Home

Burrito Store

Terminology

A burrito store is a mechanism for storing a collection of entries.

An entry is an ordered collection of revisions. Within a given store, each entry has the status of owned, delegated or loaned. The status of an entry representing the same burrito may vary between burrito stores.

A burrito store may be used by an id server. An id server provides the main id for an entry. Entry/revisions may be owned by zero or one id server.

An owned entry is an entry whose latest revision is authenticated by the id server which uses the burrito store.

A delegated entry is an entry whose latest revision is not authenticated by the id server which uses the burrito store, but to which editing rights have been delegated by an id server.

A loaned entry is an entry whose latest revision is not authenticated by the id server which uses the burrito store, and to which no editing rights have been delegated by an id server.

A revision is an unordered collection of variants containing metadata which refers, optionally, to ingredients.

A variant is one set of metadata plus optional ingredients for an entry/revision. It must be either a standard variant or a derived variant.

A standard variant is one of

- default: the base form of an entry/revision. The metadata and ingredients of this variant are immutable. If the entry/revision is owned, all ingredients must be present. If the entry/revision is delegated or loaned, ingredients may be added or removed, but must correspond to the immutable metadata.
- new: a draft for an entry for which no previous revisions exist. Metadata may be edited. Ingredients may be added, removed and updated.
- update: a draft that is based on the default variant of a revision of an existing entry. Metadata may be edited. Ingredients may be added, removed and updated.

A derived variant is generated from default according to a processor-dependant algorithm. The metadata and ingredients of this variant are immutable. No other variants may be created from a derived variant.

A template may be created from a default variant. A template contains some of the metadata of the default variant. It must not contain an entry or revision id or any other systemId.

Store Options

Burrito stores may appear in many different systems, such as

- a server that contains the definitive copies of entries (eg DBL)
- a client that edits entry/revisions (eg Paratext projects)
- an application that consumes published entries (eg "Paratext resources", YouVersion...)

A system may well handle different entries in different ways. For example, Paratext can edit owned projects but cannot directly edit Paratext resources.

Scripture Burrito can represent many different types of content, but any given system may only handle a subset of those types. Such constraints need to be implemented by the store.

Ideally, they should also be exposed transparently so that other stores can discover whether it will be possible to exchange content.

Necessary configuration options include:

acceptedVersion

If present, this provides a semantic versioning string, which may allow a range, eg "0.2.x". When absent, all versions are accepted.

acceptedFlavors

If present, this provides an array of flavor and/or flavorTypes, eg ": ["scriptureText", "gloss"] which means "accept the scriptureText flavor plus any gloss flavors". When absent, all flavors are accepted.

allowXFlavors

This provides a boolean. When false or absent, only x-flavors that are listed under acceptedFlavors are allowed. When true, x-flavors are accepted if acceptedFlavors is absent, or if acceptedFlavor includes the flavorType of the x-flavor.

ownedEntryIDServers

If present, this provides an array of idServer URLs for which a systemId must exist in the metadata in order for the burrito to be accepted as an owned entry. When absent, no specific systemId is required.

acceptedIDServers

If present, this provides an array of idServer URLs for which a systemId must exist in the metadata in order for the burrito to be accepted as a loaned entry.

creatableDerivedVariants

If present, this provides an array of derived variants that the store can create from a default variant. When absent, no derived variants may be created by the store.

acceptedDerivedVariants

If present, this provides an array of derived variants that are accepted by the system, eg ["noReferences"]. When absent, no derived variants are permitted.

validation

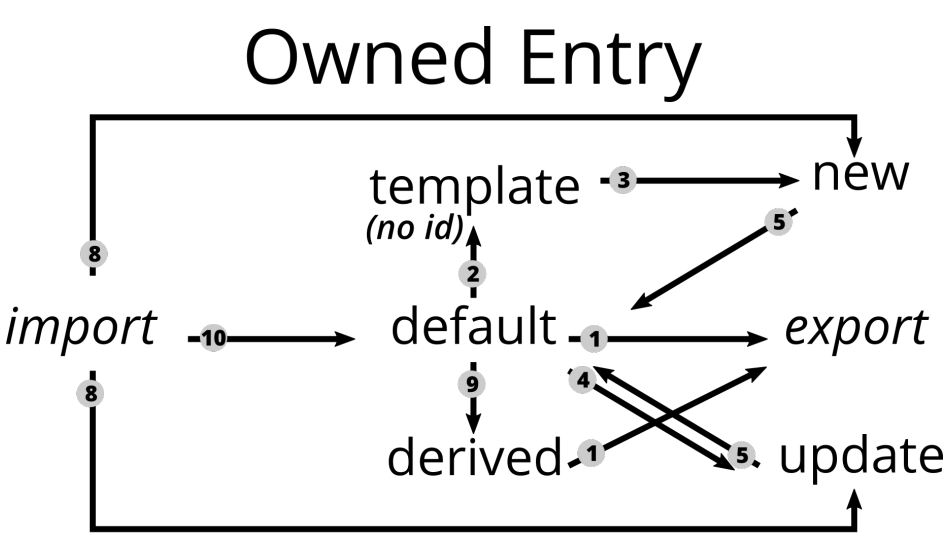
If present, this specifies the minimum level of validation that all entries must meet. When absent, the level is "catalog valid".

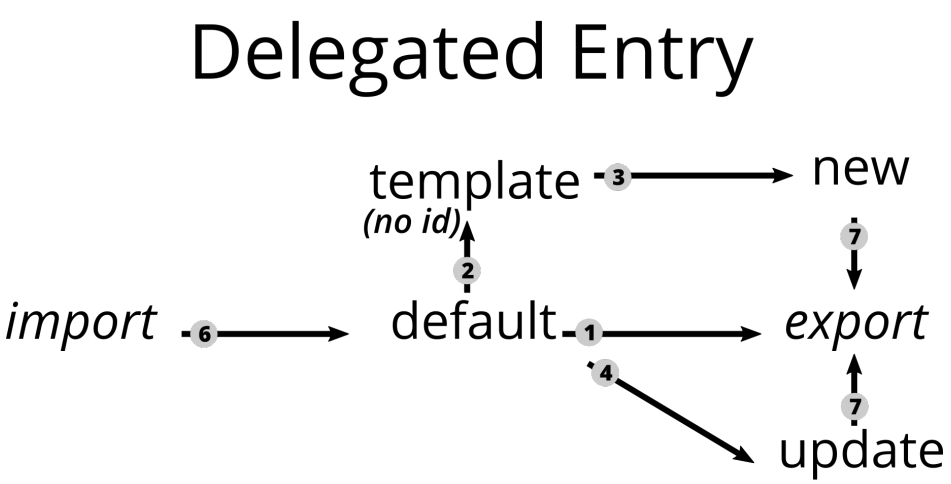
State Changes

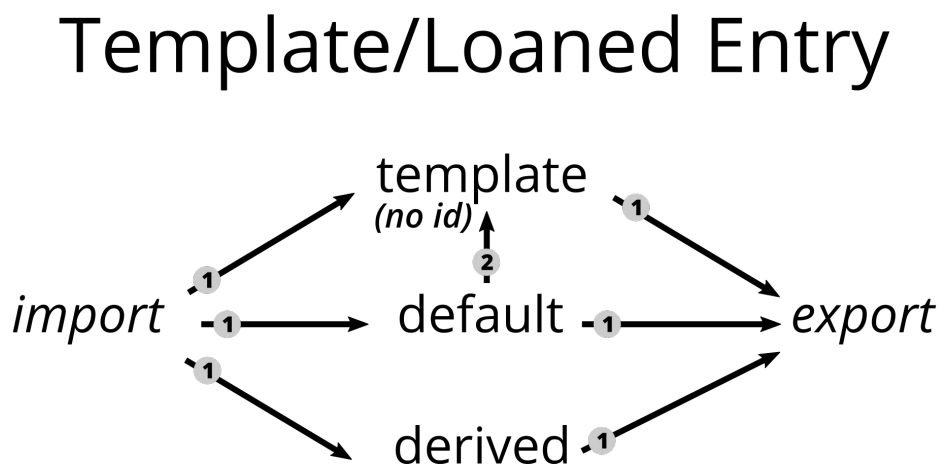
The diagram below shows how variants may be passed between burrito stores and transformed within burrito stores.

Processors may provide convenience "macros" that include several state changes, but the semantics of each state change must be respected. For example, a processor might provide a "default to new" operator, but such an operator should produce the same result as creating a template and then using it to create a new variant (state changes 2 and 3 below).

Any variant may be created in a "constructing" sub-state. This sub-state allows for non-instantaneous creation of variants, for example when the content of a default entry is being downloaded, or when the creation of a derived variant requires significant processing. variants in the constructing sub-state should not be accessible through the standard API.







1: share

The burrito is shared, without change, licenses permitting. This is an inter-store operation, so various transport formats are supported.

importFromObject(metadata)

importFromDir(path)

importFromZip(path)

exportToObject(idServerId, entryId, revisionId, variantId)

exportToDir(idServerId, entryId, revisionId, variantId, path)

exportToZip(idServerId, entryId, revisionId, variantId, path)

2: toTemplate

The burrito is stripped of ids for use as a template.

defaultToTemplate(idServerId, entryId, revisionId, templateData, filter?)

3: toNew

A draft new burrito (new id, no previous revision) is created, based on a template.

templateToNew(idServerId, templateName)

4: toUpdate

An existing burrito is used to create a draft update (same id, new revision).

defaultToUpdate(idServerId, entryId, revisionId)

5: acceptDraft

A draft is accepted as a new revision.

newToDefault(idServerId, entryId, revisionId)

updateToDefault(idServerId, entryId, revisionId)

6: receiveRevision

A new entry/revision is received from the owner.

- importFromObject(metadata)
- importFromDir(path)
- importFromZip(path)

7: sendDraft

A draft is sent to the owner.

- exportToObject(idServerId, entryId, revisionId, variantId)
- exportToDir(idServerId, entryId, revisionId, variantId, path)
- exportToZip(idServerId, entryId, revisionId, variantId, path)

8: receiveDraft

A draft is received by the owner from a delegated store.

- importFromObject(metadata)
- importFromDir(path)
- importFromZip(path)

9: makeDerived

A read-only derived variant is created.

defaultToDerived(idServerId, entryId, revisionId, derivativeName)

10: ownEntry

An entry created elsewhere is imported, with change of ownership.

importAsOwnedFromObject(metadata)

importAsOwnedFromDir(path)

importAsOwnedFromZip(path)

The possible state changes are summarized in the tables below.

Owned Entry

	default	template	derived	new	update
import	yes	no	no	yes	yes
export	yes	no	yes	no	no
to default	-	no	no	yes	yes
to template	yes	-	no	no	no
to derived	yes	no	-	no	no
to new	no	yes	no	-	no
to update	yes	no	no	no	-

Delegated Entry

	default	template	derived	new	update
import	yes	no	no	no	no
export	no	no	no	yes	yes
to default	-	no	no	no	no
to template	yes	-	no	no	no
to derived	no	no	-	no	no
to new	no	yes	no	-	no
to update	yes	no	no	no	-

Template / Loaned Entry

	default	template	derived	new	update
import	yes	yes	yes	no	no
export	yes	yes	yes	no	no
to default	-	no	no	no	no
to template	yes	-	no	no	no
to derived	no	no	-	no	no
to new	no	no	no	-	no
to update	no	no	no	no	-

Maintaining Entry State

In addition to "burrito" metadata, the burrito store requires the following state information about each entry:

Ownership

This is one of

- owned
- delegated
- loaned
- template

isConstructing

This boolean may be set during an async entry instantiation process. (In the case of an immutable variant, such as a derived variant or an imported burrito containing substantial ingredients, this process may take some time.)

Metadata schema and scope

Validation of the contents of each entry is vital to simplify processing and to maintain data integrity. Five schema are envisaged:

Template

A template is not strictly an entry, since it does not have an entry id. Templates, by their nature, are incomplete. However, whatever structure is present should be consistent with a metadata-valid document. The template schema also expects additional fields to identify the template, such as an id, name and description. (These should be stored differently to entry ids etc to avoid confusion when the template is instantiated.)

Catalog Valid

This is the lowest level of validity, which is required for all entries. Fields that are used for presenting summary information about the entry must be present and valid.

Archivist Valid

This level of validity includes catalog validity. All fields that can reasonably be human-maintained must be present and valid. (This notably excludes ingredients and publication structure.)

Metadata Valid

This level of validity includes detail validity. The entire metadata document must be valid.

Burrito Valid

This level of validity includes metadata validity. The entire metadata document must be valid and, in addition, the metadata and ingredients must conform to any conventions listed in the metadata. (This level of validation is only viable for owned entries, or for the first revision of a delegated entry, since, in other cases, the ingredients may not be present locally.)

Operations

List

These operations are suitable for producing table views, and can be used as the basis of iterative algorithms. In each case there is a keys-only function (which should be cheap), as well as a function that returns catalog-level information.

idServers()

Returns an array of idServer ids.

idServersDetails()

Returns an object with metadata for each idServer.

idServersEntries()

Returns the entry ids for each idServer.

entries(idServerId)

Returns an array of entry ids for the idServer.

entriesRevisions(idServerId)

Returns an object with the revision ids of each entry.

entryRevisions(idServerId, entryId)

Returns an array of revision ids.

entryRevisionsVariants(idServerId, entryId)

Returns an object with variant ids for each revision of the entry.

entryRevisionVariants(idServerId, entryId, revisionId)

Returns an array of variant ids.

entryRevisionVariantsDetails(idServerId, entryId, revisionId)

Returns an object with catalog information for each variant of the entry.

ingredients(idServerId, entryId, revisionId, variantId)

Returns a list of ingredient ids.

ingredientsDetails(idServerId, entryId, revisionId, variantId)

Returns an object with metadata for each ingredient.

Read Metadata

Metadata is treated differently to ingredients. It is assumed that the metadata is small enough to be returned conveniently as an object.

metadataContent(idServerId, entryId, revisionId, variantId)

Returns the metadata document as an object.

Read Ingredients

Various options are supported, some of which are, by necessity, implementation-dependent.

ingredientDetails(idServerId, entryId, revisionId, variantId, ingredientName)

Returns ingredient metadata (size, checksum, role, isSource, isCached) as an object.

ingredientContent(idServerId, entryId, revisionId, variantId, ingredientName)

Returns a stream-like object from which the (potentially large) ingredient content may be read.

ingredientLocation(idServerId, entryId, revisionId, variantId, ingredientName)

Returns a path to access the ingredient directly, eg from a local filesystem or S3.

Update Metadata

There is no "replace metadata completely" endpoint, since this would require a lot of checks to avoid esoteric data integrity issues. The metadata update model is based on the forms functionality behind DBL's Nathanael client. A form is an object containing schema-like information for each field (name, description, validation information) plus the current value. The filter specifies which portion of the metadata is referenced in an xpath-like way (mainly nested keys).

metadataForm(idServerId, entryId, revisionId, variantId, filter)

Returns a form for the variant.

submitMetadataForm(filter, form, minValidity=catalog)

Validates the form information and, if validation succeeds, updates the metadata.

Operation	default (owned)	default (not owned)	derived	new	update
metadataForm	no	no	no	yes	yes

Add, Update, Remove, Cache Ingredients

Operations exist to add and remove ingredients, either as a cache operation (when metadata is immutable) or as a modifying operation (when metadata is mutable and the intention is to change the metadata ingredients).

cacheIngredient(idServerId, entryId, revisionId, variantId, ingredientName, ingredientContent)

Make a local copy of an ingredient that exists in the owning store for this entry/revision . This is a no-op if the ingredient is already present.

uncacheIngredient(idServerId, entryId, revisionId, variantId, ingredientName)

Remove a local copy of an ingredient that exists in the owning store for this entry/revision . This is a no-op if the ingredient is not present.

addOrUpdateIngredient(idServerId, entryId, revisionId, variantId, ingredientName, ingredientContent)

Add or update an ingredient, changing the ingredients section of the metadata accordingly.

deleteIngredient(idServerId, entryId, revisionId, variantId, ingredientName)

Delete an ingredient, changing the ingredients section of the metadata accordingly.

Operation	default (owned)	default (not owned)	derived	new	update
cacheIngredient	no	yes	yes	no	no
uncacheIngredient	no	yes	yes	no	no
addOrUpdateIngredient	no	no	no	yes	yes
deleteIngredient	no	no	no	yes	yes

Validation

All data within a store should always be catalog valid. It is possible to validate against stricter schema, and to check for convention conformance.

validateMetadata(idServerId, entryId, revisionId, variantId, schema=strict)

Validates the metadata of a variant.

validateConvention(idServerId, entryId, revisionId, variantId, conventionName)

Returns a report on whether the variant's metadata and ingredients conform to the named convention.

validataMetadataConventions(idServerId, entryId, revisionId, variantId)

Returns a report on whether the variant's metadata and ingredients conform to each of the conventions listed in the metadata.

Delete

Delete operations may not be allowed by all stores.

deleteIdServer(idServerId)

Deletes an idServer from the store.

deleteEntry(idServerId, entryId)

Deletes an entry from the store. The idServer will also be deleted if this is the only entry.

deleteEntryRevision(idServerId, entryId, revisionId)

Deletes an entry/revision from the store. The entry will also be deleted if this is the only revision. The idServer will also be deleted if this is the only entry.

deleteEntryRevisionVariant(idServerId, entryId, revisionId, variantId)

Deletes a variant from the store. The revision will be deleted if this is the only variant. The entry will be deleted if this is the only variant of the only revision. The idServer will also be deleted if this is the only entry.