Administrator manual

How to create a dataflow in Reportnet
– an implementer's guide

Prepared by
Søren Roug

May 2011
Version 0.8

European Environment Agency

Version management

No	Date	Changes	Author
0.1	2008-07-21	Initial version	Søren Roug
0.2	2008-08-15	Added information on delivery mechanisms and workflow.	Søren Roug
0.3	2008-11-05	Information on webform restrictions	Søren Roug
0.4	2009-01-13	Information on translation strategies	Søren Roug
0.5	2009-01-14	Information on machine translation	Søren Roug
0.6	2009-02-11	Chapter on scripted aggregation	Søren Roug
0.7	2010-08-12	Chapter on creating your own repositories	Søren Roug
0.8	2011-05-10	Information about SPARQL queries	Søren Roug Enriko Käsper

Contents

1. Introduction

This document will show you the process of implementing a dataflow in Reportnet.

There is essentially one initial decision you must make. Is your repository going to be CDR or your own? If the latter, go to chapter 10. There is also a document titled "Implementing a Reportnet/SEIS node" that is relevant.

1.1. Roles

Since dataflows are about people doing tasks in a sequence and interacting through the system with other people, we must describe a number of roles a person can have. We have so far identified three roles of importance.

Role	Description
Data provider	The data provider is anybody who makes a delivery to either CDR or another repository. It is typically an employee of an NFP, but can also be an employee of the secretariat of a commission, if this commission must deliver to EEA.
Requester	The requester is the organisation that has requested the data. This is typically EEA, Eurostat, and OECD etc. When the data are received, the requester aggregates and analyses them.
Supervisor	A supervisor checks that the delivery is complete, and then releases it to the requester. Most often the data provider and the Supervisor is the same person, but if e.g. if the data provider is a fully automated delivery system, the supervisor role as a separate role becomes important.

1.2. Subsystems in Reportnet

Name	Description
CDR	The Central Data Dictionary is EEA's main repository for delivered data or reports. It contains a folder structure for each country where the data or reports can be uploaded on-line and made available to others.
ROD	The Reporting Obligation Database is EEA's database which contains records describing environmental reporting obligations that countries have towards international organizations, such as DG Environment, marine conventions, Eurostat, OECD, UN, UNECE, as well as the EEA itself.
DD	The Data Dictionary is EEA's database containing detailed specifications concerning the format of the data that should be submitted.
CR	XML formatted deliveries are converted to RDF and imported into the Content Registry. This then makes it possible to query the data using SPARQL e.g. for QA or information.
XMLCONV	The conversion subsystem is a web service holding scripts in XSL-T and

	XQuery language for XML files using the XML Schema identifier as they key. When requested the scripts are run on a data file.
UNS	The Unified Notification System is EEA's mechanism to notify you of new content. You can subscribe to this system and define for which items you want to receive a notification.
DIR	Eionet directory is a database with user accounts. A user account contains personal information (such as first name, surname, email address, username and password) and is used for communication and authentication purposes.

2. Creating a dataflow on CDR

2.1. Skills needed

XML Schema	Must be able to design an XML format in XML schema language.
XSL-T	Must be able to write an XSLT stylesheet that displays an XML file as HTML.
XForms	If webforms are used, must be able to write a webform in the XForms language.
Zope	Some Zope knowledge is required to create the workflow and action pages
XQuery	The QA scripts are written in the XQuery language

2.2. Steps in creating a dataflow

If you create an XML-based dataflow, these are the steps:

1. Check that the obligation is known in the reporting obligation database.
2. Define the structure of the XML instance file that will contain the delivered information as an XML schema. Store the XML schema at a permanent location with a persistent URL.
3. Create the delivery mechanism. You can:
   a) Create an XForm for the XML file format and upload it to WebQDev using the schema identifier as the key. If it is a multilingual delivery, create message catalogues in XML containing the webform labels in the supported languages.
   b) Write a script in MS-Access that exports the tables to XML
   c) ...many other options are available
4. Write one or more conversions in XSL-T. One for visual inspection in HTML. Another one to get the data exported in the desired format.
5. Write a QA script in XQuery that checks some of the more mundane issues. You can potentially check against reference data. In that case the reference data must be available in XML format.

6. Write a workflow for the dataflow. Normally this involves the steps: Draft, Automatic QA, Manual QA, End or Back to draft. You can often just reuse an existing workflow.
7. If the reporters can report in more than one language, develop a plan for translation of the deliveries, incl. potentially QA of the translations.
8. If the end product is to be a database, then you must define the database schema, and a conversion that populates the database from the XML instance files.
9. Finally, write a user guide telling the reporters how the reporting is to be made.

The entire dataflow is then assembled on http://cdrtest.eionet.europa.eu and tested before being migrated to production.

3. Creating the XML schema

CDR doesn't have a relational database such as SQL Server or MySQL. It is more like an advanced FTP-server, but using Web technology. When you make a delivery, you upload files, which are stored in a folder called an envelope. This happens both when you deliver regular files and XML. In the latter case you must define the XML file's format in the XML schema language (or DTD language).

There are basically two ways to create the XML schema. One is to do it by hand, and the other is to use the data dictionary. Choosing which one is appropriate depends on your data.

3.1. Using the data dictionary

If you data structure is already structured as a set of tables, then using the data dictionary is appropriate, because you get a number of extra features, such as automatic conversions.

When you have defined the tables, XML schemas, technical specifications and MS-Excel templates can be generated automatically, and you will also get 95% of a webform created for you.

3.2. Creating the schema by hand

A common tool to create schemas by hand is XMLSpy from Altova.

3.3. Storing the schema

It is best to store the XSD file at a persistent location. That way people can always find the newest version of the file. You then use the fully qualified URL as the schema identifier in the XML instance file. The instance file must have in its root element an *xsi:noNamespaceSchemaLocation* pointing to the schema and an *xmlns:xsi=-"http://www.w3.org/2001/XMLSchema-instance"* attribute.

This *noNamespaceSchemaLocation* attribute (alternatively the *schemaLocation* attribute) is used in by Reportnet to find relevant QA scripts and conversions.

3.4. Other considerations

Some dataflows allow the member states to submit data in their own language. Some countries deliver in English anyway, but for the others, the information will eventually have to be translated. There is a challenge in sending the right files to the right translator. Can you tell the difference between Slovak and Czech? The easiest way is to ask the countries to declare what language they are using in the file. XML specifies a mechanism using the attribute xml:lang. It should be placed on the root element and use the two-character language codes.

For further discussion on multilinguality see chapter 8.

4. Creating delivery mechanisms

4.1. MS-Excel

If you have defined the schema in the data dictionary as a dataset, it can generate an MS-Excel template for you with one sheet per table. Reportnet is then able to received a filled out template, convert it to XML and run the QA scripts.

Note, that the MS-Excel template has a sheet for *all* the tables in the dataset. It currently doesn't support uploading only one or a few of the tables with MS-Excel and the rest some other way.

The conversion process from MS-Excel to XML uses a special sheet at the end to contain information linking back to the data dictionary to aid in the conversion. The data dictionary provides information on how what elements there are, and what the name space is. You can therefore not add or remove columns from the sheets in the template unless you do it in data dictionary. You might be able to add formulas, comments etc. This has not been attempted yet.

The way the upload mechanism works in CDR is that if you upload an MS-Excel file, the conversion process will only create XML files for the sheets having content. It will however, *delete* all XML files having the same schema identifier as those found in the MS-Excel file. It is therefore dangerous to combine webforms and MS-Excel delivery. The user might delete a file created or modified by the webform.

Microsoft Office Excel 2003 has a feature that makes it possible to export a sheet in XML data format. To take advantage of this you must add *XML Maps* to the spreadsheet. No dataflows currently use this feature.

4.2. MS-Access

The easiest way to use MS-Access is to write a script in a form that exports the relevant tables as XML – even better if you can make MS-Access zip it. You then tell the user to upload the file to CDR using the webbrowser. CDR will expand zip files and be able to run the QA scripts.

4.3. DEMs

A Data Exchange Module (DEM) is the somewhat non-descriptive term we use for applications that are installed on the user's personal computer and dedicated to the

delivery for a dataflow. A DEM can upload directly to CDR, but because of issues with authentication, time-out etc., you should consider letting the DEM write a zip file, which the reporter uploads using the webbrowser.

4.4. Website for data entry

A related mechanism to the DEM is a web-based application that the data provider uses to enter data into a database. The main benefit over the DEM is that you are certain the current version of the tool is used. However, if you want to avoid committing yourself to having a website permanently available then you still need some way of transforming your database content to XML files (one per data provider) and upload them to CDR. The easiest way to do this is to let the data provider download his data as XML, then he can upload it to CDR himself and potentially there can be QA scripts installed to test the deliveries.

4.5. Webforms

CDR has the ability to provide web questionnaires for XML files. It is implemented with the XForms specification, which natively uses an XML file as the data-carrying medium. Before choosing webforms, you must evaluate the amount of data you expect the reporters to deliver, as it quickly becomes tedious to enter more than a few pages of questionnaire.

The creation of webforms is described in the Webforms User guide.

Webforms created from the data dictionary use the same XML format as the MS-Excel templates.

4.6. System-to-system delivery

The final mechanism is to give the member states instructions on how to connect their own databases to CDR. It entails creating a script that extracts the data from the member state database into XML and then calls CDR's webservice API to upload the datafile.

5. Creating conversions

One reason to create conversions is to be able to *inspect* the XML file. As you know XML is difficult to read. We therefore often create a conversion that shows the XML file as an HTML page. Sometimes also as a Google Earth KML file.

The XMLCONV can create MS-Excel files via an intermediate XML format.

Conversions are XSL-T stylesheets. The creation of them is described in the XMLCONV User Manual.

6. Creating QA scripts

Reportnet has a quality assessment service that can check XML files. The QA service provides the engines for the following query languages – XML Schema, XQuery, XSL and XGAWK. The XML Schema validation is just for a simple syntax and validation check. It also has some obscure error messages. The scripts written in query languages

can do more complex checks. E.g. check that the XML file contains information about all the stations in a given country, and that no station occurs twice. You can provide the error messages in user-friendly HTML.

But additional flexibility comes at a price. You must write the QA rules, whereas when you have an XML Schema, you can just activate it for validation.

A typical XQuery script looks like this:

```
(:=================================================================:)
(: Rule 1.c :)
(: Goes through all the Rows in the document and checks if there exist:)
(:  EWN codes with different 2 first letters (country codes):)
(: if yes, returns the code, if no, returns nothing = empty String:)
(:=================================================================:)

declare function local:checkEWNCodeStart($url as xs:string)   {
        for $pn in
fn:doc($url)//dd8:GeneralCharacterisation/dd8:Row[@status="new"]
          let $start := fn:substring($pn/dd37:EWN-Code,1,2)
         let $i :=
fn:doc($url)//dd8:GeneralCharacterisation/dd8:Row[substring(dd37:EWN-
Code,1,2) != $start]
         where fn:count($i) > 0
              order by $pn
                    return
                          <b>{$pn/dd37:EWN-Code} <br/></b>
}
;
```

The creation of QA scripts is described in the Query Service user manual.

XQuery can load external XML files containing reference data. This is very useful if you want to check whether a code is valid. Beware however, that the XQuery engine loads the file into memory, and loading too large files can have adverse effects on the system's stability.

```
(: open XML file containing the list of stations :)

let $stations :=fn:doc("http://converters.eionet.europa.eu/xmlfile/airbase-
stations.xml")

(: run through the list of stations and return EoI code:)

for $station in $stations//station
       return $station/eoi_code
```

XQuery can construct the URL to load and can load any XML that can be produced via REST. This means you can create Web Services to do the heavy lifting and let your XQuery script call it.

If you don't have the means to create a Web service or you just have a large file containing codes, you can load your reference data into the Content Registry and use SPARQL queries.

SPARQL is a Semantic Web query language with a standardised result format for XML and JSON. Your XQuery script would construct the SPARQL query, send it as a GET or POST query and read the result in XML.

The following snippet shows how to check what countries a lat/long coordinate could belong to using bounding boxes around countries. (a country can have multiple bounding boxes for a snug fit):

```
(: Define Content Registry SPARQL Endpoint URL  :)

declare variable $xmlconv:SPARQL_URL := "http://cr.eionet.europa.eu/sparql";


(: Helper function for constructing the request URL for SPARQL query :)
declare function xmlconv:getSparqlEndPointUri($sparql as xs:string) {
    let $sparql := fn:encode-for-uri($sparql)

    let $defaultGraph := fn:encode-for-
uri("http://converters.eionet.europa.eu/xmlfile/stations-min-max.rdf")

    let $uriParams := concat("default-graph-uri=", $defaultGraph,
"&amp;query=", $sparql , "&amp;format=application/sparql-results+xml")
    let $uri := concat($xmlconv:SPARQL_URL, "?", $uriParams)

    return
        $uri
}
;


(: Function checks if the given longitude and latitude belong to some of the
countries' bounding boxes. Return the list of matching countries :)

declare function xmlconv:getCountryByLongLat($lat as xs:decimal, $long as
xs:decimal) {

    let $sparql := concat("PREFIX eea:
<http://rdfdata.eionet.europa.eu/eea/ontology/>
  SELECT ?country WHERE { ?s eea:isoCode ?country .
            ?s eea:minx ?minx . FILTER ( xsd:decimal(?minx)<=", $lat, ")
            ?s eea:maxx ?maxx . FILTER ( xsd:decimal(?maxx)>=", $lat, ")
            ?s eea:miny ?miny . FILTER ( xsd:decimal(?miny)<=", $long, ")
            ?s eea:maxy ?maxy . FILTER ( xsd:decimal(?maxy)>=", $long, ")
        }")

        let $sparqlUri := xmlconv:getSparqlEndPointUri(fn:normalize-
space($sparql))

    return
       fn:doc($sparqlUri)
}
;
```

7. Designing the workflow

CDR has a document-centric workflow system that makes the envelope transit from activity to activity.

The process defining the sequence of activities to be carried out says **what** should be done and **when** by the definition of activities and transition. An activity (the *what* part of the issue) represents something to be done: giving authorization, updating a database, sending an e-mail, loading a truck, filling a form, printing a document and so on.

Transitions define the appropriate sequence of activities for a process (the *when* part of the issue).

Each activity will have an associated form or application designed to carry out the job: the **how** part.

The workflow is selected and begins when the user creates the envelope. Typically the first activity is the *Draft*. The data provider will see a page that allows him to upload files or fill out questionnaires depending on how the dataflow is designed to work.

He will then *complete* the activity, and it will *transition* to the next. In the illustration below it progresses to an automatic step called *Insert record into DB*. Depending on whether this went well the system forwards the envelope to either release or send back to draft.

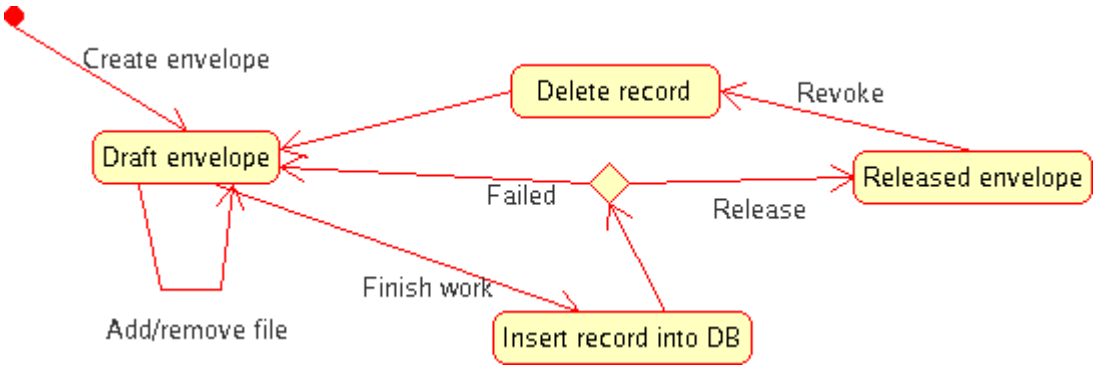


Illustration 1: Workflow with database interaction

A typical workflow runs with these activities:

1. Draft. Read it as a verb. The reporter is shown a special web page called the activity page, that gives him the possibility to add files to the envelope either by file upload or formfilling. He then clicks *Complete activity*. The system automatically goes to:
2. Release the envelope. This makes the envelope available to the general public. It signifies that the country has delivered, and the date this occurs is used to check whether the country has met the deadline or not.
3. Run Automatic QA. Before the envelope is forwarded to the requester, the system runs the automatic QA on all files and stores the feedback in the envelope as a permanent comment.
4. Step 4 can take two forms. In most cases the receiver of the data is not involved in the Reportnet process. The receiver would typically be UNEP, Helsinki Commission, ACCOBAMS etc. In that case the activity is a stop, where the *reporter* has the option to revoke the delivery and restart the drafting, presumably because of feedback received over other channels than Reportnet

   If the receiver *is* involved, this step is an activity, where the receiver can accept or reject the delivery. Rejecting sends the envelope back to draft. Accepting completes the workflow. The reporter can not modify the delivery after completion, nor delete it.

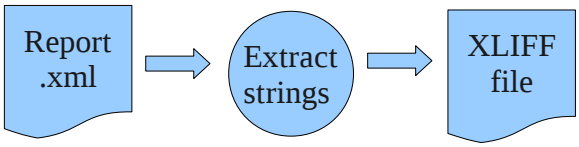
8. Translations

In some dataflows the member states can deliver in their own language. How do you deal with this? In our experience, as late as possible in the pipeline. I.e. after the texts have been extracted and imported into a database.

8.1. Example 1: Reviewing translations

However, there is a special case that is relevant to CDR. That's when the country wants to verify the translations to English. In this case – and it has so far not been tried – you put the translations in a separate file to avoid tampering with the original delivery.

Since the report is in XML you can create a stylesheet to extract the text strings. EEA's interface with the translation centre is the XLIFF, which is also XML. It looks like below:



Let's try with a more specific example. Let's assume Estonia has delivered a report in Estonian describing the effects of something and the area affected.

Original report (very simple):

```
<?xml version="1.0"?>
<report xml:lang="et">
  <effect>Bioloogilise mitmekesisuse muutumine</effect>
  <effect>Hapestumine</effect>
  <area>15000</area>
</report>
```

We create a stylesheet to generate XLIFF for the elements we know contain Estonian text. In this case it is only the element <effect>. The stylesheet does not handle repeated answers, which is an obvious optimisation. There is no need to put the text *Hapestumine* into the XLIFF file more than once.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:output method="xml" indent="yes"/>

  <xsl:template match="report">
  <xliff version="1.1">
    <file original="report.xml"
      datatype="plaintext"
      target-language="en">
      <xsl:attribute name="source-language">
      <xsl:value-of select="@xml:lang"/>
      </xsl:attribute>
      <header/>
      <body>
        <xsl:apply-templates select="effect"/>
      </body>
    </file>
  </xliff>
  </xsl:template>
```

```
  <xsl:template match="effect">
    <trans-unit>
      <xsl:attribute name="id">
        <xsl:value-of select="generate-id()"/>
      </xsl:attribute>
      <source><xsl:value-of select="text()"/></source>
      <target><xsl:value-of select="text()"/></target>
    </trans-unit>
  </xsl:template>

</xsl:stylesheet>
```

Note that the xml:lang attribute is grabbed and used for the source-language attribute in the XLIFF file. It will also be used to route the XLIFF file to the translator who understands Estonian, or to not do anything if the text is already in English.

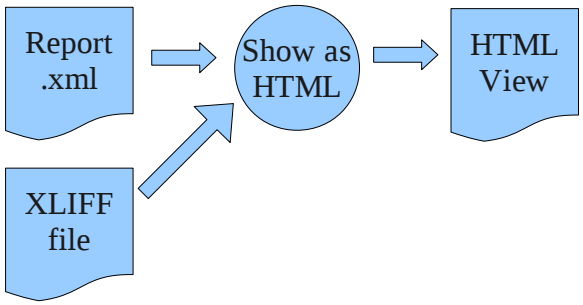
The translation centre has the quirk that it needs the source text in the target element as well as in the source. They will then replace the content of <target> with English. This is due to their use of Trados' TagEditor to translate XML. It also needs an XLIFF.ini file to configure TagEditor to only open the <target> element for translation.

The XLIFF output *after* translation looks like this:

```
<?xml version="1.0"?>
<xliff version="1.1">
  <file original="report.xml" datatype="plaintext"
    target-language="en" source-language="et">
    <header/>
    <body>
      <trans-unit id="id2651627">
        <source>Bioloogilise mitmekesisuse muutumine</source>
        <target>Biodiversity change</target>
      </trans-unit>
      <trans-unit id="id2652309">
        <source>Hapestumine</source>
        <target>Acidification</target>
      </trans-unit>
    </body>
  </file>
</xliff>
```

and is saved as translations.xlf somewhere on the Internet.

The next step is to display the translated text to the data provider. One way to do it is to display the XML as an HTML factsheet. It is more user-friendly. What you'd do is to create a stylesheet linked to the report's schema identifier as normal, but whenever there is text to display, look it up in the translation file and display the translation instead. We reuse the XLIFF file as a message catalogue.



Stylesheet to show report.xml as HTML:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
```

```
  <xsl:output method="html"/>

  <xsl:template match="/">
  <html>
    <head>
      <title>Effects of emissions</title>
    </head>
    <body>
      <table>
        <xsl:apply-templates select="//effect"/>
      </table>
    </body>
  </html>
  </xsl:template>

  <xsl:template match="effect">
    <tr>
      <th>Effect</th>
      <td><xsl:value-of select="text()"/></td>
      <xsl:variable name="sourcetext" select="text()"/>
      <td><xsl:value-of
select="document('translations.xlf')/xliff/file/body/trans-
unit[source/text()=$sourcetext]/target"/></td>
    </tr>
  </xsl:template>

</xsl:stylesheet>
```

The file looks up the content of the <effect> elements in the XLIFF file. *Hapestumine* is always translated to Acidification and *Jah* always to Yes etc. You can choose to display both the source and the target side-by-side, or whatever is convenient for your users.

Result is:

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Effects of emissions</title>
</head>
<body>
  <table>
    <tr>
      <th>Effect</th>
      <td>Bioloogilise mitmekesisuse muutumine</td>
      <td>Biodiversity change</td>
    </tr>
    <tr>
      <th>Effect</th>
      <td>Hapestumine</td>
      <td>Acidification</td>
    </tr>
  </table>
</body>
</html>
```

And will be shown in the webbrowser like this:

Effect	Bioloogilise mitmekesisuse muutumine	Biodiversity change
Effect	Hapestumine	Acidification

There are several benefits to this approach:

a) If a translation is incorrect, you can replace it in the XLIFF file.

b) If the country decides to redeliver with some changes to the XML file, you'll notice which parts of the XML file that is out of sync with the translation. They are the text segments, that don't have translations.

c) You can work with translations to more than one language simultaneously.

8.2. Example 2: Using translations on a website

After the member states have delivered you presumably want to work with the deliveries, and import them into a database. You can either translate all the non-English XML files, import those and discard the originals, or you can import the originals and then translate in the database.

This example deal with the latter option. If we go back to the XML delivery from example 1, we'd create a table for the effects for all member states and include the language code. The table could look like this:

'Effects' table		
Country	**Language**	**Effect**
EE	et	Bioloogilise mitmekesisuse muutumine
EE	et	Hapestumine
DE	de	Versauerung

You create a script that exports the 'effect' column to a table called 'translations'. You do the same for all tables and all columns with text to translate.

```
INSERT IGNORE INTO translations (transunitid, langcode, translation,
issource)
SELECT MD5(effect), language, effect, 1
FROM effects WHERE language != 'en'
```

This is the translations table:

```
CREATE TABLE translations (
  transunitid varchar(32) character set ascii NOT NULL default ''
              COMMENT 'The key is the MD5 hashcode',
  langcode char(2) character set ascii NOT NULL default ''
              COMMENT 'Language code',
  translation text NOT NULL,
  issource tinyint(1) NOT NULL default '0',
  lineno int(11) NOT NULL auto_increment
              COMMENT 'To extract in the same order as imported',
  PRIMARY KEY  (transunitid,langcode),
  KEY lineno (lineno)
) DEFAULT CHARSET=utf8 COMMENT='Table for translations';
```

From the 'translations' table you create the XLIFF file. You will want to send only those text segments to the Translation Centre that you don't already have translations for. This statement will extract the text strings written in Estonian (et) with no English (en).

```
SELECT source.transunitid,source.translation from translations AS source
LEFT JOIN translations AS target
ON source.transunitid = target.transunitid
```

```
 AND target.langcode='en'
WHERE source.langcode = 'et'
 AND source.issource = 1
 AND target.langcode IS NULL
ORDER BY source.lineno
```

When you get the translations back from the Translation Centre, you import them into the same translations table. The translation finds the source text based on a key, which is a simple MD5 hash created from the source text.

Then in your code, when you need to display the English translation for *Hapestumine*:

```
SELECT translation FROM translations
WHERE transunitid=MD5('Hapestumine')
AND ( langcode='en' OR issource=1 )
ORDER BY issource
```

You will not get a hit if your source is already in English, or your translation table doesn't have all strings. If you don't get a hit you display what you're trying to look up. You can fix that by adding the English strings. Otherwise, the query gives you at least one record. The first record is the translation, if none is available, then you get the source text, and you just display that one.

8.3. Machine translation (MT)

If all you need to do is to get an idea of what is written in the foreign language, you might want to try machine translation. The commission has a service that can be reached via a website or email address. Unfortunately it doesn't understand XLIFF and it always expects ISO-8859-X encoding. You might have to clean your text of Euro-signs etc. before sending it off to MT. Additionally, your result is always sent back as an email.

At the time of this writing (January 2009), the following languages are supported: Spanish, Danish, German, Greek, English, French, Italian, Dutch, Portuguese and Swedish. No translations for recent EU members, and the translations for the smaller languages (Danish, Swedish) can be really bad.

You might get better results with Google's free machine translation. In most cases the results are better and since late 2008 it has more languages. The location is http://code.google.com/apis/ajaxlanguage/ You can call it as a webservice API, and if you dig into the documentation, there are ways to use other languages than JavaScript.

9. Aggregation

Aggregation is the term we use for grabbing all deliveries for a particular dataflow and period, and then merge them to one dataset.

It is the least developed part of Reportnet, as it doesn't do anything for you automatically, but only provides you a few APIs to do the work manually.

If you get data delivered in binary formats; MS-Excel, MS-Access, shapefiles etc. then you have to do everything manually. You can find the deliveries at http://cdr.eionet.europa.eu/searchdataflow, and then proceed to download them all as a zipped archive.

When aggregating XML files, there is some support, but the task becomes more complex. You first have to be aware of for what purpose you're aggregating. If it is QA then you might want to declare all columns in your database CHAR, as this won't *abort* the INSERT when the reporter has entered "n/a" into a field that expects an integer. We'll proceed with the assumption you're doing QA.

The first problem is to define a relational table structure that can contain the delivered data. In most cases this is already done, and the XML schema for the delivery was derived from the table structure. In that case you change all non-char columns to char, and as I'll show, you'll add a few columns as well.

When you grab the files, there's a risk you get too much. The country can have reported twice for the same period, and it is up to you to figure out if the second envelope replaces or supplements the first envelope.

To detect when this happens, you should add the envelope URL and reporting date as extra columns.

Example:

We'll assume the countries have reported on the basic statistics of their country. In XML it will look like:

```
<?xml version="1.0"?>
<report>
  <population-size>5373870</population-size>
  <country-area>49035</country-area>
</report>
```

When you aggregate you'll want the country code in the table as well. You can get it from the envelope metadata if it isn't in the file.

'Country stats' table		
Country	**Population_size**	**Country_area**
SK	5.373.870	49.035

But if Slovakia has revised it population size, it might have reported again somewhat later:

```
<?xml version="1.0"?>
<report>
  <population-size>5379455</population-size>
  <country-area></country-area>
</report>
```

If you get the URL and reporting date from the envelope, your table will look like this:

'Country stats' table				
Country	**Envelope**	**Released**	**Population_size**	**Country_area**
SK	http://cdr.../envu8s9zq	2008-12-23 10:23:22	5.373.870	49.035
SK	http://cdr.../envy0x1os	2009-01-04 15:51:06	5.379.455	NULL

Now you can decide how you want to deal with the NULL in the last row and you have the URL of the envelope to tell the reporter if you don't accept missing values.

Some dataflows allow the reporters to provide revised data for the previous reporting periods at the same time the report for the current period. If so, you might want to add yet another envelope metadata property as a column to the table; the periodic coverage.

9.1. Extraction

If the dataflow is one that happens on a regular schedule, or your QA procedures involves letting the countries revise their deliveries, you'll want to write a script to do the extraction and aggregation work.

One way to do it is to call CDR's XML-RPC API. That way you get the envelope metadata as well. There's a method called xmlrpc_search_envelopes and it takes two arguments, the obligation identifier, and whether you want released or unreleased envelopes.

In Python the output is a list of envelopes in the form of dictionaries, where the envelope looks like this:

```
{
'dataflow_uris': ['http://rod.eionet.eu.int/obligations/367'],
'description': '546-8381-08 Rv',
'title': 'Report on all major roads, railways and airports delivery 2008',
'url': 'http://cdr.eionet.europa.eu/se/eu/noise/df5/envst0zuq',
'startyear': 2008,
'endyear: '',
'partofyear': 'Whole Year',
'country: 'http://rod.eionet.eu.int/spatial/36',
'country_code': 'SE',
'country_name': 'Sweden',
'locality': '',
'isreleased': 1,
'released': '2008-12-17T13:29:56Z',
'files': [
 ['DF5_MRail.xml',
  'text/xml',
  'http://dd.eionet.europa.eu/GetSchema?id=TBL5079',
  'Converted from – SE_NoiseDirectiveDF5.xls',
  '2008-12-17T13:52:42Z',
  ''],
 ['DF5_MRoad.xml',
  'text/xml',
  'http://dd.eionet.europa.eu/GetSchema?id=TBL5078',
  'Converted from – SE_NoiseDirectiveDF5.xls',
  '2008-12-17T13:52:43Z',
  ''],
 ['DF5_MAir.xml',
  'text/xml',
  'http://dd.eionet.europa.eu/GetSchema?id=TBL4956',
  'Converted from – SE_NoiseDirectiveDF5.xls',
  '2008-12-17T13:52:43Z',
  '']]
}
```

The 'files' key contains a list of files in the envelope. For each file you have six metadata properties; filename, media type, XML schema identifier, title, upload time and accept

time. The last one will most likely be empty, as the file presumably hasn't cleared manual QA yet and you haven't accepted it.

Here is an example on how to call the API.

```
#!/usr/bin/env python
import xmlrpclib
isreleased = 1
obligation = 'http://rod.eionet.eu.int/obligations/367'

server = xmlrpclib.ServerProxy("http://cdr.eionet.europa.eu/")
for envelope in server.xmlrpc_search_envelopes(obligation, isreleased):
    envelopeurl = envelope['url']
    startyear = envelope['startyear']
    if startyear not in (2007,2008,2009):
        continue
    files = envelope['files']
    for f in files:
        pathname = "%s/%s" % (envelopeurl, f[0])
        if f[2] == 'http://dd.eionet.europa.eu/GetSchema?id=TBL5079':
            os.system("""wget -O /tmp/downloadtmp.xml %s ; xsltproc
--stringparam countrycode %s --stringparam envelopeurl '%s' --stringparam
filename '%s' --stringparam uploadtime '%s' --stringparam accepttime '%s'
DF5_MRail.xslt /tmp/downloadtmp.xml ; rm -f /tmp/downloadtmp.xml""" %
(pathname, envelope['country_code'], envelopeurl, f[0], f[4], f[5]) )
```

The script calls CDR and gets all deliveries for obligation #367. The key is the URL of the obligation on ROD[1]. Then the script loops through the list of envelopes. If the reporting period doesn't start in 2007-2009, then it ignores it. Otherwise it downloads all files with the schema identifier of "http://dd.eionet.europa.eu/GetSchema?id=TBL5079" and runs the XSLT transformation in the file DF5_MRail.xslt on it, giving it some parameters from the envelope.

9.2. Converting XML to SQL

The next challenge is to convert the tree structured XML to table-oriented SQL. This can be easy or difficult depending on how convoluted the XML is.

Additionally, remember that SQL strings must have apostrophes escaped. That's what we have the template *globalReplace* for.

Here is an example (DF5_MRail.xslt):

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
       xmlns:dd499="http://dd.eionet.europa.eu/namespace.jsp?ns_id=499"
       xmlns:dd594="http://dd.eionet.europa.eu/namespace.jsp?ns_id=594"
       version="1.0">
  <xsl:output method="text"/>

  <xsl:param name="countrycode"/>
  <xsl:param name="envelopeurl"/>
  <xsl:param name="filename"/>
  <xsl:param name="uploadtime"/>
  <xsl:param name="accepttime"/>

  <xsl:template match="/">
    <xsl:apply-templates select="dd499:DF5_MRail/dd499:Row"/>
  </xsl:template>
```

[1] For historical reasons the URL is the old domain of ROD (rod.eionet.*eu.int*).

```
  <xsl:template match="dd499:DF5_MRail/dd499:Row">
INSERT INTO DF5_MRail VALUES (
    '<xsl:value-of select="$countrycode"/>',
    '<xsl:value-of select="$envelopeurl"/>',
    '<xsl:value-of select="$filename"/>',
    '<xsl:value-of select="$uploadtime"/>',
    '<xsl:value-of select="$accepttime"/>',
<xsl:for-each select="*"><xsl:if test=".=''">null</xsl:if><xsl:if test=".!
=''">'<xsl:call-template name="globalReplace"><xsl:with-param
name="outputString" select="."/><xsl:with-param name="target"
select="&quot;'&quot;"/><xsl:with-param name="replacement"
select="&quot;''&quot;"/></xsl:call-template>'</xsl:if><xsl:if
test="position()!=last()">,</xsl:if></xsl:for-each>
);
  </xsl:template>

  <xsl:template name="globalReplace">
    <xsl:param name="outputString"/>
    <xsl:param name="target"/>
    <xsl:param name="replacement"/>
    <xsl:choose>
      <xsl:when test="contains($outputString,$target)">
        <xsl:value-of select="concat(substring-before($outputString,$target),
$replacement)"/>
        <xsl:call-template name="globalReplace">
          <xsl:with-param name="outputString"
                          select="substring-after($outputString,$target)"/>
          <xsl:with-param name="target" select="$target"/>
          <xsl:with-param name="replacement" select="$replacement"/>
        </xsl:call-template>
      </xsl:when>
      <xsl:otherwise>
        <xsl:value-of select="$outputString"/>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>

</xsl:stylesheet>
```

The DF5_MRail is already a plain table structure, so the conversion is straight forward. It simply goes through every <dd499:Row> and converts the elements inside to values for an INSERT statement. In this XML format there are no optional elements under <dd499:Row>, and therefore the script doesn't have to take that into account.

After execution, you'll have something like this:

```
INSERT INTO DF5_MRail VALUES (
    'SE',                                                  -- country code
    'http://cdr.eionet.europa.eu/se/eu/noise/df5/envsvouvw', -- envelope url
    'DF5_MRail.xml',                                       -- filename
    '2008-12-30T14:37:47Z',                                -- upload time
    '',                                                    -- accept time
    'a','1','Västra Stambanan','57100',                    -- data
    'Flb – Söö','6568514','1622318','6561500','1605300',
    '21000','RT90 2,5 gon V 0:-15');
```

9.3. Semantic web aggregation

EEA is building a different way to aggregate the data. The idea is to use Semantic web principles. You create an XSL stylesheet that converts XML to RDF. This can then be loaded into any Semantic web tool, and the aggregation will happen automatically. Part of the project is to create a search engine, called Content Registry, which will be notified as soon as a delivery has been made to CDR. It will load the data into its database, and you can query it immediately with SPARQL.

10. Creating a new repository

A repository is the website, where deliveries are downloaded from. To be able to trace back to what an aggregated dataset is based on, there must be a location, where the original deliveries are permanently stored. CDR usually has that role, but if your needs are different, this chapter will explain how to link a new repository to the rest of Reportnet.

It isn't trivial to set up a new repository. After you've looked at the requirements, consider developing your system as a DEM – just for the drafting of the delivery. When it is time to deliver, your system would export an XML file, which the data reporter uploads to CDR. In this case, go to chapter 2.

To make it possible for the data providers to find your website, we set the principle repository field in ROD to point to your website for the obligation(s) you handle deliveries for.

The next issue is that the requesters need to know when there have been made updates to the data. There are two methods for that:

1. Provide an RDF feed of deliveries. It will be harvested every night by the Content Registry, and will show up as a list of deliveries for the obligation(s) in ROD.
2. Send a notification to the unified notification service (UNS) and everyone who's subscribed will get an email.

10.1. A note about traceability

Presumably; something will be produced from the data you have in the repository. It can be a report, website or dataset to name a few examples. That product will only be an opinion unless it is possible to audit that the conclusions it draws are supported by the data. That audit can happen any time in the future – months or years from the publish date. It is the responsibility of the report producer to state what data they retrieved from your repository and when. It is your responsibility to ensure it is possible to get exactly the same data again at a later date.

This conflict between wanting to provide always the newest and making old data available is also an issue for online newspapers, blogs, bulletin boards – any system that generates dynamic content. Industry best-practice is to have both dynamic and persistent/archival URLs for the same content.

10.2. RDF feed of deliveries

As mentioned above, you should provide an RDF feed of deliveries. It is a list of the files the requester must download from your site to get the full set of data. If he can get it all in one database extraction, then there is only one delivery. If the member states can release their data independently of each other then consider making it available as 27 deliveries. Especially if the countries want a copy of their own data.

If your data collection is periodic, there will be deliveries yearly or some other interval. In that case you must also make the old data available, because it has historical value. If you don't know what you measured last year, then you don't know if you're improving or deteriorating.

The RDF for a delivery looks like this:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:dc="http://purl.org/dc/elements/1.1/"
         xmlns="http://rod.eionet.europa.eu/schema.rdf#">
 <Delivery rdf:about="http://cdr.eionet.europa.eu/it/eu/WID/envsvgl5g">
  <dc:title>Waste Incineration Directive - Report from Italy</dc:title>
  <dc:description>Questionario relativo alle relazioni concernenti
l'applicazione della direttiva 2000/76/CE</dc:description>
  <released rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime">2009-11-
05T15:15:38Z</released>
  <link>http://cdr.eionet.europa.eu/it/eu/WID/envsvgl5g</link>
  <locality rdf:resource="http://rod.eionet.europa.eu/spatial/19" />
  <period>2006-01-01/P3Y</period>
  <obligation rdf:resource="http://rod.eionet.europa.eu/obligations/59"/>
  <hasFile rdf:resource="http://cdr.eionet.europa.eu/it/eu/WID/env-
svgl5g/Questionario_Dec2006_329_CE.xls"/>
 </Delivery>
</rdf:RDF>
```

If there are more than one delivery, you just repeat the Delivery element in the XML file.

For more information see the document Implementing a Reportnet/SEIS node.

10.3. Authentication of users

We strongly recommend to use the Eionet site directory to authenticate your users. With over 8.000 registered users there is a large chance that your data providers already have an account. If not we have a helpdesk to create accounts for them.

The site directory is an LDAP database which is available from the Internet. You connect and attempt to bind (log in) with the user name and password of the visiting user. If you succeed, then you are authenticated. Once you are logged in, you can get first name, last name, email etc. for the user.

10.4. Sending notifications

Notifications can be *pushed* into UNS via its webservice. This will ensure almost immediate notification. You can also create an RSS 1.0 feed, which will be harvested by UNS every hour or so. Every new item occurring on the feed will generate a notification. Items that are dropped from the feed will do nothing. Whether an item is a new item is determined by the URL in the rdf:about attribute.

10.5. File formats

You have your data in a database. How to make it available for the public? In principle any format you know your audience can read is allowed. But Reportnet prefers XML. This is because we're building tools to do automatic QA of XML files, which can then report on errors in the file. The QA happens when the deliveries are reported in the RDF feed as described in section 10.2. We can also convert XML to various formats, of which one is Excel.

RDF is a possibility, but the Reportnet tools are not mature enough yet to be useful.

To activate the QA and the conversion systems, your XML must follow a schema, and the instance file must have in its root element an *xsi:noNamespaceSchemaLocation* pointing to the schema location.