AtlasMap User Guide

The AtlasMap Team

Version 2.4.0-SNAPSHOT, 2022-04-04

AtlasMap User Guide

[badge]  [atlasmap]

AtlasMap is a data mapping solution with an interactive web based user interface. It simplifies configuration of an integration that handles different types of data including:

- XML schema or instance files
- JSON schema or instance files
- Java class files

To use AtlasMap, you start from AtlasMap UI and import your data shape definition files, map source fields to target fields, optionally apply transformations or conditions, and then execute defined mappings by means of the AtlasMap runtime engine. AtlasMap provides standard transformations and supports the creation of custom, user-defined transformations.

AtlasMap UI is available:

- As a standalone application
- Embedded in Syndesis, which is an open-source integration platform

To execute the mappings created through AtlasMap UI, there is a `camel-atlasmap` component that consumes an AtlasMap mapping definition and processes data mappings as a part of a Camel route https://camel.apache.org/components/latest/atlasmap-component.html

Also, There is a Camel Quarkus extension for AtlasMap https://camel.apache.org/camel-quarkus/latest/reference/extensions/atlasmap.html

This user guide is organized as follows:

Chapter 1. Quickly get started using AtlasMap UI standalone

To run AtlasMap UI standalone:

1. Obtain the most recent AtlasMap version number:
   a. Go to https://github.com/atlasmap/atlasmap/
   b. Scroll down to see the `maven central` icon with the latest version, for example: [maven central 1.42.3].
2. Download the AtlasMap standalone `.jar` file by invoking the `wget` command with the following format. Replace `$2.4.0-SNAPSHOT` with an actual version number.

```
wget https://repo1.maven.org/maven2/io/atlasmap/atlasmap-
standalone/${VERSION}/atlasmap-standalone-${VERSION}.jar
```

3. Run AtlasMap standalone by invoking the `java` command with the following format:

```
$ java -jar atlasmap-standalone-${VERSION}.jar
```

   For example:

```
$ java -jar atlasmap-standalone-1.42.3.jar
```

Although it is not a common situation, you can start AtlasMap and also load mapping definitions if you have an AtlasMap `.adm` file. An AtlasMap `.adm` file contains mapping definitions that were exported from AtlasMap. For example, suppose you want to start AtlasMap and import the `/home/aslan/Downloads/atlas-mapping.adm` file. Invoke a command such as the following:

```
$ java -Datlasmap.adm.path=/home/aslan/Downloads/atlas-mapping.adm
-jar atlasmap-standalone-1.42.3.jar
```

4. In a browser, visit http://127.0.0.1:8585/ to see the AtlasMap data mapper canvas.
5. Experiment with the data mapper:
   a. At the top of the **Source** panel, click [Import] to import a JSON or XML file that you can map fields from.
   b. At the top of the **Target** panel, click [Import] to import a different JSON or XML file that you can map fields to.
   c. To map fields:

i. Navigate to a source field and click [Create new napping].

   The **Mapping Details** panel will slide in from the right.

   [Mapping Details]

ii. Either:

   A. Navigate to a target field and click [Connect to the selected mapping] to add it to the selected mapping, or

   B. Drag and drop a target field on the source field to add the target field to the mapping.

d. With a data mapping selected, explore the optional transformations.

e. To preview a data mapping result, click [Show Mapping Preview] and enter sample data in source **Mapping preview** fields.

f. To save your work, click the **AtlasMap** menu and select **Export the current mappings and support files into a catalog (.adm)**

   [AtlasMap menu].

   Then choose a folder in which to save an AtlasMap `.adm` file. This file contains the mapping definitions.

Chapter 2. Overview of AtlasMap

The following topics provide a high-level overview of AtlasMap:

- General procedure for using AtlasMap standalone
- About AtlasMap and Syndesis
- About types of fields in AtlasMap
- About types of mappings in AtlasMap
- About transforming a field in AtlasMap
- Alternatives for supplying source values that are missing
- Descriptions of AtlasMap icons

2.1. General procedure for using AtlasMap standalone

To use AtlasMap standalone, the main steps are:

1. Download the AtlasMap standalone `.jar` file.
2. Start running AtlasMap standalone.
3. In the AtlasMap data mapping canvas:
    a. Import a source data file.
    b. Import a target data file.
    c. Map fields as needed, optionally applying transformations and/or conditions.
    d. Export the defined mappings to a `.adm` AtlasMap catalog file.

What can you do with an AtlasMap `.adm` file?

- Continue working on the mappings by importing the `.adm` file into AtlasMap.
- Execute the mappings in a Camel application that uses the `camel-atlasmap` component, which consumes `.adm` files.

2.2. About AtlasMap and Syndesis

The Syndesis open-source integration platform embeds AtlasMap. In a Syndesis integration, you add connections to applications that provide the data of interest to you. With connections in place, Syndesis offers an AtlasMap data mapper step. When you add a data mapper step, Syndesis displays the data mapper canvas with:

- Source data fields that are provided by previous connections
- Target data fields that are required by the subsequent connection

In other words, you do not import source and target data as you would in standalone AtlasMap. Likewise, when data mappings are complete, instead of exporting the mappings, you just click a

button to indicate that you are done.

When Syndesis embeds AtlasMap, it does not display the icons for importing and exporting data.

2.3. About types of fields in AtlasMap

The AtlasMap user interface displays source fields and target fields and you define the field-to-field mappings that you need.

At this time, AtlasMap does not provide whole document mapping. It is expected that a future release will support bulk mapping by detecting the same object structure in source and target files (issue #86).

An understanding of the different kinds of AtlasMap fields makes it easier for you to define the mappings you need. For the purposes of mapping, there are three field types:

- **Terminal** — A terminal field is selectable. It can be the source or target in a mapping. During execution, a path to a terminal field identifies a value.
- **Parent** — A parent field is also referred to as a complex field. A parent field is expandable. Expanding a parent field displays its descendant fields. A parent field it not selectable and cannot be directly in a mapping.
- **Collection** — A collection field is also a terminal field. It is selectable and can be the source or target in a mapping. A collection field indicates a field that holds one or more objects that are all the same type. Internally, a collection field might be an array.

For example, consider this sample XML instance document:

```
<order>
  <orderId value="0123"/>
  <items>
    <item>
      <itemId>Orange</itemId>
      <quantity value="1"/>
    </item>
    <item>
      <itemId>Apple</itemId>
      <quantity value="2"/>
    </item>
  </items>
</order>
```

Terminal fields:	`/order/orderId/@value` `/order/items/item[0]/quantity/@value`

Parent fields:	`/order`
	`/order/orderId`
	`/order/items/item[0]`

Collection fields:	`/order/items/item[]/itemId` `/order/items/item[]/quantity/@value`

When you define a mapping that uses this XML instance document as the source data, you cannot know how many items will be in an order. However, the same mapping definition works for each transaction, regardless of the number of items in the order.

2.4. About types of mappings in AtlasMap

AtlasMap supports the following general types of mappings:

- **One to one** — Map one source field to one target field.
- **Many to one** — Map multiple source fields to to one target field. You specify the delimiter character that AtlasMap inserts in the target field between the mapped source fields. The default delimiter is a space.
- **One to many** — Map one source field to multiple target fields. You specify the delimiter character that is in the source field. AtlasMap maps each delimited value to a the target field you select.
- **For each** — Iteratively map one source collection field to one target collection field.

2.5. About transforming a field in AtlasMap

In a data mapping, you can apply a transformation to one or more fields. A transformation is a function that AtlasMap performs on the selected field. For example, suppose you select a source field and specify the `Uppercase` transformation. At runtime, AtlasMap converts the source field value to uppercase and inserts the uppercase value in the target field.

To apply a transformation, select a source or target terminal field, click **Add Transformation** in the **Mapping Details** panel on the right, and select the transformation.

You can apply different transformations to different fields in the same mapping.

In a one-to-one mapping, which maps one source field to one target field, it does not matter whether you apply the transformation to the source field or the target field. In a one-to-many or many-to-one mapping, consider what the target field value needs to be when you specify a transformation. For example, consider a many-to-one mapping that combines source fields for number, street, city, and state into one target address field. If you want the strings in the target address field to all be uppercase, select the target address field and apply the uppercase transformation. If only the state needs to be uppercase, select the source state field, and apply the uppercase transformation.

You can think of a source field transformation as performing pre-processing, while a target field transformation performs post-processing.

2.6. Alternatives for supplying source values that are missing

When you are mapping fields, you might find that a source data shape does not provide a value that a target data shape requires. In these situations, AtlasMap provides two alternatives for specifying source values that you can map to target fields:

- **Define a property** — In AtlasMap, at the top of the **Source** panel, to the right of **Properties**, click the + sign to display a dialog. Enter the property name, the property's default value, and indicate the data type. Note that at runtime, if there is an environment variable or a Java system property with the same name as the property, it is possible for this default value to be overridden. Click **Save** to create a new terminal source field.
- **Define a constant** — In AtlasMap, near the top of the **Source** panel, to the right of **Constants**, click the + sign to display a dialog. Enter the value of the constant, and indicate the data type. Click **Save** to create a new terminal source field.

For example, suppose that a target data shape defines a **Layout** field whose value must be `HORIZONTAL` or `VERTICAL`. The source data shape does not provide this field. You can create a constant. Specify `HORIZONTAL` or `VERTICAL` as the value and accept `String` as the data type. After you save the new constant, you can map it to the **Layout** target field.

2.7. Descriptions of AtlasMap icons

The following table describes some of the AtlasMap icons:

Icon	Description
[Circle icon]	Completely mapped field
[Collection icon]	Collection field
[Folder icon]	Complex type that you can expand
[Lightning bolt iicon]	There is a transformation that is being applied to this field.

There are also several toolbar buttons that allow you to choose what should be displayed, such as data types, sample data previews, and mapped or unmapped fields.

Chapter 3. Importing data files into AtlasMap

Data mapping allows you to match fields in a source data shape to fields in a target data shape. The data shapes that AtlasMap can operate on are defined in JSON schema or instance documents, XML schema or instance documents, or Java classes.

To map fields, you import data shape definition files into the AtlasMap **Source** and **Target** panels. You can import multiple files into each panel. For example, suppose you import three source files and one target file. You can map fields from all three source files to the single target file.

The following topics provide examples of importing different data shape definitions:

- Importing JSON files into AtlasMap
- Importing XML files into AtlasMap
- Importing Java archive files into AtlasMap

3.1. Importing JSON files into AtlasMap

You can import a JSON schema file or a JSON instance file into AtlasMap. The procedure for importing each kind of file is the same. For example, the following JSON schema file, `JSONSchema.json`, defines properties for an `Order` object:

```
{
    "$schema": "http://json-schema.org/schema#",
    "description": "Order",
    "type": "object",
    "properties": {
      "order": {
        "type": "object",
        "properties": {
          "address": {
          "type": "object",
          "properties": {
            "street": { "type": "string" },
            "city": { "type": "string" },
            "state": { "type": "string" },
            "zip": { "type": "string" }
          }
        },
        "contact": {
          "type": "object",
          "properties": {
            "firstName": { "type": "string" },
            "lastName": { "type": "string" },
            "phone": { "type": "string" }
          }
```

```
        },
        "orderId": { "type": "string" }
      }
    },
    "primitives": {
      "type": "object",
      "properties": {
        "stringPrimitive": { "type": "string" },
        "booleanPrimitive": { "type": "boolean" },
        "numberPrimitive": { "type": "number" }
      }
    },
    "primitiveArrays": {
      "type": "object",
      "properties": {
        "stringArray": {
          "type": "array",
          "items": { "type": "string" }
        },
        "booleanArray": {
          "type": "array",
          "items": { "type": "boolean" }
        },
        "numberArray": {
          "type": "array",
          "items": { "type": "number" }
        }
      }
    },
    "addressList": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "street": { "type": "string" },
          "city": { "type": "string" },
          "state": { "type": "string" },
          "zip": { "type": "string" }
        }
      }
    }
    }
}
```

To import this file into the AtlasMap Source panel:

1. At the top of the **Source** panel, click [Import].
2. In the **Open File** dialog, navigate to the `JSONSchema.json` file and select it.
3. Click **Open**.

AtlasMap displays the fields in the **Source** panel:

[Order document].

3.2. Importing XML files into AtlasMap

You can import an XML schema file or an XML instance file into AtlasMap. The procedure for importing each kind of file is the same. For example, the following XML schema file, `XMLSchema.xml`, defines a schema for transactions related to pets:

```
<d:SchemaSet xmlns:d="http://atlasmap.io/xml/schemaset/v2"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:schema targetNamespace="http://syndesis.io/v1/swagger-connector-
template/request" elementFormDefault="qualified">
      <xsd:element name="request">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="body">
              <xsd:complexType>
                <xsd:sequence>
                  <xsd:element ref="Pet" />
                </xsd:sequence>
              </xsd:complexType>
            </xsd:element>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:schema>
    <d:AdditionalSchemas>
      <xsd:schema>
        <xsd:element name="Pet">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="id" type="xsd:decimal" />
              <xsd:element name="Category">
                <xsd:complexType>
                  <xsd:sequence>
                    <xsd:element name="id" type="xsd:decimal" />
                    <xsd:element name="name" type="xsd:string" />
                  </xsd:sequence>
                </xsd:complexType>
              </xsd:element>
              <xsd:element name="name" type="xsd:string" />
              <xsd:element name="photoUrl">
                <xsd:complexType>
                  <xsd:sequence>
                    <xsd:element name="photoUrl" type="xsd:string"
maxOccurs="unbounded" minOccurs="0" />
                  </xsd:sequence>
                </xsd:complexType>
```

```
            </xsd:element>
            <xsd:element name="tag">
              <xsd:complexType>
                <xsd:sequence>
                  <xsd:element name="Tag" maxOccurs="unbounded" minOccurs="0">
                    <xsd:complexType>
                      <xsd:sequence>
                        <xsd:element name="id" type="xsd:decimal" />
                        <xsd:element name="name" type="xsd:string" />
                      </xsd:sequence>
                    </xsd:complexType>
                  </xsd:element>
                </xsd:sequence>
              </xsd:complexType>
            </xsd:element>
            <xsd:element name="status" type="xsd:string" />
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:schema>
  </d:AdditionalSchemas>
</d:SchemaSet>
```

To import this file into the AtlasMap **Target** panel:

1. At the top of the **Target** panel, click [Import].
2. In the **Open File** dialog, navigate to the `XMLSchema.xml` file and select it.
3. Click **Open**.

AtlasMap displays the fields in the **Target** panel. In the following image, the fields are expanded to show more detail.

[Imported schema for pet transactions]

3.3. Importing Java archive files into AtlasMap

AtlasMap can import a Java archive file and display mappable fields in either the source or target panel. In the data mapper, each class-wide public entity in a class file that is in the imported JAR file becomes a mappable field.

Arrays and data types are more discretely defined.

Consider the following two files:

```
Bicycle.java:

package io.paul;
import io.paul.GeoLocation;

public class Bicycle {
    public int cadence;
    public int gear;
    public int speed;
    public float[] seatHeight;
    public String[] color;
    public GeoLocation geoLocation;
}
```

```
GeoLocation.java:
package io.paul;

public class GeoLocation {
    double lattitude;
    double longitude;
}
```

To import these files:

1. Compile the Java files and assemble the results into a Java archive file. For example, invoke the following commands to create the `Bicycle.jar` file:

   ```
   javac -cp io.paul:. -d . GeoLocation.java Bicycle.java
   jar cvf ../Bicycle.jar *
   ```

2. In the main toolbar, click the **AtlasMap** menu and select **Import a catalog (.adm)**. [Import].
3. In the **Open** dialog, navigate to the `Bicycle.jar` file and click **Open**.
4. At the top of the **Source** panel, click [+] to import the fields into the source panel.

   To display imported fields in the **Target** panel, you would click [+] at the top of the **Target** panel.

5. In the **Load Java Document From Custom Class** dialog, in the **Custom class name** field, select the class package name of the Java class. For example, in `Bicycle.jar` file, the class package name is `io.paul.Bicycle`.
6. Click **OK**. If you import the sample `Bicycle.jar` file into the the **Source** panel, you would now see something like the following. The fields that are in the imported Java class files appear in the panel into which they were imported.

[Bicycle jar file]

Chapter 4. Mapping fields in AtlasMap

The following topics provide details for mapping fields in AtlasMap:

4.1. Finding the data field that you want to map

In a relatively simple integration, mapping data fields is easy and intuitive. In more complex integrations or in integrations that handle large sets of data fields, mapping from source to target is easier when you have some background about how to use the data mapper.

The data mapper displays two columns of data fields:

- **Source** is a list of the data fields that you can map to target fields.
- **Target** is a list of the data fields that you can map source fields to.

To quickly find the data field that you want to map, you can do any of the following:

- Search for it.

  The **Source** panel and the **Target** panel each have a search field at the top. Enter the name of the field you want to map.

- Enter the names of the fields that you want to map in the **Mapping Details** panel.

  Click [Add icon] in the toolbar to add a new empty mapping. The **Mapping Details** panel will slide in from the right. In the **Sources** section, enter the name of the source field. In the **Targets** section, enter the name of the field that you want to map to.

- Expand and collapse folders to limit the visible fields.

  Once the appropriate field is visible, you can drag it to a currently mapped field in the opposite panel to add it to that mapping.

4.2. Mapping one source field to one target field

The default mapping behavior maps one source field to one target field. For example, map the `Name` field to the `CustomerName` field.

Procedure

1. In the **Source** panel, navigate to the data field that you want to map from.

   You might need to expand a folder to see the data fields that it provides.

   When there are many source fields, you can search for the field of interest by entering the field's name in the search field at the top of the **Source** panel.

2. Click [Create new mapping] to add the field as a source of a new mapping.
3. In the **Target** panel, navigate to the data field that you want to map to.
4. Click [Connect to selected mapping] to add the field as a target in the selected mapping.
5. Optionally, preview the data mapping result. This is useful when you add a transformation to the mapping or when the mapping requires a type conversion.
   a. In the toolbar, click [Show mapping preview] to display a text input field on the source field and a read-only result field on the target field.
   b. In the source field's data input field, enter text. Click somewhere outside this text box to display the mapping result in the read-only field on the target field.
   c. Optionally, to see the result of a transformation, add a transformation in the **Mapping Details** panel.
   d. Hide the preview fields by again clicking [Show mapping preview].
6. Optionally, to confirm that the mapping is defined, in the upper right, click [Grid] to display the defined mappings in a table.

   You can also preview data mapping results in this view. If preview fields are not visible, click [Show mapping preview] and enter data as described in the previous step. In the table of defined mappings, preview fields appear for only the selected mapping. To see preview fields for another mapping, select it.

   Click [Column icon] to display the data field panels again.

4.3. Example of missing or unwanted data when combining or separating fields

In a data mapping, you might need to identify missing or unwanted data when a source or target field contains compound data. For example, consider a `long_address` field that has this format:

number street apartment city state zip zip+4 country

Suppose that you want to separate the `long_address` field into discrete fields for `number`, `street`, `city`, `state`, and `zip`. To do this, you select `long_address` as the source field and then select the target fields.

You then add padding fields at the locations for the parts of the source field that you do not want. In this example, the unwanted parts are *apartment*, *zip+4*, and *country*.

To identify the unwanted parts, you need to know the order of the parts. The order indicates an index for each part of the content in the compound field. For example, the `long_address` field has 8 ordered parts. Starting at 1, the index of each part is:

1	*number*
2	*street*
3	*apartment*
4	*city*
5	*state*
6	*zip*
7	*zip+4*
8	*country*

In the data mapper, to identify *apartment*, *zip+4*, and *country* as missing, you add padding fields at indexes 3, 7, and 8. See Separating one source field into multiple target fields.

Now suppose that you want to combine source fields for `number`, `street`, `city`, `state`, and `zip` into a `long_address` target field. Further suppose that there are no source fields to provide content for *apartment*, *zip+4*, and *country*. In the data mapper, you need to identify these fields as missing. Again, you add padding fields at indexes 3, 7, and 8. See Combining multiple source fields into one target field.

4.4. Combining multiple source fields into one target field

In a data mapping, you can combine multiple source fields into one compound target field. For example, you can map the `FirstName` and `LastName` fields to the `CustomerName` field.

Prerequisite

For the target field, you must know what type of content is in each part of this compound field, the order and index of each part of the content, and the separator between parts, such as a space or comma. See Example of missing or unwanted data when combining or separating fields.

Procedure

1. In the **Source** panel, navigate to the first field that you want to combine into the target field and click [Create new mapping]. For each of the other fields that you want to combine into the target field, click [Connect to selected mapping] next to each field.

   The data mapper automatically changes the type of mapping to **Concatenate**.

2. In the **Target** panel, click [Connect to selected mapping] next to the field into which you want to map all of the source fields.

When you are done you should see a line from each of the source fields to the target field.

3. In the **Mapping Details** panel, in the **Delimiter** field, select the character that the data mapper inserts in the target field between the content from different source fields. The default is a space.
4. In the **Mapping Details** panel, under **Sources**, ensure that the source fields are in the order in which they are expected to appear in the target field.

   If necessary, increase or decrease the index number of a source field to achieve the desired order.

5. Optionally, preview the data mapping result:
   a. In the toolbar, click [Show mapping preview] and select **Show Mapping Preview** to display a text input field on each source field for the currently selected mapping and a read-only result field on the target field of the currently selected mapping.
   b. In the source data input fields, enter text. Click outside the text box to display the mapping result in the read-only field on the target field.

      If you reorder the source fields or add a transformation to the mapping then the result field on the target field reflects this. If the data mapper detects any errors, it displays informative messages at the top of the **Mapping Details** panel.

   c. Hide the preview fields by clicking [Show mapping preview] again.

      If you redisplay the preview fields, any data that you entered in them is still there and it remains there until you exit the data mapper.

6. To confirm that the mapping is correctly defined, in the upper right, click [Grid icon] to display the defined mappings. A mapping that combines the values of more than one source field into one target field looks like this: [Combine mapping].

   You can also preview mapping results in this view. Click [Show mapping preview] and enter text as described in the previous step. Preview fields appear for only the selected mapping. Click another mapping in the table to view preview fields for it.

4.5. Separating one source field into multiple target fields

In a data mapping, you can separate a compound source field into multiple target fields. For example, map the `Name` field to the `FirstName` and `LastName` fields.

Prerequisite

For the source field, you must know what type of content is in each part of this compound field, the order and index of each part of the content, and the separator between parts, such as a space or comma. See Example of missing or unwanted data when combining or separating fields.

Procedure

1. In the **Source** panel, navigate to the field that contains the content you want to separate and click [Create new mapping].
2. In the **Target** panel, for each field into which you want to map part of the source field's content, navigate to the field and click [Connect to selected mapping].

   The data mapper automatically changes the mapping type to **Split**.

   When you are done selecting target fields, you should see lines from the source field to each of the target fields.
3. In the **Mapping Details** panel, in the **Delimiter** field, select the character in the source field that indicates where to separate the source field values. The default is a space.
4. In the **Mapping Details** panel, under **Targets**, ensure that the target fields are in the same order as the corresponding content in the compound source field.

   If necessary, increase or decrease the indexes next to a target field to achieve the desired order.
5. Optionally, preview the data mapping result:
   a. In the toolbar, click [Show mapping preview] to display a text input field on the source field and read-only result fields on each target field.
   b. In the source field's data input field, enter text. Be sure to enter the separator character between the parts of the field. Click outside the text box to display the mapping result in the read-only fields on the target fields.

      If you reorder the target fields or add a transformation to a target field then the result fields on the target fields reflect this. If the data mapper detects any errors, it displays informative messages at the top of the **Mapping Details** panel.

   c. Hide the preview fields by clicking [Show mapping preview] again.

      If you redisplay the preview fields, any data that you entered in them is still there and it remains there until you exit the data mapper.
6. To confirm that the mapping is correctly defined, click [Grid icon] to display defined mappings. A mapping that separates the value of a source field into multiple target fields looks like this: [Separate mapping].

   You can also preview mapping results in this view. Click [Show mapping preview] and enter text as described in the previous step. Preview fields appear for only the selected mapping. Click another mapping in the table to view preview fields for it.

4.6. Mapping between collections and non-collections

In the data mapper **Source** and **Target** panels:

- [Collection icon] indicates a collection. If the collection contains one primitive type, you can map directly from or to that collection. If the collection contains two or more different types, the data mapper displays the collection's child fields and you can map to or from the collection's

fields.

- [Folder icon] indicates an expandable container that is a complex type. A complex type contains multiple fields of different types. A field in a complex type can be a type that is a collection, such as an array. You cannot map a complex type container itself. You can map only the fields that are in the complex type.

To toggle the display of data types, such as `(COMPLEX)`, `STRING`, `INTEGER`, click [Show types icon] in the toolbar.

The following table shows the default behavior when mapping between collection fields and non-collection fields.

When you map from this source	To this target	During execution
A collection. (No child fields appear in the data mapper.)	A field that is not in a collection.	The data mapper maps the value that is in the last element in the source collection to the target field.
A field that is in a collection.	A field that is not in a collection.	The data mapper maps the mapped field's value that is in the last element in the source collection to the target field.
A field that is not in a collection.	A collection. (No child fields appear in the data mapper.)	The data mapper maps the value that is in the mapped source field to the first (and only) element in the collection.
A field that is not in a collection.	A field that is in a collection.	The data mapper maps the value that is in the mapped source field to the first (and only) element in the collection.

Changing default behavior when mapping from a collection field

When you map from a collection field to a non-collection field, the default behavior is that the target field gets its value from the last element in the source collection. You can change this default behavior in the following ways:

- To map from the element that you choose, apply the **Item At** transformation to the source and specify an index. For example, to map the value that is in the first element that is in the collection, specify `0` for the index.
- To map all values that are in all elements that are in a source collection, apply the **Concatenate** transformation to the source collection or source collection field and optionally specify a delimiter. The default delimiter is a space. For example, consider this source collection:
  - In the first element, the value in the **city** field is **Boston**.
  - In the second element, the value in the **city** field is **Paris**.

- In the third element, the value in the **city** field is **Tokyo**.

During execution, the data mapper populates the target field with `Boston Paris Tokyo`.

Changing default behavior when mapping from a non-collection field

When you map from a non-collection field to a collection field, the default behavior is that the target collection contains one element, which contains the non-collection, source field value. You can change the default behavior when the source field contains a series of values that are separated by the same delimiter. For example, consider a non-collection, source **cities** field that contains:

```
Boston Paris Tokyo
```

You would map this to a target collection or to a target field that is in a collection. On the source **cities** field, add the **Split** transformation. During execution, the data mapper splits the value of the **cities** field at the space delimiter. The result is a collection that contains three elements. In the first element, the value of the **city** field is `Boston`. In the second element, the value of the **city** field is `Paris`. In the third element, the value of the **city** field is `Tokyo`.

4.7. Transforming source or target data

In the data mapper, after you define a mapping, you can transform any field in the mapping. Transforming a data field defines how you want to store the data. For example, you could specify the **Capitalize** transformation to ensure that the first letter of a data value is uppercase.

Procedure

1. Map the fields. This can be a one-to-one mapping, a combination mapping, or a separation mapping.
2. In the **Mapping Details** panel, under **Sources** or under **Targets**, in the box for the field that you want to transform, click [Lightning bolt icon] to the left of the trash can icon.

   [Mapping details transformation button]

   This displays a pull-down where you can select the transformation that you want the data mapper to perform. The set of available transformations is type specific.

   [Mapping details transformation button - available transformations]
3. Click the transformation that you want to perform.
4. Optional. If the transformation requires any input parameters, specify them in the appropriate input fields.
5. Optional. To add another transformation, click the arrow to the left of the trash can icon again.

Additional resource

Descriptions of available transformations

4.8. Applying conditions to mappings

In some integrations, it is helpful to add conditional processing to a mapping. For example, suppose

that you are mapping a source zip code field to a target zip code field. If the source zip code field is empty, you might want to fill the target field with `99999`. To do this, you would specify an expression that tests the zip code source field to determine if it is empty, and if it is empty, inserts `99999` into the zip code target field.

The data mapper supports expressions that are similar to a Microsoft Excel expressions, but does not support all Microsoft Excel expression syntax.

You can define zero or one condition for each mapping.

The following procedure gets you started with applying conditions to mappings. As you work with mappings and conditions, you can perform the required steps in the order that is most convenient for you.

Prerequisites

- You are mapping fields in a the data mapper UI.
- You are familiar with Microsoft Excel expressions or you have the conditional expression that you want to apply to a mapping.

Procedure

1. If data types are not already visible, display them by clicking [Show types icon].

   While this is not a requirement for specifying a condition, it is helpful to see the data types.

2. Create the mapping that you want to apply a condition to, or ensure that the currently selected mapping is the mapping that you want to apply a condition to. For example, consider this mapping:

   [lastName and firstName map to customerName]

3. In the upper right, click [Add expression] to display the conditional expression input field.

   In the expression field, the data mapper automatically displays the names of the source fields in the current mapping. For example:

   [lastName + firstName]

   In the expression input field, the order of the source fields is the order in which you selected them when you created the mapping. This is important because the default mapping behavior is that the data mapper concatenates the field values in this order to insert the result in the target field. In this example, to create this mapping, `lastName` was selected first and then `firstName` was selected.

4. Edit the expression input field to specify the conditional expression that you want the data mapper to apply to the mapping. Details about supported conditional expressions follow this procedure.

   As you specify the expression, you can enter `@` and start to enter the name of a field. The data mapper displays a list of the fields that match what you entered. Select the field that you want to specify in the expression.

When you add a field name to the expression, the data mapper adds that field to the mapping. For example, consider this conditional expression:

[if(ISEMPTY(lastName)]

During execution, if the data mapper determines that the `lastName` field is empty, it maps only the `firstName` field to the target `customerName` field. If the `lastName` field contains a value, that is, it is not empty, the data mapper concatenates the values in the source `orderId` and `phone` fields, and inserts the result in the `customerName` field. (This example shows how the logic works, but it is probably not a useful example because when there is a value in the `lastName` field, you most likely want the data mapper to simply perform the mapping and not map some other value into the target.)

+ For this example, after you complete entering the expression, the data mapping is:

+ [lastName firstName orderId phone are mapped to customerName]

+ In the conditional expression, if you remove a field name that is in the mapping that the expression applies to, the data mapper removes that field from the mapping. In other words, every field name in the mapping must be in the conditional expression.

1. If mapping preview fields are not already visible, display them by clicking [Show mapping preview].
2. Enter sample data in the source preview input field(s) to ensure that the target field or target fields get(s) the correct value.
3. Optionally, apply transformations to one or more source or target fields in the selected mapping by clicking [Lightning bolt icon] next to each field to which you want to apply a transformation and then selecting the desired transformation from the pull-down menu.

   For example, in the same mapping presented in this procedure, in the **Mapping Details** panel, you could apply the `Uppercase` transformation to the `firstName` field. You can test this by entering data in the `firstName` field's preview input field.

4. Edit the conditional expression as needed to obtain the desired result.

Supported functions in conditional expressions

- `ISEMPTY(source-field-name1 [+ source-field-name2])`

  The result of the `ISEMPTY()` function is a Boolean value. Specify at least one argument, which is the name of a source field in the mapping that you want to apply the condition to. When the specified source field is empty, the `ISEMPTY()` function returns true.

  Optionally, add the + (concatenation) operator with an additional field, for example:

  ```
  ISEMPTY(lastName + firstName)
  ```

  This expression evaluates to true if both source fields, `lastName` and `firstName`, are empty.

  Often, the `ISEMPTY()` function is the first argument in an `IF()` function.

- `IF(boolean-expression, then, else)`

When `boolean-expression` evaluates to true, the data mapper returns `then`. When `boolean-expression` evaluates to false, the data mapper returns `else`. All three arguments are required. The last argument can be null, which means that nothing is mapped when `boolean-expression` evaluates to false.

For example, consider the mapping that combines the `lastName` and `firstName` source fields in the target `customerName` field. You can specify this conditional expression:

```
IF (ISEMPTY(lastName), firstName, lastName + ',' + firstName )
```

During execution, the data mapper evaluates the `lastName` field.

- If the `lastName` field is empty, that is, `ISEMPTY(lastName)` returns true, the data mapper inserts only the `firstName` value into the target `customerName` field.
- If the `lastName` field contains a value, that is, `ISEMPTY(lastName)` returns false, the data mapper maps the `lastName` value, followed by a comma, followed by the `firstName` value into the target `customerName` field.

  Now consider the behavior if the third argument in this expression is null:

  ```
  IF (ISEMPTY(lastName), firstName, null )
  ```

  During execution, the data mapper evaluates the `lastName` field.

- As in the previous example, if the `lastName` field is empty, that is, `ISEMPTY(lastName)` returns true, the data mapper inserts only the `firstName` value into the target `customerName` field.
- However, when the third argument is null, if the `lastName` field contains a value, that is, `ISEMPTY(lastName)` returns false, the data mapper does not map anything into the target `customerName` field.

Table 1. Supported operators in conditional expressions

Operator	Description
`+`	Add numeric values or concatenate string values.
`-`	Subtract a numeric value from another numeric value.
`*`	Multiply numeric values.
`\`	Divide numeric values.
`&&` And	Return true if both the left and right operands are true. Each operand must return a Boolean value.
`||` Or	Return true if the left operand is true, or if the right operand is true, or if both operands are true. Each operand must return a Boolean value.
`!`	Not
`>` Greater than	Return true if the left numeric operand is greater than the right numeric operand.

`<` Less than	Return true if the left numeric operand is less than the right numeric operand.
`==` Equal	Return true if the left operand and the right operand are the same.

4.9. Viewing mappings

While you are using the data mapper UI, you can view the mappings that are already defined. This lets you check whether the correct mappings are in place.

Prerequisites

The data mapper canvas is visible.

Procedure

1. In the upper right, click [Grid icon] to display a list of the defined mappings.
2. To dismiss the list of mappings and redisplay the source and target fields, click [Column icon] again.

4.10. Descriptions of available transformations

TODO Generate this list automatically from annotation - https://github.com/atlasmap/atlasmap/issues/173

The following table describes the available transformations. The date and number types refer generically to any of the various forms of these concepts. That is, number includes, for example, `integer`, `long`, `double`. Date includes, for example, `date`, `Time`, `ZonedDateTime`.

Transformation	Input Type	Output Type	Parameter (* = required)	Description
`AbsoluteValue`	number	number	None	Return the absolute value of a number.
`AddDays`	date	date	`days`	Add days to a date. The default is 0 days.
`AddSeconds`	date	date	`seconds`	Add seconds to a date. The default is 0 seconds.
`Append`	string	string	string	Append a string to the end of a string. The default is to append nothing.

Transformation	Input Type	Output Type	Parameter (* = required)	Description
`Camelize`	string	string	None	Convert a phrase to a camelized string by removing whitespace, making the first word lowercase, and capitalizing the first letter of each subsequent word.
`Capitalize`	string	string	None	Capitalize the first character in a string.
`Ceiling`	number	number	None	Return the whole number ceiling of a number.
`Contains`	any	Boolean	`value`	Return true if a field contains the specified value.
`ConvertAreaUnit`	number	number	`fromUnit`* `toUnit` *	Convert a number that represents an area to another unit. For the `fromUnit` and `toUnit` parameters, select the appropriate unit from the **From Unit** and **To Unit** menus. The choices are: `Square Foot`, `Square Meter`, or `Square Mile`.

Transformation	Input Type	Output Type	Parameter (* = required)	Description
`ConvertDistanceUnit`	number	number	`fromUnit` * `toUnit` *	Convert a number that represents a distance to another unit. For the `fromUnit` and `toUnit` parameters, select the appropriate unit from the **From Unit** and **To Unit** menus. The choices are: `Foot`, `Inch`, `Meter`, `Mile`, or `Yard`.
`ConvertMassUnit`	number	number	`fromUnit` * `toUnit` *	Convert a number that represents mass to another unit. For the `fromUnit` and `toUnit` parameters, select the appropriate unit from the **From Unit** and **To Unit** menus. The choices are: `Kilogram` or `Pound`.
`ConvertVolumeUnit`	number	number	`fromUnit` * `toUnit` *	Convert a number that represents volume to another unit. For the `fromUnit` and `toUnit` parameters, select the appropriate unit from the **From Unit** and **To Unit** menus. The choices are: `Cubic Foot`, `Cubic Meter`, `Gallon US Fluid`, or `Liter`.
`CurrentDate`	None	date	Note	Return the current date.

Transformation	Input Type	Output Type	Parameter (* = required)	Description
`CurrentDateTime`	None	date	None	Return the current date and time.
`CurrentTime`	None	date	None	Return the current time.
`DayOfWeek`	date	number	None	Return the day of the week (1 through 7) that corresponds to the date.
`DayOfYear`	date	number	None	Return the day of the year (1 through 366) that corresponds to the date.
`EndsWith`	string	Boolean	`string`	Return true if a string ends with the specified `string`, including case.
`Equals`	any	Boolean	`value`	Return true if a field is equal to the specified `value`, including case.
`FileExtension`	string	string	None	From a string that represents a file name, return the file extension without the dot.
`Floor`	number	number	None	Return the whole number floor of a number.

Transformation	Input Type	Output Type	Parameter (* = required)	Description
`Format`	any	string	`template` *	In `template`, replace each placeholder (such as `%s`) with the value of the input field and return a string that contains the result. This is similar to mechanisms that are available in programming languages such as Java and C.
`GenerateUUID`	None	string	None	Create a string that represents a random UUID.
`IndexOf`	string	number	`string`	In a string, starting at 0, return the first index of the specified `string`. Return `-1` if it is not found.
`IsNull`	any	Boolean	None	Return true if a field is null.
`LastIndexOf`	string	number	`string`	In a string, starting at 0, return the last index of the specified `string`. Return `-1` if it is not found.
`Length`	any	number	None	Return the length of the field, or `-1` if the field is null.
`Lowercase`	string	string	None	Convert a string to lowercase.

Transformation	Input Type	Output Type	Parameter (* = required)	Description
`Normalize`	string	string	None	Replace consecutive whitespace characters with a single space and trim leading and trailing whitespace from a string.
`PadStringLeft`	string	string	`padCharacter` * `padCount` *	Insert the character supplied in `padCharacter` at the beginning of a string. Do this the number of times specified in `padCount`.
`PadStringRight`	string	string	`padCharacter` * `padCount` *	Insert the character supplied in `padCharacter` at the end of a string. Do this the number of times specified in `padCount`.
`Prepend`	string	string	`string`	Prefix `string` to the beginning of a string. the default is to prepend nothing.
`ReplaceAll`	string	string	`match` * `newString`	In a string, replace all occurrences of the supplied matching string with the supplied `newString`. The default `newString` is an empty string.

Transformation	Input Type	Output Type	Parameter (* = required)	Description
`ReplaceFirst`	string	string	`match` * `newString` *	In a string, replace the first occurrence of the specified `match` string with the specified `newString`. The default `newString` is an empty string.
`Round`	number	number	None	Return the rounded whole number of a number.
`SeparateByDash`	string	string	None	Replace each occurrence of whitespace, colon (:), underscore (_), plus (+), and equals (=) with a hyphen (-).
`SeparateByUnderscore`	string	string	None	Replace each occurrence of whitespace, colon (:), hyphen (-), plus (+), and equals (=) with an underscore (_).
`StartsWith`	string	Boolean	`string`	Return true if a string starts with the specified string (including case).

Transformation	Input Type	Output Type	Parameter (* = required)	Description
`Substring`	string	string	`startIndex` * `endIndex`	Retrieve a segment of a string from the specified inclusive `startIndex` to the specified exclusive `endIndex`. Both indexes start at zero. `startIndex` is inclusive. `endIndex` is exclusive. The default value of `endIndex` is the length of the string.
`SubstringAfter`	string	string	`startIndex` * `endIndex` `match` *	Retrieve the segment of a string after the specified `match` string from the specified inclusive `startIndex` to the specified exclusive `endIndex`. Both indexes start at zero. The default value of `endIndex` is the length of the string after the supplied `match` string.

Transformation	Input Type	Output Type	Parameter (* = required)	Description
SubstringBefore	string	string	startIndex * endIndex match *	Retrieve a segment of a string before the supplied match string from the supplied inclusive startIndex to the supplied exclusive endIndex. Both indexes start at zero. The default value of endIndex is the length of the string before the supplied match string.
Trim	string	string	None	Trim leading and trailing whitespace from a string.
TrimLeft	string	string	None	Trim leading whitespace from a string.
TrimRight	string	string	None	Trim trailing whitespace from a string.
Uppercase	string	string	None	Convert a string to uppercase.

4.11. Creating a custom transformation

In addition to the built-in transformations, you can define custom field action transformations. Write custom field actions in Java and then import them into the AtlasMap data mapper. After it is established in a source or target panel, the transformation appears in the list of transformations that are available for any field.

Procedure

1. Define a transformation in Java.

   The following custom transformation is applicable to String arguments. It takes the argument specified in the source panel transformation and prints it on the target side with the string 'concur-' and the user-specified concur parameter. Implement the AtlasFieldAction class as follows:

src/main/java/io/atlasmap/maven/test/PaulsFieldActions.java

```
package io.atlasmap.maven.test;

import io.atlasmap.spi.AtlasActionProcessor;
import io.atlasmap.spi.AtlasFieldAction;

public class PaulsFieldActions implements AtlasFieldAction {

    @AtlasActionProcessor
    public static String myCustomFieldAction(PaulsFieldActionsModel
myCustomFieldAction, String input) {
        return "Paul's custom field action: " + myCustomFieldAction.getPaulsParam()
+ " payload: " + input;
    }

}
```

And also implement a model class `PaulsFieldActionsModel` - model class can declare transformation parameters as follows:

src/main/java/io/atlasmap/maven/test/PaulsFieldActionsModel.java

```
package io.atlasmap.maven.test;

import java.io.Serializable;

import com.fasterxml.jackson.annotation.JsonPropertyDescription;

import io.atlasmap.v2.Action;
import io.atlasmap.v2.AtlasActionProperty;
import io.atlasmap.v2.FieldType;

public class PaulsFieldActionsModel extends Action implements Serializable {

    private static final long serialVersionUID = 1L;
    private String paulsParam = "";
    /**
     * Example of a custom field action with a string parameter.
     *
     * @param value
     *     allowed object is
     *     {@link String}
     *
     */
    @JsonPropertyDescription("Paul's custom field action parameter to display")
    @AtlasActionProperty(title = "Paul's custom field action string parameter",
type = FieldType.STRING)
    public void setPaulsParam(String value) {
        paulsParam = value;
    }

    public String getPaulsParam() {
        return paulsParam;
    }
}
```

2. Declare for Java service loader

   Then both class needs to be declared in `META-INF/services` to get them registered through Java service loader.

   src/main/resources/META-INF/services/io.atlasmap.v2.Action

   ```
   io.atlasmap.maven.test.PaulsFieldActionsModel
   ```

   src/main/resources/META-INF/services/io.atlasmap.spi.AtlasFieldAction

   ```
   io.atlasmap.maven.test.PaulsFieldActions
   ```

3. Build your Java archive file.

   The `io.atlasmap.v2`, `io.atlasmap.api` and `io.atlasmap.spi` target dependencies are most easily resolved through the use of a maven `pom.xml` file. Use the same version number as the AtlasMap standalone JAR file that you previously downloaded.

4. At the top of the AtlasMap main tool bar, click `AtlasMap > Import a Java archive (.jar)` to import your Java archive file.

   [Import a Java archive]

5. Navigate to the JAR file that contains your custom transformation and select it.

   [Import custom field action JAR]

6. In the **Mapping Details** panel, in the **Targets** section, select the transformation pull-down menu to see that your custom transformation now appears as a selectable transformation. Select it.

   [Import custom field action JAR]

7. To test your custom transformation:

   a. In the **Mapping Details** panel transformation you just added, type the string `ball` in the `Pauls Concur Param` input text box.

   b. In the AtlasMap main tool bar, click the mapping preview icon.

   c. In the **Source** panel input field, type a string, for example, `bat`.

   d. Notice the same string in the **Preview Results** field in the target panel with `concur-ball` prepended.

   [Import custom field action JAR]

When you run the mapping with custom transformation, Java classes and service declaration files you created above have to be put into classpath. Although uploaded jar file are stored in .adm file, those are only used to load mapping definition in the UI, but not at runtime.

Chapter 5. Exporting mappings to an ADM archive file

After all mappings are defined, save the mappings by exporting them to an ADM archive file:

1. Click the **AtlasMap** menu and select **Export the current mappings and support files into a catalog (.adm)**

   [AtlasMap menu].

2. In the **Export mappings** field, accept the default name or enter a name for the exported `.adm` file. The default name is `atlasmap-mapping.adm`.
3. Click **OK**.

   The catalog will be saved in your **Downloads** folder.

4. Note, you can later import the .adm file by using the **AtlasMap** menu.

Chapter 6. Resetting data files in AtlasMap

Initially, AtlasMap displays a blank canvas. You must import at least one data file into the **Source** panel and at least one data file into the **Target** panel. You can then map fields.

After you import a data file, you can remove it from the data mapper canvas by clicking the trash icon, as in the following image:

[Remove the specified file]

To remove all imported files, as well as all data mappings:

1. Click the **AtlasMap** menu.
2. In the pop-up menu, select **Reset all mappings and clear all imported documents**.
3. In the confirmation dialog, click **Confirm**.

AtlasMap displays a blank canvas.

Chapter 7. Running AtlasMap with Apache Camel

The latest version of `camel-atlasmap` component is now maintained in Apache Camel upstream https://camel.apache.org/components/latest/atlasmap-component.html

We will keep maintaining Camel2 version of `camel-atlasmap` component in this repository for a while, however for Camel3 you need to go with the one from Apache Camel upstream.

Camel Quarkus extension for AtlasMap is available here https://camel.apache.org/camel-quarkus/latest/reference/extensions/atlasmap.html

Developers who are familiar with Apache Camel can implement a Camel application that uses the `camel-atlasmap` component. A `camel-atlasmap` endpoint

- Can consume an AtlasMap `.adm` file
- Uses the content of `body` as the default source document for mappings

For example:

```
from("direct:start")
   .to("atlas:atlas-mapping.adm")
   .log("${body}")
```

If `body` contains a `java.util.Map` object, then `camel-atlasmap` uses `key` as a document ID and the corresponding value as the document payload.

You can obtain the `camel-atlasmap` component here: https://repo1.maven.org/maven2/io/atlasmap/camel-atlasmap/$2.4.0-SNAPSHOT/camel-atlasmap-$2.4.0-SNAPSHOT.jar

Chapter 8. AtlasMap Runtime Examples

8.1. Using AtlasMap API

8.1.1. AtlasMap runtime API example

https://github.com/atlasmap/atlasmap/tree/main/examples/atlasmap-example-api

This is a small example of running AtlasMap mapping by interacting with runtime API directly. It consumes `order.json` JSON file, process mapping and produce XML output. The output XML is simply printed in console.

Compile and run an example:

```
../../mvnw clean package exec:java
```

Then generated XML output is printed in console:

```
Target Document:
<?xml version="1.0" encoding="UTF-8"?><ns:XmlOE
xmlns:ns="http://atlasmap.io/xml/test/v2" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
  <ns:Address>
    <ns:addressLine1>1040 Notexisting St</ns:addressLine1>
    <ns:city>Westford</ns:city>
    <ns:state>Massachusetts</ns:state>
    <ns:zipCode>01886</ns:zipCode>
  </ns:Address>
  <ns:Contact>
    <ns:firstName>Bob</ns:firstName>
    <ns:lastName>Totton</ns:lastName>
    <ns:phoneNumber>123-456-7890</ns:phoneNumber>
    <ns:zipCode>01886</ns:zipCode>
  </ns:Contact>
  <ns:orderId>O001</ns:orderId>
</ns:XmlOE>
```

8.1.2. AtlasMap runtime API with ADM archive file example

https://github.com/atlasmap/atlasmap/tree/main/examples/atlasmap-example-api-adm

This is a small example of running AtlasMap mapping by interacting with runtime API directly. This example uses an ADM archive file to load a mapping definition from. It consumes `order.json` JSON file, process mapping and produce XML output. The output XML is simply printed in console.

Compile and run an example:

```
../../mvnw clean package exec:java
```

Then generated XML output is printed in console:

```
Target Document:
<?xml version="1.0" encoding="UTF-8"?><ns:XmlOE
xmlns:ns="http://atlasmap.io/xml/test/v2" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
  <ns:Address>
    <ns:addressLine1>1040 Notexisting St</ns:addressLine1>
    <ns:city>Westford</ns:city>
    <ns:state>Massachusetts</ns:state>
    <ns:zipCode>01886</ns:zipCode>
  </ns:Address>
  <ns:Contact>
    <ns:firstName>Bob</ns:firstName>
    <ns:lastName>Totton</ns:lastName>
    <ns:phoneNumber>123-456-7890</ns:phoneNumber>
    <ns:zipCode>01886</ns:zipCode>
  </ns:Contact>
  <ns:orderId>O001</ns:orderId>
</ns:XmlOE>
```

8.2. Running with Apache Camel 2.x

8.2.1. AtlasMap running with Apache Camel2

https://github.com/atlasmap/atlasmap/tree/main/examples/camel-example-atlasmap

Introduction

This example shows how to work with AtlasMap Data Mapper running as a part of Camel route.

Build

You will need to compile this example first:

```
mvn compile
```

Run

To run the example, type

```
mvn camel:run
```

To stop the example hit ctrl+c.

8.2.2. AtlasMap running with Apache Camel2 :: Blueprint XML

https://github.com/atlasmap/atlasmap/tree/main/examples/camel-example-atlasmap-blueprint

Introduction

This example shows how to work with AtlasMap Data Mapper running as a part of Camel route.

Build

You will need to compile this example first:

```
mvn compile
```

Run without container

To run the example, type

```
mvn camel:run
```

To stop the example hit ctrl+c.

Run on karaf container

To run the example on the karaf container

Step 1: Start karaf container

```
karaf / karaf.bat
```

Step 2: Deploy

```
feature:repo-add mvn:io.atlasmap.examples/camel-example-atlasmap-
blueprint/${VERSION}/xml/features
feature:install camel-example-atlasmap-blueprint
```

Step 3: Check the output

You will see the output by log:tail or in ${karaf}/data/karaf.log

You can see the routing rules by looking at the Blueprint XML configuration lives in `src/main/resources/OSGI-INF/blueprint`

8.2.3. AtlasMap running with Apache Camel2 :: Message Map

Introduction

This example shows how to perform data mapping from multiple source Documents and corresponding headers stored in Message Map with using AtlasMap Data Mapper running as a part of Camel route. AtlasMap expects Message Map to hold Camel Message with a Document ID as a key. The Message headers are handled as a scoped property where Document ID is equal to the scope. Syndesis provides this message mapping feature internally so that the Data Mapper step can consume multiple source Messages as well as headers. In this example, `MessageCaptureProcessor` does that part of managing Message Map.

Build

You will need to compile this example first:

```
mvn compile
```

Run

To run the example, type

```
mvn camel:run
```

To stop the example hit ctrl+c.

8.2.4. AtlasMap running with Apache Camel :: Multiple source Documents

Introduction

This example shows how to perform data mapping from multiple source Documents with using AtlasMap Data Mapper running as a part of Camel route.

Build

You will need to compile this example first:

```
mvn compile
```

Run

To run the example, type

```
mvn camel:run
```

To stop the example hit ctrl+c.

8.3. Running with Apache Camel 3.x

8.3.1. AtlasMap running with Apache Camel 3

https://github.com/atlasmap/atlasmap/tree/main/examples/camel3/camel-example-atlasmap

Introduction

This example shows how to work with AtlasMap Data Mapper running as a part of Camel route.

Build

You will need to compile this example first:

```
mvn compile
```

Run

To run the example, type

```
mvn camel:run
```

To stop the example hit ctrl+c.

8.3.2. AtlasMap running with Apache Camel3 :: Message Map

https://github.com/atlasmap/atlasmap/tree/main/examples/camel3/camel-example-atlasmap-msgmap

Introduction

This example shows how to perform data mapping from multiple source Documents and corresponding headers stored in Message Map with using AtlasMap Data Mapper running as a part of Camel route. AtlasMap expects Message Map to hold Camel Message with a Document ID as a key. The Message headers are handled as a scoped property where Document ID is equal to the scope. Syndesis provides this message mapping feature internally so that the Data Mapper step can consume multiple source Messages as well as headers. In this example, `MessageCaptureProcessor` does that part of managing Message Map.

Build

You will need to compile this example first:

```
mvn compile
```

Run

To run the example, type

```
mvn camel:run
```

To stop the example hit ctrl+c.

8.3.3. AtlasMap running with Apache Camel3 :: Multiple source Documents

https://github.com/atlasmap/atlasmap/tree/main/examples/camel3/camel-example-atlasmap-multidoc

Introduction

This example shows how to perform data mapping from multiple source Documents with using AtlasMap Data Mapper running as a part of Camel route.

Build

You will need to compile this example first:

```
mvn compile
```

Run

To run the example, type

```
mvn camel:run
```

To stop the example hit ctrl+c.