Akeneo.NET Documentation

Release

Akeneo.NET contribs

Jun 07, 2017

Contents:

Akeneo.NET is a .NET client for Akeneo PIM's RESTful API. It can be run on .NET Core as well as .NET Framework. The source code to the project can be found on Github.

CHAPTER 1

Getting Started

Install Akeneo PIM

It is assumed that Akeneo PIM is installed. To install Akeneo PIM, follow the official installation instructions or use this `docker-compose.yml` file to get started on Docker with one simple command.

Creating OAuth credentials

The calls to the API is authenticated via oauth. To create credentials follow the official instructions. Note, that if the `pardahlman/akene` docker image is used, a client is created during startup. Look through the containers logs and locate an entry similar to this

```
A new client has been added.
client_id: 1_27xlkd53wou8ogggwwwksk48s0sgsoogwkowws8wko88gcs0os
secret: 65rfqpc5a3okws0w4k0kgcwswwg0ggwg48wc40gcckso88sk44
  - Done.
```

Install Akeneo.NET client

The .NET client is published on NuGet. To install it, search for `Akeneo.NET` in the NuGet user interface, or open the Package Manager Console and enter

```
PM> Install-Package Akeneo.NET
```

Create client

The client is created with an `AkeneoOptions` options object. All fields are required in order to successfully connect to the PIM:

- `ApiEndpoint` is the URL to Akeneo PIM without any trailing slash
- `ClientId` is the OAuth client id (generated)
- `ClientSecret` is the OAuth client secret (generated)
- `UserName` is the name of a user in the PIM
- `Password` is the corresponding password the the user in the PIM

Below is an example of a complete options object.

```
var options = new AkeneoOptions
{
    ApiEndpoint = new Uri("http://localhost:8080"),
    ClientId = "1_27xlkd53wou8ogggwwwksk48s0sgsoogwkowws8wko88gcs0os",
    ClientSecret = "65rfqpc5a3okws0w4k0kgcwswwg0ggwg48wc40gcckso88sk44",
    UserName = "admin",
    Password = "admin"
}
```

That's it! Create an instance of the client with the options object

```
var client = new AkeneoClient(options);
```

Using the API

Asynchronious calls

All calls to the API is made from the `AkeneoClient`. The client implements asynchronious methods, that should be called using the `async`/`await` pattern. Each method has an Cancellation Token as an optional argument. It can be provided to cancel an ongoing operation.

Generic arguments

The API is uniform over most of it resources. This allows the client to use the same methods to query different resources. The desired resource type is provided as a generic argument to the calls. Below are some example calls using the same method but querying different resources.

```
var argument = await Client.GetAsync<NumberAttribute>("shoe_sie");
var family = await Client.GetAsync<Family>("sports_shoe");
var category = await Client.GetAsync<Category>("women");
var product = await Client.GetAsync<NumberAttribute>("nike_air");
```

Important! Despite the fact that the API is uniform, it is not complete. As of the most current version (1.7.4) the only resource that can be deleted is `Product`. Trying to delete other resources will end up with a `405 Method Not Allowed`.

Error handling

The client passes on any error messages from the API. Many of the methods returns either an `AkeneoResponse`, `AkeneoBatchResponse` or `PaginationResult`, each having a message property as well as property for the HTTP status code.

```
var response = await Client.DeleteAsync<Product>("nike_air");
if (response.Code == HttpStatusCode.NoContent)
{
    Console.WriteLine("Successfully deleted product.");
}
else
{
    Console.WriteLine($"Unsuccessfully deleted product. Message: {response.Message}");
}
```

CHAPTER 2

Attributes

This section contains information on how to work with the different types of attributes available.

Derived attribute types

There are multiple types of attributes in Akeneo PIM, each with a slightly different set of properties. in the Akeneo.NET client, each attribute type is represented by a specialized class, deriving from the abstract class `AttribteBase`. This simplifies things, as all valid properties and their corresponding types are present for each attribute type.

Attribute types

A complete list of valid attributes can be found at Akeneo's documentation. All the attribute classes are located in the namespace `Akeneo.Model.Attributes`. In the same namespace is also `AttributeType` contains a list that hold the list of attribute types.

Example

For example, the `NumberAttribute` has `NumberMin`, `NumberMax`, `DecimalsAllowed` and `NegativeAllowed` properties that is specific to it.

```
var shoeSize = new NumberAttribute
{
    Code = "shoe_size",
    DecimalsAllowed = true,
    NegativeAllowed = false,
    NumberMin = 6,
    NumberMax = 15,
    AvailableLocales = new List<string> { Locales.EnglishUs }
}
```

Attribute features

Metric Attribute

The metric attribute expects a *Metric Family* and *Metric Unit* to be supplied. In order to make this easier, classes for `MetricFamily` and corresponding units can be used

```
var weigth = new MetricAttribute
{
    Code = "weigth",
    MetricFamily = MetricFamilies.Weight,
    DefaultMetricUnit = WeightUnits.Gram
};
```

For each metric family, there is a family specific units class. In the example above, the `Weight` family has a `WeightUnits` units class. In the same way, the `Temperature` family has a `TemperatureUnits` class etc.

Retrieving attributes

Get one attribute

Attributes are queried for the API by it's unique identifier. In order to retrieve a `NumberAttribute` with the identifier `shoe_size`, the following request can be made

```
var shoeSize = await Client.GetAsync<NumberAttribute>("shoe_size");
```

The generic argument tells the client what kind of attribute it is, and the client uses that class when trying to deserialize it. If the type of the attribute is unknown, the `GenericAttribute` can be used. It holds all know attribute properties, and thus will serialize all properties from the API.

Get multiple attributes

Often, the PIM has multiple attributes of different typs. In those scenarios, it is not feasable to query after multiple attributes using only one target type. By using the attribute base class `AttributeBase` the client will look at the retrieved attribute's `type` property and serialize it to corresponding type.

```
var paginatedResult = await Client.GetManyAsync<AttributeBase>();
var attributes = paginatedResult.Embedded.Items;
var numberAttributes = attributes.OfType<NumberAttribute>();
var shoeSize = numberAttributes.FirstOrDefault(a => a.Code == "shoe_size");
```

CHAPTER 3

Products

The entity `Product` has some properties that references other resources. This documentation does not go into detail on the constraints of these properties. For more information, see the API documentation.Below are some client specific information that can be useful when creating a product.

Creating product values

Product values are created in a dictionary keyed with attribute code (`string`).

```
var values = new Dictionary<string, object>
{
    {"shoe_size", new List<ProductValue>{ new ProductValue
    {
        Locale = Locales.EnglishUs,
        Scope = AkeneoDefaults.Channel,
        Data = 9
    }}}
};
```

Different attribute expects different data. For example, `Scope` and `Locale` should not be provided if the attribute does not have unique values for locales or channels. Depending on the attribute type, the `Data` payload is expected to be of certain types.

Product values from attribute

It can be hairy to provide the correct product values. In order to make it easier, Akeneo.NET has a set of extension methods that can be used to create a product value based on an attribute. The extension methods are specific for each type, which can be helpful if unsure what kind of data to expect.

Below is an example where the product values are created from a set of attributes.

```
// load known attributes
var price = await Client.GetAsync<PriceAttribute>("list_price");
var campaignPrice = await Client.GetAsync<PriceAttribute>("campaign_price");
var startOfSales = await Client.GetAsync<DateAttribute>("campaign_start_date");

// use extension method to create product values
var product = new Product
{
    Identifier = "nike_air",
    Family = "sports_shoe",
    Enabled = true,
    Categories = {"shoe", "sport", "women"},
    Values = DictionaryFactory.Create(
        price.CreateValue(99, Currency.USD),
        campaignPrice.CreateValue(79, Currency.USD),
        startOfSales.CreateValue(DateTime.Today.AddDays(7))
    )
};
```

The `DictionaryFactory` is a convenience class that creates a dictionary from the `KeyValuePairs` returned from the `CreateValue` calls.

CHAPTER 4

Filter and Search

Akeneo PIM provides endpoints to filter and search through products and locales. In the official documentation, it is refered to simply as Filter, but the Akeneo.NET client differentiates between the search query (with its own syntax) and the general filtering.

Search Criterias

A search is performed by supplying one or more `Criteria`. There are different types of criterias, and they are all represented by a derived class. For example, search criterias for a category is created by the `Category` criteria (found in `Akeneo.Search` namespace). Each criteria has an operator (`string`) and a value (`object`). The class `Operators` contains a list of all known operators and can be used to supply values there.

```
var categoryCriteria = new Category
{
    Operator = Operators.NotIn,
    Value = new[] { "summer_sale" }
};
```

Depending on what kind of criteria that is used, different operations are valid. Each criteria has static members that represent valid operations, and expected value types.

```
var response = await Client.SearchAsync<Product>(new List<Criteria>
{
    Category.NotIn("summer_sale"),
    Family.In("sports_shoe"),
    Status.Disabled(),
    Created.Between(DateTime.Today.AddDays(-7), DateTime.Today)
});
```

The response is a `PaginationResult`.

Multiple criterias of same type

It is possible to use the same type of criteria multiple multiple times to search for products. For example: to find shoes that are not on summer sale, the following search criterias can be used

```
var response = await Client.SearchAsync<Product>(new List<Criteria>
{
    Category.NotIn("summer_sale"),
    Category.In("shoes")
});
```

This results in the following search query

```
?search={"categories":[{"operator":"NOT IN","value":["summer_sale"]},{"operator":"IN",
↪"value":["shoes"]}]}
```

Filter products

The client also provides a no-magic method for filtering products. `FilterAsync` is invoked with a query string argument (`string`) that is appended on the resource endpoint (derived from the provided generic type). To filter products via `description` attribute (similar to this example) simply perform the call.

```
var response = await Client.FilterAsync<Product>("?attributes=description");
```

In fact, `SearchAsync` internally calls this end point to perform the searchd with a query parameter created from the provided criterias.

CHAPTER 5

Updating Resources

Updating existing resources is a complex topic with a whole section in the API docs describing update behaviour based on the entity types etc. It is recommended to read up on how this works before using the endpoint.

Add or change values

One straight forward strategy to update an resource is to retrieve it from the API, modify it and the update it in the API. This is how it can be done:

```
var product = await Client.GetAsync<Product>("nike_air");
product.Enabled = false;
await Client.UpdateAsync(product);
```

This approach has two drawbacks:

1. It requires one extra round-trip to the API to retrieve the product before updating it
2. It can not be used to remove certain property values (like `string` properties on the product).

Delete values

In order to be able to remove the value of a `string` property, it needs to be passed to the API with a `null` value.

```
var response = await Client.UpdateAsync<Product>("nike_air", new UpdateModel
{
    {"variant_group", null}
});
```

This approach requires no extra round-trip, but gives the caller greater greater responsibility.

Technical information

It is how the API interprets `null` that adds complexity to the update query. A string property value is removed by passing in `null` as a value for that property. However, `null` is the default value for reference types and it is difficult to differ an “explicit null” from a “implicit null” without introducing some sort of complexity to the solution.

CHAPTER 6

Media Files

Media files are files associated with a specific attribute on a specific product. They can be uploaded and downloaded through Akeneo.NET

Upload file

In order to upload a file, use the `FileUpload` class. In order for a successful upload, all properties must be set:

- Product Identifier: the identifier of the product
- Product Attribute: the attribute code for which the file should be associated with. The attribute should be a `FileAttribute` attribute
- File Path: The path to the image.

Below is an example of of a correctly created `FileUpload`

```
var fileUpload = new MediaUpload
{
    Product =
    {
        Identifier = "nike_air",
        Attribute = "product_image_large"
    },
    FilePath = "C:\\images\\nike_air.png"
};
```

Upload the image by calling `UploadAsync`

```
var response = await Client.UploadAsync(fileUpload);
```

The call returns an `AkeneoResponse` that hold information about the outcome of the operation.

Download file

A file is downloaded through by supplying it's `mediaCode`. If the media file is recently uploaded, the media code can be retrieved from the links collection in the `AkeneoResponse`.

```
var response = await Client.UploadAsync(fileUpload);
var mediaCode = response.Links[PaginationLinks.Location].Href;
```

Otherwise, it can be retrieved from the product values.

```
var product = await Client.GetAsync<Product>("tyfon-tv-6m-0m-company");
var logoValue = product.Values["product_image_large"];
var mediaCode = logoValue.FirstOrDefault()?.Data as string;
```

Once the code is retrieved, it can be used as argument when calling `DownloadAsync`. The resource contains the original file name, as well as a `Stream` containing the image. There are extension methods for writing the file to disk.

```
var file = await Client.DownloadAsync(mediaCode);
file.WriteToFile("C:\\downloads", file.FileName);
file.Dispose();
```

If a file name is not provided, the original file name will be used.