Continuous Integration for XML and RDF Data

Sandro Cirulli

Oxford University Press

<sandro.cirulli@oup.com>

Abstract

At Oxford University Press we build large amounts of XML and RDF data as it were software. However, established software development techniques like continuous integration, unit testing, and automated deployment are not always applied when converting XML and RDF since these formats are treated as data rather than software.

In this paper we describe how we set up a framework based on continuous integration and automated deployment in order to perform conversions between XML formats and from XML to RDF. We discuss the benefits of this approach as well as how this framework contributes to improve both data quality and development.

Keywords: Jenkins, Unit Testing, Docker

1. Introduction

Oxford University Press (OUP) is widely known for publishing academic dictionaries, including the Oxford English Dictionary (OED), the Oxford Dictionary of English (ODE), and a series of bilingual dictionaries. In the past years OUP acquired a large number of monolingual and bilingual dictionaries from other publishers and converted them into the OUP XML data format for licensing purposes. This data format was originally developed for print dictionaries and had to be loosened up in order to take into account both digital products and languages other than English. Conversion work from the external publishers' original format was mainly performed out-of-house, thus producing a large number of ad hoc scripts written in various programming languages. These scripts needed to be re-run each time on in-house machines in order to reproduce the final XML. Code reuse and testing were not implemented in the scripts and external developers' environments had to be replicated each time in order to rerun the scripts.

As part of its Oxford Global Languages (OGL) programme [1], OUP plans to convert its dictionary data from a print-oriented XML data format into RDF. The aim is to link together linguistic data currently residing in silos and to leverage Semantic Web technologies for discovering new information embedded in the data. The initial steps of this transition have been described in [2] where OUP moved from monolithic, print-oriented XML to a leaner, machine-interpretable XML data format in order to facilitate transformations into RDF. [2] provides examples of conversion code as well as snippets of XML and RDF dictionary data and we recommend to refer to it for understanding the type of data modelling challenges faced in this transition.

Since the OGL programme aims at producing lean XML and RDF for 10 different languages in its initial phase and for tens of languages in its final phase, the approach of converting data with different ad hoc scripts would not be scalable, maintainable, or cost-effective. In the following chapters we describe how we set up a framework based on continuous integration and automated deployment in order to perform conversions between XML formats and from XML to RDF. We discuss the benefits of this approach as well as how this framework contributes to improve both data quality and development.

2. Continuous Integration

Continuous Integration (CI) refers to a software development practice where a development team commits their work frequently and each commit is integrated by an automated build tool detecting integration errors [3]. In its simplest form it involves a build server that monitors changes in the code repository, runs tests, performs the build, and notifies the developer who broke the build [4] (p. 1).

2.1. Build Workflow

We adopted Jenkins [5] as our CI server. Although we have not officially evaluated other CI servers, we decided to prototype our continuous integration environment with Jenkins for the following reasons:

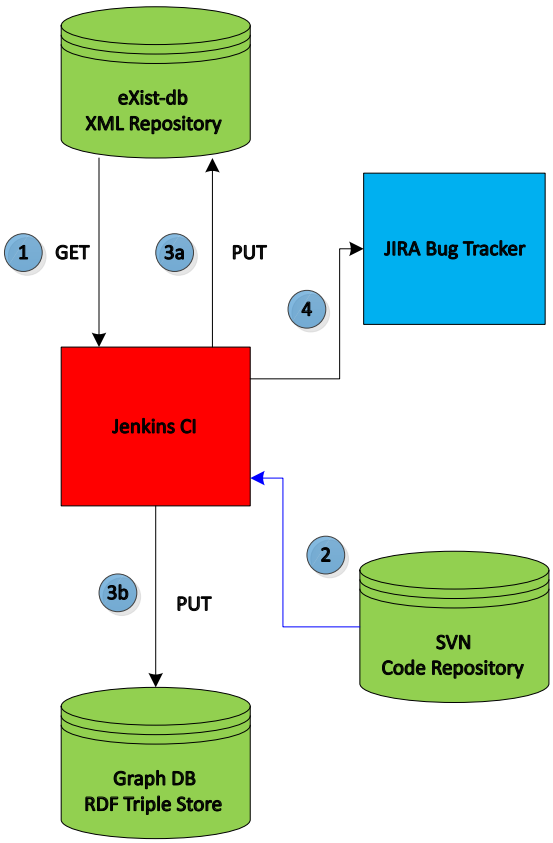
doi:10.14337/XMLLondon15.Cirulli01

- it is the most popular CI server with 70% market share [6]
- it is open source and allows to prototype without major costs
- it is supported by a large number of plugins that extend its core functionalities
- it integrates with other tools used in-house such as SVN, JIRA, and Mantis

Nevertheless, we reckon that other CI servers may have equally fulfilled our basic use cases. On the other hand, specific use cases may require different CI servers: for example, Travis CI may be a better choice for open source projects hosted on GitHub repositories due to its distributed nature whereas Bamboo may be a safer option for businesses looking for enterprise support in continuous delivery.

Figure 1, "Workflow and components for converting XML and RDF" illustrates the workflow and the components involved in converting and storing XML and RDF data via Jenkins.

Figure 1. Workflow and components for converting XML and RDF

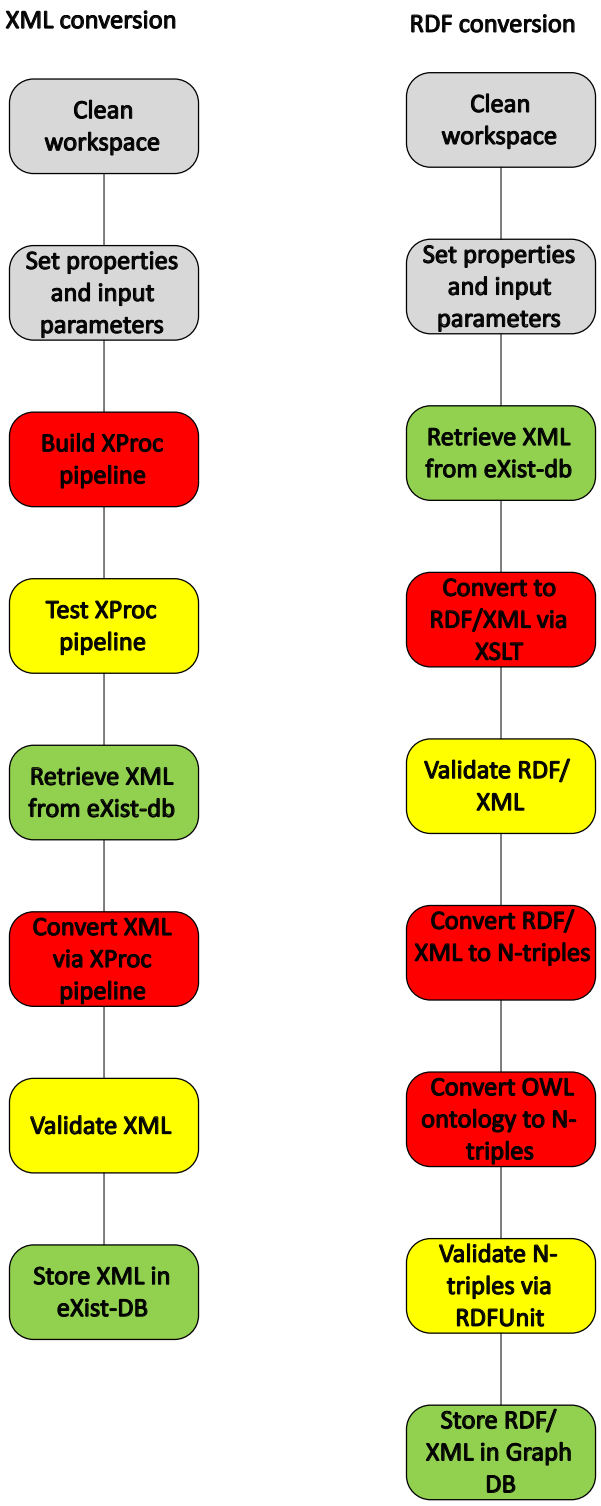


XML data in print-oriented format is stored on the XML repository eXist-db. The data is retrieved by Jenkins via a HTTP GET request (1). Code for converting print-oriented XML and building artifacts is checked out from the Subversion code repository and stored in Jenkins's workspace (2). The build process is run via an ant script inside Jenkins and the converted XML is stored in eXist-db (3a). Should the build process fail, Jenkins automatically raises a ticket in the Jira bug tracking system (4).

The same workflow occurs in the RDF conversion. XML data converted in the previous process is retrieved from eXist-db (1), converted by means of code checked out from Subversion (2), and stored in the RDF Triple Store Graph DB (3b).

The core of the build process is performed by an ant script triggered by Jenkins. Figure 2, "Build process steps for XML and RDF conversions" shows the steps involved in the build process for XML and RDF conversions.

Figure 2. Build process steps for XML and RDF conversions



2.2. Nightly Builds

Nightly builds are automated builds scheduled on a nightly basis. We currently build data in both XML and RDF for 7 datasets and the whole process takes about 5 hours on a Linux machine with 132GB of RAM and 24 cores (although only 8 cores are currently used in parallel). The build process is performed in Jenkins via the Build Flow Plugin [7] which allows to perform complex build workflows and jobs orchestration. For our project the XML ought to be built before the RDF and each build is parametrized according to the language to be converted. The Build Flow Plugin uses Jenkins Domain Specific Language (DSL), a Groovy-based scripting language. In this case we used this scripting language as it ships with the Build Flow Plugin and the official plugin documentation provides several examples of complex parallel builds. Example 1, "XML and RDF builds for English-Spanish data" shows the DSL script for building XML and RDF for the English-Spanish dictionary data.

Example 1. XML and RDF builds for English-Spanish data

```
out.println 'English-Spanish Data Conversion'
// build lexical XML full data
build( "lexical_conversion",
      source_lang: "en-gb",
      target: "build-and-store",
      build_label: "nightly_build",
      input_type: "oxbiling",
      input: "full",
      target_lang: "es")
// build RDF full data
build( "lexical_rdf_conversion",
      input_source: "database",
      source_type: "dict",
      source_language: "en-gb",
      target_language: "es",
      target: "update-rdf")
```

Builds are run in parallel and make use of the multi-core architecture of the Linux machine. For our current needs Jenkins is set to use up to 8 executors on a master node in order to build 7 datasets in parallel. Compared to a sequential build run on a single executor, the parallel build reduced by several hours the total execution time of nightly builds. In the future we foresee to increase the number of executors as we convert more datasets and to run nightly builds and other intensive process on slave nodes in order to scale horizontally.

2.3. Unit testing

Unit testing was originally included in the build process. However, since the builds took several hours before producing results, we decided to separate the building and testing processes in order to provide immediate feedback to developers. We created validation jobs in

Jenkins that poll code repositories on the SVN server every 15 minutes and run tests within minutes from the latest commit. Should tests fail, a JIRA ticket is assigned to the latest developer who committed code and the system administrator is notified via email.

Unit testing for XSLT code is implemented using XSpec [8]. [9] suggested the use of Jxsl [10], a Java wrapper object for executing XSpec tests from Java code. We took a simpler approach which does not require the use of Java code. XSpec unit tests are run within the ant task as outlined in [11] and the resulting XSpec HTML report is converted into JUnit via a simple XSLT step. Since JUnit is understood natively by Jenkins, it is sufficient to store the JUnit reports into the directory where Jenkins would expect them to be in order to take advantage of Jenkins's reporting and statistical tools. Example 2, "XSpec unit test" shows how all the XSpec HTML reports are converted into JUnit within an ant script.

Example 2. XSpec unit test

```
<for param="file">
  <path>
    <fileset dir="${test.dir}" includes="**/*.xspec"/>
  </path>
  <sequential>
    <echo>convert XSpec test results into JUnit XML</echo>
    <propertyregex override="yes" property="basename" input="@{file}"
                   regexp=".+[\\/]([^\\/]+?)\.xspec" replace="\1"/>
    <xslt in="${test.dir}/results/${basename}-result.html"
          out="${test.dir}/results/${basename}-result.junit"
          style="${shared.dir}/xsl/xspec_to_junit.xsl" force="true">
      <classpath location="${saxon.jar}"/>
    </xslt>
  </sequential>
</for>
```

RDF data is tested using the RDFUnit testing suite [12] which runs automatically generated test cases based on a given schema. The output is generated in both HTML and JUnit. Figure 3, "Report for RDFUnit tests" shows a screenshot of the HTML report (the top level domain has been hidden for security reasons).

Figure 3. Report for RDFUnit tests

TestExecution: http://rdfunit.aksw.org/data/results#f77bbf16-2ac9-11b2-8008-001018e2c9d0

Dataset	http://[redacted]/validation
Test suite	http://rdfunit.aksw.org/data/testsuite#f78f9cd2-2ac9-11b2-8008-001018e2c9d0
Test execution started	2015-03-09T06:22:09.999Z
-ended	2015-03-09T06:24:31.095Z
Total test cases	157
Succeeded	152
Failed	5
Timeout / Error	T:0 / E: 0
Violation instances	11

Results

Status	Level	Test Case	Errors	Prevalence
Success	WARN	http://[redacted]/ontology/hasTransliteration does not have rdfs:domain: http://languagehub.oup.com/ontology/String	0	0
Success	ERROR	http://[redacted]/ontology/hasGender has different range from: http://languagehub.oup.com/ontology/Gender	0	51951
Success	ERROR	http://[redacted]/ontology/hasInflection has different range from: http://languagehub.oup.com/ontology/Inflection	0	0
Success	WARN	http://[redacted]/ontology/hasWrittenForm does not have defined range: http://languagehub.oup.com/ontology/String	0	143086
Success	WARN	http://[redacted]/ontology/precededBy does not have defined range: http://languagehub.oup.com/ontology/DerivationalStep	0	0
Success	WARN	http://[redacted]/ontology/hasAbbreviation does not have rdfs:domain: http://languagehub.oup.com/ontology/LexicalForm	0	0
Success	ERROR	http://[redacted]/ontology/hasNote has different range from: http://languagehub.oup.com/ontology/Note	0	10640

As shown in Figure 2, "Build process steps for XML and RDF conversions", the XProc pipeline for the XML conversion is built on-the-fly during the build process from a list of XSLT steps stored in a an XML configuration file. This approach simplifies and automates the creation of XProc pipelines for new datasets: for example, developers converting new datasets have to create and maintain a simple XML file with a list

of steps rather than a complex XProc pipeline with several input and output ports. On the other hand, the generated XProc file needed to be tested and we therefore implemented unit tests using the xprocspec testing tool [13]. Example 3, "xprocspec unit test" shows a test that, given a valid piece of XML, expects the XProc pipeline not to generate failed assertions on the ports for Schematron reports.

Example 3. xprocspec unit test

```
<x:scenario label="test_fragment">
<x:call step="oup:main">
  <x:option name="source_lang" select="'@LANG@'"/>
  <x:input port="source">
    <x:document type="file"
                href="valid_fragment.xml"/>
  </x:input>
</x:call>
<x:context label="Schematron Validation">
  <x:document type="port"
              port="schematron_intermediate"/>
  <x:document type="port"
              port="schematron_final"/>
</x:context>
<x:expect type="xpath"
          test="count(//svrl:failed-assert)"
          equals="0"
          label="There should be no failed
                 Schematron assertions"/>
</x:scenario>
```

2.4. Benefits of Continuous Integration

Introducing Continuous Integration in our development workflow has been a big shift from how code used to be written and how data used to be generated in our department. In particular, we have seen major improvements in the following areas:

- **Code reuse**: on average, 70-80% of the code written for existing datasets could be reused for converting new datasets into leaner XML and RDF.
- **Code quality**: tests ensured that code is behaving as intended and minimized the impact of regression bugs as new code is developed.
- **Bug fixes**: bugs are spotted as soon as they appear, developers are notified instantly, and bugs are fixed more rapidly.
- **Automation**: removing manual steps made the building process faster and less error-prone.
- **Integration**: a fully automated building process reduced risks, time, and costs related to integration with existing and new systems and tools.

Figure 4, "Jenkins projects" and Figure 5, "Parametrized build" show respectively the list of Jenkins projects and a parametrized build inside the Lexical Conversion project. In Figure 4, "Jenkins projects" we illustrate on purpose a critical situation showing projects with failed builds in red, projects with unstable builds (i.e. failing unit tests) in amber, and project with successful builds in blue; the weather icon illustrates the general trend.

Figure 4. Jenkins projects

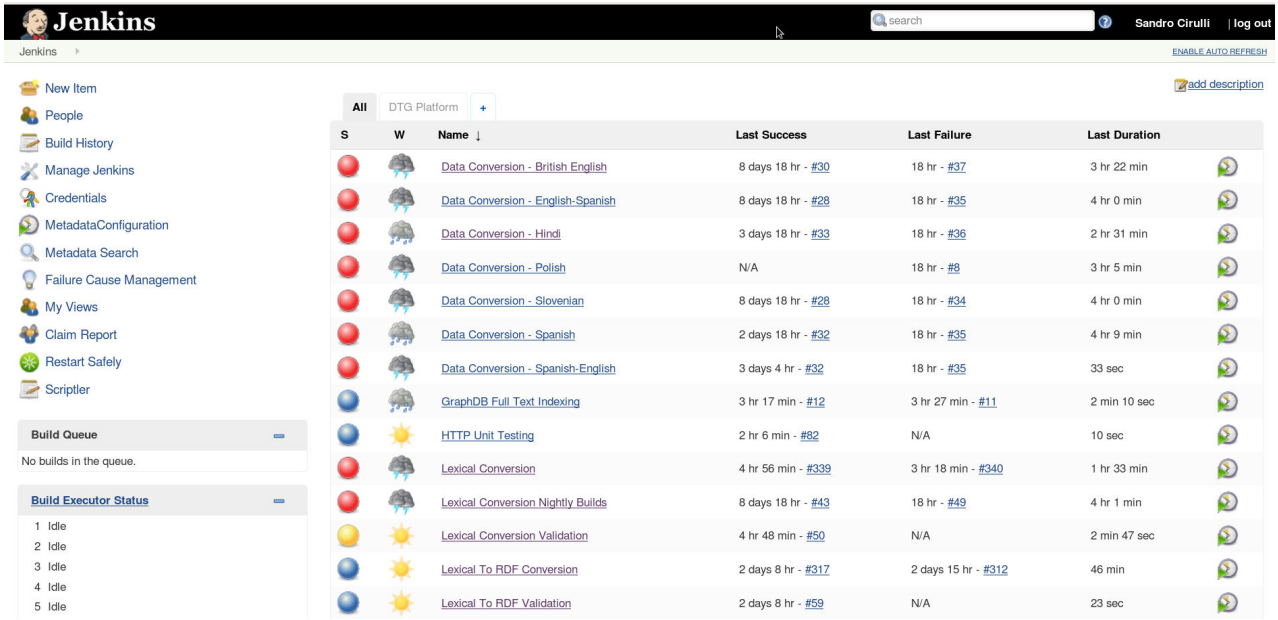
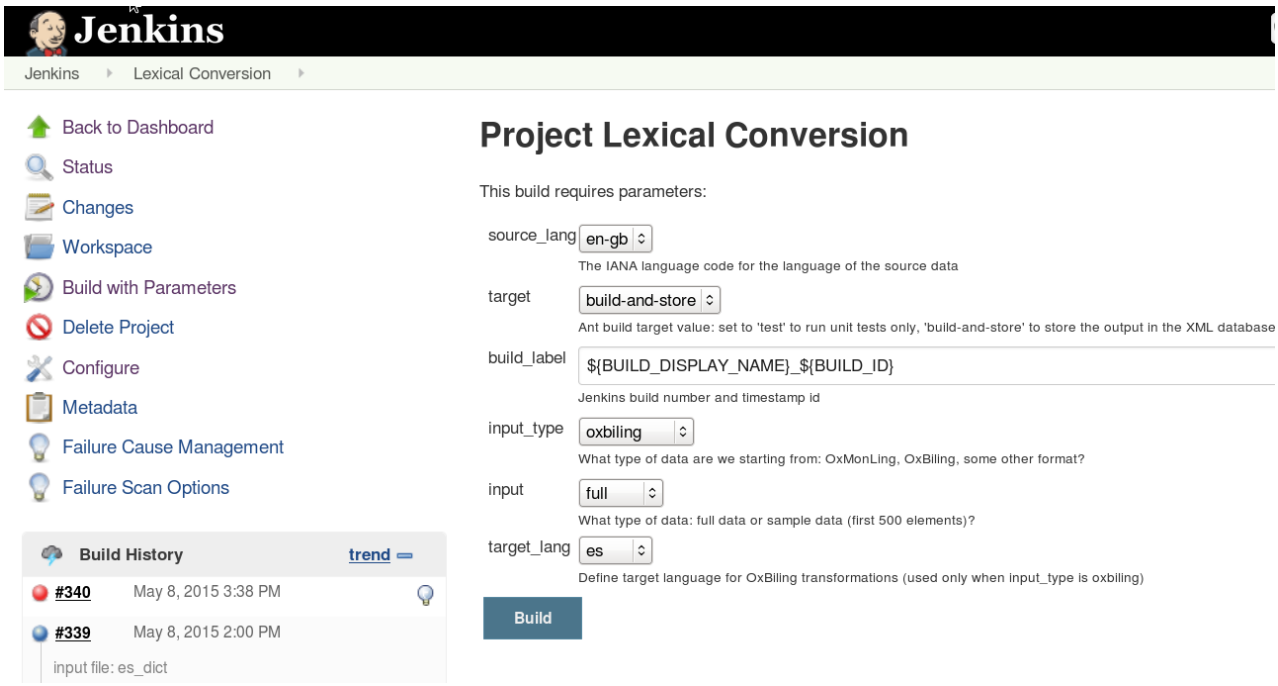


Figure 5. Parametrized build



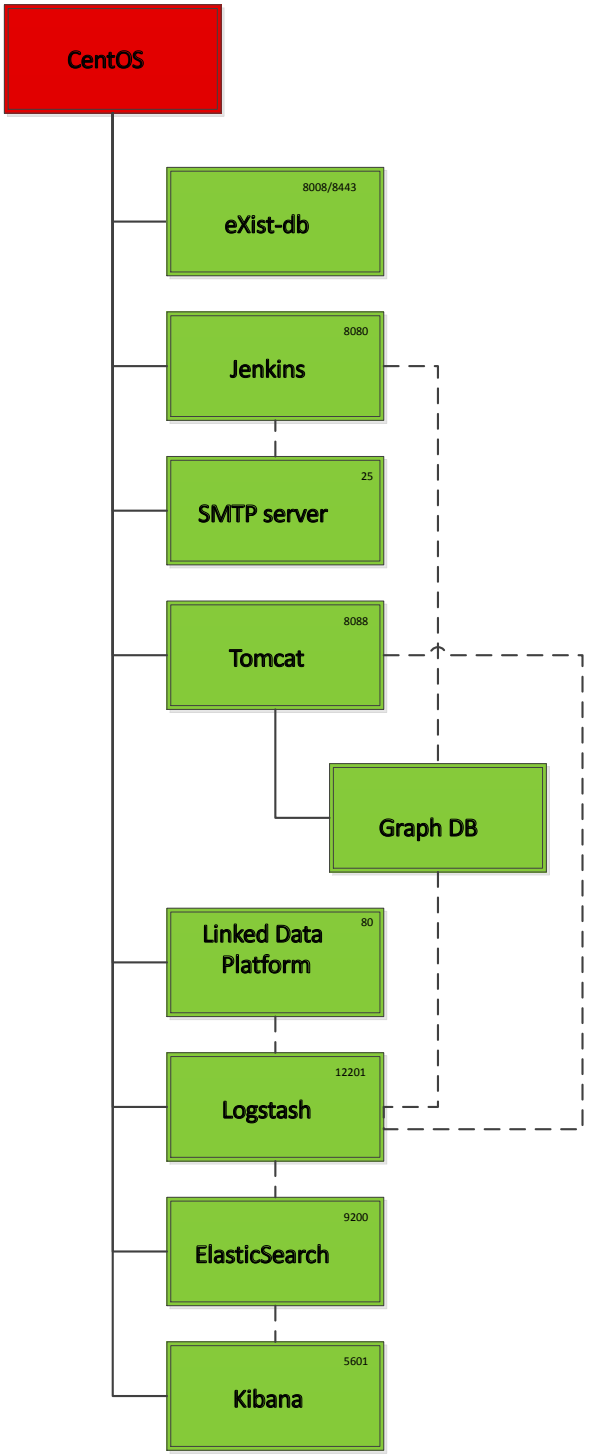
3. Deployment

One of the issues we faced when working with out-of-house freelancers is that their working environments needed to be replicated in-house in order to rerun scripts. Indeed, even within an in-house development team it is not uncommon to use different software tools and operating systems. In addition, the need of development, staging, and production environments for large projects usually causes integration problems when deploying from one environment to another.

In order to minimize integration issues and avoid the classic 'but it worked on my machine' problem, we picked up Docker as our deployment tool. Docker is an open source software for deploying distributed applications running inside containers [14]. It allows applications to be moved portably between development and production environments and provides development and operational teams with a shared, consistent platform for development, testing, and release.

As shown in Figure 6, “Docker Containers”, we based our environment on a CentOS image base. This container also deploys all the software tools employed by subsequent containers (e.g. Linux package utilities, Java, ant, maven, etc.). Separate ports are allocated to each component and re-deploying the components to a different port is simply a matter of re-mapping the Docker container to the new port. Graph DB is deployed as an application inside a Tomcat container. The Jenkins container is linked to the SMTP server container in order to send email notifications to the system administrator and to Graph DB for reindexing purposes. Most of the components send their logs to logstash which acts as a centralized logging system. Logs are then searched via ElasticSearch and visualized with Kibana. Software components like SVN and Jira are deployed on separate servers managed by other IT departments, therefore there was no need to deploy them via Docker containers.

Figure 6. Docker Containers



Example 4, "Dockerfile for deploying eXist-db" illustrates an example of Dockerfile for deploying eXist-db inside a Docker container. The example is largely based on an image pulled out from the Docker hub registry. The script exist-setup.cmd is used to set up a basic configuration (e.g. admin username and password).

Example 4. Dockerfile for deploying eXist-db

```
FROM centos7:latest
MAINTAINER Sandro Cirulli  <sandro.cirulli@oup.com>

# eXist-db version
ENV EXISTDB_VERSION 2.2

# install exist
WORKDIR /tmp
RUN curl -LO http://downloads.sourceforge.net/exist/Stable/${EXISTDB_VERSION}/eXist-db-setup-${EXISTDB_VERSION}RC2.jar
ADD exist-setup.cmd /tmp/exist-setup.cmd

# run command line configuration
RUN expect -f exist-setup.cmd
RUN rm eXist-db-setup-${EXISTDB_VERSION}RC2.jar exist-setup.cmd

# set persistent volume
VOLUME /data/existdb

# set working directory
WORKDIR /opt/exist

# change default port to 8008
RUN sed -i 's/default="8080"/default="8008"/g' tools/jetty/etc/jetty.xml

EXPOSE 8008 8443

ENV EXISTDB_HOME /opt/exist

# run startup script
CMD bin/startup.sh
```

4. Future Work

The aim of the OGL programme is to convert into lean XML and RDF tens of language datasets and our project is a work-in-progress that changes rapidly. Although we are in the initial phase of the project, we believe we have started building the initial foundations of a scalable and reliable system based on continuous integration and automatic deployment. We have identified the following areas of further development in order to increase the robustness of the system:

- **Availability**: components in the system architecture may be down or inaccessible thus producing cascading effects on the conversion workflow. In order to minimize this issue, we introduced HTTP unit tests using the HttpUnit testing framework [15]. These tests are triggered by a Jenkins project and regularly poll the system components to ensure that they are up and running. A more robust approach would involve the implementation of the Circuit Breaker Design Pattern [16] which early detects

system components failures, prevents the reoccurrence of the same failure, and reduces cascading effects on distributed systems.

- **Scalability**: we foresee to build large amounts of XML and RDF data as we progress with the conversion of other language datasets. As our system architecture matures, we also feel an urgent need to deploy development, staging, and production environments. Consequently, we plan to move part of our system architecture to the cloud in order to run compute-intensive processes such as nightly builds and to deploy different environments. Cloud computing is particularly appealing for our project thanks to auto-scaling features that allow to start and stop automatically instances of powerful machines. Another optimization in terms of scalability would be to increase the number of executors for parallel processing and to distribute builds across several slave machines.
- **Monitoring**: we introduced a build monitor view in Jenkins [17] that tracks the status of builds in real time. The monitor view also allows to display automatically the name of the developer who may have broken the last build, to identify common failure causes by catching the error message in the logs, and to assign or claim broken builds so that developers can fix them as soon as possible. We hope that this tool will act as a deterrent for unfixed broken builds and will increase the awareness of continuous integration in both our team and our department.
- **Code coverage and further testing**: we introduced code coverage metrics (i.e. the amount of source code that is tested by unit tests) for Python code related to the development of the Linked Data Platform and we would like to add code coverage for XSLT code. Unfortunately, there is a lack of code coverage frameworks in the XML community since we could only identify two code coverage tools (namely XSpec and Cakupan), one of which requires patching at the time of writing [18]. In addition, we plan to increment and diversify the types of testing (e.g. more unit tests, security tests, acceptance tests, etc.). Finally, in order to avoid unnecessary stress on Jenkins and SVN servers, we would like to replace the polling of SVN via Jenkins with SVN hooks so that an SVN commit will automatically trigger the tests execution.
- **Deployment orchestration**: the number of containers increased steadily since we started to deploy via Docker. Moreover, some containers are linked and need to be started following a specific sequence. We plan to orchestrate the deployment of Docker container and there are several tools for this task (e.g. Machine, Swarm, Compose/Fig).

5. Conclusion

In this paper we described how we set up a framework based on continuous integration and automated deployment for converting large amounts of XML and RDF data. We discussed the build workflows and the testing process and highlighted the benefits of continuous integration in terms of code quality and reuse, integration, and automation. We illustrated how the deployment of system components was automated using Docker containers. Finally, we discussed our most recent work to improve the framework and identified areas for further development related to availability, scalability, monitoring, and testing.

In conclusion, we believe that continuous integration and automatic deployment contributed to improve the quality of our XML and RDF data as well as our code and we plan to keep improving our workflows using these software engineering practices.

6. Acknowledgements

The work described in this paper was carried out by a team of developers at OUP. This team included Khalil Ahmed, Nick Cross, Matt Kohl, and myself. I gratefully acknowledge my colleagues for their precious and professional work on this project.

Bibliography

[1] *Oxford's Global Languages Initiative*. OUP. Accessed: 8 May 2015. http://www.oxforddictionaries.com/words/oxfordlanguage

[2] Matt Kohl, Sandro Cirulli, and Phil Gooch. *From monolithic XML for print/web to lean XML for data: realising linked data for dictionaries*. In Conference Proceedings of XML London 2014. June 7-8, 2014. doi:10.14337/XMLLondon14.Kohl01

[3] Martin Fowler. 2006. *Continuous Integration*. Accessed: 8 May 2015. http://martinfowler.com/articles/continuousIntegration.html

[4] John Ferguson Smart. 2011. *Jenkins - The Definitive Guide*. O'Reilly Media, Inc.. Sebastopol, CA. ISBN 978-1-449-30535-2.

[5] Jenkins CI. *Jenkins*. Accessed: 8 May 2015. http://jenkins-ci.org

[6] ZeroTurnaround. *10 Kick-Ass Technologies Modern Developers Love*. Accessed: 8 May 2015. http://zeroturnaround.com/rebellabs/10-kick-ass-technologies-modern-developers-love/6

[7] Jenkins CI. *Build Flow Plugin*. Accessed: 8 May 2015. https://wiki.jenkins-ci.org/display/JENKINS/Build+Flow+Plugin

[8] Jeni Tennison. *XSpec - BDD Framework for XSLT*. Accessed: 8 May 2015. http://code.google.com/p/xspec

[9] Benoit Mercier. *Including XSLT stylesheets testing in continuous integration process*. In Proceedings of Balisage: The Markup Conference 2011. Balisage Series on Markup Technologies. vol. 7. August 2-5, 2011. doi:10.4242/BalisageVol7.Mercier01

[10] *Jxsl - Java XSL code library*. Accessed: 8 May 2015. https://code.google.com/p/jxsl/

[11] Jeni Tennison. *XSpec - Running with ant*. Accessed: 8 May 2015. https://code.google.com/p/xspec/wiki/RunningWithAnt

[12] Agile Knowledge Engineering and Semantic Web (AKSW). *RDFUnit*. Accessed: 8 May 2015. http://aksw.org/Projects/RDFUnit.html

[13] Jostein Austvik Jacobsen. *xprocspec - XProc testing tool*. Accessed: 8 May 2015. http://josteinaj.github.io/xprocspec/

[14] Docker. *Docker*. Accessed: 8 May 2015. https://www.docker.com/whatisdocker/

[15] Russell Gold. 2008. *HttpUnit*. Accessed: 8 May 2015. http://httpunit.sourceforge.net

[16] Michael T. Nygard. 2007. *Release it! Design and Deploy Production-Ready Software*. The Pragmatic Programmers, LLC. Dallas, Texas - Raleigh, North Carolina. ISBN 978-0978739218.

[17] Jenkins CI. *Build Monitor Plugin*. Accessed: 8 May 2015. https://wiki.jenkins-ci.org/display/JENKINS/Build+Monitor+Plugin

[18] Google Groups. *XSpec Coverage*. Accessed: 8 May 2015. https://groups.google.com/forum/#!topic/xspec-users/VRlCTR5KvIU