Importing Items via basic bibliographic formats (Endnote, BibTex, RIS, TSV, CSV) and online services (OAI, arXiv, PubMed, CrossRef, CiNii)

These facilities were developed separately for JSPUI and XMLUI.

Deprecated

According to the discussion in Unable to locate Jira server for this macro. It may be due to Application Link configuration., DSpace 6 is very likely the last version whe n external sources will be extended to match and exceed all functionality offered by BTE enabled imports.

Using JSPUI

This functionality is an extension of that provided by Importing and Exporting Items via Simple Archive Format so please read that section before continuing. It is underpinned by the Biblio Transformation Engine (https://github.com/EKT/Biblio-Transformation-Engine)

About the Biblio-Transformation-Engine (BTE)

The BTE is a Java framework developed by the Hellenic National Documentation Centre (EKT, www.ekt.gr) and consists of programmatic APIs for filtering and modifying records that are retrieved from various types of data sources (eg. databases, files, legacy data sources) as well as for outputing them in appropriate standards formats (eg. database files, txt, xml, Excel). The framework includes independent abstract modules that are executed seperately, offering in many cases alternative choices to the user depending of the input data set, the transformation workflow that needs to be executed and the output format that needs to be generated.

The basic idea behind the BTE is a standard workflow that consists of three steps, a data loading step, a processing step (record filtering and modification) and an output generation. A data loader provides the system with a set of Records, the processing step is responsible for filtering or modifying these records and the output generator outputs them in the appropriate format.

The standard BTE version offers several predefined Data Loaders as well as Output Generators for basic bibliographic formats. However, Spring Dependency Injection can be utilized to load custom data loaders, filters, modifiers and output generators.

BTE in DSpace

The functionality of batch importing items in DSpace using the BTE has been incorporated in the "import" script already used in DSpace for years.

In the import script, there is a new option (option "-b") to import using the BTE and an option -i to declare the type of the input format. All the other options are the same apart from option "-s" that in this case points to a file (and not a directory as it used to) that is the file of the input data. However, in the case of batch BTE import, the option "-s" is not obligatory since you can configure the input from the Spring XML configuration file discussed later on. Keep in mind, that if option "-s" is defined, import will take that option into consideration instead of the one defined in the Spring XML configuration.

Thus, to import metadata from the various input formats use the following commands:

Input	Command
BibTeX (*.bib)	`[dspace]/bin/dspace import -b -m mapFile -e example@email.com -c 123456789/1 -s path-to-my-bibtex-file -i bibtex`
CSV (*.csv)	`[dspace]/bin/dspace import -b -m mapFile -e example@email.com -c 123456789/1 -s path-to-my-csv-file -i csv`
TSV (*.tsv)	`[dspace]/bin/dspace import -b -m mapFile -e example@email.com -c 123456789/1 -s path-to-my-tsv-file -i tsv`

RIS (*.ris)	`[dspace]/bin/dspace import -b -m mapFile -e example@email.com -c 123456789/1 -s path-to-my-ris-file -i ris`
EndNote	`[dspace]/bin/dspace import -b -m mapFile -e example@email.com -c 123456789/1 -s path-to-my-endnote-file -i endnote`
OAI-PMH	`[dspace]/bin/dspace import -b -m mapFile -e example@email.com -c 123456789/1 -s path-to-my-oai-file -i oai`
arXiv	`[dspace]/bin/dspace import -b -m mapFile -e example@email.com -c 123456789/1 -s path-to-my-arxiv-file -i arxivXML`
PubMed	`[dspace]/bin/dspace import -b -m mapFile -e example@email.com -c 123456789/1 -s path-to-my-pubmed-file -i pubmedXML`
CrossRef	`[dspace]/bin/dspace import -b -m mapFile -e example@email.com -c 123456789/1 -s path-to-my-crossref-file -i crossrefXML`
CiNii	`[dspace]/bin/dspace import -b -m mapFile -e example@email.com -c 123456789/1 -s path-to-my-crossref-file -i ciniifXML`

Keep in mind that the value of the "-e" option must be a valid email of a DSpace user and value of the "-c" option must be the target collection handle. Attached, you can find a .zip file (sample-files.zip) that includes examples of the file formats that are mentioned above.

BTE Configuration

The basic idea behind BTE is that the system holds the metadata in an internal format using a specific key for each metadata field. DataLoaders load the record using the aforementioned keys, while the output generator needs to map these keys to DSpace metadata fields.

The BTE configuration file is located in path: `[dspace]/config/spring/api/bte.xml` and it's a Spring XML configuration file that consists of Java beans. (If these terms are unknown to you, please refer to Spring Dependency Injection web site for more information.)

Explanation of beans:

```
<bean id="org.dspace.app.itemimport.BTEBatchImportService" />
```

This is the top level bean that describes the service of the batch import from the various external metadata formats. It accepts three properties:

a) **dataLoaders**: a list of all the possible data loaders that are supported. Keep in mind that for each data loader we specify a key that can be used as the value of option "-i" in the import script that we mentioned earlier. Here is the point where you would add a new custom DataLoader in case the default ones doesn't match your needs.

b) **outputMap**: a Map between the internal keys that BTE service uses to hold metadata and the DSpace metadata fields. (See later on, how data loaders specify the keys that BTE uses to hold the metadata)

c) **transformationEngine**: the BTE transformation engine that actually consisits of the processing steps that will be applied to metadata during their import to DSpace

```
<bean id="batchImportTransformationEngine" />
```

This bean is instantiated when the batch import takes place. It deploys a new BTE transformation engine that will do the transformation from one format to the other. It needs one input argument, the workflow (the processing step mentioned before) that will run when transformation takes place. Normally, you don't need to modify this bean.

```
<bean id="batchImportLinearWorkflow" />
```

This bean describes the processing steps. Currently, there are no processing steps meaning that all records loaded by the data loader will pass to the output generator, unfiltered and unmodified. (See next section "Case studies" for info about how to add a filter or a modifier)

```
<bean id="bibTeXDataLoader" />
<bean id="csvDataLoader" />
<bean id="tsvDataLoader" />
<bean id="risDataLoader" />
<bean id="endnoteDataLoader" />
<bean id="pubmedFileDataLoader" />
<bean id="arXivFileDataLoader" />
<bean id="crossRefFileDataLoader" />
<bean id="oaipmhDataLoader" />
```

These data loaders are of two types: "file" data loaders and "online" data loaders. The first 8 of them belong to file data loaders while the last one (OAI data loader) is an online one.

The file data loaders have the following properties:

a) **filename**: it is a String that specifies the filepath to the file that the loader will read data from. If you specify this property, you do not need to give the option "-s" to the import script in the command prompt. If you, however, specify this property and you also provide a "-s" option in the command line, the option "-s" will be taken into consideration by the data loader.

b) **fieldMap**: it is a map that specifies the mapping between the keys that hold the metadata in the input file and the ones that we want to have internal in the BTE. This mapping is very important because the internal keys need to be declared in the "outputMap" of the "DataLoadeService" bean. Be aware that each data loader has each own input file keys. For example, RIS loader uses the keys "T1, AU, SO ... " while the TSV or CSV use the index number of the column that the value resides.

Some loaders have more properties:

CSV and TSV (which is actually a CSV loader if you look carefully the class value of the bean) loaders have some more properties:

a) **skipLines**: A number that specifies the first line of the file that loader will start reading data. For example, if you have a csv file that the first row contains the column names, and the second row is empty, the the value of this property must be 2 so as the loader starts reading from row 2 (starting from 0 row). The default value for this property is 0.

b) **separator**: A value to specify the separator between the values in the same row in order to make the columns. For example, in a TSV data loader this value is "\u0009" which is the "Tab" character. The default value is "," and that is why the CSV data loader doesn't need to specify this property.

c) **quoteChar**: This property specifies the quote character used in the CSV file. The default value is the double quote character (").

The OAIPMHDataLoader has the following properties:

a) **fieldMap**: Same as above, the mapping between the input keys holding the metadata and the ones that we want to have internal in BTE.

b) **serverAddress**: The base address of the OAI provider (server). Base address can be specified also in the "-s" option of the command prompt. If is specified in both places, the one specified from the command line is preferred.

c) **prefix**: The metadata prefix to be used in OAI requests.

Since DSpace administrators may have incorporated their own metadata schema within DSpace (apart from the default Dublin Core schema), they may need to configure BTE to match their custom schemas.

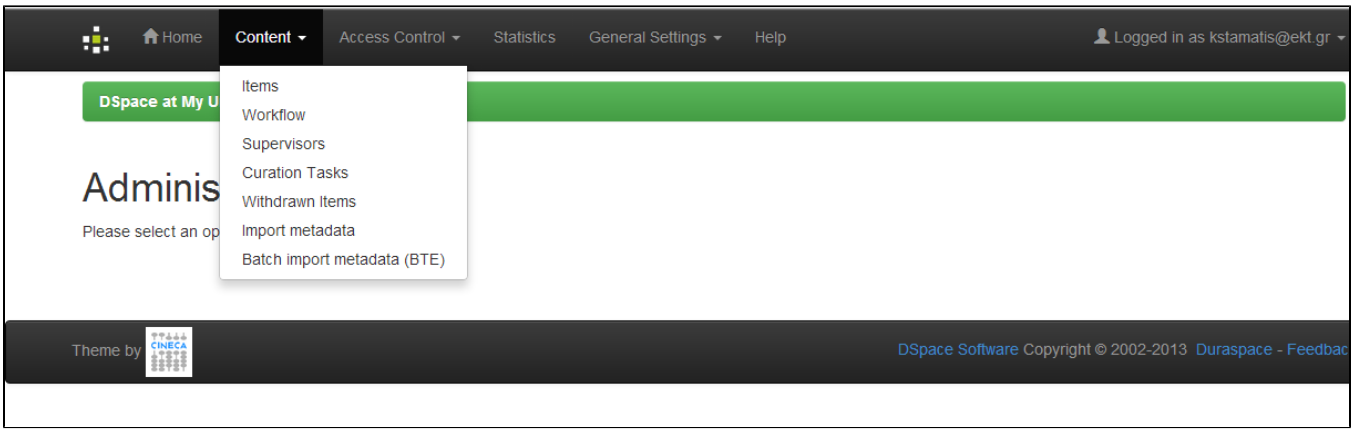
So, in case you need to process more metadata fields than those that are specified by default, you need to change the data loader configuration and the output map.

I can see more beans in the configuration file that are not explained above. Why is this?

The configuration file hosts options for two services. BatchImport service and SubmissionLookup service. Thus, some beans that are not used for the latter, are not mentioned in this documentation. However, since both services are based on the BTE, some beans are used by both services.

UI for administrators

Batch import of files can be done via the administrative UI. While logged in as administrator, visit "Administer" link and then, under the "Content" drop down menu, choose "Batch import metadata (BTE)"



In the screen that follows, select the file to upload, select the data type of the file to be uploaded (bibtex, csv, etc.) and finally, select the collections the data need to be inserted to.

Keep in mind, that the type drop down menu includes all the supported data loaders declared in the configuration XML file that are of type "file". Thus, OAI data loader is not included in this list and in case you need to create your own data loader you are advised to extend the "FileDataLoader" abstract class rather than implement the "DataLoade" interface, as mentioned in previous paragraph.

The whole procedure can take long time to complete, in case of large input files, so the whole procedure runs in the background in a separate thread. When the thread is completed (either successfully or erroneously), the user is informed via email for the status of the import.

Case Studies

1) I have my data in a format different from the ones that are supported by this functionality. What can I do?

Either you try to easily transform your data to one of the supported formats or you need to create a new data loader. To do this, create a new Java class that implements the following Java interface from BTE:

```
gr.ekt.bte.core.DataLoader
```

You will need to implement the following method:

```
public RecordSet getRecords() throws MalformedSourceException
```

in which you have to create records - most probably you will need to create your own Record class (by implementing the gr.ekt.bte.core.Record interface) and fill a RecordSet. Feel free to add whatever code you like in this method, even to read data from multiple sources. All you need is just to return a RecordSet of Records.

You may also extend the abstract class

```
gr.ekt.bte.core.dataloader.FileDataLoader
```

if you want to create a "file" data loader in which you need to pass a filepath to the file that the loader will read the data from. Normally, a simple data loader is enough for the system to work, but file data loaders are also utilized in the administration UI discussed later in this documentation.

After that, you will need to declare the new DataLoader in the Spring XML configuration file (in the bean with id=" org.dspace.app.itemimport. BTEBatchImportService ") using your own unique key. Use this key as a value for option "-i" in the batch import in order to specify that the specific data loader must run.

2) I need to filter some of the input records or modify some value from records before outputting them

In this case you will need to create your own filters and modifiers.

To create a new filter, you need to extend the following BTE abstact class:

```
gr.ekt.bte.core.AbstractFilter
```

You will need to implement the following method:

```
public abstract boolean isIncluded ( Record  record )
```

Return false if the specified record needs to be filtered, otherwise return true.

To create a new modifier, you need to extend the following BTE abstact class:

```
gr.ekt.bte.core.AbstractModifier
```

You will need to implement the following method:

```
public abstract Record modify ( Record record )
```

within you can make any changes you like in the record. You can use the Record methods to get the values for a specific key and load new ones (For the later, you need to make the Record mutable)

After you create your own filters or modifiers you need to add them in the Spring XML configuration file as in the following example:

```
<bean id="customfilter"   class="org.mypackage.MyFilter" />

<bean id="batchImportLinearWorkflow" class="gr.ekt.bte.core.LinearWorkflow">
    <property name="process">
    <list>
         <ref bean="customfilter" />
    </list>
    </property>
</bean>
```

You can add as many filters and modifiers you like to *batchImportLinearWorkflow* , they will run the one after the other in the specified order.

Using XMLUI

Introduction

This documentation explains the features and the usage of the importer framework.

Features

- Look up publications from remote sources.
- Support for multiple implementations.

Abstraction of input format

The importer framework does not enforce a specific input format. Each importer implementation defines which input format it expects from a remote source. The import framework uses generics to achieve this. Each importer implementation will have a type set of the record type it receives from the remote source's response. This type set will also be used by the framework to use the correct MetadataFieldMapping for a certain implementation. Read Im plementation of an import source for more information.

Transformation to DSpace item

The framework produces an 'ImportRecord' that is completely decoupled from DSPace. It contains a set of metadata DTO's that contain the notion of schema,element and qualifier. The specific implementation is responsible for populating this set. It is then very simple to create a DSPace item from this list.

Relation with BTE

While there is some overlap between this framework and BTE, this framework supports some features that are hard to implement using the BTE. It has explicit support to deal with network failure and throttling imposed by the data source. It also has explicit support for distinguishing between network caused errors and invalid requests to the source. Furthermore the framework doesn't impose any restrictions on the format in which the data is retrieved. It uses java generics to support different source record types. A reference implementation of using XML records is provided for which a set of metadata can be generated from any xpath expression (or composite of xpath expressions). Unless 'advanced' processing is necessary (e.g. lookup of authors in an LDAP directory) this metadata mapping can be simply configured using spring. No code changes necessary. A mixture of advanced and simple (xpath) mapping is also possible.

This design is also in line with the roadmap to create a Modular Framework as detailed in https://wiki.duraspace.org/display/DSPACE/Design+-+Module+Framework+and+Registry This modular design also allows it to be completely independent of the user interface layer, be it JSPUI, XMLUI, command line or the result of the new UI projects: https://wiki.duraspace.org/display/DSPACE/Design+-+Single+UI+Project

Implementation of an import source

Each importer implementation must at least implement interface *org.dspace.importer.external.service.other.Imports* and implement the inherited methods.

One can also choose to implement class *org.dspace.importer.external.service.other.Source* next to the Imports interface. This class contains functionality to handle request timeouts and to retry requests.

A third option is to implement class *org.dspace.importer.external.service.AbstractImportSourceService*. This class already implements both the Imports interface and Source class. AbstractImportSourceService has a generic type set 'RecordType'. In the importer implementation this type set should be the class of the records received from the remote source's response (e.g. when using axiom to get the records from the remote source's XML response, the importer implementation's type set is *org.apache.axiom.om.OMElement*).

Implementing the AbstractImportSourceService allows the importer implementation to use the framework's build-in support to transform a record received from the remote source to an object of class *org.dspace.importer.external.datamodel.ImportRecord* containing DSpace metadata fields, as explained here: Metadata mapping.

Inherited methods

Method getImportSource() should return a unique identifier. Importer implementations should not be called directly, but class *org.dspace.importer.external.service.ImportService* should be called instead. This class contains the same methods as the importer implementatons, but with an extra parameter 'url'. This url parameter should contain the same identifier that is returned by the getImportSource() method of the importer implementation you want to use.

The other inherited methods are used to query the remote source.

Metadata mapping

When using an implementation of AbstractImportSourceService, a mapping of remote record fields to DSpace metadata fields can be created. First create an implementation of class AbstractMetadataFieldMapping with the same type set used for the importer implementation. Then create a Spring configuration file in [dspace.dir]/config/spring/api. Each DSpace metadata field that will be used for the mapping must first be configured as a spring bean of class *org.dspace.importer.external.metadatamapping.MetadataFieldConfig*.

```
<bean id="dc.title" class="org.dspace.importer.external.metadatamapping.MetadataFieldConfig">
    <constructor-arg value="dc.title"/>
</bean>
```

Now this metadata field can be used to create a mapping. To add a mapping for the "dc.title" field declared above, a new spring bean configuration of a class class *org.dspace.importer.external.metadatamapping.contributor.MetadataContributor* needs to be added. This interface contains a type argument. The type needs to match the type used in the implementation of AbstractImportSourceService. The responsibility of each MetadataContributor implementation is to generate a set of metadata from the retrieved document. How it does that is completely opaque to the AbstractImportSourceService but it is assumed that only one entity (i.e. item) is fed to the metadatum contributor.

For example `java SimpleXpathMetadatumContributor implements MetadataContributor<OMElement>` can parse a fragment of xml and generate one or more metadata values.

This bean expects 2 property values:

- field: A reference to the configured spring bean of the DSpace metadata field. e.g. the "dc.title" bean declared above.
- query: The xpath expression used to select the record value returned by the remote source.

```
<bean id="titleContrib" class="org.dspace.importer.external.metadatamapping.contributor.
SimpleXpathMetadatumContributor">
    <property name="field" ref="dc.title"/>
    <property name="query" value="dc:title"/>
</bean>
```

Multiple record fields can also be combined into one value. To implement a combined mapping first create a *SimpleXpathMetadatumContributor* as explained above for each part of the field.

```
<bean id="lastNameContrib" class="org.dspace.importer.external.metadatamapping.contributor.
SimpleXpathMetadatumContributor">
    <property name="field" ref="dc.contributor.author"/>
    <property name="query" value="x:authors/x:author/x:surname"/>
</bean>
<bean id="firstNameContrib" class="org.dspace.importer.external.metadatamapping.contributor.
SimpleXpathMetadatumContributor">
    <property name="field" ref="dc.contributor.author"/>
    <property name="query" value="x:authors/x:author/x:given-name"/>
</bean>
```

Note that namespace prefixes used in the xpath queries are configured in bean "FullprefixMapping" in the same spring file.

```
<util:map id="FullprefixMapping" key-type="java.lang.String" value-type="java.lang.String">
    <description>Defines the namespace mappin for the SimpleXpathMetadatum contributors</description>
    <entry key="http://purl.org/dc/elements/1.1/" value="dc"/>
    <entry key="http://www.w3.org/2005/Atom" value="x"/>
</util:map>
```

Then create a new list in the spring configuration containing references to all *SimpleXpathMetadatumContributor* beans that need to be combined.

```
<util:list id="combinedauthorList" value-type="org.dspace.importer.external.metadatamapping.contributor.
MetadataContributor" list-class="java.util.LinkedList">
    <ref bean="lastNameContrib"/>
    <ref bean="firstNameContrib"/>
</util:list>
```

Finally create a Spring bean configuration of class *org.dspace.importer.external.metadatamapping.contributor.CombinedMetadatumContributor*. This bean expects 3 values:

- field: A reference to the configured spring bean of the DSpace metadata field. e.g. the "dc.title" bean declared above.
- metadatumContributors: A reference to the list containing all the single record field mappings that need to be combined.
- separator: These characters will be added between each record field value when they are combined into one field.

```
<bean id="authorContrib" class="org.dspace.importer.external.metadatamapping.contributor.
CombinedMetadatumContributor">
    <property name="separator" value=", "/>
    <property name="metadatumContributors" ref="combinedauthorList"/>
    <property name="field" ref="dc.contributor.author"/>
</bean>
```

Each contributor must also be added to the "MetadataFieldMap" used by the *MetadataFieldMapping* implementation. Each entry of this map maps a metadata field bean to a contributor. For the contributors created above this results in the following configuration:

```
<util:map id="org.dspace.importer.external.metadatamapping.MetadataFieldConfig"
                  value-type="org.dspace.importer.external.metadatamapping.contributor.MetadataContributor">
        <entry key-ref="dc.title" value-ref="titleContrib"/>
        <entry key-ref="dc.contributor.author" value-ref="authorContrib"/>
</util:map>
```

Note that the single field mappings used for the combined author mapping are not added to this list.