REST API conventions

The IBM® UrbanCode™ Release REST API follows a set of conventions to make interacting with it more consistent.

REST URLs

Each element type on the server is represented as a top-level URL with a plural form. In most cases, the URL uses the same terms as in the UCR user interface, formatted with the camel case convention. For example, change types are represented at the URL http://*base_url*/changeTypes/, environment reservations are at the URL http://*base_url*/environmentReservations/, and applications are at the URL http://*base_url*/applications/, where base_url is the host name and path to the server.

Required HTTP headers

Most operations in the REST API accept an input in JSON format, return an output in JSON format, or both. Following HTTP conventions, the Content-Type request header is required for operations that provide JSON input, and the Accept request header is required for operations that produce JSON output, with the media type value of application/json. As a convenience for developers, the GET method for lists and individual items (described below) also produces JSON output if the media type is text/html and the query parameter json is appended to the request URL with any value. This parameter is useful for rendering JSON output from a web browser without special tools to modify the Acceptheader. For example, to display the JSON output for all applications, type http://*base_url*/applications/?json into the location bar of a web browser.

The id property

Elements such as applications have a unique ID (UUID) property, which appears in the JSON object representation as the id property as a JSON string. You can refer to a

specific element by its UUID. You can use this UUID in places such as the URLs of REST API operations and within JSON representations of elements.

Basic operations

The basic create, read, update, delete operations are provided according to common REST API conventions. The top-level URL for an element type represents the collection of items of that type. To render the list of all elements of that type as an array of JSON objects, use the HTTP method GET with appropriate HTTP headers or the json query parameter. To create a new element, use the HTTP method POST to the top-level URL for the appropriate type and provide JSON data for the element in the body of the request. The response body includes a full JSON representation of the new element, including the server-generated id property. You can also provide an array of JSON objects to create multiple elements in a single request. In this case, the response body contains a JSON array of the new elements in the same order that they were provided in the request. Similarly, you can update the data for multiple elements by using the HTTP PUT method with a JSON array of elements. You can delete multiple elements by using the HTTP DELETE method with a JSON array that contains either full JSON elements or only the UUIDs of the elements.

To run an operation on a single existing element, append the UUID for the element to the top-level URL for the type. For example, to access the JSON data for the sample release that is provided, use the HTTP method GET with the URL http://*base_url*/releases/00000000-0000-0000-0000-000000000036/. To update the data for an element, use HTTP PUT method and provide an updated JSON object in the request body. (In this case, the UUID in the request URL takes precedence over the id property in the request body.) To delete the element, use the HTTP DELETE method; no request body is required in that case.

Output formats

When you use the create, read, and update operations (POST, GET, and PUT), you can include the optional **`format`** query parameter to adjust the JSON output for specific use cases. If you do not provide this parameter, or if the value of the parameter is not recognized, a default format is used. Each element type supports a different set of formats; these formats are listed with the reference information for the element type. Some formats show a large amount of detail and others show a small amount of detail, but the difference is only in the information that is shown; the values are the same.

You can use the list and detail formats with any element type. The list format provides summary information, such as you might show in a table of items. The detail format provides more detailed information. This format often includes the content of related

elements. For some elements, these formats are identical. In addition, element types that have a name property also have a name format. This format includes only the id and name properties, which can be useful for displaying a list of items for selection by name. As a convenience, the name format for an element type can also be produced by appending name to the top-level URL for the type, for example, GET http://*base_url*/applications/name.

JSON input conventions

When you provide JSON data as input for the create or update operations, the REST API takes into account only the properties that are writable on the element. The API ignores all other data. Read-only properties, such as computed counts or creation dates, are not updated. The API ignores unrecognized properties. Properties of related elements, other than the id property used to establish a reference, are also not updated.

For example, assume that you provide the following JSON input to create a release:

```
{
  "name": "New Release",
  "team": {
    "id": "00000000-0000-0000-0000-000000000206",
    "name": "Not the actual Team name" },
  "lifecycleModel": "00000000-0000-0000-0000-000000000006",
  "totalChanges": 42,
  "BadProperty": "xxxx"
}
```

When the API processes this input, it recognizes only the properties that are required for creating a release. Therefore, it ignores the unrecognized property BadProperty and the computed value totalChanges. In this case, the input above is equivalent to the simpler input below:

```
{
  "name": "New Release",
  "team": "00000000-0000-0000-0000-000000000206",
  "lifecycleModel": "00000000-0000-0000-0000-000000000006"
}
```

Similarly, when you update an existing element, the API changes only the properties that you specify in the JSON input. The API does not change properties that you omit. To clear the value of a property, specify an explicit JSON null value for that property.

For example, consider an existing change with the following content:

```
{
  "id": "00000000-0000-0000-0000-123456789000",
  "name": "New Feature",
  "description": "",
  "status": "In Progress",
```

```
  "type": {
    "id":"00000000-0000-1201-0000-000000000001",
    "name":"Feature",
    "icon":"feature"}
}
```

The following input associates the change with the sample release, but does not update any other properties of the change.

```
{
  "id": "00000000-0000-0000-0000-123456789000",
  "release": "00000000-0000-0000-0000-000000000036"
}
```

To reverse this operation and clear the association from this change to the sample release, you could use the following input:

```
{
  "id": "00000000-0000-0000-0000-123456789000",
  "release": null
}
```

References

In most cases, you can refer to a single element with either a JSON string that contains the UUID of the element or a JSON object that contains an id property with the same value. As described above, any other properties of the referenced element are ignored. For example, all of the following JSON objects are equivalent when you are associating a change to a release.

```
{
  "id": "00000000-0000-0000-0000-123456789000",
  "release": "00000000-0000-0000-0000-000000000036"
}
```

```
{
  "id": "00000000-0000-0000-0000-123456789000",
  "release": {"id":"00000000-0000-0000-0000-000000000036"}
}
```

```
{
  "id": "00000000-0000-0000-0000-123456789000",
  "release": {
    "id":"00000000-0000-0000-0000-000000000036",
    "name":"Sample Release"}
}
```

```
{
  "id": "00000000-0000-0000-0000-123456789000",
  "release": {
    "id":"00000000-0000-0000-0000-000000000036",
    "name":"Not the actual Release name"}
}
```

Multiple-value (collection) references

Some relationships can have multiple values. For example, a single release can be associated with multiple applications. In some cases, these relationships are not updated when you create or update the single-value side of the relationship. For example, when you create a Release, the values that you provide for the applicationsproperty have no effect. Instead, use other URLs to add or remove applications from releases:

- List the applications that are related to a release: GET /releases/{releaseID}/applications
- Add an application to a release: POST /releases/{releaseID}/applications/{applicationID}
- Remove an application from a release: DELETE /releases/{releaseID}/applications/{applicationID}

However, in other cases, particularly if an element requires a specific relationship, you can specify input for the single-value side of the relationship. For example, an application can be associated with multiple teams. In this case, when you create an application, you can provide a JSON array of UUID strings for the "teams" property. There are no special URLs for manipulating this relationship.

Key-value properties

Some element types support the storage and retrieval of free-form key-value pairs in addition to statically defined properties. This type of property often represents data that is linked to an integration provider. For example, a change that represents a IBM Rational® Team Concert™ work item might store the identifier for the work item as a property in this key-value storage.

Key-value properties are represented in the JSON data as a related element under the propertiesproperty. For example, the following code represents a change with key-value properties that are named first, second, and third:

```
{
```

```
  "id": "00000000-0000-0000-0000-123456789000",
  "name": "New Feature",
  "description": "",
  "status": "In Progress",
  "type": {
    "id":"00000000-0000-1201-0000-000000000001",
    "name":"Feature",
    "icon":"feature"},
  "properties": {
    "first":"first value",
    "second":"second value",
    "third":"third value"}
}
```

Both the key and value must be represented as JSON strings. As with statically defined properties, if a key is omitted from input, update operations do not change it. To clear a key-value pair, assign a JSON null value to the key.

Pagination

For some element types, the top-level GET request can provide a subset of elements rather than the full list. Retrieving subsets of elements like this is commonly referred to as "pagination," because it involves loading only a single "page" of data in each request.

Two alternative forms of paging are available, using either query parameters or the HTTP Range header.

To use query parameters for pagination, specify both a **`rowsPerPage`** and a **`pageNumber`** query parameter in the HTTP GET request. To retrieve the first page, specify the code pageNumber=1. For example, to retrieve the first five applications, use the request GET http://*base_url*/applications/?rowsPerPage=5&pageNumber=1.

To use the HTTP Range header, specify the start and end values for the range of items, specifying zero-based and inclusive values. For example, to retrieve the first five applications, add the following header to the request GET http://*base_url*/applications/:

```
Range: items=0-4
```

Regardless of how you request the pagination, the HTTP response uses the Content-Range header to specify the actual range that is retrieved and the total number of available items. For example, if you requested the first five changes of a list of 10, the response includes the following header:

```
Content-Range: 0-4/10
```

If you requested the first five changes, and only three changes are available, the header looks like this:

```
Content-Range: 0-2/3
```

Sorting

To sort the results of the top-level GET operation, specify the **`orderField`** and **`sortType`** parameters. The **`orderField`** parameter must contain the name of a single property to sort by. The **`sortType`** parameter must contain either the value asc to sort in ascending order, or the value desc to sort in descending order.

To sort on the properties of related elements, specify the property name with dotted-path format. For example, to sort a list of changes according to the associated release, specify the value release.name. The ordering is unspecified if any property in this path is null.

You can combine sorting with pagination and the **`format`** parameter. For example, to retrieve the first five changes in the detail format, sorted by the name of the related release in ascending order, use the following request:

```
GET http://base_url/changes/?format=detail&rowsPerPage=5&
  pageNumber=1&orderField=release.name&sortType=asc
```

Not all properties can be used for sorting. In particular, properties that represent summary data, such as item counts, generally cannot be used for sorting.

Filtering

To filter results, specify query parameters. The names of some query parameters are based on the names of the data properties on which you filter the results. For example, the following request shows the releases that were created on a specified date:

```
GET http://base_url/releases/
  ?filterFields=dateCreated
  &filterType_dateCreated=gt
  &filterClass_dateCreated=Long
  &filterValue_dateCreated=1421171883574
  &orderField=dateCreated
  &sortType=asc
```

This example uses the following query parameters:

filterFields

This parameter specifies the fields on which to filter the results. In the previous example, the code filterFields=dateCreated filters the results by a specific date. To filter on more than one field, repeat this parameter, as in the following example:

```
filterFields=dateCreated&filterFields=name
```

filterType_*propertyName*

This parameter specifies the type of filter operation to run on the specified property. The previous example had the property value filterType_dateCreated=gt. This code specifies that the filter for the **dateCreated** property shows values that are greater than the specified date. Supported filter operations are like, eq (equal to), ne (not equal), gt (greater than), ge (greater than or equal to), lt (less than), le (less than or equal to), null, notnull, range, and in.

filterClass_*propertyName*

This property specifies the data type that you provide to the filter. In the previous example, the date was in a long integer in a Unix timestamp value: filterClass_dateCreated=Long. Valid data types are Boolean, Long, String, UUID, and Enum.

Each data type supports a specific set of filter types:

- String: like, eq, ne, gt, ge, lt, le, null, notnull, range, and in
- Long: eq, ne, gt, ge, lt, le, null, notnull, range, and in
- Boolean: eq, ne, null, notnull
- UUID: eq, ne, null, notnull
- Enum: eq, ne, null, notnull

filterValue_*propertyName*

This parameter specifies the value on which to filter the results. The previous example specifies a date in Unix timestamp format. If you are using the filter types range or in, specify this parameter multiple times to define the range or set of values. No parameter is necessary for filters of the types null and notnull.

Related reference:

→ REST API reference
→ Running REST commands
→ REST commands
→ Authenticating for REST commands
→ Java client library

REST API examples

Feedback