SMARTLING

Smartling API Integrations: Best practices and considerations

The purpose of this document is to provide a comprehensive guide to best practices and key considerations for designing API integrations. It should be referenced throughout the design and development process to help you design reliable, scalable, and maintainable solutions.

Table of Contents

File Considerations

File Type(s)

Smartling supports many file types. Some integrations will naturally lead toward one file type or another based on the options readily available in the platform. If creating the files from scratch or given the choice, we recommend using JSON file type as it provides the most flexibility with supported directives.

File Naming Conventions

File names (fileUri) in Smartling are searchable, so it is important to define a naming convention that will easily enable users to find content in Smartling.

File names should include

- User friendly name for the asset (may need to be trimmed). This could be the title of the asset or other attribute that is visible to the end user.
- (optionally) an asset type
- Unique identifier for the asset

Example: The Best Surfboard-PRODUCT-123563.json

For platforms that use a path or tree structure:

- Asset path (may need to be trimmed)
- Asset ID

Example: `<branchName>/content/markening_site/product/features/en.json`
In these types of structures the asset itself is often not unique, so including the entire path ensures uniqueness. It's also important to ensure that if trimming of the path does occur that it doesn't remove the most relevant part of the path (i.e. the file/asset name).

For this structure, consider how namespaces should be determined. For example, if string sharing is desired for the same file in different branches or other variable part of the file uri. See Namespaces for Path based structures for additional details.

Keep in mind:

- Maximum fileUri is 512 characters. Some parts may need to be trimmed especially if the file path is part of the fileUri
- All UTF-8 characters are supported, including extended unicode symbols and emojis
- fileUri values are case-insensitive

File Contents

As a best practice **Smartling recommends sending the entire set of keys for an asset every time that translation is requested** as opposed to only sending changed or updated content (i.e. delta content). Smartling will detect if strings have changed or are new. Anything strings that are unchanged will remain unaffected in the platform. This approach allows for better tracking of strings in the platform.
Otherwise, if delta files are sent they need to be uniquely named so as to not remove strings that were previously sent for translation. This makes it difficult to track changes to existing strings, instead creating multiple copies.

HTML Content

We recommend the integration provide a way to identify which strings/keys contain HTML content. Smarting will parse these strings as HTML, breaking them into smaller translation units which allows for better TM leverage. When sending the content to Smartling use the **strings_format_paths** directive to identify keys containing HTML.

Sample JSON File structure

Provided below is a sample JSON file structure. While Smartling is very flexible and can accommodate a wide variety of file structures, formatting the file in specific ways can make it easier to identify which keys need to be translated and other parsing requirements.

```
{
   "smartling": {
       "variants_enabled": "true",
       "translate_paths": [
           {
               "path": "/values/*/text",
               "key": "/values/*/variant",
               "instruction": "/*/instruction",
               "character_limit": "/*/character_limit"
           }
       ],
       "string_format_paths": "html: [values/htmlValue], txt: [values/textValue]"
   },
   "values": [
       {
           "key": "store.checkout.key1",
           "textValue": {
```

```
                "text": "A discount has been applied to this order. You can't add
another discount.",
                "variant": "store.checkout.key1",
                "instruction": "these are the instructions",
                "character_limit": 50
            }
        },
        {
            "htmlValue": {
                "text": "<p><strong style=\"text-transform:uppercase\">%{code}</strong>
discount code isn't valid for the items in your cart</p><p>Please enter a new
code</p>",
                "variant": "store.checkout.key2",
                "instruction": "these are some other instructions",
                "character_limit": 100
            },
            "key": "sotre.checkout.key2"
        }
    ]
}
```

Directives used for this example. These directives can either be specified inline (most directives) as shown in the sample above. Alternatively, they can be specified through the API. See documentation for details on each directive. Note, the directives shown here are specific to JSON. Similar directives are available for some (but not all) other file types.

- string_format_paths: "html: [values/htmlValue], txt: [values/textValue]"
    - Tells Smarting which paths should be parsed as which format. Useful for enabling HTML parsing.
- translate_paths: "[{\"path\":\"/values/*/text\",\"key\":\"/values/*/variant\",\"instruction\":\"/*/instruction\",\"character_limit\":\"/*/character_limit\"}]"
    - Gives the path to the translatable strings, keys, and instructions
    - Optionally, you can specify a character limit
- variants_enabled: "true"
    - Determines whether variants are used
- namespace
    - This directive can only be set through the API

- See Namespaces and Variants section for further details.

Content Parsing

Directives

Content parsing in Smartling is controlled using directives. Specific directives are noted throughout this document. The directives that are available are based on the specific file type used for the integration.

Notranslate Fields

A common use case for an API integration is that certain fields in the source system should not be translated and therefore not sent to Smartling.

The integration must take into account:

- Fields/attributes that should never be translated.
  - Example: Fields that contain urls, fields that contain image names, fields that contain ids, etc.
- Fields/attributes that the user should be allowed to configure whether they should be translated
  - The integration should provide a way for the user to identify fields that should be excluded from translation. For example, a freeform text area.

Fields that should not be translated can either:

1. Recommended: Included in the file sent to Smartling but with a path that is different from what is looked for in the translate_paths directive. Therefore, the strings will not be ingested.
2. Not be included in the files sent to Smartling.
3. In the case of HTML content, notranslate tags can be used.

Placeholders

Another common use case is when a string that should not be translated is embedded in a larger string which does require translation. This is common when a variable is used in part of a larger string.
In this case there are two ways to handle:

1) Non-programatically through Glossary entries. This fits some use cases where the set of phrases not to translate are well known and can be clearly defined. While this approach does not require any development effort, it is not a good fit for all use cases.

This can be useful when end users are generating the content and may not be able to always identify which content in the string should not be translated. For example, someone is composing a description of a restaurant, “Welcome to Restaurant ABC….”. The name of the restaurant should not be translated but it’s contained in a larger block of text. In this case glossary entries may be a good approach.

2) By using placeholders. This is required for cases where variables are being used.

Examples:

Source String	Example Placeholder String
You will love Product A	You will love {{productName}}
You will love Product B	
Buy 2 more to receive $100 off	Buy {{number}} more to receive {{amount}} off
Buy 1 more to receive $50 off	

An added benefit of using placeholders is that it can reduce the amount of strings requiring translation by collapsing multiple strings into a single string.

The integration should consider the data that is sent for translation.

- Where should placeholders be included?
- Is there a need to allow the content creator to specify when to use placeholders? How can this be achieved technically?
- What placeholder format(s) will be used?

Refer to file type documentation for information on what placeholder formats are supported OOTB for each file type.

Custom placeholders

Sometimes there is a need for the integration to allow a configuration option for custom placeholders. This is typically a plain text field where an admin can enter the custom placeholder format. This would provide the ability to allow for custom placeholder format(s) that could be easily modified based on business requirements. These should be sent to Smartling using the custom placeholder directive.

Keep in mind that for the JSON file format, adding a custom placeholder format will override the OOTB placeholder formats. Therefore, to support both the custom placeholder format expression must be inclusive of any needed OOTB formats as well.

Namespaces & Variants

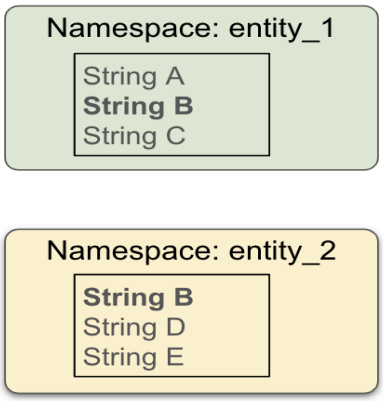
Smartling has a concept of string sharing where strings with the same text within the same namespace share a translation.

You can think of a Namespace as a container of strings. Using the API, you can define different namespaces (containers) of strings. By default, the fileUri is the namespace if none is specified.

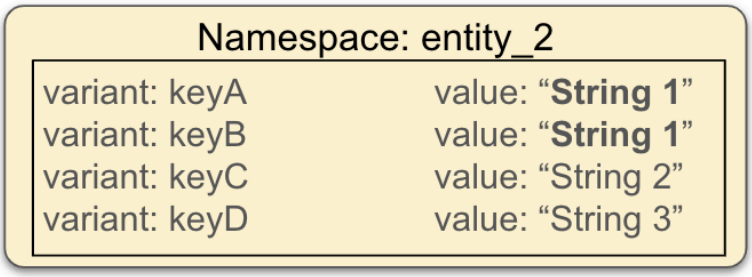
String Variants are an additional optional feature that can be used that allows for different translations of the same source text. Namespaces can be used with or without variants enabled.

At a high level, namespaces control separation of strings at the namespace level, while variants allow for separation of strings within a namespace.

No Variants



With Variants



It is important to note that the Smartmatch feature can be used across namespaces and variants, still allowing translations to be reused.

Approaches to Namespaces

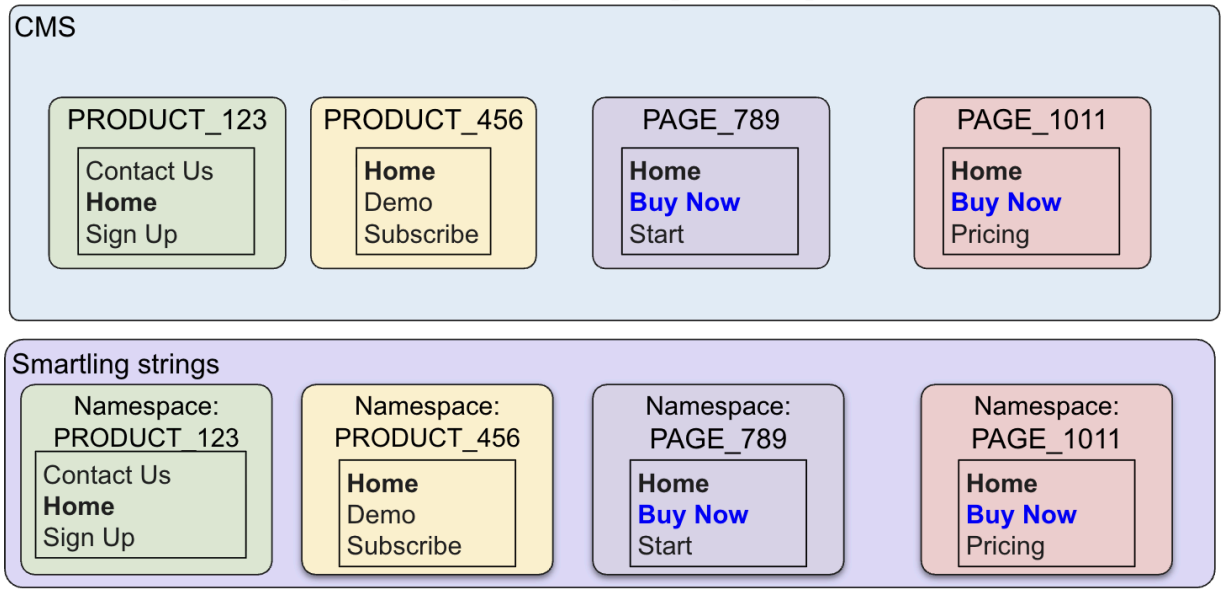
Decisions around file naming conventions and namespace creation are often closely tied together. To make this decision, the underlying data structure of the CMS should be considered. There are two general approaches that Smartling recommends:

Namespace Approach 1: Namespaces for each unique asset

This approach generally works best for systems that store strings in assets that can be uniquely identified with an id. A namespace is created for each unique asset and includes the ID of that asset. Do not include user supplied attributes or attributes that could change over time, such as a name or title.

Each namespace can be assigned to one or more files.

Examples: PRODUCT_123, PRODUCT_456, PAGE_789, PAGE_1011



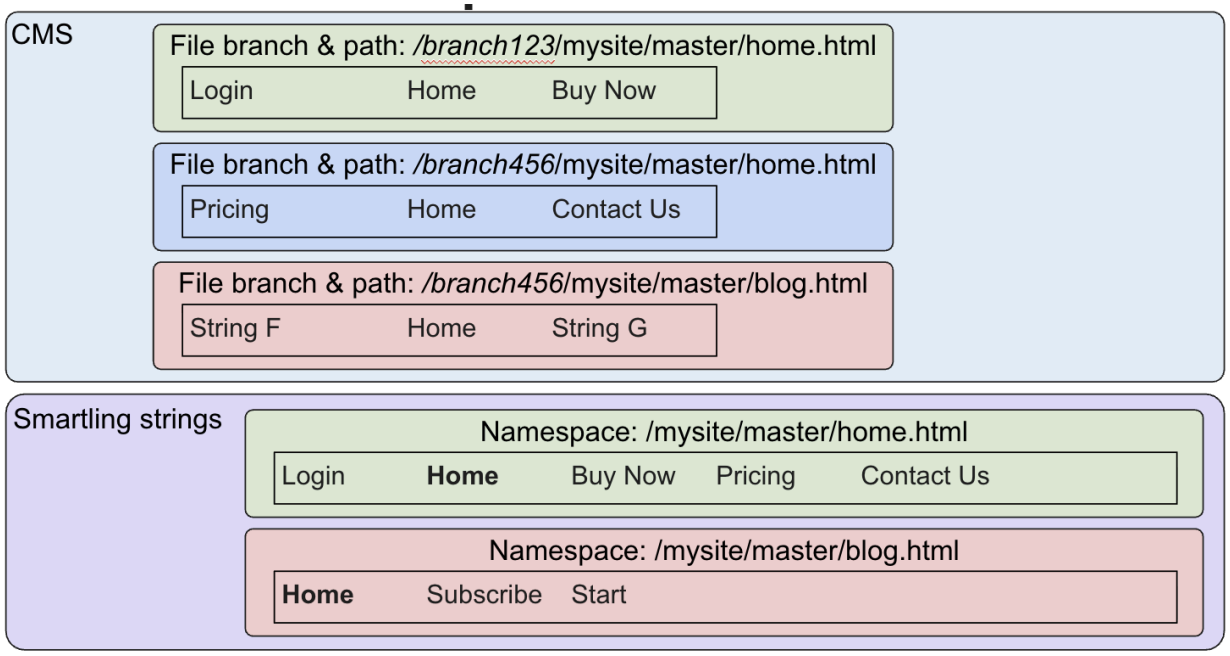
Namespace Approach 2: Files based

This approach generally works best for systems that store strings in path based assets. These systems often contain branches or other unique identifiers to distinguish versioning between assets.
When using the file path as part of the fileUri naming convention, it is typically recommended to remove the branch (or other versioning identifier) from the path when specifying the namespace.

Consider the following example where two versions of a file exist in different branches:

`<branch123>/content/markening_site/product/features/en.json`
`<branch456>/content/markening_site/product/features/en.json`

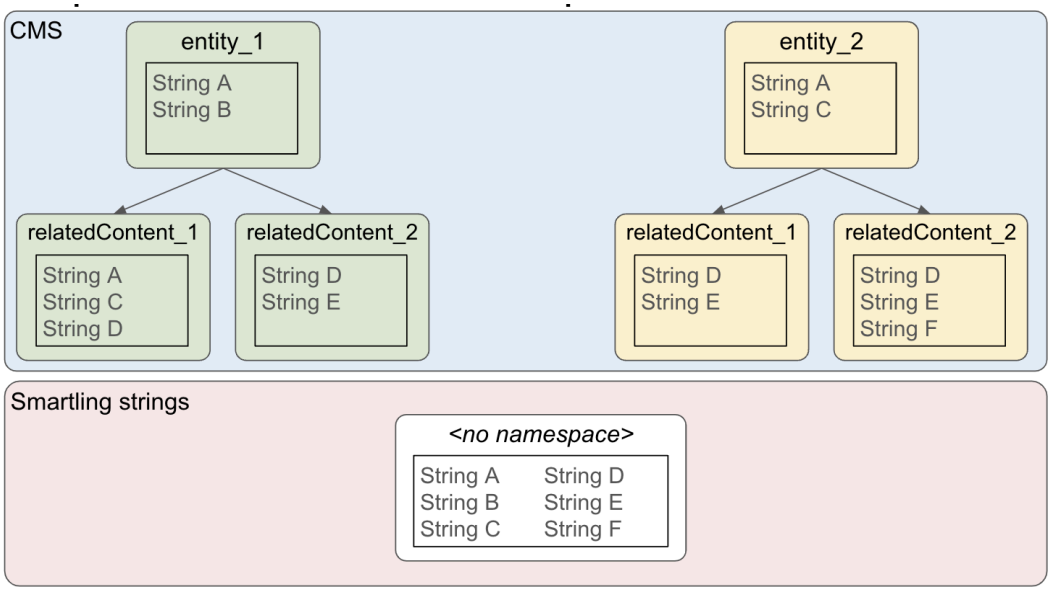
In this case, it is recommended to share strings between both versions of the file. The namespace could be set to exclude the specific branch:
`/content/markening_site/product/features/en.json`



Smartling uses this convention in the GitHub connector. You can see another example here.

No namespaces (not recommended)

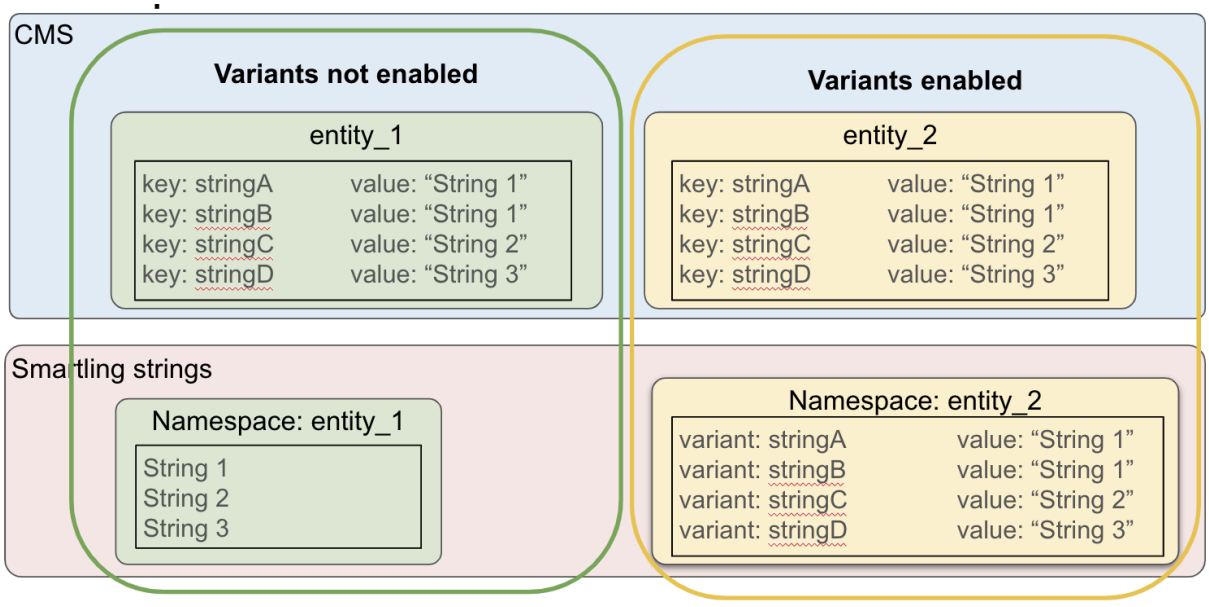
In this approach, all strings are shared across all files. This approach is **not recommended** as it does not allow adequate separation of strings to be translated differently depending on the context. This is included here as a warning.



Variants

If the source content has keys, these are often used as string variants. Variants should be enabled when there is a need to separate translations for the same string within a file. Variants can be used with or without namespaces. An additional benefit is that keys/variants are searchable in Smartling

Variants should not be used in the place of string instructions for the translator. Instead use string instructions for this purpose.



API Flow

Smartling offers multiple APIs which can be used to get strings into Smartling for translation.

Job Batches

As a general best practice, we recommend the Job batches API flow for the majority of integration use cases. Job Batches take care of asynchronous processes when uploading files and attaching them to jobs.

An alternative approach is using the Files API plus the Jobs API without Job Batches. In this approach, adding multiple files to a job requires polling at each step. The file is parsed asynchronously and can't be added to a job until it is completed. Adding a file to a job is also asynchronous, and the job can't be authorized until all background tasks are completed. This forces a third-party integration poll for every file after every step.

Job Batches hide these asynchronous processes from 3rd party integration. After it accepts the file, it processes this file through all steps up to the Job authorization. As the Job Batches Service utilizes internal Smartling events it is more efficient than just polling files and jobs.

You can find a sample postman collection here.

Strings API + Jobs

It is also possible to upload strings directly into Smartling using the Strings API and then add them to a job. While this approach is possible, for the majority of use cases the job batches API flow outlined above is a better fit and considered a best practice. One exception is if the strings are being stored in a database. In this case, it may not make sense to try to structure the strings into a file.

The Strings API has several limitations including:

- Limited directive support in the Strings API
  - As an example, you cannot control entity escaping if a string is parsed as HTML
  - ICU format is not supported
- The Strings API does not allow content to be easily managed in the dashboard because strings are not tied together as they are in a file
  - No guarantee of ordering of strings
  - No ability to easily download from the dashboard since they do not live in a file
- No way to delete or modify strings if the source content changes
- Job batches API has better performance and doesn't require polling before adding strings to a job
- No ability to include additional metadata with strings API as you can in a file

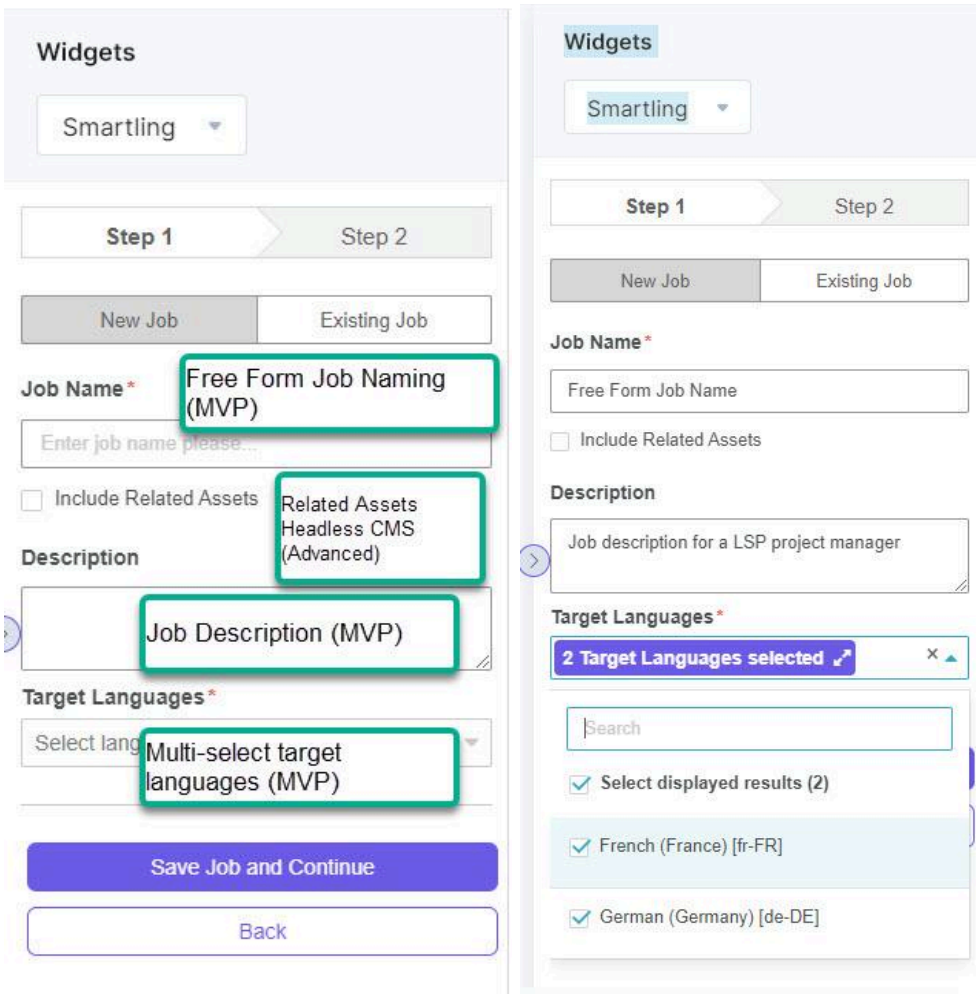
        - When the file is used as context, inline comments and metadata can serve as additional context
    - Callbacks are supported, a separate callback will be generated for each string unless job level callbacks are used. Therefore, the frequency will be increased.
    - File rewrites are not supported since there is no file

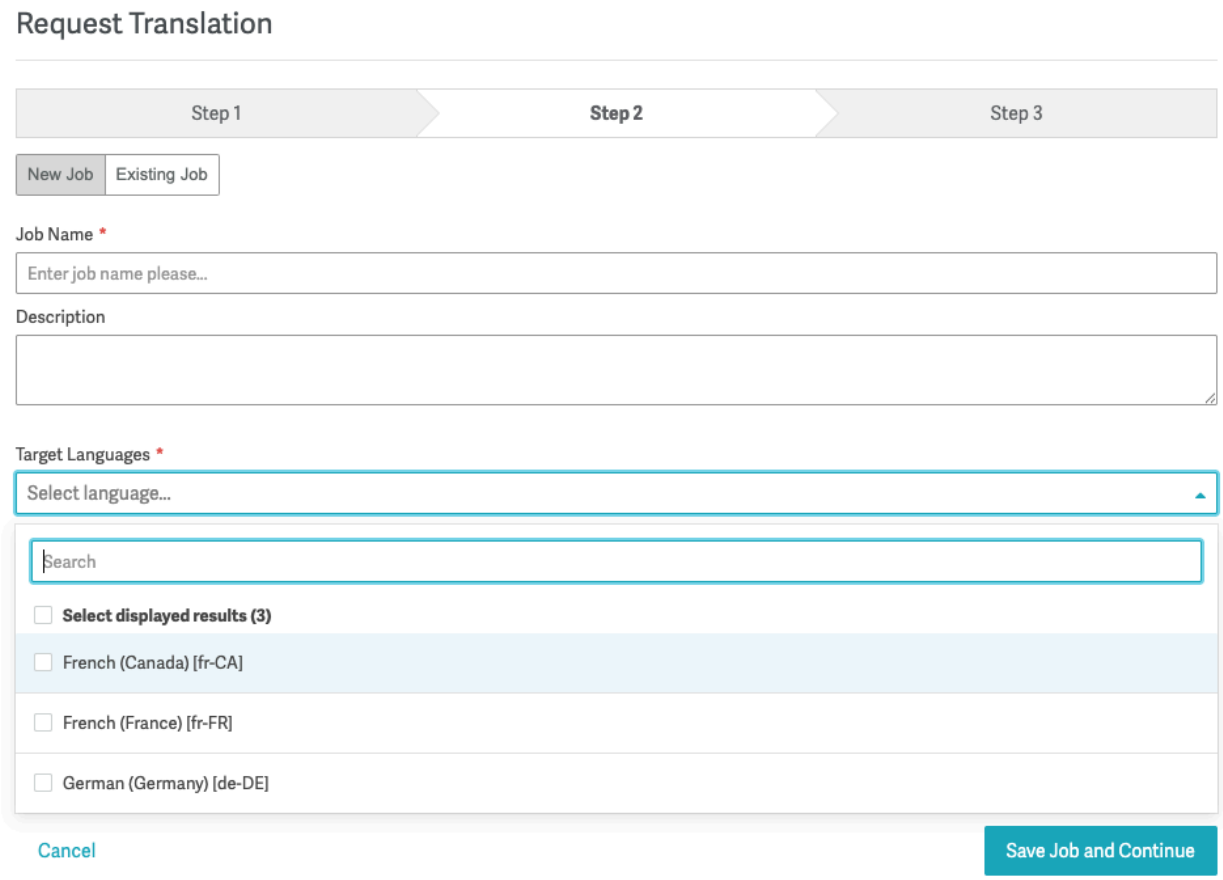
Translation Request Process

The integration should carefully consider: *What triggers content to be sent to Smartling for translation?*

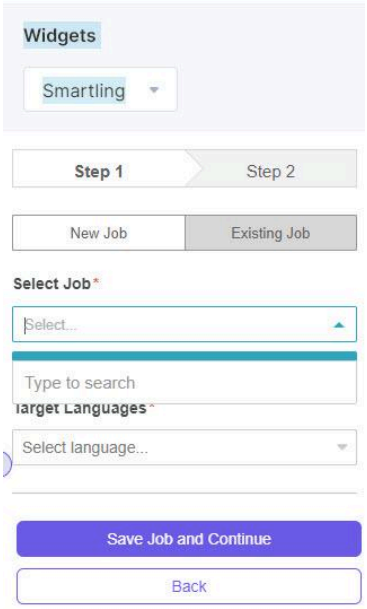
- Is this automated or manually requested by the user?
- If Automated:
    - Consider how to ensure content is “complete” before triggering translation.
    - Consider having a manual override mechanism to ensure that content can be requested on demand in case of an urgent request or issue resolution.
- If Manual:
    - Consider what the UI looks like for requesting translation.
    - What should the user be allowed to override?
    - Should a user be allowed to add content to an existing job?

UI Example - Create New Job


Widgets
Smartling
Step 1
Step 2
New Job
Existing Job
Job Name*
Free Form Job Naming (MVP)
Enter job name please...
Include Related Assets
Related Assets Headless CMS (Advanced)
Description
Job Description (MVP)
Target Languages*
Select lang
Multi-select target languages (MVP)
Save Job and Continue
Back
Widgets
Smartling
Step 1
Step 2
New Job
Existing Job
Job Name*
Free Form Job Name
Include Related Assets
Description
Job description for a LSP project manager
Target Languages*
2 Target Languages selected
Search
Select displayed results (2)
French (France) [fr-FR]
German (Germany) [de-DE]



UI Example - Add to Existing Job



Smartling Job Creation

The integration should consider how and when Smartling jobs will be created.

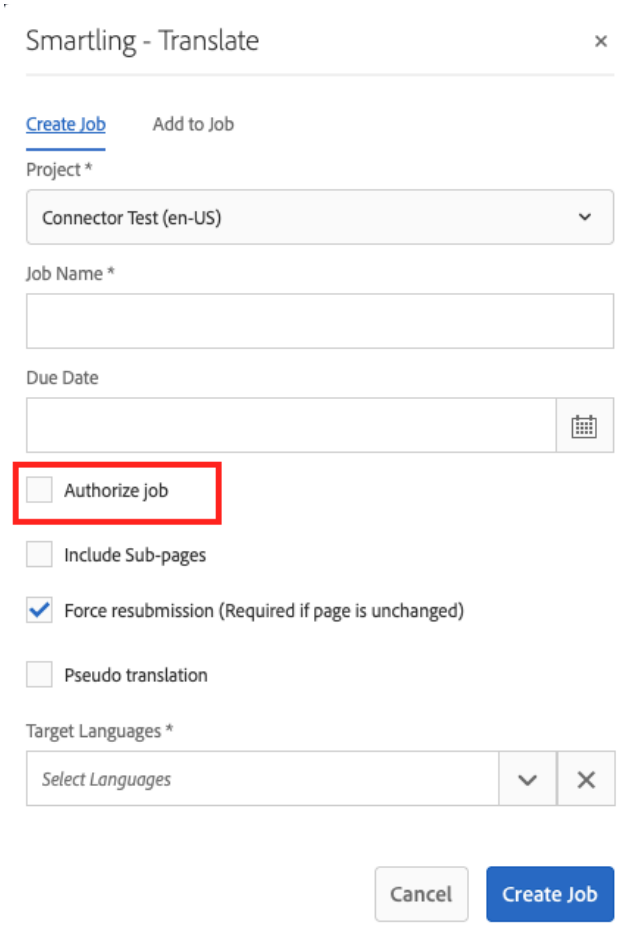
- If using the manual option for Translation Request process above: Typically the users are allowed to indicate a job name.
    - Determine if there's a need for users to add content to an existing job
- If using the automated translation request process: consider how frequently jobs should be created in Smartling.
    - "Real time" - depending on expected volume this could result in a lot of small batches. Alternatively, you can add new content to an existing job for a given timeframe (i.e. Daily job).
    - Batched / schedule (what's the schedule and how can it be changed or overridden)
- The only requirement is that job names must be unique in Smartling.
    - If the user tries to indicate a job name that is not unique an error message should be displayed to the user.
    - If jobs are created automatically define a naming convention that will result in unique job names.

Authorization

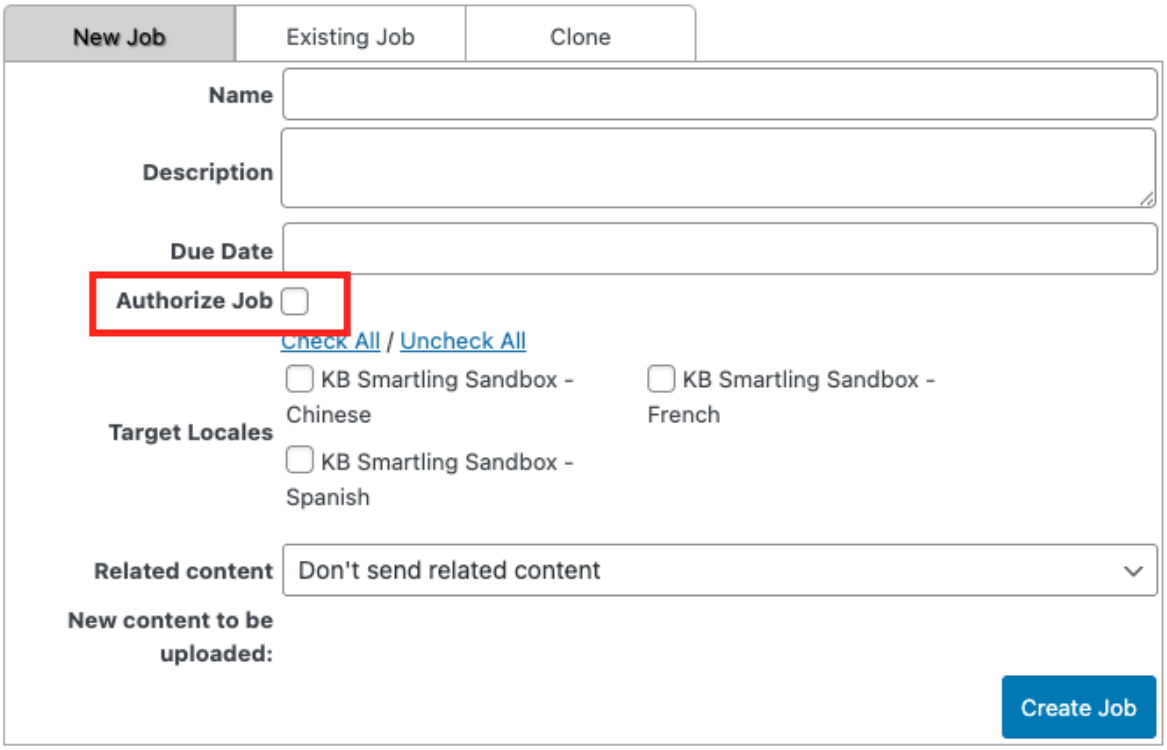
The integration should carefully consider: *How should content be authorized in Smartling?*

- Auto-authorization: Jobs are automatically authorized in Smartling.
    - This can be done using the authorize parameter on the Create Job Batch API or the Authorize Job endpoint.
    - Content will follow the default workflow unless overridden in the API.
- Manual authorization
    - Jobs will be manually reviewed in Smartling prior to authorization.
    - Users can override the default workflow at the time of authorization.
- Allow user to choose
    - If users are manually requesting content, in the request UI dialog, allow users to indicate whether the content should be automatically authorized. This is only applicable for the manual translation request process.
    - What will the default value be? Is there a configuration option to control the default?

UI Examples:


Smartling - Translate
Create Job
Add to Job
Project *
Connector Test (en-US)
Job Name *
Due Date
Authorize job
Include Sub-pages
Force resubmission (Required if page is unchanged)
Pseudo translation
Target Languages *
Select Languages
Cancel
Create Job

Smartling Upload Widget

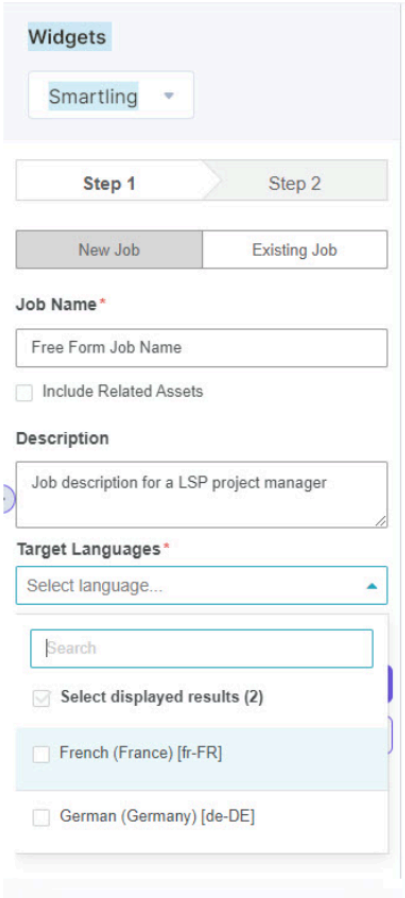


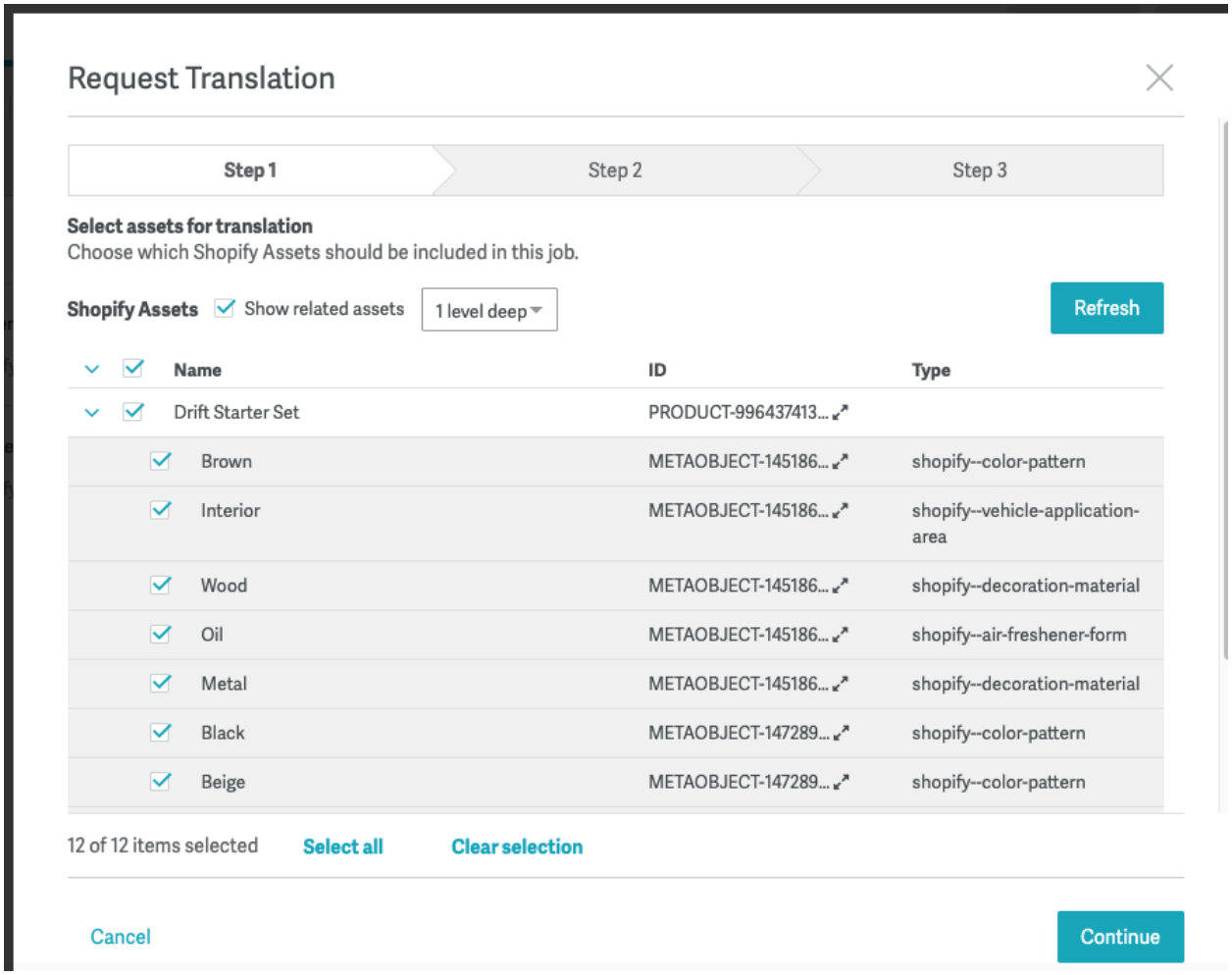
Related and Child assets

The integration should consider the entity structure of assets in the system and how assets may be related to each other.

- Child assets: In a hierarchical or path based structure, users may want to submit an asset and any children of that asset at the same time.
    - When submitting `/content/top_path/page1` allow for `/content/top_path/subfolder/childPage` to also be submitted as well.
    - Consider what the default behavior should be.
    - Consider how to allow the user to change this behavior as needed
- Related assets: Similarly, sometimes assets are related to each other that are required for the asset to be fully translated. Users may want to submit an asset and any related assets at the same time without having to individually request related assets.
    - Consider what the default behavior should be.
    - Consider how to allow the user to change this behavior as needed

UI Example


Widgets
Smartling
Step 1
Step 2
New Job
Existing Job
Job Name*
Free Form Job Name
Include Related Assets
Description
Job description for a LSP project manager
Target Languages*
Select language...
Search
Select displayed results (2)
French (France) [fr-FR]
German (Germany) [de-DE]



Translation Download Process

The integration should consider:
When should translations be downloaded?

- Pending: All saved translations, regardless of where it resides in the workflow
  - This can be be done only through a polling mechanism and setting retrievalType to pending on file download
  - Generally this is not recommended for Production usage as it can download translations that are incomplete.
- Published: (Recommended) Translations that have completed the entire workflow
  - Can be done through polling or callback
- Pre-published: Translations that are not fully published but have been configured for pre-publishing.
  - See Prepublishing with File Downloads and Callback on Prepublish for technical details

    - This is a good option when you are using a workflow that has human review or edit steps that can take time but it is important to get some translation available as soon as possible. In this case, pre-publishing makes a “first draft” of the translation available as soon as possible while still allowing for the final review to occur and translation to be updated.
- Pseudo: Pseudo translations are simulated translations, useful for testing purposes
    - This can be be done only through a polling mechanism and and set the retrievalType to pseudo on file download
- Is this a configurable option for the integration? Typical options would be Pending, Published, and Pseudo. If allowing for pseudo translation, consider where in the UI to add an option for this, for example on the Request Translation dialog. Pre-publishing is configured in Smartling.

What triggers translations to download? There are two mechanisms to monitor for translation status:

- Polling: This involved periodically checking Smartling for status updates. You can check the progress at the job level or at the file level. Generally, we recommend checking progress at the file level
    - See Checking File Translation Status for details around options for file level polling
    - What schedule will the integration poll Smartling? How will this be managed/updated? Any override option to force download of a particular asset?
- Callbacks: Callbacks allow for a more immediate notification when translations are ready, as opposed to a time-delay that is introduced with the polling option.
    - Callbacks can be set at the file and job level. Consider which webhook notifications your integration needs. We recommend using file level callbacks at a minimum. This ensures that the integration is notified every time a file is published or re-published.
    - While webhooks are retried, consider having a backup polling or manual download mechanism in case of a failure receiving the webhook.

Content updates

There are two types of content changes the integration should consider.

- Changes to source content: When changes occur to the source content, how will the updated content be resubmitted for translation?
    - **Manually**: Content will be manually re-submitted for translation following the same translation request process as for new content.

    - **Automatically:** Updated content for assets that have been previously submitted for translation will automatically be detected and sent to Smartling. The same considerations discussed in Smartling job creation should be applied.
    - Smartling recommends sending the entire set of strings when a file is updated. New or updated strings will automatically be detected. Deleted strings will be removed.
- Changes to translations: Changes to translations after a file has been published can occur if issues are found with translations that need to be corrected or if a preferred translation is desired. In these cases, we recommend updating the translation directly in Smartling rather than in the application. This is a best practice to ensure that it is saved to the Translation Memory. Additionally, in the future if the same asset is requested again, any changes made in Smartling will not be overwritten.
    - To ensure the integration is notified of changes to translations, it is recommended to implement callbacks as described in the Translation Download process. The file level callback will trigger when a translation is updated and the file is re-published.
    - If the integration uses a polling mechanism only, the only available method is to monitor the List Recently Published Files endpoint.

Context

Visual Context is an important consideration for the integration especially when humans are involved in the translation workflow. Visual context provides a visual representation of the source content to Translators, Editors, and Internal Reviewers as they translate the content.

Consider what the best option for sending visual context to Smartling is. The two main mechanisms include the Smartling Context API (recommended if possible) and the Smartling Context Javascript library.

Smartling Context API

The Smartling Context API is the preferred mechanism for integrations to use for Visual Context when possible. As part of the translation request process, if the platform can generate a preview or snapshot of the rendered content, it can be sent to Smartling without any additional user interaction required.

Considerations:

- Always upload the file first before uploading the context. Strings must be present in Smartling before they can be bound to context.

- You can control whether you want to bind the context to specific files, hashcodes (strings), or jobs. Often it is advisable to limit where the context can be bound to only the recently uploaded content.
- You can override the context for strings already bound older than a specified number of days.

Smartling Context Javascript Library

For platforms that have a web interface, the Smartling Context Javascript Library can be a viable, low code option.

Considerations:

- The Javascript library is not a fully automated solution. The site still needs to be browsed to send context to Smartling.
- Careful consideration to sensitive or personal data should be used before enabling the library on a production site. It's generally advisable to use the library in a lower environment, however if this does not mirror production it is not always feasible.

Other Considerations

Locale Mapping

Locales in the platform may not match locales in Smartling. Alternatively, clients may wish to use a different locale in the platform than is used for translation in Smartling. For example, they may wish to use the 2 letter locale code in the platform ("fr") and the 4 letter code in Smartling ("fr-fr").

The integration should provide a way to map the platform locales to Smartling locales. This mapping may be prepopulated with default information, but should allow for users to update based on business needs.

Smartling Workflow

In Smartling a workflow controls the specific translation type and steps that content flows through. Each Smartling project has a default workflow configured. However, it is common for specific content to need to be sent to a different workflow.

This can be done either manually or through the API:

- Manual option: The user can manually change the workflow in Smartling if they authorize the job in Smartling. This is part of the authorization dialog.

- Automatically: The integration can override the workflow for a by using the `localeWorkflows` parameter on the Create Job Batch API call.
    - Consider how you want to allow the user to override the workflow. Is this a UI option? A backend configuration based on the type of content?

Error Handling

Checking for errors is crucial to a robust integration. It's important to distinguish which errors are recoverable and should be retried. See retries for more details.

Take note of the possible success and error codes.

Of particular note, are 429 rate limiting errors. Smartling enforces rate limiting. We recommend the integration implements an exponential backoff algorithm to retry the API call when a 429 error is encountered.

Logging

The integration should log the following details specific to the Smartling API *in addition to any application-specific error messages*

- **Error type and code:** HTTP status code as well as Smartling error code and error key if available; or the type of network error, such as a timeout.
- **Error details:** Any additional details associated with the error. These can be found in the HTTP response, or in the exception.
- **Request & Response details:** Details of the request sent that resulted in this error response: URL, api parameters, and any additional context. Details of the response received that resulted in the error.
- Additional, non-error logging can be helpful here too in order to clarify what was happening in the application before the error occurred.
- **Request Tracking ID:** Smartling includes a request tracking number in a response header in the format *X-SL-RequestId: 369f4a6e-6f7f-4206-8a20-5485742a1d48*. Make sure to log this ID as it can help Smartling find the request in its logs.

Advanced Features

The following are considered to be features of more advanced integrations.

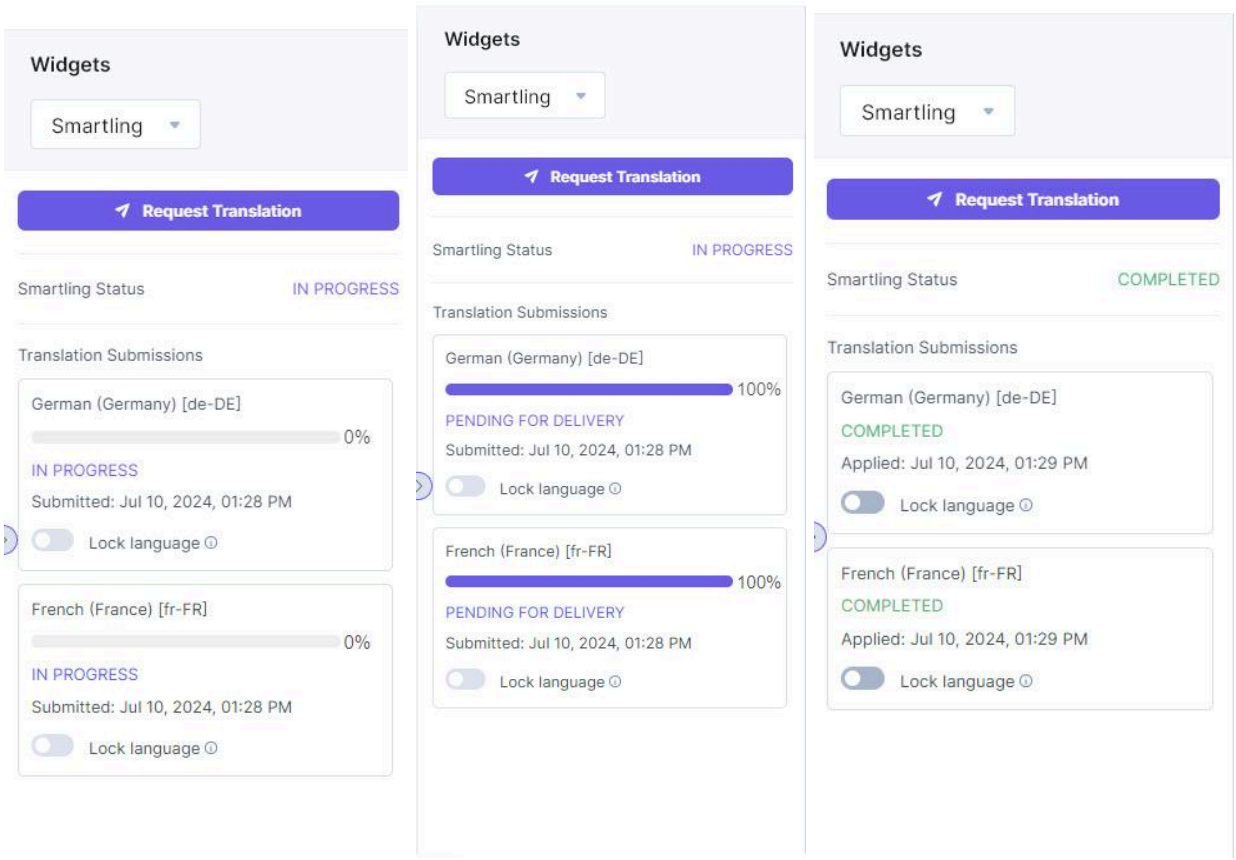
Show Translation Progress Status

It can be very helpful for content owners to have a place where they can view the translation status for a particular asset. This should include some indicator of which locale(s) were sent for translation and which were not requested for translation. Additionally, showing a percentage complete or tool tip indicating the number of words in each status (Awaiting Authorization, In Progress, Published, Excluded) can be very helpful.

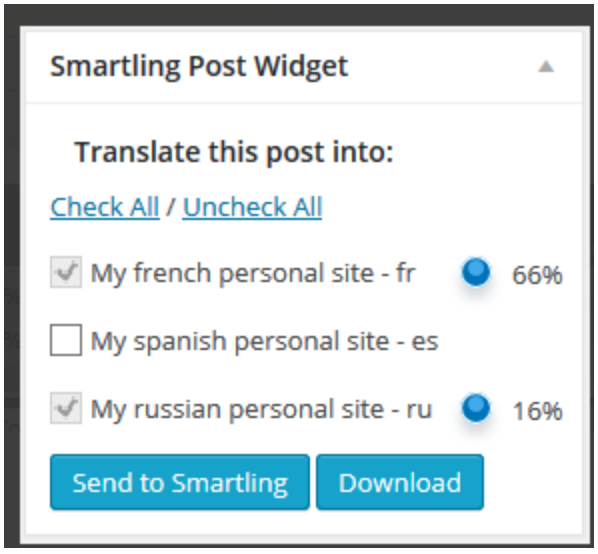
The ability to request translation and download the translation is often included in this dialog. The Request Translation (or “Send to Smartling”) is a way for the user to trigger the translation request process. The ability to download translations as a way for the user to force the delivery of translation even if not yet complete. This download delivers all pending translations.

Examples of UI:

Example 1 -



Example 2 -



Example 3 -

Multiple Projects

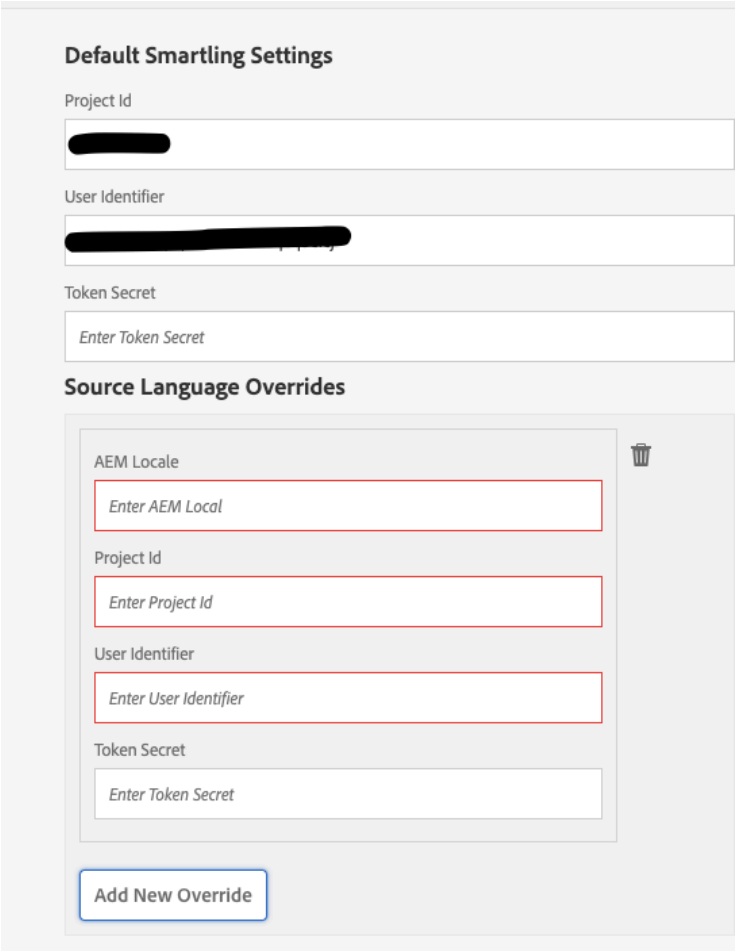
Occasionally it is necessary to send different sets of content to different Smartling projects. Here are a few reasons why this may be necessary:

- Different source language
- Very different types of content (i.e. marketing vs legal) means that different linguistic assets are required during translation

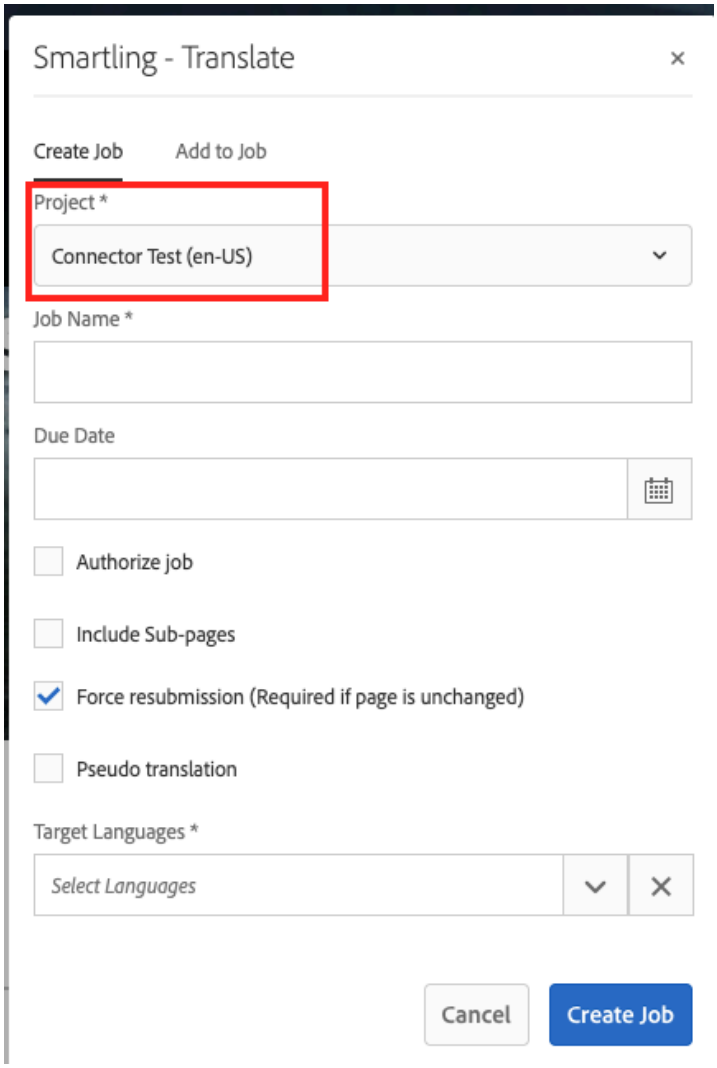
To support this advanced feature, the integration must have a way to allow the content to be sent to multiple Smartling projects. This means having the ability for an admin to configure multiple sets of API credentials and corresponding project IDs and also a way to specify which project should be used to translate which sets of content.

Project select could be done through the request translation UI or some backend config. In either case, it's important to have a default project selected to reduce unnecessary button clicks.

Example UI:
Configuration -



Request Translation Dialog -



Configurable Directives

Many different file level directives were discussed in the Content Parsing and Placeholders sections. It is important to consider both standard directives that will be OOTB with the integration as well as configurable directives that could be changed as needed.

Common cases for configurable directives:

- The most common use case for configurable directives is the ability to add custom placeholders. It is important to note which placeholder formats are included by default based on the file type the integration is using. See details for each file format for OOTB placeholders as well as the syntax for custom placeholders. For some file file formats, such as JSON, adding a custom placeholder format will override the OOTB placeholder

formats. Therefore, to support both the custom placeholder format expression must be inclusive of any needed OOTB formats as well.
- Specifying the content parsing strategy for each file type (i.e. plain text, html, icu, etc)

Third Party Integration Features

The following features are applicable to Smartling partners building their own integrations.

User Agent

The User Agent header is used to identify each integration and allows Smartling to track integration usage across clients. It is highly recommended that the integration include this header in all API requests.

Please use the following format for the User-Agent header field:

`integration-<integration name>-<integration product> / <version>`

Example: `integration-Smartling-Braze-integration/1.0`