Import API

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

There are four basic ways of triggering an import of data for the `ImpEx` extension:

1. Using the Backoffice ImpEx Import Wizard. For details, see Import Wizard.
2. Creating an Import CronJob in Backoffice. For details, see Import CronJob.
3. Using `ImpEx Import` page in SAP Commerce Administration Console. For details, see Import Through Administration Console.
4. Using the Import API, which is described here.

To support the systems with large numbers of products, SAP Commerce has a capability of multithreaded import operations. For details, see Multithreaded Import.

You have several possibilities to perform an import programmatically. The decision depends mainly on the specialized configuration needs. The basic kind of processing is the instantiation and configuration of the **Importer** class. For detailed information, Using an Importer Instance. Here you have the full range of configuration possibilities. The instantiation and configuration of an **Importer** class triggers an import 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** method. For detailed information, see Using a Method of ImpExManager. They also use an import cronjob, but you do not have to create and configure it on your own.

Using an Importer Instance

The **Importer** class is the central class for processing an import. The process for importing by directly using this class has 3 steps.

Procedure

1. Instantiate the class.

   While instantiating the **Importer** class, this CSV-stream is given using a **CSVReader** or an **ImpExImportReader**. If you only want to specify the input stream, use an **CSVReader** class, the **Importer** instantiates a corresponding **ImpExImportReader**. The usage of an **ImpExImportReader** is only needed, if special settings while instantiation are needed (settings after instantiation can be done using the **getReader()** method of the **Importer** instance).

   Example:

   ```
   CSVReader reader  = new CSVReader( "input.csv", "utf-8" );
   Importer importer = new Importer( reader );
   ```

   or:

   ```
   Importer importer = new Importer( new ImpExImportReader( reader, new MyImportProcessor() ) );
   ```

2. Configure the import process.

   You have several possibilities for configuring the import process.

   - You can configure the used **ImpExImportReader** using the **getReader** method. Here you can configure several things about the reading of the input (**skipValueLines**, **enableCodeExecution**, and so on) or item processing (**setRelaxedMode**, and so on).
   - You can set a **DumpHandler** for specifying the dump file handling. Thirdly, you can set an **ErrorHandler** to specify the process in case of an error. Fourthly, you can set the maximal amount of passes for resolving dumped value lines.

3. Trigger the import.

You can perform the import using the **importNext**, which processes the input stream until an item was processed (that means **inserted**, **updated** or **removed**). It returns the processed item. Another possibility is the usage of the **importAll** method, which calls the **importNext** method until finishing of the input stream. While and after the import process, you have several possibilities to get information about the state, for example: current pass, processed items, and so on.

Example:

```
Item item = null;
do
{
 item = importer.importNext();
 System.out.println( "Processed items: " + getProcessedItemsCountOverall() );
}
while( item != null );
```

or:

```
importer.importAll();
System.out.println( "Processed items: " + getProcessedItemsCountOverall() );
```

Using ImpExImportCronJob

When using `ImpExImportCronjob`, you have the advantage of persistent logging, as well as persistent result and settings holding.

The possible settings you can find in the API of the cron job class. The following sample shows an example configuration:

```
try
{
  // Creating import media
  ImpExMedia jobMedia = createImpExMedia( "myImportScript", "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 );

  // create cronjob
  ImpExImportCronJob cronJob = ImpExManager.getInstance().createDefaultImpExImportCronJob();
  cronJob.setEnableCodeExecution( codeexecution );
  cronJob.setJobMedia( jobMedia );

  // process import
  cronJob.getJob().perform( cronJob, true );
}
catch( UnsupportedEncodingException e )
{
  log.error( "Given encoding is not supported", e );
}
```

Using a Method of ImpExManager

The `ImpExManager` class provides methods with the qualifier **importData**. These methods all use a cron job for performing an import from a given source and help you to simplify the import call.

Important parameters for the import methods are as follows:

- **synchronous** - sets the created cronjob to be performed synchronous or asynchronous.
- **removeOnSuccess** - sets the cronJob and created medias to be removed if finished successfully.
- **codeExecution** - sets the execution of BeanShell code to be enabled.

If the **removeOnSuccess** flag is set, the resulting cronjob may not be valid anymore after import, because it is already removed.

```
InputStream is = ImpExManager.class.getResourceAsStream( csv );
 ImpExManager.getInstance().importData( is, "windows-1252",
   CSVConstants.HYBRIS_FIELD_SEPARATOR, CSVConstants.HYBRIS_QUOTE_CHARACTER, true );
```

```
CSVConstants.HYBRIS_FIELD_SEPARATOR, CSVConstants.HYBRIS_QUOTE_CHARACTER, true );
```

Another set of methods for import are the **importDataLight** methods. These are lightweight methods, note that they have prefix **light**, using no cronjob, and so no persistent and logging offset.

```
InputStream is = ImpExManager.class.getResourceAsStream( csv );
ImpExManager.getInstance().importDataLight( is, "windows-1252", true );
```

Data Inclusion

Often you want to separate your ImpEx script logic from the data for example if the data is provided by a foreign system, and you do not want to touch this file by adding a ImpEx header. This is possible by using the `include` methods of the `ImpExReader` instance registered at the BeanShell context.

Storing both header definitions and data in one single file is a possible approach for smaller files. However, if you want to separate your **ImpEx** script logic from the data, for example, if you need to process externally supplied data (from a customer or supplier), or if your file exceeds several hundred lines of data, you probably want to split it. The **ImpEx** extension allows you to include other files within the parse process using the **include** methods of the **ImpExReader** class.

Basically there are three possibilities for inclusion, first the reference of input streams outside of the platform like files from file system, second the referencing of medias from the platform, and the third alternative is the direct inclusion of data from a database.

Inclusion of Data using Input Streams

The inclusion of input streams within the parse process can be achieved by using the **includeExternalData** methods (marked with the **BeanShell** annotation) of the **ImpExReader** instance registered as **ImpEx** variable at the BeanShell context. Just add such a call to the script and the stream is included directly at the position of the call within the parse process. So be sure that there is a header written at the main script before calling an include containing only raw data.

```
INSERT_UPDATE Product;code[unique=true];...
              "#% impex.includeExternalData(ImpExManager.class.getResourceAsStream(""myDataFile.csv""), ""utf-8"", 1, -1 );"
```

The common parameters used at most method signatures are:

Name	Type	Default value	Description
linesToSkip	int	0	Amount of lines that are skipped when start reading from external data.
columnOffset	int	-1	The column offset compared to **ImpEx** standard: if the first column already contains data use -1.
encoding	string	UTF-8	The encoding of the external CSV data.
delimiter	char	;	Field separator used in external data.

An other method signature alternative allows to set a **CSVReader** instance instead of an input stream where you can configure some additional parameters like the maximal buffer size for reading a cell, which is spread over many lines (By default a **ImpEx** CSV cell can only consist of 10000 lines).

```
INSERT_UPDATE Product;code[unique=true];...
              "#% CSVReader reader = new CSVReader( ImpExManager.class.getResourceAsStream(""myDataFile.csv""), ""utf-8"" );"
              "#% reader.setMaxBufferLines(1000001);"
              "#% impex.includeExternalData( reader, 1, -1 );"
```

Note

You can chain includes of data, so you can include data within an already included data. Please do not use the **ImpExManager.importData()** methods within BeanShell code, because it starts a completely separate import and with that a completely different context as for example the BeanShell interpreter and the resolving of dumped lines.

Inclusion of Data using Media

If you already have your data file added to the platform as an instance of a **ImpExMedia** type, you can simply include the media using the following BeanShell call:

```
INSERT_UPDATE Product;code[unique=true];...
              "#% impex.includeExternalDataMedia( ""MyDataMediaCode"" );"
```

The advantage of using the media type is you can do the configuration using the properties of the media instance, so you just give the **code** of the media as parameter.

Unfortunately there can be medias with the same code, so it is necessary to provide the import process with a collection of all possibly included medias. You can only do it by using an import procedure where an import cronjob takes part, because the collection of medias is stored at the cronjob instance.

Summarized, to include data using a media, you have to use an import proceeding with cronjob, and you have to set the collection of all included medias at this cronjob (the attribute is called **externalDataCollection**).

Inclusion of Data Through Database Statements

Using the BeanShell, you can also include external data directly from a database connection. Therefore you first have to open a JDBC connection using the **initDatabase** method. Secondly, you can include the data giving a SQL-query whose result set fits the defined header line.

```
#  Initialization
               INSERT_UPDATE XYType; $code[unique=true]; $mandant; $typ; baseProduct(code, catalogVersion(catalog(id[default='myCatalog']),version
               #% impex.initDatabase( <dburl>, <user>, <password>, <driver.class>);
               "#%
               impex.includeSQLData(
               "" SELECT ""+
               ""   myProduct.ProductID, myProduct.Tenant, Variant.myVariantID, (
               myProduct.ProductID + '-base::' + CAST( myProduct.Tenant AS varchar( 2 ) ) ) ""+
               "" FROM DB.SpecialProduct as Product JOIN DB.SpecialProductVariant as VARIANTE ""+
               ""   ON myProduct.ProductID = Variant.ID and myProduct.Tenant = Variant.Tenant""+
               "" WHERE ""+
               ""   Variant.myVariantID > 0 AND Product.variant ='xytype'""
               );
               "
```

Both methods and their signatures can be accessed by using the registered **ImpEx** variable of type **ImpExReader**.

Resolving

In contrast to export, ImpEx import provides a resolving mechanism to deal with value lines that cannot be imported by the time the parser runs across them. These lines are imported as far as possible, the unresolved lines are dumped into a temporary file and read in again when the current file has completed parsing.

The following code snippet gives an example of a case where a line may remain unresolved:

```
INSERT Address; appartment; owner(Principal.uid)
                ;testApp; testUser
INSERT Customer;uid[unique=true]
;testUser
```

Lines 1 and 2 try to create an address using the user **testUser** as owner. However, **testUser** is defined in lines 3 and 4. Therefore, **testUser** does not exist when the attempt to create the address is attempted, and **testUser** cannot be assigned to the address at this point of time. The parser then writes the address definition into a temporary file. When the main file has completed, that temporary file is read in again. Now **testUser** exists, and the address can be finished. Such a further run is called **pass** and an execution of the sample script can result in the following log:

```
Starting import synchronous using cronjob with PK=141025265286919088 and name=00000014-ImpEx-Import
INFO - Starting import with pass 1
INFO - Starting pass 2
INFO - Import finished successfully within 00:00:00:328 (hh:mm:ss:ms)
```

The dump file is stored by default at the used cronjob (attribute **unresolvedDataStore** at **Log** tab) and contains in the above example the following lines:

```
insert Address;appartment;owner(Principal.uid)
,,cannot create due to unresolved mandatory/initial columns;test;testUser
```

If all mandatory columns in the value line were given and successfully translated, an item is created despite the fact that maybe other non-mandatory attributes are not resolved. After creation, an update of the resting columns is made. The second line of the dumped line example shows the value line, which was not completed successfully. The first column of the value line holds three pieces of important information separated by commas. The first field contains the type of the item already created when mandatory attributes were resolved, but not all non-mandatory ones. The second holds the PK of this created item. The third contains an error message, which explains the reason not all attributes were resolved successfully.

If the item is already created but not all attributes were resolved, the already set attributes are ignored with the **ignoreAP** flag. So just search for attributes not marked with that

If the item is already created but not all attributes were resolved, the already set attributes are ignored with the **<ignore>** flag. So just search for attributes not marked with that flag. The following content of such a dump file shows that a value line representing a Product could not be imported completely because the detail attribute was not resolved. You can see that the first and second attribute of the value line contain an **<ignore>** marker, which means that the product was created setting these two attributes. The third attribute was not resolved so there is no **<ignore>** marker.

```
INSERT Product;code;catalogVersion[allowNull=true];detail(code)
Product,288342436307920,, column 3: cannot resolve value 'testDetail' for attribute 'detail';<ignore>testProduct;<ignore>;testDetail
```

If a dump file cannot be imported completely, again, another dump file is written, and the next pass starts. If a dump file has the same size as a dump file of its following pass the import is aborted with following log:

```
ERROR (master) [ImpExImportJob] Can not resolve lines any more ... aborting further passes (at pass 2).
 Finally could not import 1 lines!
```

In this case, you have to evaluate (as described above) the last dump file stored at the used cronjob.

Please be aware that only header and value lines are dumped, no BeanShell. So there is no chance to execute BeanShell code at a second pass.

Using Header Abbreviations

ImpEx provides a way to shorten length column declarations by using regexp patterns and replacements.

Although the ImpEx header definition language provides a most flexible way of using custom column translators, their declaration can grow long. You can shorten them using the ImpEx alias syntax.

The following example shows how to use this syntax to shorten classification columns declaration by giving them a new syntax: Put this into your SAP Commerce platform `local.properties` file:

```
impex.header.replacement.1 =
   C@(\\w+) ...
   @$1[ system='\\$systemName', version='\\$systemVersion',
        translator='de...ClassificationAttributeTranslator']
```

Note that the line has been wrapped to make it more readable - you have to leave it on one line, of course. Also note the Java string notation has to be used, that's why there are double '\'s.

All parameters starting with *impex.header.replacement* are parsed as **ImpEx** column replacement rules. The parameter has to end with a number that defines the priority of the rule. This way ambiguous rules can be sorted.

So what's this for? The first part of the property C@(\w+) defines the new abbreviation pattern to be used to declare classification attribute columns with. The second part is the replacement text including the attribute qualifer match group $1. In fact, it contains the original special column declaration. Both parts are to be separated by '...'.

So all that is needed now to declare classification attribute column is this:

```
$systemName=MySys
$systemVersion=1.0

INSERT_UPDATE Product; code[unique=true]; C@attr1; C@attr2 ; ...
```

Now the whole translator definition is hidden and you do not need to declare any helper alias for that.

Nevertheless both alias definitions are mandatory to tell the classification column which system and version it should take its attribute from.