Delivery GraphQL API reference

October 25, 2022 • Jan Cerman • 20 min read

GraphQL is a query language that prioritizes giving clients exactly the data they request and no more. Compared to the Delivery REST API, the Delivery GraphQL API exposes only a single endpoint for making queries.

Each Kontent.ai project has its own GraphQL schema generated from the project's content model. The schema is generated dynamically at request time and is always current.

Not familiar with GraphQL?

If you're starting out with GraphQL, we recommend you first learn the fundamentals in How to GraphQL.

Introduction

The Delivery GraphQL API is a read-only API. The GraphQL API accepts GET and POST requests to the base URL `https://graphql.kontent.ai/<YOUR_PROJECT_ID>`.

Use the API to deliver specific content to your website or app. The API responses are cached in a CDN. This makes the content quickly available from wherever you are.

Make POST or GET requests

You can make queries to the Delivery GraphQL API using either POST or GET requests. For both POST and GET requests, the base URL is the same. The difference is in how you specify your GraphQL query. The following examples show how to query for a specific article using POST and GET requests.

POST request

For POST requests, specify your GraphQL query in the body of your HTTP request. The maximum request body size is 8 kB.

cURL

```
curl --request POST \
  --url 'https://graphql.kontent.ai/<YOUR_PROJECT_ID>' \
  --header 'content-type: application/graphql'
  --data-raw '{article(codename:"my_article"){title}}'
```

Request body format depends on the `content-type` header

With the `content-type` header set to `application/graphql`, you need to provide your GraphQL query directly in the request body. If you set the header to `application/json`, wrap your GraphQL query in valid JSON like this `{"query":"{<YOUR_GRAPHQL_QUERY>}"}`.

GET request

For GET requests, specify your GraphQL query using the query parameter named `query`. With query parameters, you're limited by the maximum URL length of 2000 characters.

cURL

```
curl -g --request GET 'https://graphql.kontent.ai/<YOUR_PROJECT_ID>?query=
{article(codename:"on_roasts"){title}}'
```

Authorization

By default, the Delivery GraphQL API doesn't require authentication. If you've enabled secure access for your Kontent.ai project or want to use preview, you need to authorize your requests with an API key.

To get your API key, go to ⚙ **Project settings** > **API keys**. The Delivery API keys provide access to a single Kontent.ai project. You will need different API keys for each of your project environments.

To authorize your API requests, add the *Authorization* header to your request in the following format: `Authorization: Bearer <YOUR_API_KEY>`. Requests with an incorrect or missing `Authorization` header will fail with an error.

cURL

```
curl --request POST \
  --url 'https://graphql.kontent.ai/<YOUR_PROJECT_ID>' \
  --header 'content-type: application/graphql'
  --header 'Authorization: Bearer <YOUR_DELIVERY_API_KEY>'
  --data-raw '{<YOUR_GRAPHQL_QUERY>}'
```

Preview content

To get the latest version of your content items from your project, you need to use the preview base URL and authorize your requests.

- Preview base URL: `https://preview-graphql.kontent.ai/<YOUR_PROJECT_ID>`
- Authorization: Add the *Authorization* header to your request in the following format: `Authorization: Bearer <YOUR_PREVIEW_API_KEY>`.

To get your Preview API key, go to ⚙ **Project settings** > **API keys**.

cURL

```
curl --request POST \
  --url 'https://preview-graphql.kontent.ai/<YOUR_PROJECT_ID>' \
  --header 'content-type: application/graphql'
  --header 'Authorization: Bearer <YOUR_PREVIEW_API_KEY>'
  --data-raw '{<YOUR_GRAPHQL_QUERY>}'
```

Explore your GraphQL schema

You can explore the GraphQL schema of your Kontent.ai project in your web browser with GraphiQL online.

1. Open GraphiQL online in your browser.
2. Enter the GraphQL base URL of your project.
3. Start sending queries.

GraphiQL supports specifying custom headers, which you use to make API calls to projects with secure access enabled or to preview content.

API limitations

API requests limit

Requests made to the Delivery GraphQL API count towards the overall *API Calls* limit set in our Fair Use Policy.

Query complexity limit

Before processing your query, the GraphQL API calculates the following:

- Query complexity – The maximum number of content items and components your query can return. The maximum complexity is **2000**. See complexity examples.
- Query total depth level – The sum of maximum nesting levels of the queries in your API request. The maximum query depth is **124**. See depth examples.

Query complexity examples

If your query complexity exceeds 2000, the GraphQL API rejects your request and returns a `QUERY_TOO_COMPLEX` error.

You can find whether your authorized requests are close to the complexity limit by looking at the `X-Complexity` header.

Example 1

The following query can return up to 1000 Articles. Its complexity is 1000.

GraphQL

```graphql
query GetArticles {
  article_All(limit: 1000) {
    items {
      title
    }
  }
}
```

Example 2

The following query can return:

- Up to 100 Articles – complexity is 100
- Up to 20 linked content items for each Article – complexity is 100 × 20

The total complexity of the query is 2100.

GraphQL

```graphql
query GetArticlesWithRelatedArticles {
  article_All(limit: 100) {
    items {
      title
      relatedArticles(limit: 20) {
        items {
          _system_ {
            name
          }
        }
      }
    }
  }
}
```

Query total depth limit examples

If your query exceeds the nesting limit of 124, the GraphQL API rejects your request and returns an error.

Example 1

The total depth of the following query is 6. The root query in the API request goes six levels deep.

GraphQL

```graphql
# Total query depth = 6
query GetImagesInArticle {
  article_All {      # level 1
    items {          # level 2
      bodyCopy {     # level 3
        assets {     # level 4
          items {    # level 5
            url      # level 6
          }
        }
      }
    }
  }
}
```

Example 2

The total depth of the following query is 6. The two root queries both go three levels deep. The sum of the maximum depth of the root queries is 6.

GraphQL

```graphql
# Total query depth = 6
query GetNavigationWithArticles {
  article_All { # level 1
    items {     # level 2
      title     # level 3
      url       # level 3
    }
  }
  navigationItem_All {
    items {     # level 2
      title     # level 3
      url       # level 3
    }
  }
}
```

Rate limitation

The GraphQL API enforces a rate limitation based on resource consumption.

For cached requests served from our CDN, we don't enforce any rate limits. You can make an unlimited number of repeated requests to the CDN.

For uncached requests that reach the GraphQL API, we enforce a rate limitation of 1000 resource units per second and 40,000 resource units per minute. To see how many resources your requests consumed, check the `X-Request-Charge` header.

When you reach the resource limit for a given time period, the API rejects the request and responds with a 429 HTTP error. This error comes with the `Retry-After` header that tells you how many seconds you need to wait before retrying your request. Each failed request is perfectly safe to retry. If you begin to receive 429 errors, reduce the frequency of your requests.

Errors

The Delivery GraphQL API returns standard HTTP status codes to indicate the success or failure of a request. In general, status codes in the `2xx` range indicate a successful request, status codes in the `4xx` range indicate errors caused by an incorrect input (for example, providing incorrect API key), and status codes in the `5xx` range indicate an error on our side.

If your query contains a mistake, the API might return a `200` with an error message in the response body.

Status code	Description
`200` OK	The request was successful. However, the API might return an error message. For example, if your query is too complex or fails to execute correctly.
`400` Bad Request	The request wasn't successful. Check your request for a missing required parameter or an invalid query parameter value.
`403` Forbidden	The API request was forbidden. Check if your subscription plan comes with GraphQL enabled.
`404` Not Found	The specified project doesn't exist.
`405` Method Not Allowed	The requested HTTP method is not supported for the specified resource.
`429` Too Many Requests	The rate limit for the API has been exceeded. Try your request again after a few seconds as specified in the `Retry-After` header.
`5xx` Internal Error or Service Unavailable	Something went wrong on our side. Try your request again after a few seconds and use a retry policy.

Query content

For every content type in your project, the API generates two GraphQL root queries. For example, for a content type named *Article*, you get the following queries:

- `article` query for retrieving a single content item.
- `article_All` collection query for retrieving multiple content items.

Both of these queries work with a GraphQL type named *Article* that is generated from the content type.

Get a content item

To get a single content item, you need to provide the item's type and the item's identifier. The identifier can be either codename or internal ID.

GraphQL

```graphql
query GetArticleByCodename {
  article(codename: "my_article") {
    title
    slug
  }
}

query GetArticleById {
  article(id: "c706af79-2a4f-41e4-9bec-8c8383e00944") {
    title
    slug
  }
}
```

List content items

To get a list of content items, you need to use the `<typeName>_All` (as in `article_All`) query. By default, the items are ordered alphabetically by codename.

GraphQL

```
query GetArticles {
  article_All {
    items {
      title
      slug
    }
  }
}
```

Order content items

If you want to get your items in a specific order, use the `order` argument in your query. The `order` argument requires that you provide a field name and specify whether the order should be ascending or descending.

GraphQL

```
query GetOrderedItems {
  # Get articles ordered by their title from Z to A
  article_All(order: {title: desc}) {
    items {
      title
    }
  }
  # Blog posts ordered by codename from A to Z
  post_All(order: {_system_: {codename: asc}}) {
    items {
      title
    }
  }
}
```

Get localized content

By default, the GraphQL API returns content in the default language. To get content items in a specific language, use the `languageFilter` argument in your queries.

If the requested content items don't have content in the specified language, the API follows language fallbacks as specified in your project's localization settings. To check the language of the returned content items, specify the `language` field of the content item's `_system_` object.

GraphQL

```graphql
query GetSpanishArticles {
  # Requests articles in Spanish
  article_All(languageFilter: {languageCodename: "es-ES"}) {
    items {
      _system_ {
        language {
          _system_ {
            codename
          }
        }
      }
      title
      slug
    }
  }
}
```

Filter content

To retrieve content items based on a specific criteria, you can filter the content items using the `where` argument. The `where` argument can be applied only to root collection queries, which are based on your content types such as `article_All`, `navigationItem_All`, and so on.

In the `where` argument, you can specify a single condition with one filter or combine multiple filters using the AND and OR operators.

GraphQL

```graphql
query GetNonArchivedArticles {
  article_All(where: {_system_: {workflowStep: {notEq: "archived"}}}) {
    items {
      title
    }
  }
}
```

Filters

You can use the following filters on specific system fields and element fields.

- The filterable system fields are limited to the `id`, `codename`, `name`, `lastModified`, `collection`, `workflowStep`, and `language` fields.
- The filterable element fields are limited to fields based on the date & time, linked items, multiple choice, number, text, and taxonomy elements.

Filter	Description	Example condition	Use with types

`eq`	Checks whether the field value matches exactly to the specified filter value.	`where: {title: {eq: "My article"}}`
		 `where: {_system_: {id: {eq: "<item-id>"}}}`
			DateTime, Number, String

`notEq`	Checks if the field value is different than the specified value.	`where: {_system_: {workflowStep: {notEq: "archived"}}}`	DateTime, Number, String
`lt`	Checks if the field's value is less than the specified value.	`where: {postDate: { lt: "2021-01-01T00:00:00Z"}}`	DateTime, Number
`lte`	Checks if the field's datetime value is less than or equal to the specified value.	`(where: {_system_: {lastModified: {lte: "2021-01-01T00:00:00Z"}}}`	DateTime, Number
`gt`	Checks if the field's datetime value is greater than the specified value.	`where: {postDate: {gt: "2021-01-01T00:00:00Z"}}`	DateTime, Number
`gte`	Checks if the field's datetime value is greater than or equal to the specified value.	`where: {price: {gte: 10.5}}`	DateTime, Number
`in`	Checks if the field value matches at least one of the specified array values.	`where: {_system_: {collection: {in: ["hr", "marketing"]}}}`	DateTime, Number, String
`notIn`	Checks if the field value is different than the specified array values.	`where: {_system_: {collection: {notIn: ["default"]}}}`	DateTime, Number, String
`isEmpty`	Checks if the field value is null or empty.	`where: {relatedArticles: { isEmpty: true }}`	DateTime, Number, String, Array
`containsAll`	Checks if the array field value matches all of the specified array values.	`where: {topic: { containsAll: ["gadget", "security"] }}`	Array
`containsAny`	Checks if the field value matches at least one of the specified array values.	`where: {topic: { containsAny: ["nature"] }}`	Array

Combine filters with AND and OR

To combine multiple filters in your `where` arguments, you can use the `AND` and `OR` operators. Both operators take an array of at least two values, and can be nested in one another.

GraphQL

```graphql
query GetArticlesByComplexCondition {
  article_All(where: {AND: [
    {OR: [
      {title: {eq: "On Roasts"}},
      {title: {eq: "Donate with us"}},
    ]},
    {summary: {isEmpty: false}}
  ]}) {
    items {
      title
    }
  }
}
```

Paging

To paginate through collection fields, use the `limit` and `offset` arguments combined with the predefined `totalCount` field.

- `limit` (`Int`) – Specifies the number of objects to retrieve. If not specified, the API returns up to 10 objects. The maximum `limit` value is 20.
- `offset` (`Int`) – Specifies the number of objects to skip. If not specified, the `offset` is 0 and the API returns the first page of results.

GraphQL

```graphql
# Paginating articles
query GetPaginatedArticles {
  article_All(limit: 10, offset: 10) {
    items {
      title
    }
    offset      # Returns the specified offset
    limit       # Returns the specified limit
    totalCount  # Returns the total number of content items that match the query
  }
}

# Paginating linked content in rich text
query GetPaginatedLinkedItems {
  article_All {
    items {
      bodyCopy {
        html
        linkedItems(offset: 10, limit: 10) {
          items {
            _system_ {
              codename
            }
          }
          offset      # Returns the specified offset
          limit       # Returns the specified limit
          totalCount  # Returns the total number of content items that match the query
        }
      }
    }
  }
}
```

Schemas

The GraphQL schema for your Kontent.ai project is dynamically generated based on your content model. The schema is generated at request time and is always current. This also means that any changes to your content model immediately affect your GraphQL schema.

How schema names are generated

The names of fields and types in your GraphQL schema are generated from the codenames of your content types, content type snippets, and the elements that define them.

The original codename is stripped of underscores (`_`), converted to PascalCase, and used as a GraphQL name. For content type snippets, the converted codename is prefixed with an underscore.

The GraphQL type names and field names must be unique. If two codenames lead to an identical GraphQL name, the GraphQL schema fails to build correctly. For example, the codenames `button` and `button_` would lead to the same GraphQL name. In such case, adjust your codenames to avoid collisions.

Reserved field names

The codenames of your types, snippets, or elements must not be one of the following: `Array` , `Boolean` , `DateTime` , `Float` , `Guid` , `Int` , `String` .

Examples of codename conversion to GraphQL names:

Original codename	GraphQL object type name	GraphQL query name
Content type `article`	`Article` for single item `Article_All` for multiple items	`article` for single item `article_All` for multiple items
Content type `fact_about_us`	`FactAboutUs` for single item `FactAboutUs_All` for multiple items	`factAboutUs` for single item `factAboutUs_All` for multiple items
Content type snippet `metadata`	`_metadata` for the GraphQL field	None. Elements under the content type snippet are available as subfields of the `_metadata` field.

Collection fields

The collection fields in the GraphQL schema can hold objects such as content items, assets, taxonomy terms, linked items, and more. You can use filters and paging on the collection fields to specify what kind of objects you want.

GraphQL

```graphql
type Article implements _Item {
  ...
  teaserImage(          # Asset element
    offset: Int!
    limit: Int!
    totalCount: Int!
  ): _AssetCollection!
  relatedArticles(      # Linked items element
    offset: Int!
    limit: Int!
    totalCount: Int!
  ): _ItemCollection!
  personas(             # Taxonomy element
    offset: Int!
    limit: Int!
    totalCount: Int!
  ): _TaxonomyTermCollection!
}
```

When retrieving lists of objects using collection fields, you need to specify the `items` field and at least one of its subfields. You can also specify the predefined `totalCount` field to find how many items the collection contains.

GraphQL

```graphql
query HowToQueryCollectionFields {
  article_All {
    items {
      teaserImage {
        items {
          name
        }
        totalCount # Returns the total number of assets inserted in the asset element
      }
      personas {
        items {
          _system_ {
            codename
          }
        }
        totalCount # Returns the total number of selected taxonomy terms
      }
      relatedArticles {
        items {
          ... on Article {
            title
          }
        }
        totalCount # Returns the total number of items linked in the element
      }
    }
    totalCount # Returns the total number of articles matching the query
  }
}
```

Content items

Every GraphQL type for retrieving content items consists of the following fields:

- The predefined `_system_` field with the content item's metadata.
- The individual fields for every content element as defined by the item's content type.

For example, the following is a GraphQL type for the content type named *Article*.

GraphQL

```graphql
type Article implements _Item {
  _system_: _Sys!         # The content item's metadata.
  title: String!          # Text element
  teaserImage(            # Asset element
    offset: Int = 0
    limit: Int = 10
  ): _AssetCollection!
  postDate: DateTime      # Date & time element
  bodyCopy: _RichText!  # Rich text element
  relatedArticles(      # Linked items element
    offset: Int = 0
    limit: Int = 10
  ): _ItemCollection!
  personas(             # Taxonomy element
    offset: Int = 0
    limit: Int = 10
  ): _TaxonomyTermCollection!
  urlPattern: String!   # URL slug element
}
```

System metadata in the `_system_` fields

Some of the GraphQL types come with the `_system_` field. The `_system_` field is defined by the `_Sys` type, which contains your content items' metadata. For example, the content item's last modification date, codename, and so on.

The `_Sys` type is used in the GraphQL types generated from your content types.

GraphQL

```
type _Sys {
  name: String!     # The content item's display name.
  codename: String! # The content item's codename.
  language: _Language!  # Metadata of the content item's language.
  type: _ContentType!   # Metadata of the content item's content type.
  lastModified: DateTime! # ISO-8601 formatted date and time of the last change to user-content of the content item. The value is not affected when moving content items through workflow steps.
  collection: _Collection!      # Metadata of the content item's collection.
  workflowStep: _WorkflowStep!  # Metadata of the content item's current workflow step.
  id: Guid! # The content item's internal ID.
}
```

Partial system metadata

You'll also find the `_system_` field in the predefined system GraphQL types such as `_Language`, `_ContentType`, `_Collection`, and `_WorkflowStep`. In these types, the `_system_` field contains partial metadata such as the object's `name` or `codename`.

GraphQL

```
type _Collection {
  _system_: _CollectionSys! # The collection's predefined system fields.
}

type _CollectionSys {
  codename: String! # The collection's codename.
}

```

```
type _ContentType {
  _system_: _ContentTypeSys! # The content type's predefined system fields.
}

type _ContentTypeSys {
  name: String!     # The content type's display name.
  codename: String! # The content type's codename.
}

type _Language {
  _system_: _LanguageSys! # The language's predefined system fields.
}

type _LanguageSys {
  name: String!     # The language's display name.
  codename: String! # The language's codename.
}

type _WorkflowStep {
  _system_: _WorkflowStepSys! # The workflow step's predefined system fields.
}

type _WorkflowStepSys {
  codename: String! # The workflow step's codename.
}
```

Content element schemas

Assets

Assets have two predefined GraphQL types. The used type differs based on whether the asset itself is used in asset elements or rich text elements. Both types define a set of common fields: `url`, `name`, `description`, `type`, `size`, `width`, and `height`.

Assets in asset elements

When the asset is used in asset elements, the schema uses the predefined `_Asset` type. The `_Asset` type contains the `renditions` field referencing customized images.

To display customized images on the website or an app, you first need to apply custom query parameters to the asset's original URL.

GraphQL

```
type _Asset implements _AssetInterface {
  renditions: _AssetRenditionCollection # List of renditions available for this asset.
  url: String!  # The asset's absolute URL.
  name: String  # The asset's display name.
  description: String   # The asset's alt text description for a specific language.
  type: String  # The file's MIME type.
  size: Int     # The file's size in bytes.
  width: Int    # The image's width in pixels.
  height: Int   # The image's height in pixels.
}

type _AssetRenditionCollection {
  offset: Int!
  limit: Int!
  totalCount: Int!
  items: [_Rendition!]! # Individual asset rendition objects.
}

type _Rendition {
  preset: String!         # The image preset's codename.
  preset_id: String!      # The image preset's ID.
  rendition_id: String!  # The rendition's ID.
  query: String!          # Parameters set for the image rendition. Need to be
concatenated with the original asset's URL to customize the image.
  width: Int!              # The rendition's width.
  height: Int!           # The rendition's height.
}
```

Assets in rich text elements

When the asset is used in rich text elements, the schema uses the predefined `_RichTextAsset` type. The `_RichTextAsset` type has with an additional field named `imageId`. The `imageId` field helps you identify the assets that are referenced in the rich text's `html` field.

GraphQL

```
type _RichTextAsset implements _AssetInterface {
  imageId: String!  # Identifier of the asset as used in the rich text element.
  url: String!      # The asset's absolute URL.
  name: String      # The asset's display name.
  description: String   # The asset's alt text description for a specific language.
  type: String      # The file's MIME type.
  size: Int         # The file's size in bytes.
  width: Int        # The image's width in pixels.
  height: Int       # The image's height in pixels.
}
```

Content type snippet

Content type snippets are transformed into separate GraphQL types whose name is prefixed with an underscore character (`_`). To query the elements of the snippet, you need to specify the subfields of the snippet field.

For example, a snippet named *Metadata* will have the following GraphQL type.

GraphQL

```graphql
type Metadata {
  metaTitle: String!
  metaDescription: String!
}

type Article implements _Item {
  _system_: _Sys! # The content item's predefined system fields.
  ...
  _metadata: Metadata! # The Metadata type with subfields for each element in the
snippet
}
```

Custom element

Custom elements are transformed into a GraphQL type with a single `value` field.

GraphQL

```graphql
type _CustomElement {
  value: String # The custom element's value as a string.
}
```

Create your own custom element or discover existing ones in the custom element sample gallery.

Date and time element

Date & time elements are transformed into GraphQL `DateTime` fields. The value of `DateTime` fields is an ISO-8610 formatted string such as `2021-11-18T08:43:19Z`.

Linked items element

Linked items elements are transformed into GraphQL collection fields.

GraphQL

```graphql
type Article implements _Item {
  _system_: _Sys! # The content item's predefined system fields.
  ...
  relatedContent: _ItemCollection! # Linked items element without content type
limitation
}
```

If the linked items element is limited to a specific content type, the GraphQL field's data type reflects the limitation.

GraphQL

```graphql
type Article implements _Item {
  _system_: _Sys! # The content item's predefined system fields.
  ...
  relatedArticle: Article # Linked items element limited to a single Article content type
  relatedArticles: Article_RelatedArticles_Collection! # Linked items element limited Article content types
}

type Article_RelatedArticles_Collection {
  offset: Int!
  limit: Int!
  items: [Article!]!
}
```

Multiple choice element

Multiple choice elements are transformed into a GraphQL collection field that lets you query the selected multiple choice options.

GraphQL

```graphql
type _MultipleChoiceOptionCollection {
  offset: Int!
  limit: Int!
  items: [_MultipleChoiceOption!]! # Specifies the selected multiple choice options.
}

type _MultipleChoiceOption {
  _system_: _MultipleChoiceOptionSys! # The multiple choice option's predefined system fields.
}

type _MultipleChoiceOptionSys {
  name: String!      # The multiple choice option's display name.
  codename: String! # The multiple choice option's codename.
}
```

Number element

Number elements are transformed into GraphQL `Float` fields.

Rich text element

Rich text elements have a predefined GraphQL type, which contains the `html`, `itemHyperlinks`, `linkedItems`, `components`, and `assets` fields. The data type of the `linkedItems`, `components` and `itemHyperlinks` fields can change depending on the limitations set for the rich text element.

With no limitations for the type of linked items, components, and item links, the GraphQL type for rich text is the following.

GraphQL

```graphql
type Article implements _Item {
  _system_: _Sys!   # The content item's predefined system fields.
  ...
  bodyCopy: _RichText! # Rich text element without type limitations
}

type _RichText {
  html: String! # The rich text's HTML output. Contains references to assets, links to content items, linked content, and components.
  itemHyperlinks: _ItemCollection!  # Contains the content items referenced in hyperlinks.
  assets: _RichTextAssetCollection! # Contains the assets inserted into the rich text.
  linkedItems: _ItemCollection!     # Contains the content items inserted into the rich text.
  components: _ItemCollection!      # Contains the components inserted into the rich text.
}
```

If the components, linked items, or item links you can insert into rich text are limited to a specific content type, the GraphQL API reflects the limitation by generating several custom types.

For example, if the *Body copy* rich text element in an *Article* content type has the components and linked items limited to a content type named *Tweet*, you'll see the following GraphQL types.

GraphQL

```graphql
type Article implements _Item {
  _system_: _Sys!   # The content item's predefined system fields.
  ...
  bodyCopy: Article_BodyCopy! # Rich text element with custom type limitations
}

type Article_BodyCopy {
  html: String!
  itemHyperlinks: _ItemCollection!
  assets: _RichTextAssetCollection!
  components: Article_BodyCopy_Components_Collection!
  linkedItems: Article_BodyCopy_LinkedItems_Collection!
}

type Article_BodyCopy_Components_Collection {
  offset: Int!
  limit: Int!
  items: [Tweet!]!
}
```

```

type Article_BodyCopy_LinkedItems_Collection {
  offset: Int!
  limit: Int!
  items: [Tweet!]!
}
```

Taxonomy element

Taxonomy elements are transformed into a GraphQL collection field that lets you query the selected taxonomy terms.

GraphQL

```graphql
type _TaxonomyTermCollection {
  offset: Int!
  limit: Int!
  items: [_TaxonomyTerm!]! # Specifies the selected taxonomy terms.
}

type _TaxonomyTerm {
  _system_: _TaxonomyTermSys! # The taxonomy term's predefined system fields.
}

type _TaxonomyTermSys {
  name: String!     # The taxonomy term's display name.
  codename: String! # The taxonomy term's codename.
}
```

Text element

Text elements are transformed into GraphQL `String` fields. These fields contain plaintext.

URL slug element

URL slug elements are transformed into GraphQL `String` fields. These fields contain plaintext.