

# How to model complex data types in Azure Cognitive Search

11/27/2020 • 8 minutes to read •  +6

## In this article

[Example of a complex structure](#)

[Indexing complex types](#)

[Creating complex fields](#)

[Updating complex fields](#)

[Searching complex fields](#)

[Selecting complex fields](#)

[Filter, facet, and sort complex fields](#)

[Next steps](#)

External datasets used to populate an Azure Cognitive Search index can come in many shapes. Sometimes they include hierarchical or nested substructures. Examples might include multiple addresses for a single customer, multiple colors and sizes for a single SKU, multiple authors of a single book, and so on. In modeling terms, you might see these structures referred to as *complex*, *compound*, *composite*, or *aggregate* data types. The term Azure Cognitive Search uses for this concept is **complex type**. In Azure Cognitive Search, complex types are modeled using **complex fields**. A complex field is a field that contains children (sub-fields) which can be of any data type, including other complex types. This works in a similar way as structured data types in a programming language.

Complex fields represent either a single object in the document, or an array of objects, depending on the data type. Fields of type `Edm.ComplexType` represent single objects, while fields of type `Collection(Edm.ComplexType)` represent arrays of objects.

Azure Cognitive Search natively supports complex types and collections. These types allow you to model almost any JSON structure in an Azure Cognitive Search index. In previous versions of Azure Cognitive Search APIs, only flattened row sets

could be imported. In the newest version, your index can now more closely correspond to source data. In other words, if your source data has complex types, your index can have complex types also.

To get started, we recommend the [Hotels data set](#), which you can load in the **Import data** wizard in the Azure portal. The wizard detects complex types in the source and suggests an index schema based on the detected structures.

#### ⓘ Note


Support for complex types became generally available starting in `api-version=2019-05-06`.

If your search solution is built on earlier workarounds of flattened datasets in a collection, you should change your index to include complex types as supported in the newest API version. For more information about upgrading API versions, see [Upgrade to the newest REST API version](#) or [Upgrade to the newest .NET SDK version](#).

## Example of a complex structure

The following JSON document is composed of simple fields and complex fields. Complex fields, such as `Address` and `Rooms`, have sub-fields. `Address` has a single set of values for those sub-fields, since it's a single object in the document. In contrast, `Rooms` has multiple sets of values for its sub-fields, one for each object in the collection.

JSON

 Copy

```
{
  "HotelId": "1",
  "HotelName": "Secret Point Motel",
  "Description": "Ideally located on the main commercial artery of the city in the heart of New York.",
  "Tags": ["Free wifi", "on-site parking", "indoor pool", "continental breakfast"]
  "Address": {
    "StreetAddress": "677 5th Ave",
    "City": "New York",
    "StateProvince": "NY"
  }
}
```

```
    },
    "Rooms": [
      {
        "Description": "Budget Room, 1 Queen Bed (Cityside)",
        "RoomNumber": 1105,
        "BaseRate": 96.99,
      },
      {
        "Description": "Deluxe Room, 2 Double Beds (City View)",
        "Type": "Deluxe Room",
        "BaseRate": 150.99,
      }
      . . .
    ]
  }
}
```

## Indexing complex types

During indexing, you can have a maximum of 3000 elements across all complex collections within a single document. An element of a complex collection is a member of that collection, so in the case of Rooms (the only complex collection in the Hotel example), each room is an element. In the example above, if the "Secret Point Motel" had 500 rooms, the hotel document would have 500 room elements. For nested complex collections, each nested element is also counted, in addition to the outer (parent) element.

This limit applies only to complex collections, and not complex types (like Address) or string collections (like Tags).

## Creating complex fields

As with any index definition, you can use the portal, [REST API](#), or [.NET SDK](#) to create a schema that includes complex types.

The following example shows a JSON index schema with simple fields, collections, and complex types. Notice that within a complex type, each sub-field has a type and may have attributes, just as top-level fields do. The schema corresponds to the

example data above. `Address` is a complex field that isn't a collection (a hotel has one address). `Rooms` is a complex collection field (a hotel has many rooms).

JSON

 Copy

```
{
  "name": "hotels",
  "fields": [
    { "name": "HotelId", "type": "Edm.String", "key": true, "filterable": true },
    { "name": "HotelName", "type": "Edm.String", "searchable": true, "filterable": false },
    { "name": "Description", "type": "Edm.String", "searchable": true, "analyzer": "en.lucene" },
    { "name": "Address", "type": "Edm.ComplexType",
      "fields": [
        { "name": "StreetAddress", "type": "Edm.String", "filterable": false, "sortable": false, "facetable": false, "searchable": true },
        { "name": "City", "type": "Edm.String", "searchable": true, "filterable": true, "sortable": true, "facetable": true },
        { "name": "StateProvince", "type": "Edm.String", "searchable": true, "filterable": true, "sortable": true, "facetable": true }
      ]
    },
    { "name": "Rooms", "type": "Collection(Edm.ComplexType)",
      "fields": [
        { "name": "Description", "type": "Edm.String", "searchable": true, "analyzer": "en.lucene" },
        { "name": "Type", "type": "Edm.String", "searchable": true },
        { "name": "BaseRate", "type": "Edm.Double", "filterable": true, "facetable": true }
      ]
    }
  ]
}
```

## Updating complex fields

All of the [reindexing rules](#) that apply to fields in general still apply to complex fields. Restating a few of the main rules here, adding a field to a complex type doesn't require an index rebuild, but most modifications do.

## Structural updates to the definition

You can add new sub-fields to a complex field at any time without the need for an index rebuild. For example, adding "ZipCode" to `Address` or "Amenities" to `Rooms` is allowed, just like adding a top-level field to an index. Existing documents have a null value for new fields until you explicitly populate those fields by updating your data.

Notice that within a complex type, each sub-field has a type and may have attributes, just as top-level fields do

## Data updates

Updating existing documents in an index with the `upload` action works the same way for complex and simple fields -- all fields are replaced. However, `merge` (or `mergeOrUpload` when applied to an existing document) doesn't work the same across all fields. Specifically, `merge` doesn't support merging elements within a collection. This limitation exists for collections of primitive types and complex collections. To update a collection, you'll need to retrieve the full collection value, make changes, and then include the new collection in the Index API request.

## Searching complex fields

Free-form search expressions work as expected with complex types. If any searchable field or sub-field anywhere in a document matches, then the document itself is a match.

Queries get more nuanced when you have multiple terms and operators, and some terms have field names specified, as is possible with the [Lucene syntax](#). For example, this query attempts to match two terms, "Portland" and "OR", against two sub-fields of the Address field:

```
search=Address/City:Portland AND Address/State:OR
```

Queries like this are *uncorrelated* for full-text search, unlike filters. In filters, queries over sub-fields of a complex collection are correlated using range variables in [any or all](#). The Lucene query above returns documents containing both "Portland, Maine" and "Portland, Oregon", along with other cities in Oregon. This happens because each clause applies to all values of its field in the entire document, so there's no concept of a "current sub-document". For more information on this, see [Understanding OData collection filters in Azure Cognitive Search](#).

## Selecting complex fields

The `$select` parameter is used to choose which fields are returned in search results. To use this parameter to select specific sub-fields of a complex field, include the parent field and sub-field separated by a slash (`/`).

```
$select=HotelName, Address/City, Rooms/BaseRate
```

Fields must be marked as Retrievable in the index if you want them in search results. Only fields marked as Retrievable can be used in a `$select` statement.

## Filter, facet, and sort complex fields

The same [OData path syntax](#) used for filtering and fielded searches can also be used for faceting, sorting, and selecting fields in a search request. For complex types, rules apply that govern which sub-fields can be marked as sortable or facetable. For more information on these rules, see the [Create Index API reference](#).

## Faceting sub-fields

Any sub-field can be marked as facetable unless it is of type `Edm.GeographyPoint` or `Collection(Edm.GeographyPoint)`.

The document counts returned in the facet results are calculated for the parent document (a hotel), not the sub-documents in a complex collection (rooms). For example, suppose a hotel has 20 rooms of type "suite". Given this facet parameter

`facet=Rooms/Type`, the facet count will be one for the hotel, not 20 for the rooms.

## Sorting complex fields

Sort operations apply to documents (Hotels) and not sub-documents (Rooms). When you have a complex type collection, such as Rooms, it's important to realize that you can't sort on Rooms at all. In fact, you can't sort on any collection.

Sort operations work when fields have a single value per document, whether the field is a simple field, or a sub-field in a complex type. For example, `Address/City` is allowed to be sortable because there's only one address per hotel, so

`$orderby=Address/City` will sort hotels by city.

## Filtering on complex fields

You can refer to sub-fields of a complex field in a filter expression. Just use the same [OData path syntax](#) that's used for faceting, sorting, and selecting fields. For example, the following filter will return all hotels in Canada:

```
$filter=Address/Country eq 'Canada'
```

To filter on a complex collection field, you can use a **lambda expression** with the [any and all operators](#). In that case, the **range variable** of the lambda expression is an object with sub-fields. You can refer to those sub-fields with the standard OData path syntax. For example, the following filter will return all hotels with at least one deluxe room and all non-smoking rooms:

```
$filter=Rooms/any(room: room/Type eq 'Deluxe Room') and Rooms/all(room: not room/SmokingAllowed)
```

As with top-level simple fields, simple sub-fields of complex fields can only be included in filters if they have the **filterable** attribute set to `true` in the index definition. For more information, see the [Create Index API reference](#).

# Next steps

Try the [Hotels data set](#) in the **Import data** wizard. You'll need the Cosmos DB connection information provided in the readme to access the data.

With that information in hand, your first step in the wizard is to create a new Azure Cosmos DB data source. Further on in the wizard, when you get to the target index page, you'll see an index with complex types. Create and load this index, and then execute queries to understand the new structure.

Quickstart: portal wizard for import, indexing, and queries

---

Is this page helpful?

 Yes  No

---