Structure 3 API Changes

1. State of the API

In Structure 3 we had to change API in an incompatible way because the underlying architecture of the product had changed. If you have integration with Structure 2, most likely it won't work with the new Structure and some effort is needed to migrate the code.

As of Structure versions 3.0 – 3.1, the new API is not yet finalized and thorough documentation is not yet published. We plan to spend additional effort on making the APIs simple, stable and well-documented and publish the final documentation then.

Until that time, it's possible to use the current non-published API with Structure 3, however:

- There's no public documentation on it. The sources of the API artifact are published, but they mostly don't have javadocs yet.
- There will be backwards-incompatible changes while we finalize the API. The concepts and interfaces will stay mostly the same, but some classes may be moved and optimized. This is less likely to impact REST API, although we plan to introduce new REST APIs that would be simpler than the low-level API we have now.
- There will be new interfaces that would make it easier to deal with the new concepts. Right now it may be a little "low-level" and somewhat more complicated than it needs to be.

Although the documentation about the new API is not available, Structure team will be happy to assist you in migrating your code to work with Structure 3.

This article lists some of the most frequently used API calls. If you need to do something that is not covered by this article or have any questions, please write us at support@almworks.com

2. Conceptual Changes

2.1. Forests and Rows

In Structure 2, a structure's content was called a ***forest*** . That is still the case, however, the forest now contains ***rows*** rather than issues. A row has a Row ID – a long integer primary key for the row. Given row ID, you can retrieve information about the item displayed in that row.

The data structure that represents a forest didn't change. Previously a forest was represented by an array of pairs `(issueID, depth)`. Now the forest is represented by an array of pairs `(rowID, depth)`.

The concept of a row may seem superfluous, but it's actually required for uniquely identifying a specific position in a forest. Issue ID (or Item ID) is not sufficient because an issue can be located at multiple places in the forest.

2.2. Items

Each row has an associated ***item*** – an issue, a folder, a project or any other type of items. In Structure 3, item types are extendable and an add-on may provide additional types of items to Structure. An item is identified by ***Item Identity*** , which consists of:

- Item Type, represented by a complete module key of a JIRA plugin module that provides the item type, and
- Item ID, represented by *either* a long integer (for example, issue ID for issues) or by a string (a user key for users).

Sometimes item type is omitted; in that case the implied item type is "issue".

Some of the most popular item types are:

Item Type	Module Key	Meaning of Item ID	Comments
Issue	`com.almworks.jira.structure:type-issue`	Issue ID (long)	Default when item type is not specified
Folder	`com.almworks.jira.structure:type-folder`	Either folder ID (long) or folder i18n name (string)	Folders are introduced in Structure 3
User	`com.almworks.jira.structure:type-user`	User key (string)	
Generator	`com.almworks.jira.structure:type-generator`	Generator ID (long)	A generator is an automation rule embedded in the structure.
Page	`com.almworks.structure.pages:type-confluence-page`	Page ID (ID modulo 1e9)	Confluence pages are added as a type by Structure.Pages extension

2.3. Attributes

Attributes are a generalization of a JIRA's issue fields. An attribute is something that can be calculated for an item. For example, an issue has such attributes as "summary", "key", "priority". But purely Structure-related values are also attributes, such as "sequential number", or "aggregate progress", or "sum of story points". The attributes can also be retrieved for any types of items – for a Confluence page (provided by Structure.Pages extension), "summary" would be the title of the page, "labels" would be the labels, and a new attribute "author" would provide the initial author of the page.

Attributes are identified by an attribute specification, or ***attribute* spec**. It is usually represented as a JSON object with an ID and parameters.

2.4. Concept Comparison

	Structure 2	**Structure 3**
A structure's content is ...	Forest	Forest
Things that can be placed into a structure are ...	Issues	Items (including issues)
Forest consists of ...	Issues	Rows
A position in a forest is identified by ...	Issue ID	Row ID
A value in the Structure grid is displayed by ...	Column	Column
A column requests from the server ...	Fields	Attributes

3. REST API

3.1. Retrieving Structure Forest

```
GET /rest/structure/2.0/forest/latest?s={%22structureId%22:$id}
```

This method retrieves a content of a structure. If structure has generators, the generated content is returned. Generators are preserved in the forest.

Parameters:

- `$id` – structure ID

Return value sample:

```
{
  "spec":{"structureId":171},
  "formula":"10394:0:4/356,10332:0:14707,10374:1:5/240,10348:2:14717",
  "itemTypes":{
    "4":"com.almworks.jira.structure:type-generator",
    "5":"com.almworks.jira.structure:type-folder"
  },
  "version":{
    "signature":-1659607419,
    "version":1
  }
}
```

In this reply, the most important part is "formula", which contains serialized information about the forest, much like in Structure 2.
Each component (delimited by comma) represents a row and looks like this: `10374:1:5/240`. In this example, the numbers are:

- `10374` is the row ID,
- `1` is the row depth,
- `5/240` is the item identity. If the row contains an issue, it's just issue ID, otherwise it has the format of `<item type>/<long item id>`, or `<item type>//<string item id>`. Item type is a number, which is expanded in the "itemTypes" map in the reply.

3.2. Updating a Structure Forest

```
POST /rest/structure/2.0/forest/update
```

Parameters:

```
{
  "spec": { "structureId": <id> }, // use structure ID
  "version": { "signature": <signature>, "version": <version> },  // use last seen signature and version
  "actions": [
     {
         "action": "add",
         "under": 0,        // at the top level
         "after": 123,      // after row ID 123 (not issue iD!)
         "before": 456,   // before row ID 456
         "forest": "-100:0:10001"   // insert issue 10001, -100 is the temporary row ID which will be mapped
into the real row ID when the method returns
     },
     {
         "action": "move", // works like previously, only row IDs instead of issue IDs
         "rowId": 123,
         "under": 456,
         "after": 0,
         "before": 124
     },
    {
       "action": "remove",
       "rowId": 442
     }
  ]
}
```

3.3. Creating a structure

```
POST /rest/plugins/structure/2.0/structure
```

Parameters:

```
{
   "name": "my structure",
   "description": "my description",
   "permissions": [   ]    // same format you see when you GET structure
}
```

3.4. Deleting a structure

```
 DELETE /rest/plugins/structure/2.0/structure/<id>
```

4. Java API

4.1. Versions

As Structure 3 API is finalized, it's getting a lot of refactoring and version changes. A new Structure version may have a backward-incompatible API, although incompatibilities may be isolated and your code has a good chance to work fine. However, the major version is promoted every time a backward-incompatible change is made, therefore you need to carefully set up the version of imported API packages – either set them optimistically (for example, `[12,15)` – up to version 15) and test your integration with a new release to see that there are no errors; or set the version as usual – for example, `[12,13)` – but then you might need to recompile with each new release of the API. The latter approach is recommended for in-house customizations.

Version	Supported JIRA Versions	Introduced in Structure Version	OSGi Import	OSGI Import (Optimistic)
12.0.0	JIRA 6.3+	3.0.0	"[12,13)"	"[12,15)"
12.1.0	JIRA 6.3+	3.0.1	"[12.1,13)"	"[12.1,15)"
13.0.0	JIRA 6.3+	3.1.0	"[13,14)"	"[13,16)"

13.0.1	JIRA 6.3+	3.1.1	"[13,14)"	" [13,16)"

The API versions and sources are available from the public Maven repositories – http://mvnrepository.com/artifact/com.almworks.jira.structure/structure-api

4.2. Retrieving Structure's Forest

To get the content of a structure, you need to use `ForestService` interface, which can be injected. It has `getForestSource()` method that will return a `ForestSource` given `ForestSpec`, which is a specification of what kind of forest you are retrieving. For getting just a content of a structure, use `ForestSpec.structure(structureId)`. Once you have a `ForestSource`, you can use `forestSource.getLatest().getForest()` to retrieve an instance of `Forest` – which should be familiar from the Structure 2 API.

But now `Forest` contains row IDs, not issue IDs, so to get information about what issues (or other items) are in the forest, you need to "dereference" each row ID.

4.3. Working with Rows

For working with rows, use `RowManager`. To get an item ID from a row ID, use `rowManager.getRow(rowId).getItemId()`. This gives you `ItemIdentity` instance. To see if it is an issue, use `CoreIdentities.isIssue(itemId)` and to get issue ID in that case, use `itemId.getLongId()`.

To get all row IDs for a given issue ID (for example, to find an issue in a forest), you can use `rowManager.findRows(CoreIdentities.issue(issueId))`. These row IDs may be from multiple forests, so you need to see if the forest that you have contains some of those IDs.

4.4. Getting Totals and Other Values

To calculate totals or other Structure-calculated values, you need to use `StructureAttributeService`.

`StructureAttributeService.getAttributeValues()` has the following parameters:

- `ForestSpec` – use the same forest spec that you use to retrieve the forest;
- row IDs – you need to specify for which rows (not issues!) the values are requested;
- a collection of `AttributeSpec` – specify which attributes are requested.

You need to build a list of attribute specs to specify what to calculate. There are several ways to get a correct attribute spec:

- Some specs are defined in `CoreAttributeSpecs`.
- You can build a spec using `AttributeSpecBuilder`.
- You can parse a JSON representation of a spec into a `Map`, then extract "id" and "params".

Examples:

Attribute	Spec

Story Points	`AttributeSpecBuilder`
	`  .create(“customfield", ValueFormat.NUMBER)`
	`  .params()`
	`  .set("fieldId", 10000) // 10000 – the id of "Story Points" custom field`
	`  .build()`

Story Points	`AttributeSpecBuilder`
	`  .create("sum", ValueFormat.NUMBER)`
	`  .params()`
	`  .setAttribute(storyPoints)  // storyPoints = the attribute spec for Story Points`
	`  .set("distinct", true)  // exclude duplicates`
	`  .build()`

4.5. Changing Structure

To change a structure, you need to use `UpdatableForestSource.apply()` method. Each update is a separate transaction – the concept of a ForestTransaction used in Structure 2 has been removed.

To get an instance of `UpdatableForestSource` you need to cast `ForestSource` retrieved from `ForestService`.

Examples:

Operation	Code
Add an issue with ID 10200 to structure, under parent row with ID 1040, after row with ID 1900 and before row with ID 2000	`forestSource.apply(new UpdatableForestSource.Update.Add( CoreIdentities.issue(10200), 1040, 1900, 2000))`
Remove rows with IDs 10100 and 10102	`forestSource.apply(new UpdatableForestSource.Update.Remove( LongArray.create(10100, 10102)))`
Move row with ID 1010 as the first row under parent row with ID 1040, before a row with ID 1060	`forestSource.apply(new UpdatableForestSource.Update.Move( LongArray.create(1010), 1040, 0, 1060))`