Project N°: 262608

Acronym: Data without Boundaries

Deliverable D12.3

(Harvesting and Ingest System)

Work Package 12

(Implementing Improved Resource Discovery for OS Data)

Reporting Period:	**From: Month 18**	**To: Month 48**
Project Start Date:	**1st May 2011**	**Duration: 48 Months**
Date of Issue of Deliverable:	**1st June 2014**	
Document Prepared by:	**Partner 6, 16, 18**	**NSD, MT, UEssex**

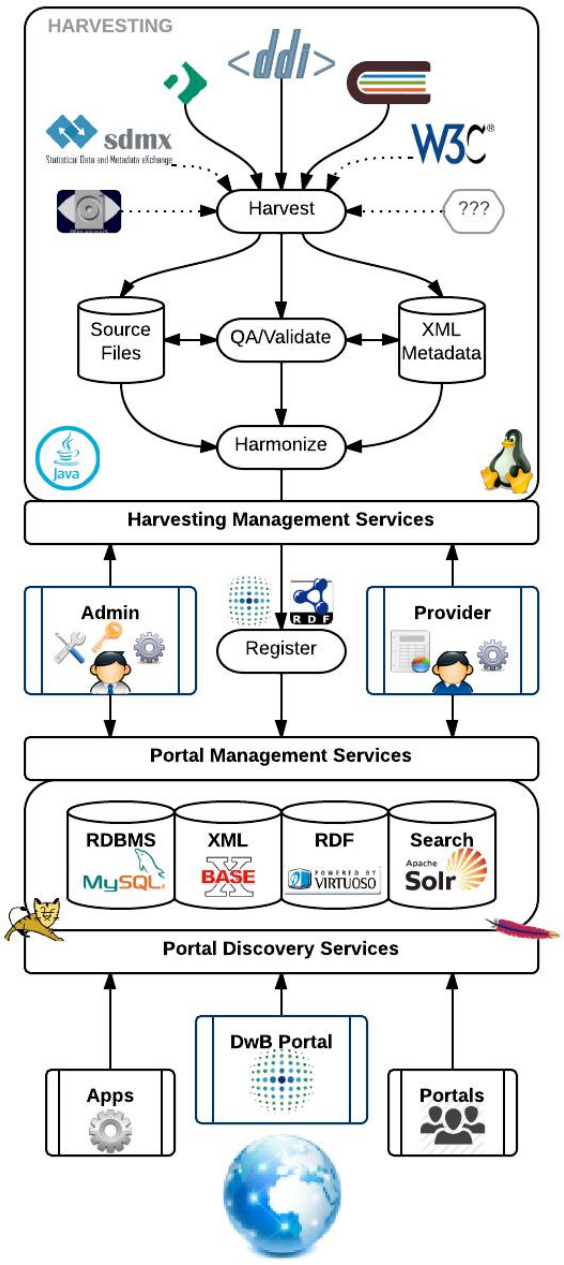
Combination of CP & CSA project funded by the European Community
Under the programme "FP7 - SP4 Capacities"
Priority 1.1.3: European Social Science Data Archives and remote access to Official Statistics

The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 262608 (DwB - Data without Boundaries)

TABLE OF CONTENTS

0. Overview

The diagram on the right provides a high level overview of the architecture of the DwB OS Resource Discovery Portal being implemented under WP12. It has been designed based on inputs from other work packages (particularly WP8 and WP5), consultations with domain experts, community trends and best practices, as well as internal research.



It is composed of the following components:

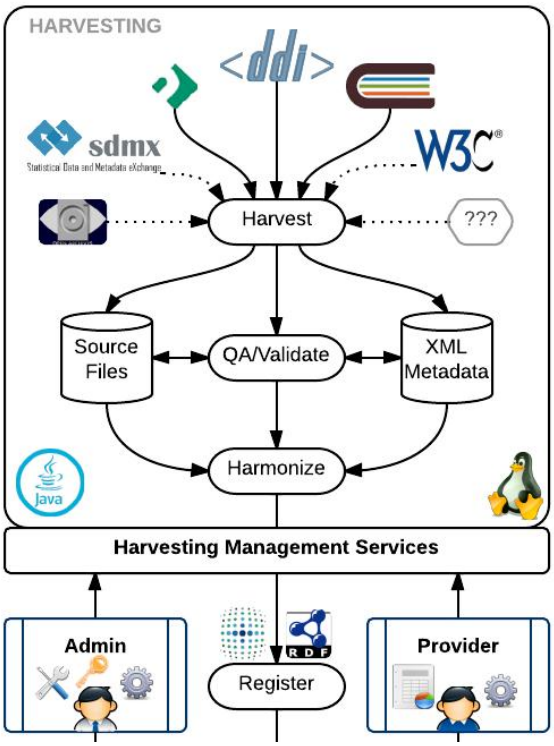
- The Harvesting Framework and related tools to facilitate the retrieval and ingest of metadata in various formats from participating providers.
- An Administrator Dashboard and underlying management services to configure and monitor the infrastructure, and orchestrate the various tasks.
- A Provider Portal enabling participating agencies to gain insights on their information present in the system. This include reports (quality assurance, usage, harvesting) and tools to control their profile, visibility of metadata, or profile information.
- The Storage Framework (databases) where the information is hosted and indexed in various shapes and form (Relational, XML, RDF) in order to support the search, discovery and other services. The metadata is based on commonly used standards and the DwB Metadata Model.
- The Discovery Services exposing the platform and information to the outside world, and enabling integration in applications or web site, along with the Discovery Portal user interface.

The implementation leverages several open sources technologies and packages, chosen for their robustness. The portal prototype implementation is woven around them in an innovative fashion to deliver the foundations of an enterprise grade scalable solution.

This document focuses on the Harvesting and Ingest System and is Part 3/5 of the project deliverable.

1. Objectives

- The purpose of ingest is to import and synchronize, on a regular basis, metadata in various shapes and forms from external sources made available by providers for publication in the portal.
- Such processes should preferably be fully automated.
- At the end of the ingestion process, the metadata is in a single standard harmonized format (DwB Discovery RDF, aka 'DwB-Disco') that can be registered in the database and indexed for search and retrieval purposes.
- As early as possible in the ingest process, the metadata will be transformed into a source-agnostic format to minimize the need for source-specific tools.



What can be ingested?

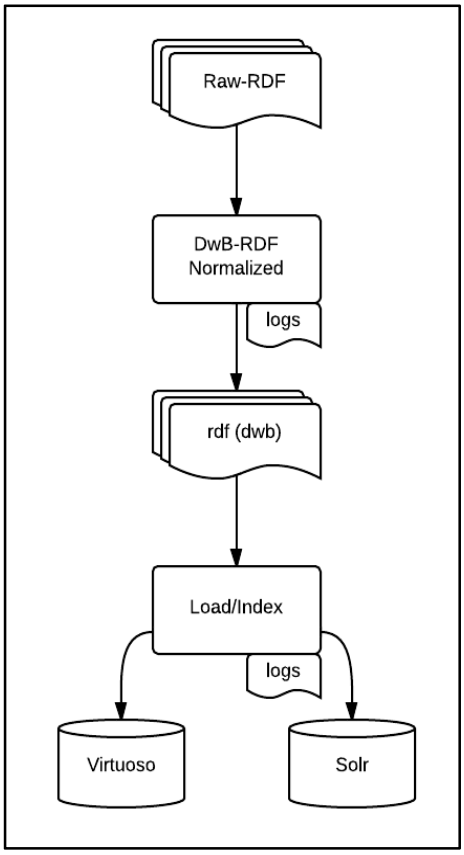
- In theory, any format that can be mapped to DwB-Disco can potentially be ingested by the discover portal.
- However, a minimal set elements must be present to ensure that: data can be properly indexed; we can display relevant/useful content to the end user.
- Candidate metadata formats include DDI, SDMX, OAI, and the like.
- Note that the term (data) 'source' is used here to mean a combination of provider and collection, where a provider is a combination of a country code and an agency, thus 'no.nsd.nsddata' is a source that has provider = no.nsd and collection = nsddata.

Ingestion workflow

The following table provides a high level overview of the envisioned ingest steps, indicating their internal process dependency (or agnosticity) in relation to specific standards, delivery mechanisms or source. This illustrates that by design, the workflow consist of a thin layer of standard/source-specific processes, and a majority of source-agnostic and reusable tasks.

Ingestion steps (*)	Source/standard
1) Harvesting	specific
2) Produce harvesting report	specific
3) Conversion to Raw-RDF	agnostic
4) Produce conversion report	agnostic
5) Normalization	agnostic
6) Produce normalization report	agnostic
7) Loading	agnostic
8) Produce loading report	agnostic
9) Indexing	agnostic
10) Produce indexing report	agnostic
11+) Discovery, other downstream processes	agnostic

(*) Orchestrated by Jenkins



2. Harvesters

The harvesting process is a "pull" mechanism whereby metadata is retrieved from a remote source for ingest. Such an operation typically takes place in an automated fashion by calling a webservice (SOAP, REST) or other API, or downloading files (e.g. over http, ftp, or other protocol). Each source is expected to expose data in a known harmonized/standard format.

As mentioned, the standards of interest for the DwB portal relate to descriptive metadata for statistical data and include DDI, SDMX, OAI, and the likes. For the implementation of the prototype portal, the focus is on supporting various versions and flavors of DDI, as it is used by most if not all potential providers identified so far.

Harvesters visit identified metadata sources periodically to identify added, modified or deleted metadata records, and to bring them into the portal. There are a number of harvesters at play since the different types of sources and their attendant access mechanisms (e.g. web services, file-servers, applications) behave differently and have different content.

The different harvesters are developed and packaged as libraries that can be built and used in the portal environment. Typically, harvesters have one platform/service specific part and one generic part. The generic part will be largely identical regardless of the platform/service being harvested.

For the time being, one specialized harvester component has been developed: “Nesstarvester” - a harvester for Nesstar servers. Nesstarvester is a specialized harvesting tool leveraging the Nesstar public API and enabling users to perform full or incremental download of a Nesstar server catalog or sub-catalog.

A component “BaseXSync” has also been implemented to push the harvested metadata (stored as XML files in a local folder) into a BaseX database. This generic tool can be used with any incoming XML metadata.

Harvesting tools for the collections and data formats listed below may be implemented during this project, assuming that access to associated web services or downloadable files becomes available in time for implementation:

- CIMES;
- MISSY;
- Colectica;
- Integrated European Census Microdata (IECM See: D12.6 and 12.7);
- SDMX.

Other formats/source types not listed above, that may emerge from unconfirmed or yet to be identified providers, will be considered, assuming such integration effort is within the capacity of this work package.

3. Transformers

While standards in theory aims to address harmonization of the metadata content, we often observe significant differences when dealing with real world metadata. These variations can be broken down into three nested categories:

- *Standard level* - sources based on the various metadata standards, as mentioned above;
- *Version level* - within a standard, the use of different versions (e.g. DDI 1.2.2, 2.5, 3.x, discovery);
- *Template/flavour level* - the use of elements of the standard for different purposes and the presence/absence of optional elements, which can be driven by institutional practices, templates, or software being used. DDI-L also comes with different ways to serialized identifiers and references (using URNs or structure XML), which further plays into this aspect.

Amongst these variations, the one that is the most resource intensive to address is the flavour level, as this is often provider, department or even individual specific.

Example of variations we have identified so far for the portal include:

- DDI-C
    - DDI-C Nesstar (1.2.2)
        - Flavours: NSD / UKDA / FORS / etc.
        - DDI-C CIMES (with additional non-DDI-info on Series. Only series + study level metadata)

- DDI-L
    - DDI-L Colectica 3.1
        - Flavours: Denmark
    - DDI-L Colectica 3.2
    - DDI-L SledgeHammer 3.1 or 3.2
    -
- DDI-Disco
    - DDI Disco MISSY

In addition to these "structural" variations, we also have "content" variations, whereby elements of information are filled differently by different providers. This is particularly challenging for element and attributes that are expected to be based on controlled vocabularies where we often see significant variations. It is also further complicated by the multilingual aspect of the DwB environment. These issues are further discussed in the enhancements and harmonization stages below.

Finally, bringing together metadata from multiple sources also raises the issue of uniquely identifying entities, but this can somewhat be addressed by assigning unique identifiers upon ingest and maintaining mappings to source-specific identifiers. This however assumes that the source identifier does not change over time (which is normally the case, but not always). As each provider typically assigns its own identifiers to objects, it is also difficult to determine whether two entities represent the same thing (e.g. same survey or variable).

The ingest workflow must take all of the above into account and reflect these layers in its design. Our solution is therefore designed in a very modular fashion and is composed of independent specialized tools, chained with each other, to turn incoming metadata into a single raw DwB-Disco format, ready for further processing by a single pipeline for enhancements and registration.

Transformation to Raw-RDF

As mentioned above, input to the discovery portal will be heterogeneous. This heterogeneity will be dealt with at an early stage through the transformation to Raw-RDF. The Raw-RDF is a common denominator for metadata in the portal, which all incoming metadata will be transformed in to. This transformation of incoming metadata into a common, canonical form greatly simplifies downstream processing (including enhancement, harmonization and indexing) of metadata. A benefit of RDF compared to e.g. XML is that RDF is semantically strong, and designed for linkage.

The ontology (model) of the Raw-RDF builds upon the ontology developed under the DDI Discovery Vocabulary (DISCO - http://www.ddialliance.org/Specification/RDF/Discovery). DISCO can be viewed as an RDF-based subset of DDI, designed to support discovery and search, and it fits both DDI-C and DDI-L formats. Mappings from DDI-L to DISCO existed prior to this deliverable, but mappings from DDI-C to DISCO were produced as part of DwB D12.3.

As part of the mapping process, inventory reports were produced for a number of Nesstar Servers (at UKDA, NSD, FORS and Statistics Canada) currently in production. The rationale for this was to try to identify frequently used metadata elements that are usable for discovery, but not currently part of the

DISCO vocabulary. This is in order to extend the Raw-RDF ontology beyond the DISCO one.

A set of transformers must be developed to convert the different flavours of input metadata into Raw-RDF to ensure that the Raw-RDF is reasonably homogenous before it is sent downstream for further processing.

DwB-Disco Normalizing

To improve the usefulness and discoverability of harvested metadata automatically, rule-based adjustments and enhancements often need to be applied to the metadata before ingestion. This is primarily to ensure consistency of the search features, in particular faceting.

The rules may be provider specific (e.g. always add “Norway” as the Nation-element for records from Norwegian providers unless the element is already populated) or more generic (e.g. standardize date-strings, spell-check, etc).

WP12 recognizes that such normalizing actions are potentially sensitive in nature, as the portal should not attempt to override a data provider’s documentation. Therefore, the enhancement and harmonization processes during the work of WP12 will primarily be used to demonstrate the use-case for such post-harvest processes.

It is thought that these experiments can not only provide guidance for the process flow of subsequent versions of the portal, but also be a source of useful feedback to providers in the form of suggestions for pre-harvest metadata enrichment (as the normalizing actions and their effects will be logged).

4. Registration

Registration in this context refers to the the following tasks:

- Validation - ensure that the final RDF file complies with DwB Portal specifications;
- Load - import the RDF into the Virtuoso database;
- Indexing - enable Solr to support test search and faceting;
- Other processes - to be identified (such as subscription/notification).

Registration takes places once metadata has completed the harvesting and harmonization / enhancement steps and is ready for publication.

Validation

This step in to ensure that the final DwB-Disco complies with all system requirements in terms of structure (valid RDF) and content (minimal metadata, use of valid terms in controlled vocabulary base elements, etc.)

In the prototype implementation, this step will be minimal as not metadata is coming from outside our

harvesting system. It is however conceivable that down the road, providers could directly submit metadata in DwB-Disco format to the portal, bypassing the need for harvesting and harmonization.

Loading the Database

Once the final version of the DwB-Disco file has been produced, it will be loaded in the Virtuoso RDF database for storage, retrieval, and querying. This step will simply take the RDF-XML files and load them into the database over a JDBC connection, as well as removing any entries that no longer exists. Once loaded, it becomes available to applications through the Virtuoso SPARQL end-point service, and accessible for rendering by the portal user interface.

Indexing for Search

Once stored in the database, the metadata also needs to be indexed by the search engine. This process will convert the DwB-Disco in to a standard Solr document format using a transform and submit it to the Solr server via the REST service. The indexing process is driven by the Solr core definitions which describe the fields to be indexed for full text search and faceting. At this point, the metadata becomes discoverable through the portal or Solr REST service.

Other steps

Down the road, additional steps could be added to the registration process. In particular, various notification services could be put in place to inform data providers that their metadata has been updated, or users that new content is available.

5. Reporting

Throughout the harvesting and related post-processing task, reports will be generated for the purpose of monitoring the internal processes, as well as providing content and quality assurance reports to the data providers or public users.

Various report-generation mechanisms will be used for implementation, such as XSL transformations, XQuery (leveraging BaseX), SPARQL (Virtuoso), or even REST (Solr). The outputs may result in static HTML reports or web pages dynamically produced from the various database contents.

Access to internal or provider-specific reports will be controlled by user/password or other authentication/authorization mechanisms.

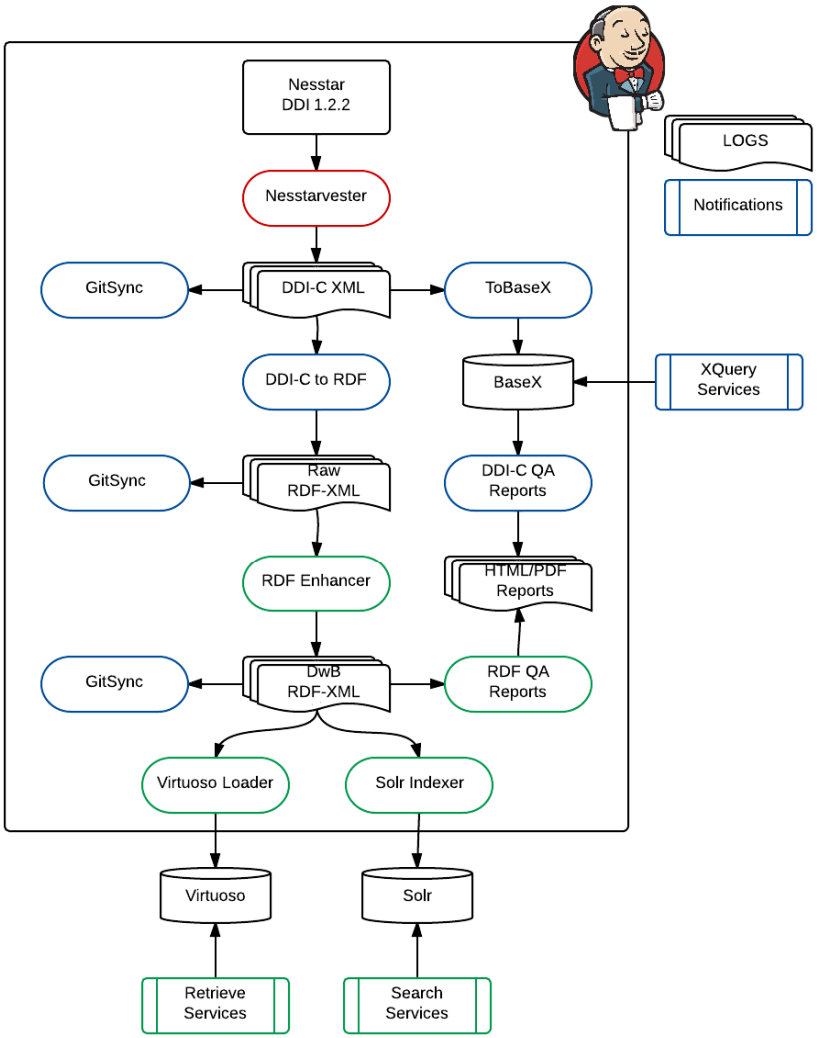
Example of reports include:

- Ingestion reports
    - Ingestion errors/issues;
- Provider Reports
    - List of titles of ingested metadata records from given providers, along with dates, number of variables;

    - List of removed titles (i.e. data sets that have been removed from the source and therefore also from the portal holdings);
    - List of enhancements;
- Public Reports
    - Aggregate reports of ingestion activity for collections from all providers;
    - Growth reports (quantification of holdings, geographic and temporal coverage).

6. Orchestration

A high level view of the ingest workflow is illustrated below, using a Nesstar server/Nesstarvester use case as an example:



The workflow can be broken down the following stages:

1. Harvesting from the source: collecting on a regular basis metadata from provider, resulting into new, updated or deleted content;
2. Conversion to raw RDF: transformation of the incoming metadata into a RDF serialization based on the DwB/DDI discovery model;

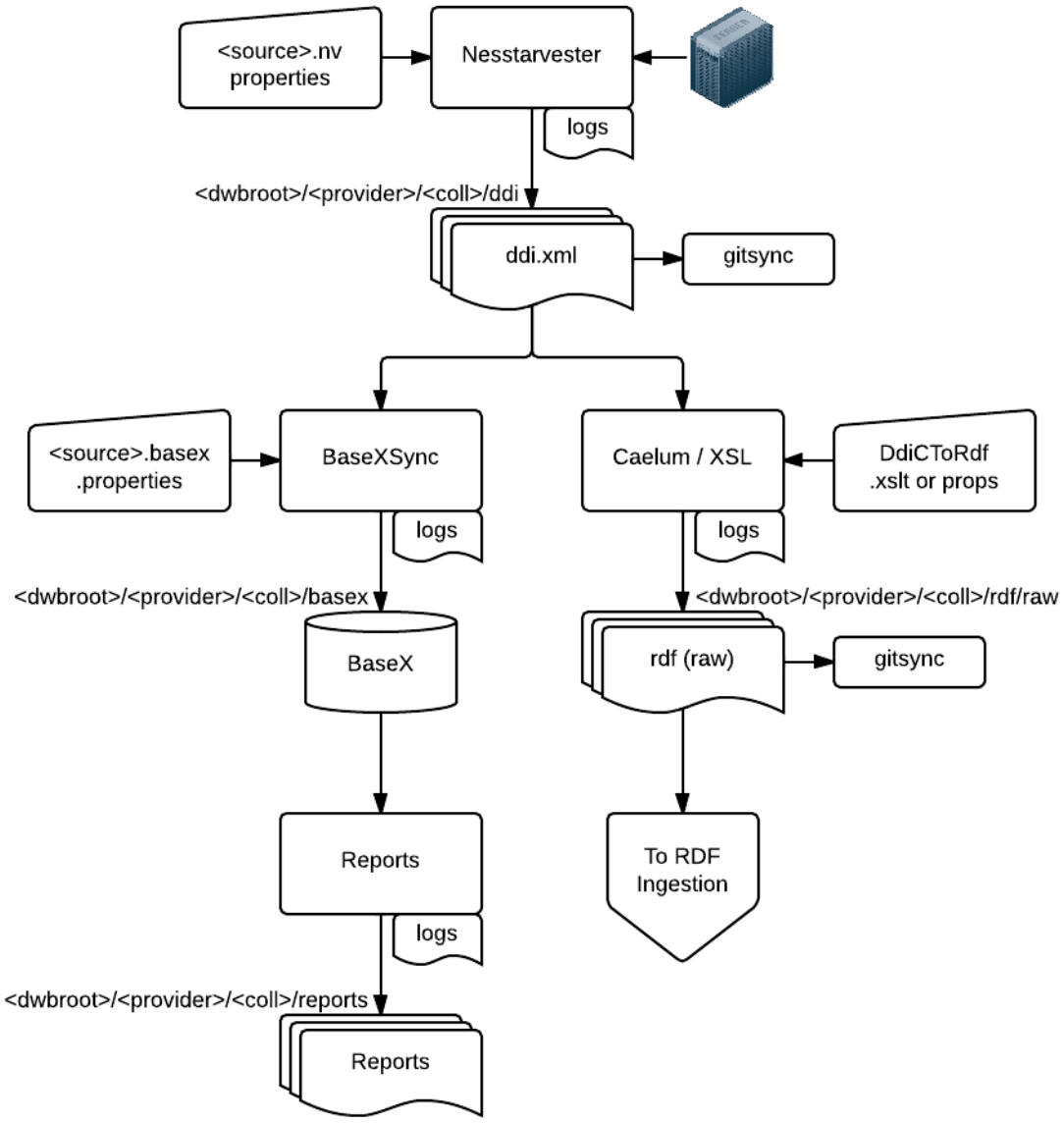
3. Harmonization and enhancements: further processing of the source specific content for alignment on the portal metadata elements driving search and presentation;
4. Registration: add/update/delete of harvested and processed documents into the database, and indexing by the search engine.

Along the way, all processes will result in:

- Files being created, updated, or removed from the system. To keep track of these file system changes over time, we propose to mirror these changes in a revision control environment[1] based on the GIT[2];
- Log files capturing the various activities, which we will keep for monitoring and auditing purposes;
- The production of reports for various audiences like system administrators, metadata providers, or portal users.

The figure below illustrates the workflow in more detail for the ingest of DDI-XML from a Nesstar server.

[1] http://en.wikipedia.org/wiki/Revision_control
[2] http://en.wikipedia.org/wiki/Git_(software)



Orchestration with Jenkins

The various processes involved in the ingestion are orchestrated by the open source Jenkins[3] package.

The basic unit of work in Jenkins is a job (i.e. a runnable task). Jobs are 'built' (i.e. run) so a successful build is said to occur when a job runs to completion without error. A job can be standalone, upstream or downstream of other jobs. A successful build of an upstream job triggers the build of its downstream jobs. This inherent causal execution behaviour of Jenkins allows workflow to be constructed easily and simply by creating and linking a number of jobs, and the built-in monitoring and display capabilities make it easy to detect trends, notify interested parties and examine reasons for failure. All this is done mostly by configuration rather than coding, which makes Jenkins a great environment for rapidly building and running workflows such as the one we have designed for the portal.

[3] http://jenkins-ci.org/

Workflow Activity

A workflow is made up of a number of activities, which are executed in some order. The basic anatomy of an activity (as used here) is a follows:

- Jenkins job;
- Ant script;
- Executable code.

Jenkins job

The Jenkins job is responsible for calling an Ant script. The job may pass command line parameters to the Ant script (as required). Any Jenkins job can be invoked as a REST web service, so parameters can be passed in the URL. The Jenkins job also takes care of email notification of interested parties on success and/or failure of the build, trend reporting, console output logging and calling any downstream jobs.

Ant script

Whilst it is not obligatory to use Ant as the scripting language, it provides platform independence and a simple mechanism for testing the executable with various parameter combinations outside of the Jenkins environment. That way we can be sure that the individual workflow activities work as expected, before combining them into a workflow.

Also the use of targets provides additional flexibility, as one or more targets can be called by the Jenkins job in any order. Typically we call a 'print environment' target for debugging purposes, so we can check that file paths etc are set up as expected.

Executable code

This is the real workhorse, with the Ant script acting as a wrapper and the Jenkins job acting as a User Interface component. Typically it is a Java JAR file that accepts command line components (such as environment variables telling it where to get and/or put data), but can be any executable type that Ant supports. The executable must incorporate basic error handling and throw some sort of exception if it does not run to completion, so an error code can be returned to Jenkins (via the Ant script). Otherwise Jenkins will assume success and call any downstream jobs (which are likely to fail as they probably depend on the output from the current job).

Workflows

So now we have a mechanism to execute and report on the success or otherwise of individual steps, we can combine them into a workflow.

We use two basic mechanisms for executing the Jenkins jobs - external dependencies and internal dependencies.

External Dependencies

A Jenkins job can be triggered by an event outside of the Jenkins environment causing its URL to be called. Typical examples are post-commit hooks in source code management systems such as Subversion or GIT, which use CURL or similar mechanisms to call a job that rebuilds the executable from source, everything a change is committed.

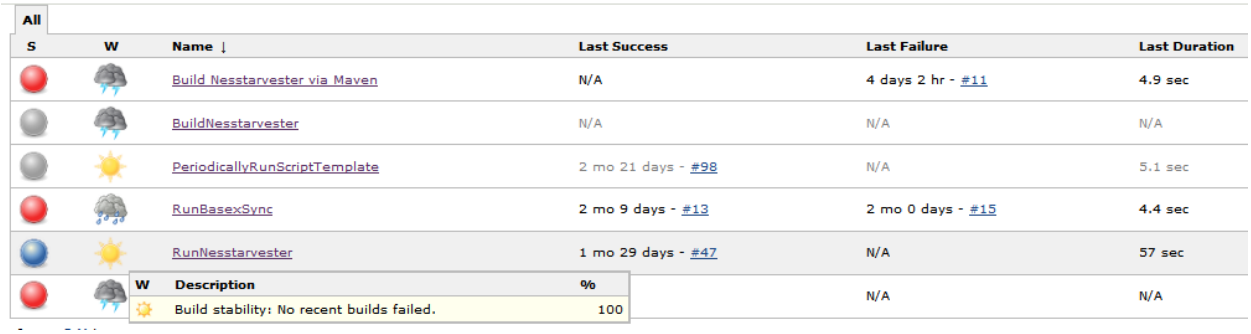
For example, the URL of the Jenkins job that rebuilds the Nesstar metadata harvester ('Nessarvester') is http://localhost/jenkins/job/Build%20Nesstarvester%20via%20Maven/. This mechanism is used to ensure that the workflow is always using the latest versions of the executables.

Internal Dependencies

Internal dependencies are modelled using the upstream/downstream relationships described above. Thus a simple workflow can be constructed using a network of upstream and downstream jobs. Executing the first job causes forward chaining until completion of the workflow (unless some job does not build successfully). Upstream jobs typically produce outputs that downstream jobs are dependant on, thus Nesstarvester deposits metadata files in a location that BaseXSync reads from etc.

Jenkins Dashboard

Much of the reporting is done automatically via Jenkins. Out of the box it provides a rich set of User Interface components that show the health of the various workflow activities.



Workflow Activity Health Checks

From left to right the columns show for each job: status (blue for success, red for failure, grey for disabled); weather report (aggregated status, i.e. recent build trend); last success (absolute time since last successful build); last failure (absolute time since last failed build); last duration (absolute time taken for most recent build to complete).

Executable Build Change Logs

The figure below shows a change log for a build of the Nesstarvester executable, as well as audit trail details (when it was last run and by which user).

Build #47 (Feb 6, 2014 4:51:13 PM)

Changes

1. Added properties files for Nesstarvester and BaseXSync (detail)
2. Current location for ddi output is not visible between Jenkins builds, (detail)
3. Revert "Current location for ddi output is not visible between Jenkins builds, so tried moving it to different location
4. As suspected, unable to create directory above Junkins installation dir. (detail)
5. Renamed shellScript folder into reources (detail)
6. Changed 'esds' to 'ukdataservice' (detail)
7. Removed trailing spaces from URLs (as port number has to be appended) (detail)
8. Temporary hard-code of path to output folder, pending introduction of (detail)
9. Updated value of prop.dir in line with restructured repository (detail)
10. removed temporary full path hardcode as unable to create ddi directory (detail)
11. Redmine Issue #2101 (detail)
12. Change value of prop.dir (detail)
13. Prepended ${input_dir} to value of outputFolder (detail)
14. Updated value of input_dir (detail)
15. Updated value of input_dir (detail)
16. Nasty hard coded path name used for outputFolder value - for test purposes only (detail)
17. Wrong file - should be updating basex properties file! (detail)
18. Nasty hardcode of path to syncFolder in lieu of environment variable shared by Jenkins and BaseXSync (detail)
19. Replaced 'workspace' with 'job' (detail)
20. Added trailing slash to syncFolder path (detail)
21. Added more binaries (detail)
22. Basic log4j properties to try to prevent warnings from Harvester and (detail)
23. Uses env variable defined in Ant file (to specify location of harvested (detail)
24. Explicitly echo value of input directory, for debug (detail)
25. Added output_dir variable so can test if Nesstarvester can evaluate it (detail)

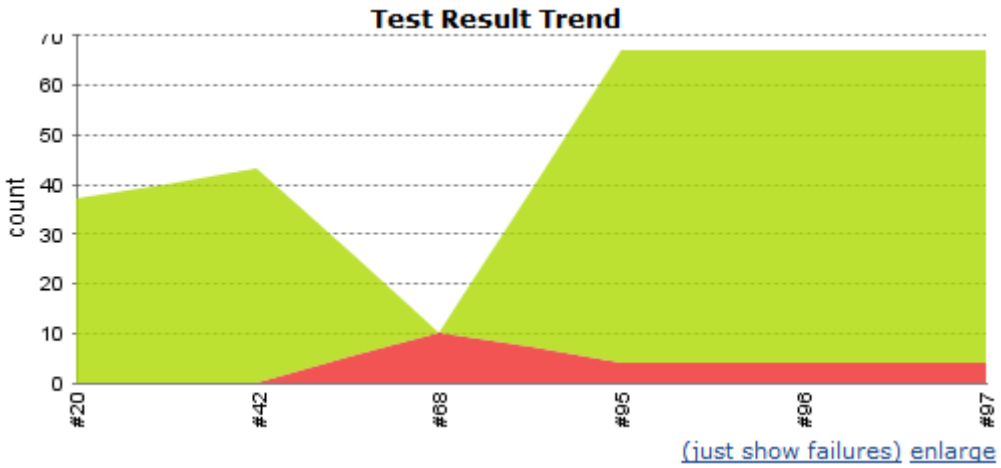
Started by user John Shepherdson

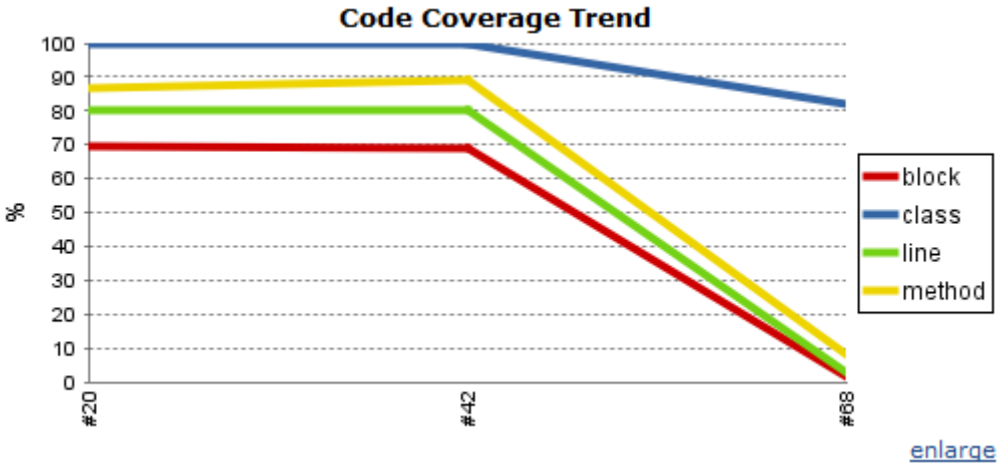
Revision: 02f15bb82da79e4a53b09bc3359a192f897e139c

- origin/master
- origin/HEAD

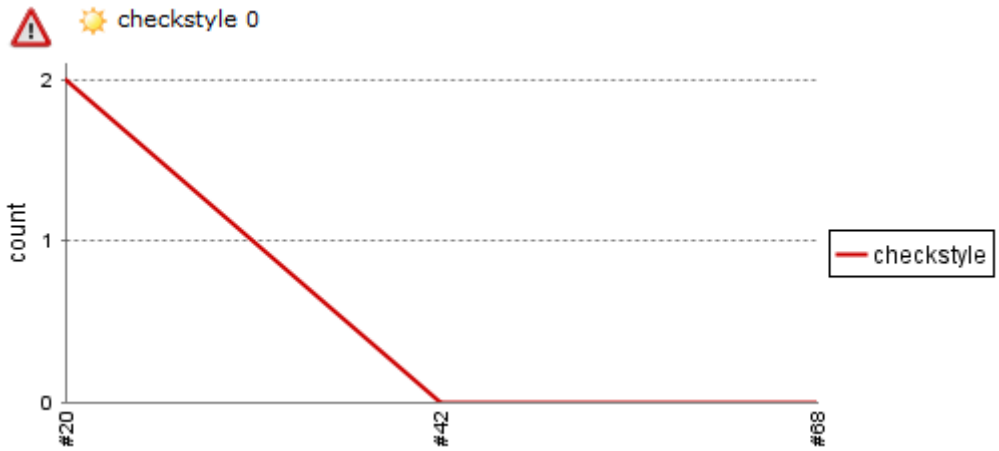
Executable Code Quality Assurance

Various plugins are available to show code QA metrics, such as unit test results trends, test coverage and adherence to coding standards, as shown in the figures below.




Code Coverage Trend
100
90
80
70
60
50
40
30
20
10
0
%
#20
#42
#68
block
class
line
method

enlarge


checkstyle 0
2
1
0
count
#20
#42
#68
checkstyle