Export API

You can trigger an export using the export API in a number of ways. These include using the back end management interfaces, as well as triggering it programmatically.

You can trigger an export of data for the **ImpEx** extension in the following ways:

1. In Backoffice using the ImpEx Export Wizard. See Export Wizard.
2. In Backoffice, using an ImpExExportCronjob. See Export Using an ImpexExportCronJob.
3. In SAP Commerce Administration Console, using the `ImpEx` extension page. See Export Through Administration Console.
4. Using the export API which is described here.

While at an import script, the data to be imported is specified via value lines, an export script has a different structure to define the set of items to be exported, as well as the export file format. Find more information on the structure of an export script, see Structure of an Export Script. The available validation modes for an export script are described in Validation Modes.

You have several possibilities to perform an export programmatically. The decision depends mainly on the specialized configuration needs.

The basic kind of processing is the instantiation and configuration of the `Exporter` class. Here you have the full range of configuration possibilities. The instantiation and configuration of an `Exporter`does an export cronjob too, but it additionally provides the features of a cronjob, that is: all settings, results, and logs are stored as persistent, which is strongly preferred. The third convenient alternative is the usage of the API methods of the `ImpExManager`. They also use an export cron job, but you do not have to create and configure it on your own.

Export Using an Exporter Instance

The `Exporter` class is the central class for processing an export. You can use this class directly for exporting data in three steps.

Procedure

1. Define your export configuration.

   All configuration is done by instantiation of an `ExportConfiguration`. This is a container for all configuration you can set for the exporter and is passed at instantiation of the Exporter. The constructor needs an `ImpExMedia` where the export script is managed by and the validation mode given as `EnumerationValue`.

   ```
   ExportConfiguration config = new ExportConfiguration( impexscript, ImpExManager.getExportOnlyMode() );
   ```

   Configuration of additional settings can then be done via the setter methods of the object, for example for setting the result medias to use explicitly. Note that otherwise they are created automatically.

   ```
   config.setDataExportTarget(datatarget);
   config.setMediasExportTarget(mediastarget);
   ```

2. Create an instance of the class.

   With the created `ExportConfiguration` you can now create an instance of the **Exporter** class using:

   ```
   Exporter exporter = new Exporter( config );
   ```

3. Start the export.

You can start the export process using a simple call of the `export` method:

```
Export export = exporter.export();
```

The returned `Result` item contains all result medias accessible via getter methods.

Export Using an ImpexExportCronJob

You can generate an export using the ImpEx export cron job.

When using the `ImpExExportCronjob`, you have the advantage of persistent logging, as well as persistent result and settings holding. You can create a cron job using the `createDefaultImpExExportCronJob` method of the `ImpExManager` class. Possible settings are provided in the API of the cron job class. For further details, see Using the ImpExImportCronJob.

A sample configuration looks like this:

```
// Creating export media
ImpExMedia jobMedia = createImpExMedia( "myExportScript", "UTF-8" );
jobMedia.setFieldSeparator( ';' );
jobMedia.setQuoteCharacter( "\"" );
jobMedia.setData( new DataInputStream( ImpExManager.class.getResourceAsStream("myScript.impex")),
   jobMedia.getCode() + "." + ImpExConstants.File.EXTENSION_CSV, ImpExConstants.File.MIME_TYPE_CSV );

// Creating an ExportConfiguration
ExportConfiguration config = new ExportConfiguration( impexscript, ImpExManager.getExportOnlyMode() );

// Creating export cronjob
ImpExExportCronJob cronJob=createDefaultExportCronJob( config );

// export
cronJob.getJob().perform( cronJob );

// Get result
Export export=cronJob.getExport();
```

Using an Export Method of the ImpEx Manager

You can export data using dedicated methods provided by the `ImpExManager` class.

The `ImpExManager` class has different methods with the qualifier `exportData`. These methods all use a cronjob for performing an export with given `ExportConfiguration` and help you to simplify the import call. The only important additional parameter for this methods is `synchronous`. This parameter defines if the created cronjob is performed synchronous or asynchronous.

```
// Creating an ExportConfiguration with an media containing the script
ExportConfiguration config = new ExportConfiguration( impexscript, ImpExManager.getExportOnlyMode() );
// Export
Export export = ImpExManager.getInstance().exportData( config, true );
```

Another set of methods for export are the `exportDataLight` methods. These are lightweight methods using no cronjob and so no persistent and logging offset.

```
// Creating an ExportConfiguration with an media containing the script
ExportConfiguration config = new ExportConfiguration( impexscript, ImpExManager.getExportOnlyMode() );
// Export
Export export = ImpExManager.getInstance().exportData( config );
```

Structure of an Export Script

An export script needs to specify the target file to export items to, a header line for defining how to export the items, and a statement specifying which items to export.

The structure of a typical export script using the export of all languages is shown in the following example.

```
"#% impex.setTargetFile( ""language.csv"", true, 1, -1 );" // 1. where to export
        insert_update Language;active;fallbackLanguages(isocode);isocode[unique=true];name[lang=de];name[lang=en] // 2. how to export
        "#% impex.exportItems( ""Language"" , true );" // 3. what to export
```

The sample shows a typical script. Here all languages of the system are to be exported to a file called `language.csv` using the given header. The order of the three statements is important.

1. The first line is used to specify where to export the items followed by the next header. Using the given file name, a file is created at the resulting archive where all items are written to and which are exported using the next header line. The **setTargetFile** method is located at the **Exporter** class and comes in different signatures all covering a different combination of the parameters. The signature using all parameters is:

   ```
   public void setTargetFile( final String filename, boolean writeHeader, int linesToSkip, int offset )
   ```

   where the parameters are:

Name	Type	Default	Description
filename	String	Type code configured at next header line.	Name for target file within resulting archive where the items of the next header are written.
writeHeader	boolean	true	If set, the header is written as a comment to the first line at the target data file.
linesToSkip	int	0	Parameter is used at generated include statement at the generated import script. In the import case, it means the amount of lines that are skipped when start reading from external data.
columnOffset	int	-1	Specifies the column offset of the target data file compared to ImpEx standard: if the first column already contains data use -1 - is set at generated include statement at generated import script.

   If you use a signature with less parameters, the default values are used for the missing ones. Be aware that you do not have to give such a statement, it is optional. In that case, all default values are used (a file is created using the name of the type configured at the header line).

2. The second line is a header line describing how to export items. This is analogous to the import case except for the header mode (INSERT, UPDATE, and so on), which is ignored at the export. However a header mode has to be given, furthermore the header is copied to the generated import script, and so it is useful to give a correct one.

3. The **exportItems** call gathers the set of items to export and exports them using the set header and target file. The method is located at the **Exporter** class and comes in different signatures of different nature.

   - **exportItems** by item set:

     ```
     public void exportItems( Collection<Item> items )
                         public void exportItems( String[] pklist )
     ```

     These methods export given items where the items can be passed either as a list of PK's (String) or directly using a Collection of items.

   - **exportItems** by type code:

     ```
     public void exportItems( String typecode )
                         public void exportItems( String typecode, int count )
                         public void exportItems( String typecode, boolean inclSubTypes )
                         public void exportItems( String typecode, int count, boolean inclSubTypes )
     ```

     It gathers items by selecting all items of a given type code and exports them. The parameters are:

Name	Type	Default	Description
typecode	String	-	The **typecode** where all items are gathered and exported.
count	int	1000	A range parameter - many FlexibleSearches are performed each covering **count** items of the type - does not limit the overall count.
inclSubTypes	boolean	false	If true, subtypes of given type are

			considered.

- **exportItems** by FlexibleSearch:

> **Note**
>
> **Paging of Search Result**
>
> Exporter API by default uses pagination of search results, therefore, to have accurate results, your FlexibleSearch queries must contain the **ORDER BY** clause, for example **ORDER BY {pk}**. If you disable pagination or you use some method that cares also for ordering of results, then the **ORDER BY** clause is not needed. For more details read FlexibleSearch, section **Paging of Search Result**.

```
// since 3.1-RC
                    public void exportItemsFlexibleSearch( String query )
                    public void exportItemsFlexibleSearch( String query, Map values, List resultClasses, final boolean failOnUnknownFields,
                    final boolean dontNeedTotal, int start, int count )
                    // since 3.1-u6
                    public void exportItemsFlexibleSearch( String query, int count )
```

It gathers items using a given FlexibleSearch query. Be aware that the resulting items have to be compatible with the current header.

You can generate a script for all types of your platform by using the **ScriptGenerator** in Backoffice. For details, see Script Generator.