a pre-requisite.
  - Sign in using the Onshape ID (account created silently on first use)

    - Sign in with partner product account credentials
    - Create a new partner account
- The application must be capable of managing/displaying documents in excess of 20. The application must display reasonable performance when reading documents, workspaces, elements, and parts. At scale, an account may have thousands of documents, many with multiple workspaces and each with multiple elements. Suggested strategies include:
    - Using a `Next` button to load the next 20 documents
    - Using infinite scroll (loading the next 20 if the scrollbar reaches the bottom of the dialog)
    - Displaying the most recently-opened documents first
    - Displaying a counter of documents/workspaces/elements read
    - Using progressive loading
- The application should correctly list valid documents when `per document app access` is turned on.
- The application should correctly handle selection of versions.
- The application should correctly handle selection of workspaces (branches).
- The application should correctly handle/display elements that are:
    - Part Studios that contain nothing
    - Assemblies that contain nothing
    - Part Studios that contain only surfaces
    - Part Studios that contain only wire data (e.g., helices)
- The application should appropriately handle revocation of a grant.

Addendum B

App Store Testing Criteria

- The application should have a descriptive name
- The application summary should be accurate
- The redirect URLS should be valid
- The iframe URL should be valid
- The `Grant` (permissions) request should be no more than is needed
- The `Application Type` should be correctly set
- Team visibility should be set (optional)
- The category should be appropriate
- The application description should be accurate
- The Sign-In URL should be valid
- The pricing summary should be accurate
    - i.e., trials should not be listed as `Free`; `Free for xx days and then $xx/month` is more accurate.
- All pay plans should have accurate descriptions.
- The support URL should point to a resource for help (the resource should NOT be an FAQ page, unless that page also contains one of the other options):
    - Support ticketing system (e.g., Zendesk, Jira, etc.)
    - Web page with a telephone number
    - Web page with an email address
    - Forum
- The EULA link should point to an English Language EULA.

Quality Considerations

Core App Quality

Onshape users expect high-quality apps. App quality directly influences the long-term success of your app in terms of installs, user rating and reviews, engagement, and user retention.

This page helps you assess the core aspects of quality in your app, through a compact set of quality criteria and associated tests. All Onshape apps should meet these criteria.

Before publishing your apps, test them against these criteria to ensure that they function well. Your testing should go well beyond what's described here; the purpose of this page is to specify the essential quality characteristics all apps should display, so that you can cover them in your test plans.

Functionality

These criteria ensure that your app provides the expected functional behavior, with the appropriate level of permissions.

Area	Description
Permissions	The app requests only the *absolute minimum* permissions that it needs to support core functionality.

Compatibility, Performance, and Stability

These criteria ensure that apps provide the compatibility, performance, stability, and responsiveness expected by users.

Area	Description
Stability	The app does not crash, force close, freeze, or otherwise function abnormally.
Performance	The app loads quickly or provides onscreen feedback to the user (e.g., a progress indicator or similar cue) if the app takes longer than two (2) seconds to load.
Visual quality	The app displays graphics, text, images, and other UI elements without noticeable distortion, blurring, or pixelation.

Security

These criteria ensure that apps handle user data and personal information safely.

Area	Description
Data	All private data is stored in the app's internal storage. All data from external storage is verified before being accessed. No personal or sensitive user data is logged to the system or app-specific log.
Networking	All network traffic is sent over SSL.

Onshape App Store

These criteria ensure that your apps are ready to publish on Onshape App Store.

Area	Description
App Details page	The app's feature graphic follows guidelines such as: - The app listing includes a high-quality feature graphic. - The feature graphic does not resemble an advertisement. - The app's screenshots or videos do not represent the content and experience of your app in a misleading way.
User support	User-reported bugs are addressed if they are reproducible.

Test procedures

These test procedures help you discover various types of quality issues in your app. You can combine the tests or integrate groups of tests together in your own test plans. See the sections above for references that associate criteria with these test procedures.

Type	Description
Core suite	Navigate to all parts of the app: all screens, dialogs, settings, and all user flows.
Security	- Review all data stored in external storage. - Review how data loaded from external storage is handled and processed.