



Custobar integration for Magento 2

Developer Guide

Basics

How is data exported to Custobar?

In order for data to get exported to Custobar, there must have following configurations:

1. Mapping data configuration: [Configuring mapping data](#)
2. Website and sync validation configuration: [Configuring logic restrictions](#)
3. Data resolver configuration: [Data resolving for export](#)

These configurations are done per entity type, for example customers and products.

When these are done, it's possible to schedule that entity for export: [Scheduling export](#). Scheduling happens as the data is updated in Magento. The linked documentation also mentions how to create schedules based on entities that are only related to the configured entities (for example customer address customer).

Once entity is scheduled for export, crons will handle the sending of data to Custobar API: [Exporting data](#)

By default entities are exported individually but it's also possible to execute mass export: [Mass exporting data](#)

One API call however can contain data of multiple entities but the calls are always entity type specific.

Configuring logic restrictions

Are there any checks as to whether or not module logic executes?

Yes, some are implemented using Api/ExecutionValidatorInterface. There are also website restrictions done via Api/WebsiteValidatorInterface and data sync restrictions done via Api/SyncValidatorInterface.

Can I modify the validator logic?

As Api/ExecutionValidatorInterface is always used as interface, the default implementation can be replaced via etc/di.xml, as long as the new implementation implements the same interface.

The default implementation can be also customised as follows:

```
<type name="Custobar\CustoConnector\Model\Validation\ExecutionValidatorChain">
  <arguments>
    <argument name="validators" xsi:type="array">
      <item name="check_if_config_allows" xsi:type="object"
>Custobar\CustoConnector\Model\Validation\ExecutionValidator\CheckIfConfigAllo
ws</item>
    </argument>
  </arguments>
</type>
```

Each item under "validators" must implement Api/ExecutionValidatorInterface. You can add either new items or replace existing components by using the same item name. If you wish to disable existing validation then just turn the item's xsi:type to null.

Website restrictions

What the default implementation of Api/WebsiteValidatorInterface does is that it resolves the given entity's website ids via Model/Validation/WebsiteValidator/EntityWebsiteResolverInterface and then compares them to the configurations in System / Configuration / Custobar / CustoConnector. If there's any match at all between the entity's website ids and the websites in admin configuration, then entity is considered the belong in allowed websites.



Where does the website restriction apply for Custobar?

In general if a website is not allowed, no Custobar logic is intended to be executed in that website's scope. This includes statistics collecting and scheduling any data syncs.

How do I modify the website validation?

If you have configured a new mapping data ([Configuring mapping data](#)) or you want to just update the website validation, you will want to look into this configuration in module's etc/di.cml:

```
<type name="Custobar\CustoConnector\Model\Validation\WebsiteValidator">
  <arguments>
    <argument name="entityResolvers" xsi:type="array">
      <item name="Magento\Catalog\Model\Product" xsi:type="object"
>Custobar\CustoConnector\Model\Validation\WebsiteValidator\EntityWebsiteResolv
er\ByWebsiteIds</item>
      <item name="Magento\Customer\Model\Customer" xsi:type="object"
>Custobar\CustoConnector\Model\Validation\WebsiteValidator\EntityWebsiteResolv
er\ByWebsiteId</item>
      <item name="Magento\Newsletter\Model\Subscriber" xsi:type="object"
>Custobar\CustoConnector\Model\Validation\WebsiteValidator\EntityWebsiteResolv
er\ByStoreId</item>
      . . . .
    </argument>
  </arguments>
</type>
```

What's here is configuration of website resolvers per entity type. So if the product is validated, the website ids for the product are fetched via `Custobar\CustoConnector\Model\Validation\WebsiteValidator\EntityWebsiteResolver\ByWebsiteIds`. You can replace this to anything else, as long as the object implements interface `Model/EntityIdentifierResolverInterface`. Same thing with adding a resolver for a new entity.

Schedule restrictions

In order to check if an entity can be scheduled, `Api/SchedulingValidatorInterface` is used. This can be very useful if you wish to apply entity specific restrictions instead of restricting all logic.

The default implementation for `Api/SchedulingValidatorInterface` is almost exactly the same as `Api/ExecutionValidatorInterface`, the only difference is later doesn't take any params. This interface can be replaced via `etc/di.xml` or it can be extended as follows:



```
<type
name="Custobar\CustoConnector\Model\Validation\SchedulingValidatorChain">
  <arguments>
    <argument name="validators" xsi:type="array">
      <item name="check_has_mapping_data" xsi:type="object"
>Custobar\CustoConnector\Model\Validation\SchedulingValidator\CheckHasMappingD
ata</item>
      <item name="check_is_allowed_website" xsi:type="object"
>Custobar\CustoConnector\Model\Validation\SchedulingValidator\CheckIsAllowedWe
bsite</item>
    </argument>
  </arguments>
</type>
```

Each item under "validators" must implement Api/SchedulingValidatorInterface. You can add either new items or replace existing components by using the same item name. If you wish to disable existing validation then just turn the item's xsi:type to null.

Configuring mapping data

What is mapping data?

Mapping data is a set of entity type specific data, that contains following information:

- What entity in Magento the data relates to
- What entity in Custobar the data relates to
- How are Magento and Custobar fields for the entity mapped

If entity type has mapping data configured, then the entity type can be considered "mapped" or "tracked" and it will be included in the module's logic related to Custobar, like data export.

What is done with mapping data?

Mapping data defines what entities can be exported to Custobar and what kind of data goes out from Magento to Custobar. See [Data resolving for export](#) on more information regarding how this data is used.

By default following entities are configured with mapping data:

- Catalog products
- Customers
- Newsletter subscribers
- Orders
- Stores

How do I fetch the mapping data?

For this you want to use `Api/MappingDataProviderInterface`. This allows you to either fetch specific mapping data or all of it. For fetching specific mapping data it's suggested you use `Model/EntityIdentifierResolverInterface` to resolve the identifier for that entity.

How can I modify mapping data?

This can for most of the part be done via module's `etc/di.xml` file, but some parts can be modified from admin as well.

Existing data

Some of the existing mapping data is established as follows.

```

<virtualType name="Custobar\CustoConnector\Model\MappingData\CatalogProduct"
type="
Custobar\CustoConnector\Model\MappingData">
  <arguments>
    <argument name="data" xsi:type="array">
      <item name="identifier"
xsi:type="string">Magento\Catalog\Model\Product</item>
      <item name="target_field" xsi:type="string">products</item>
      <item name="field_map_config"
xsi:type="const">Custobar\CustoConnector\Model\Config::
CONFIG_MAPPING_PRODUCT</item>
    </argument>
  </arguments>
</virtualType>

<virtualType name="Custobar\CustoConnector\Model\MappingData\Store" type="
Custobar\CustoConnector\Model\MappingData">
  <arguments>
    <argument name="data" xsi:type="array">
      <item name="identifier"
xsi:type="string">Magento\Store\Model\Store</item>
      <item name="target_field" xsi:type="string">shops</item>
      <item name="field_map" xsi:type="array">
        <item name="id" xsi:type="string">external_id</item>
        <item name="custobar_name" xsi:type="string">name</item>
      </item>
    </argument>
  </arguments>
</virtualType>

<type name="Custobar\CustoConnector\Model\MappingDataProvider">
  <arguments>
    <argument name="mappingDataModels" xsi:type="array">
      <item name="Magento\Catalog\Model\Product" xsi:type="object"
>Custobar\CustoConnector\Model\MappingData\CatalogProduct</item>
      <item name="Magento\Store\Model\Store" xsi:type="object"
>Custobar\CustoConnector\Model\MappingData\Store</item>
    </argument>
  </arguments>
</type>

```

So what happens here is that we create virtual types for store and product mapping data based on the type Model/MappingData. These are then added to the Model/MappingDataProvider, keyed by unique identifier. Now when the data provider is called with the intention to retrieve product mapping data, you will receive the Custobar\CustoConnector\Model\MappingData\CatalogProduct object.



You can replace each mapping data for each entity entirely with your own mapping data by creating your own virtual type. Or even own entity, as long as it implements `Api/Data/MappingDataInterface.php`.

You can also modify just parts mapping data like this from another module, as long as the virtual types name and type match with the existing virtual type:

```
<virtualType name="Custobar\CustoConnector\Model\MappingData\Store" type="
Custobar\CustoConnector\Model\MappingData">
  <arguments>
    <argument name="data" xsi:type="array">
      <item name="target_field" xsi:type="string">new_target_field</item>
    </argument>
  </arguments>
</virtualType>
```

Removing or disabling mapping data

If you wish to disable or remove existing mapping data, then you can do this:

```
<type name="Custobar\CustoConnector\Model\MappingDataProvider">
  <arguments>
    <argument name="mappingDataModels" xsi:type="array">
      <item name="Magento\Catalog\Model\Product" xsi:type="null" />
    </argument>
  </arguments>
</type>
```

In general if no mapping data can be resolved for an entity, it won't be included in any of the Custobar module logic.

Adding new one

```
<virtualType name="Vendor\CustobarExtensionModule\Model\MappingData\Entity"
type="
Custobar\CustoConnector\Model\MappingData">
  <arguments>
    <argument name="data" xsi:type="array">
      <item name="identifier"
xsi:type="string">Vendor\Module\Model\Entity</item>
      <item name="target_field" xsi:type="string">field_in_custobar</item>
      <item name="field_map" xsi:type="array">
```




```
<item name="field_in_magento"
xsi:type="string">field_in_custobar</item>
</item>
</argument>
</arguments>
</virtualType>

<type name="Custobar\CustoConnector\Model\MappingDataProvider">
<arguments>
<argument name="mappingDataModels" xsi:type="array">
<item name="Vendor\Module\Model\Entity" xsi:type="object"
>Vendor\CustobarExtensionModule\Model\MappingData\Entity</item>
</argument>
</arguments>
</type>
```

If you add a new one, make sure to also go through other documentations to add sufficient configurations for your entity. Adding mapping data alone is not enough to get data exported to Custobar.

Field mapping

This is where the admin configurations come in, if so desired. In admin, you can see for example following in Stores / Configuration / Custobar / CustoConnector / Field Mapping:

Product Field Map [global]

```
name:title
sku:external_id
custobar_minimal_price:minimal_price
custobar_price:price
type_id:mage_type
configurable_min_price:mv_configurable
```

The mapping consists of rows, where left value is a field in Magento and the right value is the field in Custobar. These must be separated via colon. How this is connected to the mapping data is by following:

```
<virtualType name="Custobar\CustoConnector\Model\MappingData\CatalogProduct"
type="
Custobar\CustoConnector\Model\MappingData">
<arguments>
<argument name="data" xsi:type="array">
```



```
.....
    <item name="field_map_config"
xsi:type="const">Custobar\CustoConnector\Model\Config::
CONFIG_MAPPING_PRODUCT</item> <!-- This one! -->
    </argument>
</arguments>
</virtualType>
```

So if you give value to field_map_config property, that configuration will be used to determine the field mapping for mapping data. If no value is set then the mapping will be empty.

As you might've already seen in above examples, it's also possible to define the mapping via di.xml:

```
<virtualType name="Custobar\CustoConnector\Model\MappingData\CatalogProduct"
type="Custobar\CustoConnector\Model\MappingData">
    <arguments>
        <argument name="data" xsi:type="array">
            <item name="field_map" xsi:type="array">
                <item name="field_in_magento"
xsi:type="string">field_in_custobar</item>
                </item>
            </argument>
        </arguments>
    </virtualType>
```

This works with a similar principle, add properties under field_map with Magento field as name and Custobar field as value.

If you use both field_map and field_map_config, then field_map is used a base, which the admin configuration then fills up further.

I want to add custom logic to resolving the field mapping, how does this happen?

If you need to add field mappings conditionally or have complex logic regarding the resolving of fields, you can utilise the same structure used to resolve the field mapping from admin config.

So what happens in the Model/MappingDataProvider is that it takes the mapping data configurations from di.xml and then calls to Model/MappingDataProvider/DataExtenderInterface on each. Depending on the implementation for the interface, the mapping data can be modified. The admin configuration is applied using this interface.



By default the Model/MappingDataProvider/DataExtenderInterface has preference to Model/MappingDataProvider/DataExtenderChain, which is meant to allow you to execute multiple different instances of Model/MappingDataProvider/DataExtenderInterface on the mapping data. You can of course replace this via di.xml to only execute one instance of Model/MappingDataProvider/DataExtenderInterface.

The admin configuration has been added as part of the chain like this:

```
<type
name="Custobar\CustoConnector\Model\MappingDataProvider\DataExtenderChain">
  <arguments>
    <argument name="dataExtenders" xsi:type="array">
      <item name="adjust_field_mapping_from_config" xsi:type="object"
>Custobar\CustoConnector\Model\MappingDataProvider\DataExtender\AdjustFieldMap
pingFromConfig</item>
    </argument>
  </arguments>
</type>
```

So if you want to modify the mapping data further, you must create a class that implements Model/MappingDataProvider/DataExtenderInterface and then add it to the chain as follows:

```
<type
name="Custobar\CustoConnector\Model\MappingDataProvider\DataExtenderChain">
  <arguments>
    <argument name="dataExtenders" xsi:type="array">
      <item name="my_custom_extender" xsi:type="object"
>Vendor\CustobarExtensionModule\Model\Path\To\Extender</item>
    </argument>
  </arguments>
</type>
```

If you want to disable any of the existing components in the chain, just turn the xsi:type to null.

Data resolving for export

What the data exported to Custobar is exactly?

The data that is sent in the export is based on the Magento entity resolved from schedule. Once the Magento entity is resolved like this, mapping data ([Configuring mapping data](#)) is used to determine the exact fields that will be sent to Custobar.

If the Magento entity already contains the data mentioned in the field mapping, then that data should be automatically sent during the export. However if no data can be found that matches the mapping, then that data cannot be exported (or it will go as empty).

How do schedules and Magento entities relate to each other?

Each row in custobar_schedule table is meant to match to some entity in default Magento. These can also match custom entities but it's not certain how common this is.

How these rows in custobar_schedule are connected to each entity is via following information:

Column in custobar_schedule	Description
entity_id	Integer that should match the corresponding identifier value of entity, for example catalog_product_entity.entity_id
entity_type	Determines which type of entity this is, currently the full class name is used, for example Magento\Catalog\Model\Product
store_id	Either filters the entity with store specific entity or configures which store some attributes are taken from

How is the Magento entity resolved from schedule?

The interface meant to do the actual conversion is Api/EntityDataResolverInterface. This interface has various methods where above information is provided and the return value is always either an array or instance of the entity.

The default implementation of this interface allows extensions via di.xml:

```
<type name="Custobar\CustoConnector\Model\EntityDataResolver">
  <arguments>
```

```
<argument name="components" xsi:type="array">
<item name="Vendor\Module\Model\Entity"
xsi:type="object">Path\To\Resolver</item>
</argument>
</arguments>
</type>
```

The only thing necessary to remember with your object is that must implement interface `Model/EntityDataResolver/ResolverComponentInterface`. If you use existing item name (like `Magento\Catalog\Model\Product`) you can replace existing resolver via separate model. This way you can also turn the `xsi:type` to "null" in order to disable existing resolvers.

By default this kind of configurations exist for:

- Catalog products
- Customers
- Orders
- Newsletter subscriptions

See the module's `etc/di.xml` for further examples.

How can I modify the product data?

The product resolving differs a bit from other types since it uses interface `Model/ResourceModel/Product/ProductProviderInterface`. This can be either replaced with your own implementation via `di.xml` or you can look into extending the default implementation's logic via `di.xml`:

```
<type
name="Custobar\CustoConnector\Model\ResourceModel\Product\ProductProvider">
<arguments>
<argument name="preProcessors" xsi:type="array">
<item name="xxxxx" xsi:type="object">xxxxx</item>
</argument>
<argument name="postProcessors" xsi:type="array">
<item name="xxxxx" xsi:type="object">xxxxx</item>
</argument>
</arguments>
</type>
```

By default there are some components added like above, see the module `etc/di.xml` for examples.



The difference between "preProcessors" and "postProcessors" is that between the execution of these happens the collection load. So if you want to include something in the collection query, modify "preProcessors". If you want to do something with the loaded collection, do it in "postProcessors".

If you use existing item name you can replace existing processors with your own. This way you can also turn the xsi:type to "null" in order to disable existing processors.

If you want to add a new/update existing processor, regardless of it going under "preProcessors" or "postProcessors", it must implement Model/ResourceModel/Product/ProductProvider/CollectionProcessorInterface. The processor must also return the method param as response.

How can I resolve the data for export?

You can use Api/MappedDataBuilderInterface. Before using this, you need to have the entity object available, as described above.

Keep in mind that Api/MappedDataBuilderInterface will construct the data for export based on the entity's current data, so this is not a reliable way to check what has been sent in the past.

How is the exported data constructed exactly?

The default implementation of Api/MappedDataBuilderInterface uses interface Model/MappedDataBuilder/DataExtenderInterface in order to apply custom fields on the entity. If there's no such interface configured for the entity, then the entity moves forward as is.

After that the entity is then ran through Magento\Framework\DataObject\Mapper with the field mapping from mapping data, which produces Magento\Framework\DataObject then contains only the mapped values.

How can I customise the data construction?

If you need to add fields outside of the mapping configurations, you might want to consider adding plugin to Api/MappedDataBuilderInterface. However, this is not ideal as then you will need to do checks on the processed entity type.

On the other hand if you want to modify how the mapped values are constructed or add support for new mapping fields, Model/MappedDataBuilder/DataExtenderInterface is your answer. Here's how the default implementation of Api/MappedDataBuilderInterface uses the interface for example with store entity:



```
<virtualType
name="Custobar\CustoConnector\Model\MappedDataBuilder\DataExtender\Store"
type="
Custobar\CustoConnector\Model\MappedDataBuilder\DataExtenderChain">
  <arguments>
    <argument name="dataExtenders" xsi:type="array">
      <item name="add_basic_data" xsi:type="object"
>Custobar\CustoConnector\Model\MappedDataBuilder\DataExtender\Store\AddBasicDa
ta</item>
    </argument>
  </arguments>
</virtualType>

<type
name="Custobar\CustoConnector\Model\MappedDataBuilder\DataExtenderProvider">
  <arguments>
    <argument name="dataExtenders" xsi:type="array">
      <item name="Magento\Store\Model\Store" xsi:type="object"
>Custobar\CustoConnector\Model\MappedDataBuilder\DataExtender\Store</item>
    </argument>
  </arguments>
</type>
```

So when store entity is given to Model/MappedDataBuilder:

1. Model/MappedDataBuilder/DataExtenderProviderInterface is used to get the virtual type Model/MappedDataBuilder/DataExtender/Store
2. Model/MappedDataBuilder/DataExtender/Store is a virtual type of Model/MappedDataBuilder/DataExtenderChain which takes the store entity and loops it through an array of Model/MappedDataBuilder/DataExtenderInterface instances, for example Model/MappedDataBuilder/DataExtender/Store/AddBasicData
3. Model/MappedDataBuilder/DataExtender/Store then returns the modified store entity back

You can add custom logic to existing configurations by creating your own object that implements Model/MappedDataBuilder/DataExtenderInterface and then adding it to existing DataExtenderChain virtual type as such:

```
<virtualType
name="Custobar\CustoConnector\Model\MappedDataBuilder\DataExtender\Store"
type="
Custobar\CustoConnector\Model\MappedDataBuilder\DataExtenderChain">
```



```
<arguments>
  <argument name="dataExtenders" xsi:type="array">
    <item name="custom_logic" xsi:type="object"
>Vendor\CustobarExtensionModule\Model\MappedDataBuilder\DataExtender\Entity\Cu
stomLogic</item>
  </argument>
</arguments>
</virtualType>
```

If you use same item name, then you can replace existing components or even disable them if you change the `xsi:type` to null.

If you want to configure this same structure for new entity, then you can more or less look at existing configurations and do something similar:

```
<virtualType
name="Vendor\CustobarExtensionModule\Model\MappedDataBuilder\DataExtender\MyEn
tity" type="
Custobar\CustoConnector\Model\MappedDataBuilder\DataExtenderChain">
  <arguments>
    <argument name="dataExtenders" xsi:type="array">
      <item name="add_basic_data" xsi:type="object"
>Vendor\CustobarExtensionModule\Model\MappedDataBuilder\DataExtender\MyEntity\
AddBasicData</item>
    </argument>
  </arguments>
</virtualType>

<type
name="Custobar\CustoConnector\Model\MappedDataBuilder\DataExtenderProvider">
  <arguments>
    <argument name="dataExtenders" xsi:type="array">
      <item name="Vendor\CustobarExtensionModule\Model\MyEntity"
xsi:type="object"
>Vendor\CustobarExtensionModule\Model\MappedDataBuilder\DataExtender\MyEntity<
/item>
    </argument>
  </arguments>
</type>
```


Exporting Data

Executing the export

Data export to Custobar can be executed by calling `Api/ExportInterface`. However you must have an array of schedule objects to give as parameters. It doesn't matter if the array has schedule objects from different entity types, these will be grouped during the execution by the default implementation. As response you will receive `Api/Data/ExportDataInterface` per entity type, which contains info on what schedules failed and succeeded, what kind of request data was sent and what kind of response was received.

How the default implementation for this interface works is as such:

1. Construct initial `Api/Data/ExportDataInterface` objects per entity based on the given schedules via `Model/Export/ExportData/InitializerInterface`:
 - a. If no mapping data ([Configuring mapping data](#)) can be found for the entity type, the schedules of that entity type will be skipped and set for removal
 - b. If no data can be resolved for export ([Data resolving for export](#)), that schedule will be skipped and marked as failed
 - c. Json data for the API call is constructed based on the resolved data for export
2. `Model/Export/ExportData/ProcessorInterface` is called on each valid `Api/Data/ExportDataInterface` object
 - a. `Model/Export/ExportData/Processor/ExecuteExport` is executed to do the actual API call and some error handling, the `Api/Data/ExportDataInterface` will be updated with response data and failed/successful schedule ids
 - b. `Model/Export/ExportData/Processor/AdjustSchedules` is executed to update the schedule data based on the API call results

The API calls are always entity type specific and one API call can contain multiple entities of that entity type.

Configuring export data initializer

The default implementation for `Model/Export/ExportData/InitializerInterface` groups the given schedules by entity type and then creates `Api/Data/ExportDataInterface` object for each.

Constructing that data can be modified via `etc/di.xml`:

```
<type name="Custobar\CustoConnector\Model\Export\ExportData\Initializer">
  <arguments>
    <argument name="components" xsi:type="array">
      <item name="add_mapping_data" xsi:type="object"
>Custobar\CustoConnector\Model\Export\ExportData\.....</item>
```



```
<item name="add_mapped_data_rows" xsi:type="object"
>Custobar\CustoConnector\Model\Export\ExportData\.....</item>
  <item name="add_request_data_json" xsi:type="object"
>Custobar\CustoConnector\Model\Export\ExportData\.....</item>
</argument>
</arguments>
</type>
```

So to add your own handling for the export data, you can add a new item under the "components". Just make sure the object implements interface Model/Export/ExportData/Initializer/InitializerComponentInterface. You can also replace existing component by using the same name. If you need to remove existing components, turn the xsi:type to null.

Keep in mind that with the default components the order is important:

- add_mapping_data is intended mainly for validating that the mapping data exists, so it should be first to prevent needlessly executing others
- add_mapped_data_rows defines data needed by add_request_data_json, so they must be in that order

Configuring export data processing

The default implementation for Model/Export/ExportData/ProcessorInterface loops through other objects of the same interface configured to it via etc/di.xml:

```
<type name="Custobar\CustoConnector\Model\Export\ExportData\ProcessorChain">
  <arguments>
    <argument name="processors" xsi:type="array">
      <item name="execute_export" xsi:type="object"
>Custobar\CustoConnector\Model\Export\ExportData\Processor\ExecuteExport</item
>
      <item name="adjust_schedules" xsi:type="object"
>Custobar\CustoConnector\Model\Export\ExportData\Processor\AdjustSchedules</it
em>
    </argument>
  </arguments>
</type>
```

If you want to add your own handling for the export data, you can add a new item under the "processors". Just make sure the object implements interface Model/Export/ExportData/ProcessorInterface. You can also replace existing processors by using the same name. If you need to remove existing processors, turn the xsi:type to null.



Keep in mind that with the default components the order is important: `execute_export` triggers the actual API call and `adjust_schedules` uses that data to update the schedules.

Calling the export API

The export itself is executed in `Model/Export/ExportData/Processor/ExecuteExport`. What this does is:

1. Use `Model/CustobarApi/ClientUrlProviderInterface` to resolve the target url for export
2. Use `Model/CustobarApi/ClientBuilderInterface` to create client for the API call
3. Give the client the export data in json format
4. Call the API
5. If failure, log all relevant information for debugging and set failed schedule ids on `Api/Data/ExportDataInterface`
6. If success, log the information for debugging and set successful schedule ids on `Api/Data/ExportDataInterface`

Configuring the API call

Currently if you need to modify the API call logic, you will need to replace the `Model/Export/ExportData/Processor/ExecuteExport` processor set on `Model/Export/ExportData/ProcessorChain` via `etc/di.xml`. You can see how to do this in earlier sections of this page.

See following sections on how to modify the client and url generations.

Export API url

When the client for the API call is constructed, the target url is always `https://<domain>.custobar.com/api/<target>/upload`. Interface `Model/CustobarApi/ClientUrlProviderInterface` is used to resolve this so if the url needs to be changed entirely, you can replace the interface with your own implementation.

In the default implementation:

- Domain is the same as the "Company Domain" in the admin configuration. If dev mode is turned on, this turns into "dev" regardless of what's set in that field.
- Target is the same as the `target_field` configured in mapping data ([Configuring mapping data](#)), see below

```
<virtualType name="Custobar\CustoConnector\Model\MappingData\CatalogProduct" type="
```

```
Custobar\CustoConnector\Model\MappingData">
  <arguments>
    <argument name="data" xsi:type="array">
      .....
      <item name="target_field" xsi:type="string">products</item>
      .....
    </argument>
  </arguments>
</virtualType>
```

Authentication

When the client for the API call is constructed, token authentication will be used as authentication method. The token will be taken from "API Token" admin configuration. This token must be created in Custobar's settings.

If the authentication needs to be modified, you can create your own implementation based on Model/CustobarApi/ClientBuilderInterface.

Where is export called from currently?

At the moment only via cron job "custoconnector_callcustobarapi".

What the cron does is:

- Check if the export can be called via Api/ExecutionValidatorInterface, stop if doesn't pass
- Check if mass export is happening, stop if it is
- Call Model/Schedule/ExportableProviderInterface to retrieve schedules possible to export
- Call Api/ExportInterface with the retrieved schedules

What happens when export fails?

The schedules will be retried up to 7200 times (not sure where the number comes from though). After this they will not be retried again and cron job "custoconnector_schedule_clean_up" will clean them up.

Usually if the schedule is invalid in the first place, like the entity type is not mapped anymore ([Configuring mapping data](#)), then those schedules will be immediately set for removal.

Logging

General

The module does two kinds of logging:

- Logs are stored in files
- Logs are stored in database

File logging

File logs are always located in `<project_root>/var/log/` and the log file names are prefixed with "custoconnector-". For example system level logs go to file `custoconnector-system.log`.

You cannot see file logs in admin.

By default file logging is not active, as in Docker environments there seem to be some permissions issues with generating the logs files. Ideally there would be admin configuration where you can toggle which logging methods are used, but for now this has simply been left out of the `di.xml` configuration of the logger interface. If wanted to be enabled, add following configuration in `di.xml`:

```
<type name="Custobar\CustoConnector\Model\Logger">
  <arguments>
    <argument name="components" xsi:type="array">
      <item name="log_file" xsi:type="object"
>Custobar\CustoConnector\Model\Logger\LoggerComponent\LogFile</item>
    </argument>
  </arguments>
</type>
```

Database logging

These logs are stored in table "custoconnector_log". Alongside timestamp, log type and message, there's also field for "context data" that can contain more details about the log.

You can view the logs added to database from System / Custobar / Custobar Logs.



When anything's logged, what's the difference between the two?

Currently there's no difference between the two, as the module uses a structure that allows logging things to both file and database at once.

How do I log anything to Custobar logs?

If you want to log something Custobar specific, always use `Api/LoggerInterface`.

If you give the `contextData` param any values, make sure that these are in array format. The `contextData` will be saved to database as json and the admin view will attempt to prettify the json, which won't work as well if the `contextData` contains json strings in it.

How can I configure the logging?

If you want to disable either the file or database logging, for now you can create configuration like this in `di.xml`:

```
<type name="Custobar\CustoConnector\Model\Logger">
  <arguments>
    <argument name="components" xsi:type="array">
      <item name="log_file" xsi:type="object"
>Custobar\CustoConnector\Model\Logger\LoggerComponent\LogFile</item>
      <item name="log_data" xsi:type="null" /> <!-- So change the type to null
-->
    </argument>
  </arguments>
</type>
```

Potential future improvements

- Log cleanup for old entries can save disk space on server
- Configuration for allowing admins to choose which logs they want to use

Mass exporting data

Triggering mass export

Currently nothing triggers mass export automatically, this needs to be done from admin "System / Custobar / Custobar Status".

In order to execute mass export, there needs to be an instance of `Api/Data/InitialInterface` created for entity type. Then following happens via cron job "custoconnector_initial_schedule_population":

1. `Model/Initial/InitialRunnerInterface` is called on each mapped entity found via `Api/MappingDataProviderInterface`
2. If instance of `Api/Data/InitialInterface` cannot be found, skip the entity type
3. Call to `Model/Initial/Entity/CollectionResolverProviderInterface` to fetch entity type based collection object
4. Filter the collection with page and page size set on on the initial, so only a certain chunk of data is retrieved via collection
5. Generate schedules based on the entities provided by the collection: [Scheduling export](#)
6. Initial entity's data is updated so that next set of entities can be picked from next page of the collection or the entity is marked as finished

The generated schedules are then later caught by the actual export handling ([Exporting data](#)).

What exact is initial entity?

Initial is set of data that indicates the status of mass export. It contains page information for filtering the related entity type's collection and timestamps to tell when the export has started and when it has finished. There is also a status flag to indicate what's happening with the export:

- 0 = pending, nothing is done
- 1 = processed, export finished
- 2 = running, export is going on

Initial entity data is located in database in table "custoconnector_initial".

Creating initial entity

Currently the only way that initials get created by the system is when prompted in admin (System / Custobar / Custobar Status). And how this on the background is via: `Model/Initial/PopulatorInterface`. The default implementation acts as follows, per entity type:

1. Get a collection of the entity type via `Model/Initial/Entity/CollectionResolverProviderInterface`, without applying any filters
2. Apply page size of 500, then retrieve the last page number from the collection
3. Create initial with following data:
 - a. `page = 0`
 - b. `pages =` the last page number from the collection mentioned above
 - c. `entity_type =` the requested one
 - d. `created_at =` current time
 - e. `status = 2`

If there is already an initial with same `entity_type`, then the other information will just be updated on the existing one.

Showing mass export status

In admin, under "System / Custobar / Custobar Status" you can view the mass export statuses for each mapped entity type ([Configuring mapping data](#)).

Record type	Status	Export %	Last Export	
Products	Export running Started at 2020-06-24 06:41:25	41 %	2020-06-24 06:39:51	<input type="button" value="Cancel"/>
Customers	Processed	-	2020-06-24 06:39:51	<input type="button" value="Rerun"/>
Events	Not running	-		<input type="button" value="Run"/>
Sales	Not running	-	2020-06-24 06:39:52	<input type="button" value="Run"/>
Shops	Processed	-	2020-06-24 06:39:52	<input type="button" value="Rerun"/>

The data shown on each row is based on the initial entity data available for each entity type. If there's no initial data available, row is still shown but with less data (see "Events" in above screenshot).

The listing refreshes automatically every 10 seconds for rows that have been in running status since page was loaded.

Cancelling mass export

Hitting "Cancel" or "Cancel All" will update the initial data to status 0 or 1 (depending on `processed_at` timestamp availability), which is when nothing happens with that initial.



Cancelling initial will not delete the already generated schedules at the moment, but no further schedules should be generated.

After cancelling, if you run the export, the schedule generation will start from the beginning.

Scheduling export

General

Data export to Custobar by the module happens based on a new entity "schedule" established by the module. On high level, schedule is generated when configured entity is created/updated. Crons check the generated schedules and send data to Custobar API based on the schedules.

What is a schedule?

It's a set of data that indicates a single Magento entity needs to be exported to Custobar. Schedule does not contain data on what will be actually sent, only references to Magento entity and the schedule's status.

Unlike how the name would imply, schedule does not indicate exact time for export execution.

Where are schedules stored?

In database the table "custoconnector_schedule".

Can you see schedules in admin?

At the moment no.

How are schedules created by default?

The module introduces one observer to event "model_save_after":
Observer/GenerateScheduleOnEntitySave. What it does is following:

1. Validate module logic execution via Api/ExecutionValidatorInterface, stop the observer execution if this doesn't pass
2. Convert the entity to entity that can be scheduled via Api/EntityDataConverterInterface and use the resulting entity in following steps
 - a. For example, customer address is not meant to be exported to Custobar as it is, but it is related to customer which is wanted to be exported
 - b. See further down on more about this
3. Check if the entity type is configured via Api/EntityTypeResolverInterface, stop the observer execution if this doesn't pass
4. Validate if entity can be scheduled via Api/SchedulingValidatorInterface, stop the observer execution if this doesn't pass
5. Resolve the store ids for the entity (currently done in the observer itself but could be later refactored to it's own interface)

6. For each resolved store id, call `Api/ScheduleGeneratorInterface`, which then creates the schedule to database

If exceptions are thrown, these are caught and logged.

When does the export happen based on the schedules?

Currently cron job "custoconnector_callcustobarapi" is responsible of this. What happens is that it checks unprocessed schedules and then for each, based on the related entity, constructs the data that will be sent to Custobar API and then sends that data.

What happens to schedules after the export?

If success, the current timestamp will be stored on schedule after the export. Once this is set, cron job "custoconnector_schedule_clean_up" knows to delete the schedule when it runs.

If there have been errors, the schedule is marked for retry until a hardcoded error limit 7200 is reached. Reason behind this high error limit is currently unknown.

What can I use to create schedules?

There are three options.

Api/ScheduleBuilderInterface + Api/ScheduleRepositoryInterface

The builder interface allows you to construct instance of `Api/Data/ScheduleInterface` by just giving data array. The resulting schedule is then saved to database via the repository class.

The default implementations for the two do not have much logic in them so there's not really any DI based extension points created, but if logic needs to be changed plugins work and it's always possible to create your own implementation for the interface.

Api/ScheduleGeneratorInterface

This is the easiest option since it's practically a wrapper for the first option. However, to respect the earlier data structure and schedule generation flow, the default implementation only allows creating new schedules. Rescheduling is currently not available through this interface. If you want to bypass this, you'll need to create your own implementation for the generator interface or then use the builder and repository classes separately.

Due to the simple implementation this one also doesn't really have any DI based extension points, but as mentioned the interface's default implementation is replaceable via DI and plugins should apply to the default implementation as well.

Model/ScheduleFactory + Api/ScheduleRepositoryInterface

This is practically the raw option of the first option, except you will need to fill the schedule object yourself before saving. Avoid this unless absolutely necessary as it can break the consistent flow in the module.

Is it possible to cancel schedules?

There's no functionality for this by default but if you know the schedule id you can always use the Api/ScheduleRepositoryInterface to delete the schedule.

Is it possible to for example schedule customer for export based on it's address updates?

Customer address updates already schedule the related customer for export, so yes. However, following example does only work when generating schedules via Observer/GenerateScheduleOnEntitySave. Anything that doesn't fit there needs to be implemented separately.

As mentioned earlier, when the observer runs, the updated entity will be converted to another entity via Api/EntityTypeConverterInterface. If there's already mapping configuration ([Configuring mapping data](#)) for the given entity, then no conversion will be done and the given entity is returned. If there's no mapping configuration, then the conversion is executed.

Doing this requires following configurations, here's example with customer address:

```
<type name="Custobar\CustoConnector\Model\EntityTypeResolver">
  <arguments>
    <argument name="components" xsi:type="array">
      <item name="Magento\Customer\Model\Address" xsi:type="object"
>Custobar\CustoConnector\Model\EntityTypeResolver\ResolverComponent\CustomerAd
dress</item>
    </argument>
  </arguments>
</type>

<type name="Custobar\CustoConnector\Model\EntityDataConverter">
  <arguments>
    <argument name="converters" xsi:type="array">
      <item name="Magento\Customer\Model\Address" xsi:type="object"
>Custobar\CustoConnector\Model\EntityDataConverter\ToMapped\CustomerAddress</i
tem>
    </argument>
  </arguments>
</type>
```

```
</type>
```

First configuration makes it possible to resolve entity type string from given object, so if customer address entity is given then it will always return "Magento\Customer\Model\Address", which can be then used as identifier for various components. The object added here must always implement Model/EntityTypeResolver/ResolverComponentInterface.

Then the second configuration adds the actual conversion component to the converter interface, with the address type as identifier. The exact logic is defined in Model/EntityDataConverter/ToMapped/CustomerAddress, which implements Api/EntityDataConverterInterface.

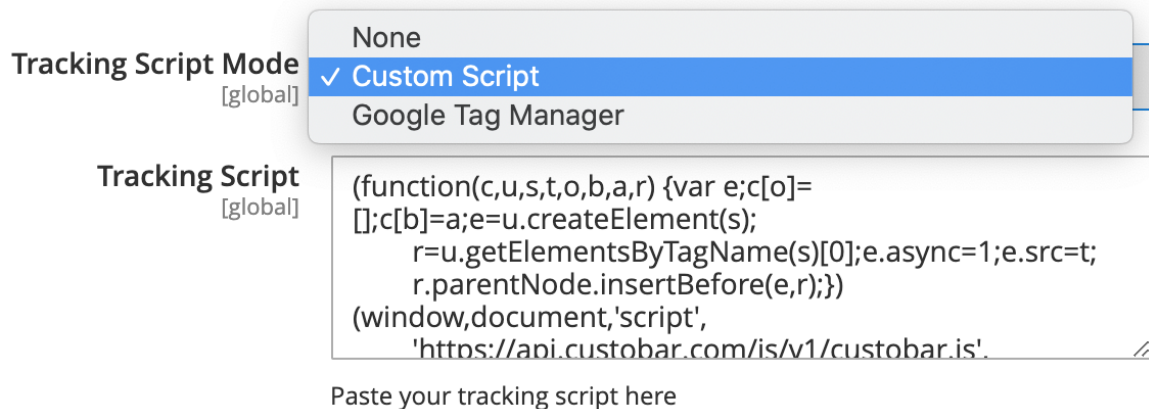
Tracking script

Installing the tracking script

The module supports both installation methods described in <https://www.custobar.com/docs/api/tracking/>. These are:

- Manual installation, inserting own Custobar tracking script to all pages
- Installation with Google Tag Manager

Both options can be configured, but only one will be used in runtime. This can be selected from "Stores / Configuration / Custobar / CustoConnector":



The image shows a configuration interface with two sections:

- Tracking Script Mode** [global]: A dropdown menu with three options: "None", "✓ Custom Script" (selected), and "Google Tag Manager".
- Tracking Script** [global]: A text area containing the following JavaScript code:

```
(function(c,u,s,t,o,b,a,r) {var e;c[o]=  
[];c[b]=a;e=u.createElement(s);  
r=u.getElementsByTagName(s)[0];e.async=1;e.src=t;  
r.parentNode.insertBefore(e,r);})  
(window,document,'script',  
'https://api.custobar.com/is/v1/custobar.is')
```

Below the text area is the label "Paste your tracking script here".

By default "Custom Script" is set, as this is what the earlier implementation of the module has only used.

If "Tracking Script" config is empty, nothing will be done. When adding the script, make sure to not include the <script> tags (see above image).

Option "Google Tag Manager" does not have any configurations specific to it at the moment.

Both "Google Tag Manager" and "Custom Script" options have been already configured in the code based on the examples in <https://www.custobar.com/docs/api/tracking/>. So if no customisation outside of default functionality is needed, you should be able to simply set this setting from Magento and follow the API documentation for rest of the steps. See rest of this page to see how to include more fields and such, if necessary.

Customising custom script behaviour

If the current store view is in allowed websites (see [Configuring logic restrictions](#)), tracking script is set and "Tracking Script Mode" is set to

"Custom Script", template

view/frontend/templates/custoconnector/statistics/custom-script.phtml will be added on all pages. This is done by view/frontend/templates/custoconnector/statistics.phtml, which has been added to default.xml layout as such:

```
<referenceContainer name="before.body.end">
  <block name="custobar_statistics"
    class="Custobar\CustoConnector\Block\Statistics"
    template="custoconnector/statistics.phtml">
    <block name="custobar_statistics_custom_script"
      as="custom_script"
      class="Custobar\CustoConnector\Block\Statistics"
      template="custoconnector/statistics/custom-script.phtml">
      .....
    </block>
  </block>
</referenceContainer>
```

What the custom-script.phtml does at the moment is insert the tracking script from admin configuration and then it renders blocks under it. Currently these blocks add the following part from the tracking API documentation:

```
cstbrConfig.customerId = {{CustomerID}};
cstbrConfig.productId = {{ProductID}}
```

These are done in separate templates:

- view/frontend/templates/custoconnector/statistics/config/customer.phtml
- view/frontend/templates/custoconnector/statistics/config/product.phtml

This is so that it's possible to customise either one of these specifically or even add new fields if needed.

Adding support for new field

You will first need a block that can give you the information needed by the block. See for example Block/Statistics/Config/Customer.



After this, you will need to create template for this block, that adds the new field on `cstbrConfig` object. See for example `view/frontend/templates/custoconnector/statistics/config/customer.phtml`.

After this, you need create `default.xml` under your customisation module's `view/frontend/layout` folder, with following:

```
<referenceBlock name="custobar_statistics_custom_script">
  <block name="custobar_statistics_custom_script_new_field"
    class="Custobar\CustoConnector\Block\Statistics\Config\NewField"
    template="custoconnector/statistics/config/new-field.phtml" />
</referenceBlock>
```

Clean caches and the new field should be added alongside the default ones.

Modifying existing field logic

By default when current customer/product id is added, they will be only added if that information is available. Product id is also actually the product sku. If you this needs to be changed, one thing you can do for example for customer, in `default.xml`:

```
<referenceBlock name="custobar_statistics_custom_script_customer"
  remove="true" />

<referenceBlock name="custobar_statistics_custom_script">
  <block name="custobar_statistics_custom_script_customer_customised"
    class="My\CustoConnectorModule\Block\Statistics\Config\Customer"
    template="My_CustoConnectorModule::custoconnector/statistics/config/customer.phtml" />
</referenceBlock>
```

In other words delete the original block and add a new one with different name that uses your own custom block and template.

You can also just leave out the bottom part if you wish to remove one of the fields.

Customising GTM behaviour

If the current store view is in allowed websites (see [Configuring logic restrictions](#)) and "Tracking Script Mode" is set to "Google Tag Manager", template `view/frontend/templates/custoconnector/statistics/gtm.phtml` will be added on all pages. This



is done by `view/frontend/templates/custoconnector/statistics.phtml`, which has been added to `default.xml` layout as such:

```
<referenceContainer name="before.body.end">
  <block name="custobar_statistics"
    class="Custobar\CustoConnector\Block\Statistics"
    template="custoconnector/statistics.phtml">
    <block name="custobar_statistics_gtm"
      as="gtm"
      class="Custobar\CustoConnector\Block\Statistics\GTM"
      template="custoconnector/statistics/gtm.phtml">
      .....
    </block>
  </block>
</referenceContainer>
```

What the `gtm.phtml` does at the moment is add following from the tracking API documentation:

```
window.dataLayer = window.dataLayer || [];
window.dataLayer.push({
  cb_customer_id: YOUR_CUSTOMER_ID,
  cb_product_id: YOUR_PRODUCT_ID
});
```

However, this implementation borrows a bit from the custom script. What the custom script implementation does is add first the tracking script, which defines JS variable called `"cstbrConfig"`. This is then filled up with the Custobar fields from customer and product specific templates.

What the GTM implementation does is it first defines this `"cstbrConfig"` JS variable as empty, then uses the same customer and product specific templates to fill it up and then convert it to data pushed to GTM. Similar to custom script implementation this can also be customised to include new fields or modify the existing ones.

Adding support for new field

If you have created block and template for your field for the custom script implementation, you can reuse the same ones. If not or you need custom logic for GTM, you will need to create those. See examples from:

- `Block/Statistics/Config/Customer.php`
- `view/frontend/templates/custoconnector/statistics/config/customer.phtml`



After this, you need create default.xml under your customisation module's view/frontend/layout folder, with following:

```
<referenceBlock name="custobar_statistics_gtm">
  <block name="custobar_statistics_gtm_new_field"
    class="Custobar\CustoConnector\Block\Statistics\Config\NewField"
    template="custoconnector/statistics/config/new-field.phtml" />
  <arguments>
    <argument name="gtm_field_mapping" xsi:type="array">
      <item name="cb_new_field" xsi:type="string">newField</item>
    </argument>
  </arguments>
</referenceBlock>
```

Here you need to pay attention especially to the "gtm_field_mapping" configuration. You want that "cb_new_field" matches a data layer variable in GTM and that "newField" is the one you added to the "cstbrConfig" JS variable in your new template. This configuration allows you to automatically map the necessary fields so you don't need to override the template used to do the actual push to data layer.

Clean caches and the new field should be sent to GTM alongside the default ones.

Modifying existing field logic

This can be done in similar manner as with custom script implementation:

```
<referenceBlock name="custobar_statistics_gtm_customer" remove="true" />

<referenceBlock name="custobar_statistics_gtm">
  <block name="custobar_statistics_gtm_customer_customised"
    class="My\CustoConnectorModule\Block\Statistics\Config\Customer"
    template="My_CustoConnectorModule::custoconnector/statistics/config/customer.phtml" />
</referenceBlock>
```

You can also modify the field mapping, if you wish. However if you only modify existing field's logic and not the field itself this is likely unnecessary.

```
<referenceBlock name="custobar_statistics_gtm">
  <arguments>
    <argument name="gtm_field_mapping" xsi:type="array">
```



```
<item name="cb_customer_id"  
xsi:type="string">newCustomerIdField</item>  
  </argument>  
</arguments>  
</referenceBlock>
```