Bulk Repository Access API

The bulk API **version 2024.1** is used to store and retrieve nodes in batches at the moment of invocation.[1] It is intended for CRUD operations on (larger) sets of nodes. it is **not** intended as a delta-oriented API that takes "modification commands" as arguments.

Table of Contents

Conventions used in this document

- ALL-CAPS key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP14 (RFC2119, RFC8174) when, and only when, they appear in all capitals, as shown here.
- Footnotes refer to more discussions and rationale, but are non-normative.

Use Cases

We describe the use cases from a client's perspective.

- "I'm starting up, which partitions do you know about?", i.e. list possible contents.

- "Let's have a new slate, add a new partition", i.e. partition creation.
- "This part is not needed any more, remove the partition", i.e. partition deletion.
- "Give me the whole subtree under this node", i.e. actual bulk data retrieval
- "Here is a node forest, store it as I hand it to you", i.e. actual bulk data storage.
    - Sub-case with same behavior: "I'm a very simple client who only knows load and save (like a file), so whatever I send you is the truth", i.e. click-save-button.
- "I'm starting up, get me the current partition contents", i.e. initial load.
- "I want to create a guaranteed new node, which id to use?", i.e. retrieval of unused ids.

Validity assumptions

We don't need to know anything about languages for this API.[2]

We never sent partial nodes or features.[3] Thus, we can only update or receive a complete node with all its features. We cannot update only some of the features, or part of a feature (e.g. start of a string property value).

We assume transactional semantics for each modifying command.[4] A command either succeeds completely, or fails completely and does not change the repository's content.

We do support invalid models w.r.t.[5]

- Violations of metamodel constraints, e.g. whether a Concept instance without parent is a partition, a feature instance can be part of its node instance, or a node mentioned as annotation is an Annotation instance.
- Unresolvable references, i.e. a reference target node id unknown to the repository.

We enforce conformance to[6]

- Tree structure (i.e. at most one parent for each node)
- Symmetry between containment/annotation and parent
- No unresolvable containment/annotation ids
- No unresolvable parent ids
- A child node id MUST appear exactly once in all of its parent's containments
- An annotation node id MUST appear exactly once in all of its parent's annotations
- Every node MUST (directly or indirectly) be contained in a partition, except partitions themselves
- Partitions MUST NOT have a parent

Out of scope

We assume mapping API is separate from bulk API[7].

For now, we do not support paging, as paging tree nodes is non-trivial.[8]

For now, we don't include any specific protections agains denial-of-service attacks.[9]

We don't describe the binding to any specific protocol (e.g. HTTP) as part of this specification; We only describe the semantics.[10]

Deleted nodes

We don't have an explicit delete command for non-root nodes.[11] We delete non-root nodes indirectly by sending their parent without the deleted node mentioned as child. We can deletePartitions via explicit command.

We don't support *orphans* for now. They are immediately deleted.[12]

Deleted nodes don't exist anymore in the repository from the client's point of view. They might still exist in other contexts (e.g. another branch), or physically within the repository for internal reasons (e.g. storage optimization, concurrent editing support). A deleted node MUST NOT appear in any responses according to this API.[13]

A repository MAY consider the deleted node's id to be *unused*, and thus allow to re-use it. A repository also MAY disallow re-using previously deleted node ids.

Client ids

Each client MUST provide a unique *client id* when connecting to the repository.[14] The client id is an id-compatible string. A client MAY connect more than once to the same repository at the same time with the same client id. It is out of scope of LionWeb to guarantee the uniqueness of the client id. The repository does not apply any uniqueness checks.

Unknown node ids

An *unknown node id* is a node id that's not the node id of any node present in the repository. An unknown node id can be present as a reference target (as we support unresolvable references).

How to handle unknown node ids in createPartitions and store commands?[7][15]

Scenarios:

Description: Node id …	Valid format	Format fits repo	Present in repo	Reserved	Repo action
already present in repository	□□	□□ (repo accepted it)	□□	□□	error (createPartitions) / update node (store)

Description: Node id …	Valid format	Format fits repo	Present in repo	Reserved	Repo action
requested by this client	□□	□□ (repo handed it out)	□	□□	create node
requested by other client	□□	□□ (repo handed it out)	□	□□	error
used previously by deleted node, repo disallows re-use	□□	□□ (repo handed it out)	□	□□	error
used previously by deleted node, repo supports re-use	□□	□□ (repo handed it out)	□	□	create node
invented by this client, requested by other client	□□	□□ (repo handed it out)	□	□□	error
invented by client, matches repo format Example: `123` for repo with internal long ids	□□	□□	□	□	create node
invented by client, violates repo format Example: `ab123` for repo with internal long ids	□□	□□	□	□	create node
invented by client, invalid Example: `he!!o`	□□□	□□	□	□	error

Responses

The repository signals *success* or *failure* with each response. We call this part of the reponse `success` flag.

Besides the main response, the repository can reply each command with zero or more additional `messages`.[16] Each message MUST have the following properties:

- `kind` is an id-compatible string identifying the message type. Some message kinds are pre-defined in this specification. A repository MAY reply with other, additional message kinds.
- `message` is a human-readable string describing the message.
- `data` is a flat map with arbitrary keys and values. All values MUST be strings, the keys MUST be id-compatible. A *kind* might imply presence of specific keys in `data`.

Message kinds for all commands

Kind	Description	Success	Implied data
`` TODO	Invalid node id	false	`invalidNodeId` Id of non-partition node.
`` TODO	Node with same id sent more than once	false	TODO
`` TODO	Invalid tree due to concurrent update[17]	false	TODO

Commands

listPartitions: List available partitions

Lists all non-language partitions accessible in the repository.

Calling this command MUST NOT change repository contents.

> **NOTE** We might add filter capabilities in the future.

Parameters

None.

Response

Contains a SerializationChunk with all accessible Partitions in the Repository. The partitions are sent as complete nodes.[18] Does NOT include Languages or partition children/annotations.

Message kinds

None.

createPartitions: Create new partitions

Creates new partitions in the repository.[19]

Each sent node is its own partition. Thus, we cannot send the contents (i.e. (indirect) annotations/containments) of a partition; We can send them in a later store call. We also MUST NOT mention any annotation/containment node ids in the partition nodes, as they cannot be part of the same request, and we don't allow moving nodes in this operation. We MAY send properties and references.[20]

Nodes MUST use Unknown node ids.

Parameters

`nodes`

SerializationChunk containing all nodes we want to add as new partitions.

Response

Does not contain a `chunk`.

Message kinds

Kind	Description	Success	Implied data
`PartitionHasParent`	Partition node states parent id	false	`nodeId` node id of requested partition that has a parent.
`PartitionAlreadyExists`	Partition node id already exists	false	`nodeId` Id of already existing node.
`` `TODO`	Partition node id not reserved for this client	false	TODO
`PartitionHasChildren` `PartitionHasAnnotations` TODO	Partition node lists contained or annotated nodes	false	`nodeId` node id of requested partition that has children or annotations.
`EmptyChunk`	Empty Request	true	None

deletePartitions: Delete partitions and all their contents

Deletes all mentioned partitions, including all (transitive) annotations and children.

All mentioned node ids MUST be ids of partition nodes.

All (transitive) annotations and children become orphans.

Parameters:

`nodeIds`

List of node ids to delete.

Response

Does not contain a chunk.

Message kinds

Kind	Description	Success	Implied data
`NodeIsNotPartition`	Node with that id is not a partition	false	`nodeId` Id of non-partition node. `parentNodeId` Id of the parent node.
`` `TODO`	Node with that id does not exist	true	TODO

retrieve: Get nodes from repository

Retrieves subtrees nested in the listed node ids.[21]

Calling this command MUST NOT change repository contents.

> **NOTE** We might add advanced filtering capabilities in the future, or introduce an additional querying API.

Parameters

`nodeIds`

List of node ids we want to retrieve from the repository. Can be partition node ids and/or nested node ids.

`depthLimit`

Limit the depth of retrieved subtrees. Optional parameter, defaults to *infinite*. If present, MUST be an integer >= 0, with

- 0 meaning "return only the nodes with ids listed in `nodeIds` parameter",
- 1 meaning "return the nodes with id listed in the `nodeIds` parameter and their direct children/annotations",
- 2 meaning "return the nodes with id listed in the `nodeIds` parameter, their direct children/annotations, and the direct children/annotations of these",
- etc.

> **NOTE** There's no *magic value* of `depthLimit` to express *infinite* depth. We need to omit the parameter if we don't want to limit the depth.

Response

SerializationChunk containing all nodes according to `nodeIds` and `depthLimit` parameters. Does NOT include the definition of UsedLanguages, only their MetaPointers.

Message kinds

Kind	Description	Success	Implied data
`IdNotFound`	Node with requested id does not exist	true	`nodeId` Id of non-existent node
`EmptyIdList`	Empty id list	true	None
`IdsIncorrect`	Invalid ids parameter	false	`nodeIds` The invalid `nodeIds` parameter
`DepthLimitIncorrect`	Invalid depthLimit parameter	false	`depthLimit` The invalid `depthLimit` parameter.

store: Put nodes into repository

Creates new nodes, or updates existing nodes in the repository.

We always process one node in its entirety, i.e. we cannot update parts of the node with this command.[3]

A node id referenced as parent, containment, reference, or annotation *can* be mentioned in the same request, but *can* be omitted if a node with that id already exists in the repository. This way, we can move subtrees and add arbitrary references without sending unchanged nodes.

We consider that node to be *new* if it has an unknown node id. Otherwise, we consider the node to be *updated* (i.e. if a node with the same id already exists in the repository).

We do not support different modes.[22]

Parameters

`nodes`

SerializationChunk containing all nodes to store to the repository.

Semantics

After completing this call, all sent nodes MUST have exactly the sent contents in the repository. We must send containments/annotations from the parent's view, because we need to know the containment and index of the contained node/annotation within its parent.

If we move a contained/annotation node `C` from its previous parent `A` to its new parent `B`, we MUST send `B`, and MAY omit `A`.[23] This means we can have implicit changes in `A`.

The whole call fails, without any changes to the repository, if it would lead to a malformed model.[6] The repository MUST NOT validate the metamodel constraints of the sent nodes.[5]

The repository MUST support changing meta-pointers, e.g. a node's `classifier.language`, a

property's `property.version` or an enumeration literal's `key`.[24]

Response

Does not contain a chunk.

Message kinds

Kind	Description	Success	Implied data
`NullChunk`	Chunk missing	false	None
`` `` **TODO**	Node id mentioned as annotation/child in more than one parent	false	TODO
`` `` **TODO**	Move would create loop in tree	false	TODO
`ParentMissing`	Parent / child / annotation node id unknown	false	`nodeId` Id of node that doesn't have a parent.
`` `` **TODO**	Parent doesn't match child/annotation	false	TODO
`` `` **TODO**	New node id not reserved for this client	false	TODO

ids: Get available ids

Provides unused valid ids.

The repository

- MUST NOT hand out the same unused ids to any other client.
- MAY hand out the same unused ids to the same client more than once.
- MUST NOT contain any node with any of the provided ids.

The ids MUST exclude the built-in ids.

Calling this command MUST NOT change repository contents themselves (besides the internal knowledge of reserved ids).

We don't assume leases, i.e. ids handed out to one client are "owned" by that client forever. Rationale: Otherwise, the repository must track sessions, and run housekeeping on leases. This would exclude simple repository implementations.

We assume infinite id space.

Parameters

`count`

Number of ids requested. The repository MUST return between one (inclusive) and `count` (inclusive) ids. It MAY return less than `count` ids.

Response

List of ids guaranteed to be free.

Message kinds

None.

Mapping CRUD operations

list

listPartitions for partitions, retrieve for descendants.

read

retrieve call with requested node ids (both partitions and other nodes).

create

createPartitions for partitions, store call that sends a node with a *new id*, including all its features.

update

store call that sends a node (both partitions and other nodes) with an *existing id*, including all its features (both updated and unchanged).

delete

deletePartitions for partitions (including all descendants), for others store of the parent node without mentioning the deleted node.

move

Assume we want to move node `N` from its current parent `S` to its new parent `T`.

store call that sends `T` with all its features, including `N` in the children.

> **NOTE** We cannot move partitions, as we cannot nest them.[25]

[1] Repo API: Bulk read/write #25

[2] How to separate between simple and advanced Bulk API? #203

[3] We always transmit complete nodes in bulk API #211

[4] Bulk API assumes transactional semantics for changing operations #229

[5] Does bulk API validate against languages? #226

[6] Which kinds of invalid nodes does Bulk API accept? #223

[7] Provide id mapping API #94

[8] Do we need a paging functionality in bulk API? #204

[9] Denial-of-service protection is out of scope (at least for now) #237

[10] Level of detail on API specifications #148

[11] Explicit delete operation in bulk API? #221

[12] Optionally support orphans in repository #219

[13] What does it mean to delete a node #220

[14] A client must identify to repository with unique id #241

[15] Can Repositories have stricter requirements on node IDs than LIonWeb (e.g. only longs)? #70

[16] How to report additional info in bulk API? #236

[17] Report model update issues due to concurrent edits separately? #238

[18] Return whole nodes when querying existing partitions #202

[19] How to create and delete partitions #216

[20] Can we send features during createPartition()? #225

[21] Don’t provide `closure` retrieve mode in simple bulk API #201

[22] Which modes to support in bulk API store? #230

[23] When moving a node, do we need to mention both source and target parent? #227

[24] Node update: do we allow concept change? #69

[25] Repo API: Do we need model partitions? #29