IBM FileNet Content Manager

Technology Preview: Content Services GraphQL API Developer Guide

Copyright

Before you use this information and the product it supports, read the information in "Notices" on page 45.

Contents

Abstract

This document provides reference information for developers who want to use the Content Services GraphQL API in a FileNet Content Manager container environment.

Background information

What is the Content Services GraphQL API?

The Content Services GraphQL API provides a schema and an easy-to-understand query language system that simplifies application development for your Content Platform Engine. The API schema definition of types and fields matches Content Engine Java API object model closely, with necessary and desirable extensions for natural GraphQL developer consumption.

How do I access the Content Services GraphQL API?

The Content Services GraphQL API are available as a separately deployed container to work with FileNet Content Manager container deployments of V5.5.3, V5.5.3 iFix1, or later.

In V5.5.3 and V5.5.3 iFix1, the GraphQL API is available as a technology preview.

It is recommended for GraphQL API early adopters who are using this technology preview release to update FileNet Content Manager container deployments to V5.5.3 iFix1. Certain new functionalities are only available in V5.5.3 iFix1. Refer to later sections and for additional details.

Developer references

For a general introduction, start with Developing FileNet P8 applications.

Supported platforms

The files provided support IBM Cloud Private and certified Kubernetes container environments where the Content Platform Engine 5.5.3 APIs are allowed. For details see the IBM Software Product Compatibility Reports.

Interfaces and output types

The schema defines many interfaces that match the CE class hierarchy, for example Containable, Versionable, Document. In some cases, the instantiable output types in the schema simply represent the concrete type that implements these interfaces, for example, DocumentType.

The suffix Type is used for output types such as these that always have a corresponding interface that they implement. It is a best practice when shaping the output data to select interfaces rather than an output type such as DocumentType. The reason is that the concrete type of an instance can change depending on the CE class it is associated with, but the interface will always be implemented by whatever type is returned.

For example, a future release might introduce a WorkflowDefinition interface that is a superset of Document and has a corresponding output type WorkflowDefinitionType. The output type will implement all of the interfaces that DocumentType implements, including Document, and in addition implements WorkflowDefinition. However, if DocumentType was selected in some GraphQL query, that selection would no longer apply to a WorkflowDefinitionType. However, the Document interface would continue to apply.

Search API restrictions and constraints

From argument single class or multiple joined classes

In V5.5.3 the "from" argument in any of the search queries – documents(), folders(), etc. – can only take a single class. The full join syntax that is supported in the CE query SQL syntax is not supported.

You can however include a class alias so that the alias can be used in other tests in the where condition. For example:

```
{
  documents(
    repositoryIdentifier:"OS1"
    from:"Document d WITH INCLUDESUBCLASSES"
    where:"(ISCLASS(d,MyDocument) OR
ISCLASS(d,MyOtherDocument)) AND d.[DateCreated] >
20180815T070000Z AND d.[IsCurrentVersion] = True"
    orderBy:"DocumentTitle"
    pageSize:20
  )
  {
    ...
  }
}
```

Multiple joined classes in the "from" argument are supported starting with V5.5.3 iFix1. The SELECT part of the eventual query is implicitly constructed using information in the output data shaping part of the GraphQL query. Note that the SELECT properties are not explicitly specified in the GraphQL query itself. In V5.5.3 the implicitly generated SELECT statement does not use any class specifier.

SQL statements that involve multiple joined classes must identify each property by using a class specifier, or there can be no ambiguity in a property and the class it comes from. There would typically be ambiguity in the properties that must be selected to satisfy API requirements. In V5.5.3 iFix1 the property selection is handled in a way that there are no ambiguities.

Refer to later examples with details involving joined classes.

OrderBy argument and data shaping

In V5.5.3, any properties that are referenced in the orderBy argument must also be represented in the output data shaping section of the GraphQL query, either by selecting a well-known field corresponding to that property or explicitly selecting that property as part of the properties field. The API does not automatically add orderBy properties to the SELECT portion of the SQL.

This is fixed in the V5.5.3 iFix1 GraphQL Technology Preview update release. orderBy properties are now automatically added into the SELECT portion of the SQL.

Unsupported property type

The Binary type of property is not supported in this release. It is ignored when selected as a property to return in the output data shaping and there is no way to specify a binary property when setting properties.

Class and property identifiers

In general, only symbolic names can be used as the identifiers of classes and properties.

User and group discovery

A User/Group Discovery API is not provided in this technology preview release.

Batching support

There is no capability in this release to execute the changes from multiple mutations in a single transactional batch. Multiple mutations can be specified in a single GraphQL document, but they will be executed one at a time.

Uploading content

Several document mutations allow content to be uploaded, along with other changes to the document. A multi-part form POST must be used to issue the GraphQL mutation and accompanying file upload part or parts. The uploaded data is specified by setting the value of a “content” field in the mutation using a GraphQL variable. The variable name itself maps to a part in the multi-part form that holds an uploaded file. Refer to examples in a later section for additional details.

The V5.5.3 GraphQL API only support content uploads up to 5MB, but this is relaxed to unrestricted content upload size in V5.5.3 iFix1.

Metadata types and secondary objects

In general, ClassDescription types and their embedded PropertyDescription types are served out of the metadata cache by the API. The PropertyDescriptionObject type has a propertyDefaultObject field that can refer to a variety of object types in CE that are not stored in this cache. This field will currently be returned as a minimal instance of the appropriate type containing identifying information only, for example, id, className and objectReference fields. Selecting non-identifying fields will result in null being returned for those fields and may produce errors if those fields are defined as non-nullable.

The choiceList field, selectable from PropertyDescription, is fully supported in these technology preview releases. The appropriate ChoiceList object or objects will be retrieved to the appropriate depth based on the output data shaping.

Some other secondary objects that are exposed in the CE metadata are not currently supported by the GraphQL API and not exposed by these metadata types, for example, AuditAs and MarkingSet.

Additional object change actions

The API currently exposes a few convenience mutations for certain core types in the CE object model. It is possible to perform a variety of other change actions using the generic changeObject() mutation. For example, a Promote or Demote action could be performed using this mutation. Refer to examples in a later section for additional details.

Note that cancelling a checkout is not a separate type of action. In V5.5.3 there is not a specific mutation for cancelling a checkout. A checkout can be cancelled by deleting whatever document is the reservation document of a VersionSeries. The reservation can be selected as an output field of any Document or VersionSeries.

In V5.5.3 iFix1, a new cancelDocumentCheckout mutation is added. See the samples section for more details.

Page size of enumeration properties

The API uses a default page size of 50 to return the results for any enumeration style properties. This includes fields such as containedDocuments or subFolders of the Folder interface. The corresponding properties can also be selected through the generic properties field. The results are returned in types such as DocumentSet, FolderSet or IndependentObjectSet. This release allows the default page size to be modified if desired by configuring a JVM system property. For example:

```
-Decm.content.graphql.enumPropertyPageSize=20
```

Example queries and mutations

Certain new functionality is only available in V5.5.3 iFix1 and is marked accordingly in the following samples.

It is recommended for GraphQL API early adopters who are using this technology preview release to update FileNet Content Manager container deployments to V5.5.3 iFix1.

Retrieving a document

```
{
  document(
    repositoryIdentifier:"OS1"
    identifier:"{D0D98E6A-0000-CD12-BB33-2DC8E5CABEE1}"
  )
  {
    className
    id
    name
    majorVersionNumber
    minorVersionNumber
  }
}
```

Retrieving a folder by path

Documents or folders can be identified by path as well as by GUID.

```
{
  folder(
    repositoryIdentifier:"OS1"
    identifier:"/Folder for Browsing"
  )
  {
    className
    id
    name
    pathName
  }
}
```

Retrieving metadata - classDescription

Metadata in the way of class descriptions and property descriptions can be obtained through the API. This example obtains a class description and its property descriptions. The filter argument to the propertyDescriptions field is optional but can be used to include only property descriptions that match certain criteria.

```
{
  classDescription(
    repositoryIdentifier:"OS1"
    identifier:"SupportingDocument")
  {
     id
    propertyDescriptions(filter: {isHidden:false,
isSystemOwned:false})
    {
      name
      dataType
      isReadOnly
      isOrderable
      isSystemOwned
      cardinality
      settability
    }
  }
}
```

Retrieving choice lists

Choice lists of property descriptions can be selected as a field of the PropertyDescription interface returned by the previous query. A client may want to cache choice lists and retrieve choice lists in a separate call as in this example. Choice lists can be retrieved by ID or name.

```
{
  choiceList(repositoryIdentifier:"OS1"
  identifier: "Entry Choices")
  {
    id
    dataType
    choiceValues
    {
      name
      choiceStringValue
    }
  }
}
```

Retrieving subclasses of a certain Class (5.5.3 iFix1)

This query supports getting all subclasses of a single class description. It also allows paging. Example of getting all subclasses and continuous query.

```
query
{
    subClassDescriptions(repositoryIdentifier: "OS1",
    identifier: "Invoice",
    pageSize: 5)
     {
        classDescriptions {
            id
            name
            superClassDescription {
                name
            }
        }
        pageInfo {
             token
            totalCount
         }
    }
}

query
{
    moreSubClassDescriptions(token: “<insert_token>”)
     {
        classDescriptions {
            id
            name
            superClassDescription {
                name
            }
        }
        pageInfo {
             token
            totalCount
         }
    }
}
```

Convenience security principal fields

The schema exposes several convenience security principal fields that correspond to CE system properties holding only a string identifier of the principal - usually a short name. Examples of these fields include creatorUser, lastModifierUser and ownerPrincipal (user or group). These fields can be selected and shaped to return more detailed information about the user or group.

```
{
  document(
    repositoryIdentifier:"OS1"
    identifier:"{D0D98E6A-0000-CD12-BB33-2DC8E5CABEE1}"
  )
  {
    className
    id
    name
    majorVersionNumber
    minorVersionNumber
    creator
    creatorUser {
      id
      shortName
      displayName
      distinguishedName
      email
    }
  }
}
```

Shaping of complex fields

The schema maps closely with the CE object model of classes and properties. Many fields of an interface or type correspond to CE object valued properties – single, list or enum. These fields can be further shaped in the output data to return these additional objects. In most cases the data is requested from CE in a single round trip using the appropriate property filter.

Shaping a single-value object:

```
{
  document(
    repositoryIdentifier:"OS1"
    identifier:"{D0D98E6A-0000-CD12-BB33-2DC8E5CABEE1}"
  )
  {
    id
    name
```

```
        versionSeries {
          id
          isReserved
          isVersioningEnabled
        }
      }
    }
```

Permissions are one example of dependent objects in the CE object model. Dependent objects are returned as an array of objects. The fields of the dependent objects can be shaped:

```
    {
      folder(repositoryIdentifier:"OS1",
      identifier:"/Folder for Browsing/GraphQL Folder") {
        id
        name
        permissions {
          permissionSource
          inheritableDepth
          ... on AccessPermissionType {
            granteeName
            granteeType
            accessType
            accessMask
          }
        }
      }
    }
```

Many CE properties reference a collection of other independent objects. These collections are returned as an enumeration. In GraphQL the collections are returned as a type such as DocumentSet, FolderSet or IndependentObjectSet. These types return the collection itself and also page information if all the items could not be returned in a single page. The fields of the Set object and the independent objects themselves can be shaped:

```
    {
      folder(
        repositoryIdentifier:"OS1"
        identifier:"/Folder for Browsing")
      {
        name
        id
        containedDocuments {
          documents {
            id
            name
            mimeType
            majorVersionNumber
            minorVersionNumber
          }
          pageInfo {
            token
          }
        }
      }
    }
```

Retrieving content elements

The content of documents can be obtained by requesting the contentElements field.

```
  {
    document(
      repositoryIdentifier:"OS1"
      identifier:"{A0CC2C69-0000-C618-A0A4-3FA294C1AE56}"
    )
    {
      id
      name
      contentElements {
        contentType
        elementSequenceNumber
        ... on ContentTransfer {
          contentSize
          retrievalName
          downloadUrl
        }
      }
    }
  }
```

The downloadUrl field of a ContentTransfer type of element returns a URL that can be accessed to stream the content for that element back to the client.

Creating a document

This example shows a mutation to create a document.

- The document can be filed in a folder at the time it is created by specifying the fileInFolderIdentifier argument.
- The documentProperties argument is an input type with fields corresponding to some common system properties that can be set.

- The properties field of this type allows any scalar (non-object) custom properties to be set. The type of the properties field is an array of custom GraphQL scalars. Each custom scalar is a key and value pair with the key part as a property symbolic name and the value as whatever type of value is appropriate for that property.
- The document can be checked in at the time it is created by specifying the checkinAction argument. The options in this field default to checking in a major version but the checkinAction argument itself must be specified to check in the document.

```
mutation {
    createDocument(
        repositoryIdentifier:"OS1"
        fileInFolderIdentifier: "/Folder for Browsing"
        documentProperties: {
            name: "Doc with Content"
            properties:[
              {CustomStringProperty:"Test value"}
              {CustomIntegerProperty:12345}
            ]
        }
        checkinAction: {}
    )
    {
        id
        name
    }
}
```

Check out a document

The following example shows the mutation to check out a document.

- It returns the source document that is being checked out.
- The newly checked out Reservation object can be obtained by selecting the **reservation** field.
- The newly checked out reservation properties can optionally be set inside the SubCheckoutActionInput object that is assigned to the **checkoutAction** field.

```
mutation {
    checkoutDocument(
     repositoryIdentifier:" OS1",
     identifier:"{40A9876C-0000-CF10-900C-02E3975438A5}",
     checkoutAction: {
          reservationProperties:
          [
               {DocumentTitle: "reservation doc"}
          ]
     })
  {
    id
    name
    reservation
    {
      id
      name
      dateCreated
    }
  }
}
```

Cancel checkout of a document (5.5.3 iFix1)

Checkout of document can be cancelled. The mutation of cancel checkout can take any version of the document in the VersionSeries as input. This mutation will delete the reservation object and cancel the checkout. The deleted Reservation object will be returned.
For example:

```
mutation
{
        cancelDocumentCheckout(repositoryIdentifier:" OS1",
        identifier:"{10AF776C-0000-C71D-BF17-73C39940CBE0}")
        {
                id
        }
}
```

Updating custom single value object properties

Properties can also be updated through an objectProperties field. Any properties can be updated through objectProperties by using a more verbose schema than what can be accomplished by using the properties field and the custom key and value scalars. The main purpose of objectProperties is to allow object valued properties to be specified.

The following example shows the setting of a single-value OVP using an object reference. Most of the time only the identifier field of the object reference needs to be specified. The classIdentifier field defaults to the required class of the property and the repositoryIdentifier field defaults to the source object’s repository.

```
mutation {
  updateDocument(
    repositoryIdentifier:"OS1"
    identifier:”{A0CC2C69-0000-C618-A0A4-3FA294C1AE57}”
    documentProperties:{
      properties: [
        {CustomStringProperty: "Updated value"}
      ]
      objectProperties: [
        {
          identifier:"PrimaryContact"
          objectReferenceValue:{
            identifier:"{A0CC2C69-0000-C618-A0A4-
3FA294C1AE56}"
          }
        }
      ]
    }
  )
  {
    id
    name
  }
}
```

Updating permissions

Permissions are an example of a dependent object in the CE object model. Most dependent objects are system classes corresponding to system properties and as such there are well known input fields for setting permissions.

The following mutation updates the permissions of a document.

```
mutation {
  updateDocument(
    repositoryIdentifier:"Daph1"
    identifier:"{A0CC2C69-0000-C618-A0A4-3FA294C1AE56}"
    documentProperties:{
      permissions:{
        replace:[
          {
            type:ACCESS_PERMISSION
            inheritableDepth:OBJECT_ONLY
            accessMask:998903
            subAccessPermission:{
              accessType:ALLOW
              granteeName:"CEAdmin"
            }
          }
        ]
      }
    }
  )
  {
    id
    name
  }
}
```

The input permissions to a mutation are an example of input types that correspond to a hierarchy of classes in the CE object model (the different classes of Permission). They are represented as input types by a field that indicates the concrete class of object (the type field). Other fields may be common to the base class of object, for example, inheritableDepth. The subAccessPermission sub field references another input type with fields specific to that concrete class of object.

Uploading content

Content must be uploaded by issuing a multi-part form POST to issue the GraphQL mutation and accompanying file upload part or parts. The uploaded data is specified by setting the value of a “content” field in the mutation using a GraphQL variable. The variable name itself maps to a part in the multi-part form that holds an uploaded file.

If there is only one content element in the document, it can be specified using the content field of the documentProperties argument. The following example shows what the mutation might look like:

```
mutation ($contvar:String) {
  createDocument(
    repositoryIdentifier:"OS1"
    fileInFolderIdentifier: "/Folder for Browsing"
    documentProperties: {
      name: "Doc with Content"
      content:$contvar
    }
    checkinAction: {}
  )
  {
    id
    name
  }
}
```

The document contentType in the previous example will be whatever mime type is submitted in the file part of the multi-part form.

This example shows how this mutation and corresponding file upload and the variable itself might be passed using a curl command. The variable values can be null as it is the variable name itself that maps to the part in the multi-part form.

```
curl localhost:8080/graphql -F graphql='{"query":"mutation
($contvar:String)
{createDocument(repositoryIdentifier:\"OS1\"
fileInFolderIdentifier: \"/Folder for Browsing\"
documentProperties: {name: \"Doc with Content\"
content:$contvar} checkinAction: {} ) { id name } }",
"variables":{"contvar":null} }' -F contvar=@test.txt
```

Multiple content elements with uploaded content (ContentTransfer) as well as references to content (ContentReference) can be updated using the contentElements field of the documentProperties argument.

The following example shows what the mutation might look like, still with only a single content element (although it might contain multiple elements):

```
mutation ($contvar:String) {
  createDocument(
    repositoryIdentifier:"OS1"
    fileInFolderIdentifier: "/Folder for Browsing"
    documentProperties: {
      name: "Doc with Content"
      contentElements:{
        replace: [
          {
            type: CONTENT_TRANSFER
            contentType: "text/plain"
            subContentTransfer: {
              content:$contvar
            }
          }
        ]
      }
```

```
        }
        checkinAction: {}
      )
      {
        id
        name
      }
    }
```

In V5.5.3 iFix1, the previous example allows caller to explicitly specify contentType that will overwrite the mime type submitted in the file part of the multi-part form.

The following example shows how this mutation might be specified using a curl command:

```
curl localhost:8080/graphql -F graphql='{"query":"mutation
($contvar:String)
{createDocument(repositoryIdentifier:\"OS1\"
fileInFolderIdentifier: \"/Folder for Browsing\"
documentProperties: {name: \"Doc with Content\"
contentElements:{replace: [{type: CONTENT_TRANSFER
contentType: \"text/plain\" subContentTransfer:
{content:$contvar} }]} } checkinAction: {} ) { id name }
}", "variables":{"contvar":null} }' -F contvar=@test.txt
```

Document searches

A search specifically for documents can be executed using the documents() query. The from, where, and orderBy arguments in general follow the CE query SQL syntax for those parts of the SQL statement. As explained earlier, the from argument can only specify a single class in V5.5.3, but in V5.5.3 iFix1 may contain joined classes (but must include a document class). The argument is optional to query for the base Document class, and if not specified, will query against the base Document class.

```
{
  documents(
    repositoryIdentifier:"OS1"
    from:"Document"
    where:"[DateCreated] > 20180815T070000Z AND
[IsCurrentVersion] = True"
    orderBy:"DocumentTitle"
    pageSize:20
  ) {
    documents {
      dateCreated
      id
```

```
          name
          majorVersionNumber
          minorVersionNumber
          mimeType
        }
        pageInfo {
          token
        }
      }
    }
```

If there are more results than can be returned in a page, a non-null string will be returned for the token field of PageInfo. The next page of results can be obtained using moreDocuments().

```
    {
      moreDocuments(
        token: "<insert_token>"
      ) {
        documents {
          dateCreated
          id
          name
          majorVersionNumber
          minorVersionNumber
          mimeType
        }
        pageInfo {
          token
        }
      }
    }
```

The output data should be shaped the same way as the original documents() call. The specific properties being selected and the depth of any complex nested data is established in the original call and cannot be changed in the continuation call.

Get total count of an object search (5.5.3 iFix1)

With 5.5.3 iFix1, various options can be passed to the searches. These are the options that are supported in the CE query SQL syntax. The COUNT_LIMIT option allows the totalCount field, a member of pageInfo, to be returned with a value. The option itself indicates the limit of matching items to be counted beyond the result items being returned as the current page of results. Refer to the Content Engine documentation for further details about this option. The following example involves a search for documents:

```
{
  documents(
    repositoryIdentifier:"OS1"
    from:"Document"
    where:"[DateCreated] > 20180815T070000Z AND
[IsCurrentVersion] = True"
    orderBy:"DocumentTitle"
    options:"COUNT_LIMIT 5000"
    pageSize:20
  ) {
    documents {
      dateCreated
      id
      name
      majorVersionNumber
      minorVersionNumber
      mimeType
    }
    pageInfo {
      token
      totalCount
    }
  }
}
```

With the previous search example, the **totalCount** field will be returned with the total number of matching items, up to the count supplied as the COUNT_LIMIT value. If COUNT_LIMIT is reached before all matching items have been counted, totalCount will be returned as a negative value to indicate this.

Generic repository object searches

A search for any searchable class of object can be executed with the repositoryObjects() query. This example assumes there is a CmAbstractPersistable sub-class called CustomContact. It has properties such as CustomString and CustomAddresses. CustomAddresses is an Enum object valued property associated with another CmAbstractPersistable sub-class of CustomAddress with properties such as CustomCity and CustomStreet.

The follwoing example shows how Enum properties, included with the generic properties field, can be further expanded and shaped in the output data.

```
{
  repositoryObjects(
    repositoryIdentifier:"OS1"
    from:"CustomContact"
    where:"DateCreated > 20190101T080000Z"
    orderBy:"DateCreated DESC"
    pageSize:20
  )
  {
    independentObjects {
      className
      objectReference {
        repositoryIdentifier
        classIdentifier
        identifier
      }
      properties(includes: ["Creator", "DateCreated",
"CustomString", "CustomAddresses"]) {
        id
        label
        type
        cardinality
        value
        ... on EnumProperty {
          independentObjectSetValue {
            independentObjects {
              className

properties(includes:["CustomCity","CustomStreet"]) {
                id
                label
                type
                cardinality
                value
              }
            }
          }
        }
      }
    }
    pageInfo {
      token
    }
  }
}
```

Additional pages can be obtained by calling moreIndependentObjects(). Any search can be continued by calling moreIndependentObjects(), even partial results returned for enum style fields and properties.

```
{
  moreIndependentObjects(
    token:"<insert_token>"
  )
  {
    independentObjects {
      className
      objectReference {
        repositoryIdentifier
        classIdentifier
        identifier
      }
      properties(includes: ["Creator", "DateCreated",
"CustomString", "CustomAddresses"]) {
        id
        label
        type
        cardinality
        value
        ... on EnumProperty {
          independentObjectSetValue {
            independentObjects {
              className

properties(includes:["CustomCity","CustomStreet"]) {
                id
                label
                type
                cardinality
                value
              }
            }
          }
        }
      }
    }
    pageInfo {
      token
    }
  }
}
```

Multiple repository object searches

Multiple related searches can be executed to get back one contiguous set of results using the repositoryObjectSearches() query. The individual searches could be to obtain all the containees of a folder, for example, using similar filter and sorting criteria for the different types of objects. The searches might also be a stored search of some type including both documents and folders.

```
{
  repositoryObjectSearches(
    repositoryIdentifier:"OS1"
    searches:[
      {
        from:"Folder"
        where:"[DateCreated] > 20180815T070000Z"
        orderBy:"FolderName ASC"
      }
      {
        from:"Document"
        where:"[DateCreated] > 20180815T070000Z AND
[IsCurrentVersion] = True"
        orderBy:"DocumentTitle ASC"
      }
    ]
    pageSize:15
  )
  {
    independentObjects {
      className
      objectReference {
        identifier
      }
      ... on Folder {
          dateCreated
          id
          name
          pathName
      }
      ... on Document {
        dateCreated
        id
        name
        majorVersionNumber
        minorVersionNumber
        mimeType
      }
    }
    pageInfo {
      token
    }
  }
}
```

The search is continued using the moreIndependentObjects() query as with a normal search. Once all the rows are exhausted for the first search, the second search is executed, and so on. A page that falls at the transition from one search to the next will have rows from both searches returned.

Generic object query

The generic object() query can be used to retrieve any object in the Content Engine even if there is not a specific query method available for that type of object.

The following example retrieves a custom Role object.

```
    {
      object(
        repositoryIdentifier:"OS1"
        classIdentifier:"CustomRole"
        identifier:"{D34D69DC-7359-45D9-9FCE-0C0711A49F07}"
      )
      {
        className
        properties(includes:["CustomInteger","CustomString"]) {
          id
          type
          cardinality
          label
          value
        }
      }
    }
```

Search with JOIN (5.5.3 iFix1)

With 5.5.3 iFix1, complex conditions involving joins can now be specified as the search criteria. With the single object type searches (documents(), folders() or repositoryObjects()) the properties selected are still from a single class but the search criteria can involve multiple joined classes. The following example involves a document class (LoanApplication) and a class representing a lender.

```
    {
```

```
  documents(
    repositoryIdentifier:"OS1"
    from:" LoanApplication a INNER JOIN Lender e ON a.Lender
= e.This"
    where:" e.LenderName LIKE 'Local%'"
    orderBy:" a.SubmittedDate DESC"
    pageSize:20
  ) {
    documents {
      id
      name
      properties(includes: ["Lender", "LoanType",
"LoanAmount", "SubmittedDate"]) {
        id
        label
        type
        cardinality
        value
        ... on ObjectProperty {
          objectValue {
            className
            ... on CmAbstractPersistable {
              creator
              dateCreated
              properties(includes: ["LenderName"]) {
                id
                label
                type
                cardinality
                value
              }
            }
          }
        }
      }
    }
    pageInfo {
      token
    }
  }
}
```

Full text search (5.5.3 iFix1)

With support for joined classes in 5.5.3 iFix1, full text searches can also be issued (involving a join with the ContentSearch class).

```
{
  documents(
    repositoryIdentifier: "OS1"
    from: "Document d inner join contentsearch c on d.this
= c.queriedobject"
    where: "(contains(d.*, 'FileNet OR IBM'))"
    orderBy: "d.DocumentTitle ASC"
    options: "COUNT_LIMIT 500"
    pageSize: 10) {
    documents {
      className
      id
      name
      mimeType
      creator
      dateCreated
      isReserved
      accessAllowed
    }
    pageInfo {
      totalCount
      token
    }
  }
}
```

Repository rows search (5.5.3 iFix1)

For use cases that involve returning properties from multiple joined classes, 5.5.3 iFix1 introduces a new repositoryRows() search. With this search the caller has control over and specifies the entire SQL statement and can be any SQL supported by the CE query syntax. The return information involves a collection of RepositoryRow types. The only field in this type is **properties**. Unlike the properties field returned in other types or interfaces such as Folder and Document, the properties field of RepositoryRow does not have an **includes** argument. The properties returned are exactly what is specified in the SQL statement.

Through the SQL statement, it is possible for the caller to specify an alias for certain selected properties. This may be desirable for example when selecting the same property on multiple classes. There is a new **alias** field of the CommonProperty interface that indicates whatever alias is specified by the caller. It is the same as the property symbolic name by default.

It is still possible to further shape object valued properties that are selected in the SQL just like shaping complex fields or object valued properties in the single object type searches. The following example selects Lender, an object valued property, as one of the row properties even though properties of the Lender class are being selected directly as one of the joined classes. This is for illustration purposes only; it is preferable to select the necessary properties from the joined class.

```
{
 repositoryRows(
   repositoryIdentifier:"OS1"
   sql:"SELECT a.Id, a.ApplicantName, a.DateLastModified,
a.Lender, a.LoanType, a.LoanAmount, d.Id AS LenderId,
d.LenderName, d.DateLastModified FROM LoanApplication a
INNER JOIN Lender d ON a.Lender = d.This WHERE d.LenderName
LIKE 'Local%' OPTIONS (COUNT_LIMIT 40)"
   pageSize:5
 )
 {
   repositoryRows {
     properties {
       id
       alias
       label
       type
       cardinality
       value
       ... on ObjectProperty {
         objectValue {
           className
                 ... on CmAbstractPersistable {
                      creator
                      dateLastModified
                      properties(includes: ["LenderName",
"PrimaryContact"]) {
                           id
                           label
                           type
                           cardinality
                           value
                      }
                 }
            }
          }
     }
   }
   pageInfo {
     totalCount
     token
   }
```

```
     }
    }
```

To retrieve additional pages of results started with repositoryRows(), use the moreRepositoryRows() query.

```
    {
     moreRepositoryRows(
       token:”<insert_token>”
     )
     {
       repositoryRows {
         properties {
           id
           alias
           label
           type
           cardinality
           value
           ... on ObjectProperty {
             objectValue {
               className
                     ... on CmAbstractPersistable {
                          creator
                          dateLastModified
                          properties(includes: ["LenderName",
    "PrimaryContact"]) {
                               id
                               label
                               type
                               cardinality
                               value
                          }
                     }
                }
             }
         }
       }
       pageInfo {
         totalCount
         token
       }
     }
    }
```

Generic object mutation

The generic changeObject() mutation can be used to execute most kinds of change actions in the Content Engine. It can be used to execute changes against types of objects for which there is not currently a specific mutation available or when change actions (pending actions in the CE Java API) must be combined in some custom manner.

The following example creates a custom role object:

```
mutation {
  changeObject(
    repositoryIdentifier:"OS1"
    properties:[
      {DisplayName:"My Custom Role"}
      {CustomString:"Test value"}
      {CustomInteger:93287}
    ]
    actions:[
      {
        type:CREATE
        subCreateAction:{
          classId:"CustomRole"
        }
      }
    ]
  )
  {
    className
    objectReference {
      repositoryIdentifier
      classIdentifier
      identifier
    }

     properties(includes:["Id","DisplayName","CustomString"
,"CustomInteger"]) {
      id
      label
      type
      cardinality
      value
    }
  }
}
```

The following example creates a custom CmTask object, connecting it to a Coordinator object (a Folder) and promoting it twice to the working state:

```
mutation {
  changeObject(
    repositoryIdentifier:"OS1"
    properties:[
      {CustomString:"A Task"}
      {CustomInteger:73216}
    ]
    objectProperties:[
      {
        identifier:"Coordinator"
        objectReferenceValue:{
          repositoryIdentifier:"OS1"
          classIdentifier:"Folder"
          identifier:"/Folder for Browsing/GraphQL Folder"
        }
      }
    ]
    actions:[
      {
        type:CREATE
        subCreateAction:{
          classId:"CustomTask"
        }
      }
      {
        type:CHANGE_STATE
        subChangeStateAction:{
          lifecycleAction:PROMOTE
        }
      }
      {
        type:CHANGE_STATE
        subChangeStateAction:{
          lifecycleAction:PROMOTE
        }
      }
    ]
  )
  {
    className
    objectReference {
      repositoryIdentifier
      classIdentifier
      identifier
    }
```

```
        properties( includes:["Id", "CustomString",
    "CustomInteger", "Coordinator"]) {
          id
          label
          type
          cardinality
          value
          ... on ObjectProperty {
            objectValue {
              className
              ... on Folder {
                id
                name
                pathName
              }
            }
          }
        }
      }
    }
```

Generic mutations with dependent objects

Any kinds of dependent objects, including system dependents, must be passed using generic input types when calling the generic changeObject() mutation. The reason for using the generic mutation for a document would be to combine other actions in an otherwise unsupported way but this is only for example purposes.

The following example updates a Document object with permissions.

```
    mutation {
      changeObject(
        repositoryIdentifier:"OS1"
        classIdentifier:"Document"
        identifier:"{90E1A36A-0000-C215-94FA-6732D7E65840}"
        objectProperties:[
          {
            identifier:"Permissions"
            dependentObjectListValue:{
              replace:[
                {
                  classIdentifier:"AccessPermission"
                  properties:[
                    {InheritableDepth:0}
                    {GranteeName:"CEAdmin"}
                    {AccessType:1}
                    {AccessMask:998903}
                  ]
                }
                {
                  classIdentifier:"AccessPermission"
                  properties:[
                    {InheritableDepth:0}
                    {GranteeName:"intg_admin"}
                    {AccessType:1}
                    {AccessMask:998903}
                  ]
                }
                {
                  classIdentifier:"AccessPermission"
                  properties:[
                    {InheritableDepth:0}
                    {GranteeName:"CEAdminGroup"}
                    {AccessType:1}
                    {AccessMask:998903}
                  ]
                }
                {
                  classIdentifier:"AccessPermission"
                  properties:[
                    {InheritableDepth:0}
                    {GranteeName:"#AUTHENTICATED-USERS"}
                    {AccessType:1}
                    {AccessMask:131201}
                  ]
                }
                {
                  classIdentifier:"AccessPermission"
                  properties:[
                    {InheritableDepth:0}
                    {GranteeName:"pwtest100"}
                    {AccessType:1}
                    {AccessMask:131201}
                  ]
                }
              ]
            }
          }
        ]
      )
      {
        className
        ... on Document {
          id
          name
          versionStatus
          majorVersionNumber
          minorVersionNumber
        }
      }
    }
```

Notices

Trademarks