ID Spike Report

Ari Nordstrom <ari.nordstrom@lexisnexis.co.uk>

Table of Contents

Executive Summary

There is a huge business need for uniquely and reliably identifying content at LexisNexis - documents, fragments such as clauses or fillable fields, etc. Currently no reliable identification is possible, preventing the implementation of mission-critical features in New Lexis, from automatically suggesting an update to an end user's Workfolder precedent to allowing the end user to make informed decisions about the nature of the changes in the update (minor amends vs significant updates). Essentially, lacking reliable identification means that anything that involves versioning content will remain out of reach, leading to duplicated content and lower quality, unreliable processes and missing features.

As introducing a central document management system at LexisNexis is currently unfeasible, we instead suggest implementing a system that tracks document[1] transactions between participating systems, following the documents across their lifecycles and logging every individual transaction using the available document information to build a version history for each tracked document.

The system will create and manage unique and persistent identifiers for every tracked document, map the document transaction information to a versioning tree of every identifier and thus provide a passive versioning system to any subscribing system. Each subscribing system will be able to query the tracking system for any earlier transactions involving a document identified using a local identifier or other metadata.

The tracking system will then be able to provide the required New Lexis functionality, from alerting an end user of the existence of a later version of a local Workfolder precedent to allowing for comparisons between the document versions. Additional features, from avoiding duplicated search results and intelligent bookmarking to efficient document assembly and disassembly, are also made possible.

[1] Any type of resources, actually, including documents, images, etc.

Among the advantages of implementing a tracking system is also the fact that very few changes are needed in existing systems. The tracking system itself is intended to be lightweight and can be implemented without needing to go beyond XML markup technologies.

Introduction

The document at hand attempts to describe the current problems having to do with *identifiers*[2] at LexisNexis. Identifiers identify content, from individual paragraphs or even single words or phrases (for example, fillable fields) to parts of documents (for example, clauses) and complete documents (for example, precedents). Some may be in XML while others exist as MS Word files, RTF and such.

Note

While there are fewer available use cases for images and other binary content, these should be identifiable as well, whenever used.

Terminology

It should be helpful to begin by defining the terminology used:

- *Identifiers* are simply strings used to name content. Frequently, they must follow well-defined rules (as in the case of XML IDs, with constraints specified by the XML 1.0 specification), while in other cases, a database ID is created and used by a system (for example, ECHo document IDs).

- *(ID) naming schemas* are frequently suggested and used to ensure a consistent identifier composition (for example, *UUID*s or "Universally Unique Identifiers", as frequently found in LexisNexis XML content) and increase the likelihood of uniqueness if used in the context of a system, company, document lifecycle, etc.

- *Scope* defines the expected confines of identifiers, that is, where they are used as defined by the participating systems, content lifecycles, publishing chains, and so on. The scope defines where the identifiers are expected to be recognisable, usable and, sometimes, unique. It is important to note that several scopes may apply to a single ecosystem of content.

- *Persistence* is the ability of the identifier to retain its original semantics over time; specifically, the timespan must be at least as long as the lifetime of the system(s) that use the identifier. Indirectly, then, persistence is also the ability of the system(s) to recognise and resolve (as in pairing an identifier to a document) the identifier within the defined scope over the lifetime of the scope.

  It follows that persistence can only be guaranteed if there is a well-defined scope and the means to fully control the creation and management of the identifiers.

  ### Note

  And yes, we are fully aware of the rejection of the term "permanence" in favour of "persistence" in the context of PGUIDs as defined by the Global Architecture and Research Group at LexisNexis (see [id-pguid-gar]). "Persistence" is used here, as this is the term used by the URN Syntax RFC (see [id-rfc2141]).

- *Control*, then, is the ability of some system to own the process of creating and updating identifiers as needed so that the corresponding document contents can be consistently accessed while in scope, thus guaranteeing persistence. In a single source environment, the central document management system should obviously be the system in control.

[2] Also known as "PGUIDs", "UPIs", "UIDs", etc. Here, we will simply use "identifier".

Problem Description

Currently, there is no way to reliably identify content across its lifecycle at LexisNexis. The identifiers used are not guaranteed to be either unique or persistent, as there is no central resolver service to ensure uniqueness, persistence and scope to the identifier so the corresponding content can be looked up by an application somewhere in the publishing chain.

For example, a clause in a future clause bank might be included in an end user's precedent and require an update months later. As the clause cannot currently be uniquely identified across its lifecycle, there is no way to reliably “trace back” to the right clause and establish that a current clause in the clause bank is, in fact, a later version of the one fetched months ago.

It is important to note that currently, even if identifiers of some description were added, there is no support for ID management in the workflows of the existing systems; therefore, there is also need for an infrastructure to handle IDs, to resolve them and to update them over time.

Use Cases

While the need for persistent identifiers is evident across LexisNexis, the use cases identified in this sprint all focus on New Lexis. Here are five cases, none of which is currently possible:

1. Bookmarking via the address bar URL when browsing NL

   This is about bookmarking documents in the [New Lexis] cloud, possibly Word documents. As the contents of the cloud are updated over time, a bookmark would most probably need to pinpoint a specific version of a document, allowing the user to determine if a) there is a later version of the document that was bookmarked, and possibly b) if the later version is still relevant.

2. “Update” feature in NL Workfolders

   This is about an end user getting the latest (or simply a specified newer one, if allowing for historical versions) version of a local (Workfolders in the cloud) document, frequently with local edits. In other words, an edited local document needs to be identifiable as an earlier version of a document later uploaded to New Lexis.

3. Ensuring there are no duplicate search results

   This is about eliminating duplicates when search results are returned in the New Lexis cloud. Obviously, with correct identification and versioning, there shouldn't be any duplicates, but relevant here is also what the search is based on. Metadata? Keywords? Document identifiers? But also: Is a search inside documents envisioned?

4. Clause Bank - requirement to be able to tell customers that their clause/precedent has been updated, as per the Workfolders case, above

   As stated, this is a variant of item 2 and requires both identifying and versioning the clauses/precedents. Note that a generic variant here would be to uniquely identify and version *any* type of resource, from documents fragments to documents, binary content, and so on.

5. Need to be able to explain to a customer whether the update is actually something that will affect the legality of the document and needs their consideration or is, in fact a minor amend from a PSL that shouldn’t be of any concern.

   On the surface, this is simply 2 and 4 revisited, but actually it requires comparing versions in addition to merely identifying and versioning them. Thus, it likely requires going inside a document to identify structural components *and/or* identifying different workflow statuses *and/or* differing metadata. How is one such update (minor amend) differentiated from another (major change to document)? This adds a workflow dimension to versioning (as in, "what signifies one change from the other?").

Studying the above, it should be easy to see that all use cases require unique and persistent identifiers. Let's group them and try to make sense of the actual requirements:

Versioning of documents	Items 2 and 4 are basically the same, and actually about the ability to version handle document identifiers rather than unique identifiers by themselves. Note that 5 also requires versioning documents.
Identifying documents in applications	Items 1 and 3 are about identifying documents (perhaps including historical versions) in applications using New Lexis, in addition to the versioning requirements (2 and 4). For example, while browsing documents in New Lexis is most likely about browsing HTML or Word content in an application, for the end user to be able to bookmark a document (probably a browser page with embedded content), the application needs to be able to identify the underlying source document relative to the rendered content so it can a) identify the unique document, b) its version, and possibly c) its workflow status. All this requires resolving a local application identifier and relating it with a source document that resides in a different system, using different identifiers. The uniqueness and applicability of search results (item 3) depend on how the search is done, in addition to the basic process of resolving the search parameters to first local (in the application) documents and then possibly to other systems. It is unlikely that the local application will contain multiple versions of a document, which means that historical versions, if allowed, require searching outside the local application.
Comparing document contents	Item 5, while requiring everything required by the other items, suggests either means of reading and comparing content, that is, going inside each document version, or having sufficient metadata attached to the versions being compared to identify the required change criteria. For unstructured formats such as Word or RTF, reliably comparing contents can be quite difficult[3]. For structured ones, a comparison is easier but does suggest the use of structural identifiers. Both suggest a diffing tool, implemented in the application.

Analysis

Current Systemic Obstacles

The fact that none of the five use cases presented in the previous section are possible today is due to several systemic factors:

- The LexisNexis publishing ecosystem (the “scope”, using the terminology in the section called “Terminology”) consists of a multitude of largely independent systems. This means that there is no single source document management system, nothing that can *control* ID creation and management. Today, any system can produce changes undetected and undetectable by another, which means that there is no way to guarantee scope or persistence.

[3]This is in terms of fulfilling the requirement; a “fuzzy” comparison is easier but non-conclusive.

- Several different and sometimes conflicting methods of identifying content are used by the various systems. For example, a system might use several different document identifiers to identify what should be a single document passing through different stages of its lifecycle, throwing away the earlier one, while another might rely on a file naming schema on what's essentially a file system.

  Conversely, there are cases where the same document identifier has been used to identify two conflicting versions of a single document. For example, a well-documented case involves using a single identifier to identify two versions of a document, each with different bias, a “buyer's version” and a “seller's version”.

  Also, very frequently structural identifiers will be replaced by new ones or removed altogether when documents are passed from one system to another, without logging the change in any way. For example, both editorial and fabrication have processes that do this; some are due to bugs while others are by design.

- There is no single source format. Some documents are now produced and updated in XML, but others are written in Word and stored in RTF format. RTF is not an “intelligent” format, lacking the semantic capabilities of XML, and can therefore not use structural identifiers.

- The many conversions (to, for example, Word for smart precedent applications) cannot be seen as merely a part of a publishing workflow where the resulting publication is simply an end result, requiring no further interaction with the system that produced it.

  For example, an end user might require a later version of a smart precedent Word file, having used a previous version to create a local precedent and now wanting to update that document. Quite possibly, there is also a need to find out the differences between the local version and the latest published one in the application.

Types of Identifiable Content

Studying the current state of LN documentation and how the various LN systems regard identifiers, and adding to that the earlier efforts to define (persistent/permanent global unique) identifiers, the current types of content to be identified are many and frequently conflicting:

- *Structural components*, basically nodes inside XML documents. These typically use `ID` type attributes. For example, a paragraph or a fillable field might have an `xml:id` attribute to separate it from other elements of the same type.

- *Semantic components*, usually more or less equal to “documents”. Any block of information used as a single identifiable unit is basically a semantic component. For example, a reusable clause in an envisioned clause bank (assuming a physical file to represent the clause) or inside a current precedent would be considered to be such a component. Other examples include attachments, drafting notes, and so on.

- *Documents*, perhaps best described as self-sufficient (XML or otherwise) physical documents.

- Auxiliary files used by the above. Typically, these might be images or other media files.

- *Links*, that is, pointers to any of the above. Normally, these pointers live inside a document.

Studying the above, if considering the various information types, above, it should be obvious that naming one type of information should be similar to naming another; identifying an image is actually not different from identifying a document, which in turn is not dfferent from identifying a semantic component. *It should be possible to use the same naming convention regardless of the type of content.* Indeed, using the same naming conventions should actually be preferable as it simplifies handling, both manually and by systems.

Structural components, that is, nodes inside documents, can be somewhat different, however, as in order to be usable in practice, they need to be identified using ID attributes in XML nodes. As such,

they are bound by constraints in the XML specification, a constraint that does not have to be imposed on other types of names.

Several previous PGUID efforts have focussed on defining the same unique namespaces for structural and semantic identifiers alike. There is, for example, code in place for various pipelines that will produce UUID-style values for `xml:id` and other attributes (sometimes replacing earlier UUID-style attribute values). While long and probably unique[4], these values are not persistent; they are, as already stated, frequently changed, and there is no resolver in place to give the strings meaning to an outside system.

Arguably, UUIDs and other complex naming conventions are overkill for structural identifiers. An XML ID needs only to be unique within a physical document (a well-defined scope by the XML 1.0 specification), and it would only be useful in the context of the XML document that contains it, which leads to uniqueness and persistence requirements for identifying that document.

Versioning and Workflow Considerations

The identification of documents is largely meaningless without versioning. For example, if a document identified using a single, fixed identifier changes, while the identifier might be used by the originating system to always identify the latest version (with the earlier one thrown away), the identifier's meaning would vary depending on the end user[5], with a user possessing an earlier version the document assigning one meaning to the identifier and another user possessing a later version assigning another.

This, of course, is not a problem if the users never need to use the identifier to communicate with the document provider, but this is not the case at LexisNexis. Versioning needs to be taken into account when identifying a document, thus requiring the versioning of the source documents *and* their identifiers.

This leads to the question: *what is a version?*

Versions, to put it simply, identify change to content in a system. While every (saved) change will cause a new version, such a definition is meaningless in itself. Change means different things to different users of a document: An editor, for example, does have use for several work-in-progress versions as this gives her the ability to go back to an earlier version, should a later one prove unusable. A reviewer, on the other hand, would only be interested in the versions submitted for review. An end user would have still other requirements, needing only the approved versions that denote *significant* changes.

This hints at workflow statuses in addition to versioning. It is quite possible to have a single version of a document move from “draft” to “review” to “approved” and, finally, “published”, but just as likely that there would be several versions between each workflow status. Indeed, the review process is often iterative and each status repeated several times.

Versioning, then, is about significant change (with the definition of “significant” differing from one user to another). Workflow, while related, is a status change to a document, with or without content change. Both must be taken into account when designing document identifiers and the systems that support them.

Proposed Solution

Basic System Support

There is, of course, a “simple” solution for much of the above: implement a central document management system used as a single source of documents in XML format only, and in control of every document lifecycle stage for the documents in scope. This solution, while by far the best one in terms of fullfilling every foreseeable requirement in addition to the ones listed above, is currently

[4] There is a probability, albeit small, built into the code that they aren't.
[5] Assuming, for the sake of argument, that the identifier was made available to the user, either directly or indirectly.

unworkable. There is no practical way of implementing such a system without severely compromising "business as usual".

Instead, we propose implementing a tracking system for every system that requires unique identifiers. The system, basically a database, tracks documents handled by other systems and generates local "abstract" identifiers for every tracked content type (tracking is performed on documents and structures inside them, but also on links). The tracking system maps the identifiers against the corresponding identifiers, URLs and other related information used by the other systems.

The tracking system also adds local versioning and workflow capabilities to the abstract identifiers. When there is an event affecting a tracked document (for example, when it is updated, converted, moved, etc), the event is logged by the tracking system, updating the version and/or workflow status of the abstract identifier according to business logic defined in the tracking system.

The great advantage of a tracking system is that *no central solution is forced; the systems "subscribing" to the service can be brought in when needed, requiring few changes to those systems.*

ID Naming Conventions

Rather than having to deal with multiple types of identifiable content (see the section called "Types of Identifiable Content"), we suggest defining two types of content, each requiring their own naming conventions:

- Documents
- Nodes inside documents

Nodes inside documents require structural identifiers, essentially XML IDs, that only need to be unique within the document in which they live. Documents, on the other hand, are semantic units of information that may or may not contain structural identifiers, but need to be named using identifiers that are globally unique and persistent.

Note

A node inside a document is uniquely identified when including both its ID value and the identifier of the document that contains it.

To ensure persistence, the document identifier needs to handle different versions of the document. The tracking system that manages the identifier also needs to handle workflow status changes to the document. The workflow information is not necessarily part of the identifier itself, but needs to be coupled with its versioning.

Document Identifiers

The basic document identifier, without versioning information, must be unique within LexisNexis. It follows that the naming conventions and syntax must be owned and controlled by LN; otherwise, a change to the syntax might affect how the identifiers are resolved. Also, a third party could conceivably use the naming conventions, again risking name collisions.

The Uniform Resource Name (URN) RFC (see [id-rfc2141]) defines a basic syntax that allows for all this, as URN namespaces can be registered with IANA (see http://www.iana.org/assignments/urn-namespaces/urn-namespaces.xhtml), ensuring uniqueness for the registered namespace within the scope of the standard.

A URN schema should be implemented for any document-level identification (XML documents, images, Word documents, and so on) and a LexisNexis namespace registered for the purpose. A URN identifying a LexisNexis document might then look like this:

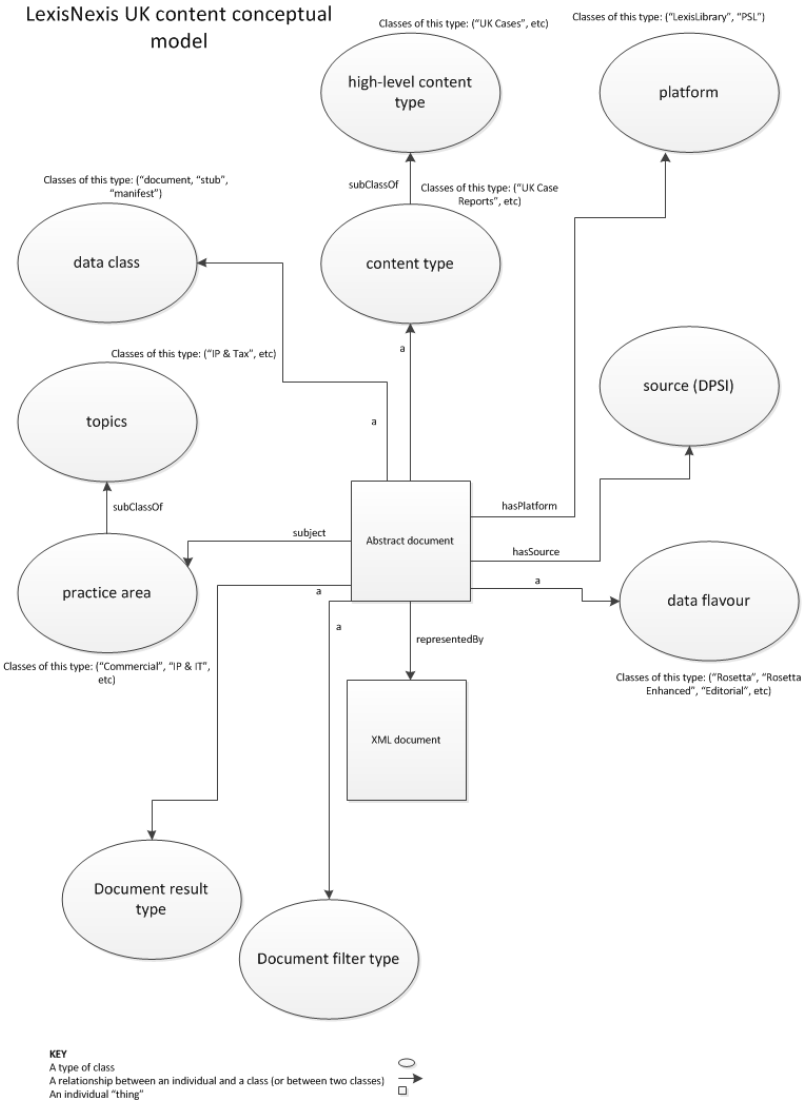
```
urn:lexisnexis:r1:ukdeg:precedent:1234567890:en-GB:123
```

The semantic components of a URN are separated using colons, so the above translates to “a registered URN in the namespace *lexisnexis*, revision 1 describing a UKDEG precedent with the base document number *1234567890*, language English, country GB, version *123*”. The descriptive part of the URN, that is, the string following the full namespace identification `urn:lexisnexis:r1`, identifies the practice area, the type of document, its basic ID, its language and country, and a version of that document.

This example is a gross oversimplification and probably wrong, as a UKDEG document could be argued to be a specific downstream workflow version of a source document authored in XML. To get the example URN right, however, is not the point here; far more important is the basic principle of the identifier where concerns are separated according to the well-defined lifecycle of a “semantic document”. For a URN to be truly useful in a document management setting, modelling its syntax according to relevant business requirements is strongly urged.

The modelling of a LexisNexis URN schema should probably follow the ontology for the conceptual model of the contents used by LN, as defined by this illustration:

Figure 1. LNUK Content Conceptual Model, Version 4



Structural Identifiers

When compared to document identifiers, structural identifiers come with fewer requirements. If given a context of a unique and persistent document ID, any XML ID-compliant string will suffice. A

semantic/structural ID combination (i.e. `URN#ID` or similar) will then always be unique for as long as the `URN` is tracked and the `ID` is unique within the document.

Also, there is no need to add versioning to the structural identifier, as any changes to the node can be fully described by versioning the parent document's identifier. In fact, changing the structural identifier is actually harmful. Consider an XML fragment such as this:

```
<doc>
    <chapter xml:id="id-chapter">
        <title>Some Title</title>
        <para>Some content.</para>
    </chapter>
</doc>
```

A change to `/doc/chapter[@xml:d='id-chapter']` is easily identifiable *if* keeping the ID unchanged:

```
<doc>
    <chapter xml:id="id-chapter">
        <title>Some Title</title>
        <para>Some content.</para>
        <para>Some change.</para>
    </chapter>
</doc>
```

Changing the ID would make a comparison of the two difficult while adding no real value.

Copying and pasting, as well as reusing contents by linking to the chapter twice in the same parent document, would both create a problem, however, since the ID would be duplicated and the document no longer valid. This, however, is a business logic problem, not an ID problem, and can be resolved using a variety of techniques from a “paste with ID renumbering” function to scoped includes, such as the ID mechanism used by DITA when including topics in DITA maps.

Document Management Issues

While the topic of document management semantics is outside the scope of this document, we do wish to make a couple of points regarding the use of identifiers in terms of document management:

- A block of (XML-tagged) information such as a clause can be fully identified inside a document if it uses a structural identifier and the containing document is identified using a document identifier.
- If the clause is broken out of the containing document and made to be a reusable document by itself, it should - and will - get its own document identifier *in addition to* the structural identifier. The latter will still reside in the clause's root element, unchanged, and the relation between the new document identifier and the former containing document's identifier will be logged.
- If the clause is then reinserted into a document, the structural identifier is again kept intact, still identifying a `clause` element. The clause fragment outside the document is still tracked using its own identifier, the assumption being that the clause continues to live as a separate document, in addition to being inserted into a new document.

  In practice, the insertion of the clause contents into a new document should be regarded as a *fork*, that is, that the clause is duplicated into two separate entities. There is no guarantee that the two copies will be simultaneously and identically updated in the future, so while a relation can be maintained between the new document and the separate clause document, the benefit of such a relation is questionable. Far better is to *link to* the clause document from every new document requiring it and keep a single source of the clause.
- Finally, keeping the clause in a separate, single-source XML document does not in any way imply that the clause cannot be updated. Quite the contrary; it can and it should. Links to the clause should *always* define a specific version of it.

Logging Events

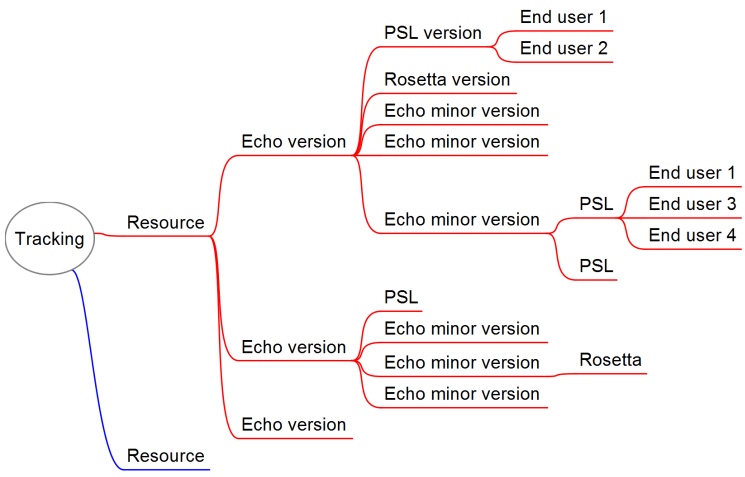
The author of this document presented a whitepaper about “multilevel versioning” at Balisage: The Markup Conference in 2014 (see [id-balisage2014-nord]). While this is not the place to explain the details, essentially the paper is about a lightweight, scoped XML-based versioning language designed to log events created by, in the paper, an XML-based database's own versioning module, enriching what is otherwise merely a linear recording of every save of a stored document.

Conceptually, the versioning XML lends itself well to the tracking system described above (see the section called “Basic System Support”), as it is meant to log events triggered by another application (in this case, the basic versioning module) but not control them. Some customisation will be necessary to include more detailed data about the system causing an event and any relevant document properties, but also adjust the versioning model because the XML in the whitepaper is designed to be used in a centralised system that lends itself well to a single tree structure. A decentralised system has several key differences:

- Each version in both the source and the target (for example, when a document is published from ECHo to a target) is a “production version”; one is not a subversion of the other.
- Each system *may* produce minor versions. These *may* then be published to other systems.
- There is not necessarily an originating document; the XML starts from the assumption that the various versions of a document do belong together when, in a decentralised case, they may only have a fleeting relation, if any.

Disregarding the intricacies of the actual XML markup, a conceptual event tracking structure might look like this:

Figure 2. Tracking Versions and Events



Here, an abstract document, created by the tracking system to log what happens to an actual physical document, is represented by a Resource node. Each subnode is an actual physical document in some system, represented as a version of the abstract Resource. For example, the subnodes of the topmost Echo version represent PSL and Rosetta versions of that Echo version, but the visualisation also includes minor Echo versions, that is, minor updates to the basic one[6].

Note that while *ECHo* is here seen as the “originating” or controlling system, it doesn't have to be.

[6]This, apparently, is not realistic.

There are several important variations of this basic concept: For example, instead of the above where events are logged in a single structure, it might be better to create a wholly abstract document versioning tree that links to nodes in separately maintained "event trees" representing each system, their timing and type of event deciding how those events are represented in the abstract document versioning tree.

Using the Logged Information

The tracking system itself is designed to be a passive observer, but from the subscribing system point of view, each event is actually a transaction, which means that when using the logged information, the usage is about handling previous transactions in relation to a current document, not merely about reading past events.

To be usable in real life, the tracking system needs to provide services to the subscribing systems that help track a document through its transactions from one system to the other. A system providing precedents in Word format to end users needs to be able to query the tracking system if a specific Word file in a user's Workfolders is an earlier version of the current Word file seemingly sharing the same local document identifier[7]. This is only possible if the system is able to track earlier transactions relating to the local document ID; a query to the tracking system would the include that identifier, available metadata, the user's ID[8], and so on. By using this information when searching through the logged version tree, the tracking system should be able to provide a reliable answer.

While the changes to the subscribing systems should be small, they do need to provide some information when logging transaction information sent to the tracking system and when querying the tracking system, including the following:

- A system identifier for the system itself.
- Current document metadata, from name to author(s) to date, and so on.
- Data about the usage of a document by various parties known to the system (in particular, users of that system; in other words, transactional usage information)
- If the current document is in XML format, information about the document's structure, especially any structural IDs, might prove useful.

Conclusions

Solution

Briefly, this paper suggest solving the current identifier-related problems as follows:

- Define two types of identifiers: document identifiers and structural identifiers
- Define a LexisNexis URN namespace for document identifiers
- Use XML ID-compliant IDs (such as those generated by most reasonably competent XML editors today) for the structural identifiers
- Implement a tracking system that logs events to documents handled by LN systems, representing the events in an XML-based structure that defines an abstract document versioning model
- Allow document version and workflow queries from the participating systems so the logged information can be used to retrieve other versions of a document

[7] That might be anything from a file name to a local database ID.
[8] User IDs are not covered by this document.

Use Cases

How are the use cases (see the section called “Use Cases”) handled by the suggested solution?

Bookmarks

If a document is browsed to and bookmarked, the bookmark should include the name (as in "semantic identifier"), version and possibly workflow status of the document. Any relevant metadata should be included as well, but the best way would be to add that to the specific version in a tracking system.

Note that the metadata would include a date. Also note that any older versions need to be saved.

This requirement can be easily handled if the documents in the cloud are uniquely identified and tracked (in other words, versioned).

Update Feature, Workfolders

This requires keeping track of several things:

- Any updates to the document during editing and publishing. At least every published version needs to be logged, which implies both versioning and workflow. Note that if the document is published in several other places (in addition to NL), these need to be logged as well.
- The version(s) in Workfolders.
- Possibly, local (end user) changes.

At the very least, meeting these requirements means that the documents need unique names and versions, but most likely also workflow statuses.

Meeting these requirements is achieved using unique names and versioning the corresponding documents by logging the events that place new versions in the cloud.

Avoiding Duplicate Results in Searches

Basically, this requirement is fulfilled as soon as the documents published in the cloud are uniquely identifiable, including their versions. *if* duplicates end up in the cloud, then the naming is flawed.

Note that the search needs to be “ID-aware”, that is, it needs to base its search on a set of identfiers (and versions of them) as defined by the tracking system, not the participating systems (such as ECHo). This means that a search should be based on a resolver service provided by the tracking system rather than a participating system, as the latter will not have direct access to the complete history of a document.

Tracking Clause/Precedent Updates

This is basically a variant of the Workfolders update requirement. Meeting it requires unique naming and versioning of the clauses, and tracking them, but obviously also tracking any updates to Workfolders. Again, the resolver must be provided by the tracking system.

Minor Amends and Major Changes

There are several ways to meet these requirements. One is to log the changes in metadata and let any search be based on that. Another is to scope the versioning (as described in my whitepaper) and base the business logic on the version changes. A minor change is OK, while a major change requires action.

However, keeping track of *what* the actual change is means that there needs to be pointers to the nodes that are affected. This, in turn, requires structural identifiers for those nodes.

Note that the structural identifiers still only needs to be unique inside the physical document. In fact, it is advantageous if the structural ID remains unchanged. The changing document, including the node, can then be easily tracked.

Assuming that the ID is “ID”, the old version of the document is “URN1” and the new “URN2”, this is the old version:

```
URN1#ID
```

And this is the new:

```
URN2#ID
```

In other words, fragment identifiers are sufficient.

Note, however, that both of the above would have to be logged by the tracking system and a suitable *presentation* of the corresponding content in the converted Word files (versions 1 and 2) be produced and displayed to the end user.

Next Steps

As next steps, we suggest the following:

- The registration of a LexisNexis URN namespace with IANA.
- Further analysis. There are other initiatives involving identifiers, including a GAR Wiki document about PGUIDs (see [id-pguid-gar]), all of whch should probably be aligned with the one at hand.
- A Proof of Concept (POC) should be implemented, using either a minimal subset of systems and data, simulated test data, or a combination of both. The POC should be exploratory in nature and include a number of variants of the basic versioning XML.

  The POC system should be an XML database, as this makes implementing both the versioning XML and any queries both easier and faster. It also aligns well with the overall strategy aiming to move content to XML at LexisNexis.

Bibliography

[id-pguid-gar] PGUID Page, from the GAR Wiki http://garwiki.regn.net/mediawiki/index.php/PGUID.

[id-rfc2141] URN Syntax http://tools.ietf.org/html/rfc2141.

[id-balisage2014-nord] Nordström, Ari. “Multilevel Versioning.” Presented at Balisage: The Markup Conference 2014, Washington, DC, August 5 - 8, 2014. In Proceedings of Balisage: The Markup Conference 2014. Balisage Series on Markup Technologies, vol. 13 (2014). doi:10.4242/BalisageVol13.Nordstrom01. http://www.balisage.net/Proceedings/vol13/html/Nordstrom01/BalisageVol13-Nordstrom01.html.