RDF Site Summary 1.0 Modules: Rich Equivalents

Author

Jon Hanna, Spin Solutions

Version

Latest Version http://purl.org/rss/1.0/modules/richequiv/

Status

Draft

Rights

Description

This module defines elements defining properties which are equivalent to the title and description properties defined by the core RSS1.0 Spec, but allowing for the use of xml elements as content.

Namespace Declarations

- **xmlns:reqv="http://purl.org/rss/1.0/modules/richequiv/"**

Model

<channel>, <item>, <textinput> elements

- **<reqv:title>** ANY
- **<reqv:description>** ANY

Motivation

RSS has always defined a title and description element. The RSS1.0 Spec defines these as containing Parsed Character Data (i.e. plain text), but authors have desired a way to use richer content, in particular HTML, in the rendering of these elements.

The "solution" hit upon was to abuse the text-based nature of XML and HTML and to store the text of an XML fragment as the content of the element. For example to transmit the HTML fragment:

```
<p>A description.</p>
```

The author would treat the HTML as text and encode it XML producing:

```
&lt;p&gt;A description.&lt;/p&gt;
```

They would then make that the content of the relevant element:

```
<description>&lt;p&gt;A description.&lt;/p&gt;</description>
```

When being read by an RSS parser that understood this convention (note that it is not documented in **any** of the RSS specs) and which successfully determined that the convention was being used in this case it would then convert the text back into the HTML fragment, (hopefully) check it for potential security risks, and then use this HTML in the rendering of the description.

There has been much debate about the validity, or even sanity, of this approach (some arguments against are given in Appendix C). In the end though no matter who has the strongest position in the debate the difference

will be problematic because either style can produce RSS content that will break on parsers written to use the other convention. Heuristics can help this problem but can quickly become a complicated piece of code as one refines them for more outside cases, and are never guaranteed to work since there is no way to know for certain whether the author intended to transmit the XML element or the actual mark-up itself (in the above example we can't be certain the author didn't want the rendering to be of a less-than symbol followed by a p, and so on).

This module aims to bypass this debate by introducing elements which have the same semantics as the <title> and <description> elements, but which allow for any well-formed XML fragment (a well-formed fragment is any XML that would be well-formed where it wrapped with another element).

This solves the determinism problem, since there is no double-encoding it is clear what is XML and what is text, it allows the content to be used sensibly with RDF with no overhead for non-RDF users, simplifies implementation (the XML is already XML, no need to parse it twice - especially awkward for RSS parsers that work on XML trees, such as XSLT-based parsers), and as a bonus offers a safe way to introduce content from other XML applications with full backwards compatibility to parsers which don't support them.
With the use of a few techniques this module can be as easy to use for even naïve xml-as-text parsers. This is important for ensuring that RSS implementations that start with such a mechanism aren't discouraged from using it and opt for the double-encoding technique.

Relation to mod_content

To some degree mod_content solves a similar problem; the transmission of arbitrary XML content, primarily HTML. However the purpose of that XML is different than in the case of mod_content, where the idea is to transmit more complete pieces of content rather than descriptions.
Some people may be using mod_content as a more "civilised" alternative to the double-encoding technique, and they will hopefully welcome this module. It is possible that the same XML may be a good value for both the <content:item> and <reqv:desciption> elements. As such it may be appropriate for a <content:item> element to have an rdf:resource attribute that points to a fragment identifier reflecting the value of an id attribute on an element that is the content of the <reqv:desciption> element. However resolving such references is impossible without a validating XML parser - which is beyond the requirements for processing other RSS elements.

Syntax

<reqv:title> and <reqv:description> can appear where <title> and <description> as defined in RSS1.0 can appear, and have the same meaning.

They MUST have an attribute with a namespace name of http://www.w3.org/1999/02/22-rdf-syntax-ns#, a local name of parseType and a value of "Literal" (in other words they must be <reqv:title rdf:parseType="Literal"> and <reqv:description rdf:parseType="Literal">).

They can contain any XML content. The type of the content is indicated by the use of namespaces. <reqv:title rdf:parseType="Literal" xmlns=""> can be used to contain XML from the default namespace (i.e. which doesn't use namespaces).

Multiple occurances of each element is allowed, although rendering parsers are expected to ignore all but one. rdf:Alt or other RDF collections MUST NOT be used, to preserve the equivalence with the related RSS elements, and to ease non-RDF based implementations.

Although document order isn't significant when considering the RSS as RDF, there is no reason why document order can't be used in determining which element to use in rendering. As such the suggested method for determining which element to use in the case of multiple equivalents being available is to use the first element in document order which the renderer is capable of using. Rendering parsers are free however to make a choice based on implementation-specific criteria. If a rendering parser **does** use criteria other than document order they MUST be deterministic; in other words if the parser repeatedly encounters the same RSS and no applicable settings have been changed it MUST always pick the same element as before.

Note on Charset Encoding

The elements defined in this document are conceived as transmitting XML elements and text nodes, not the text that represents them. As such the encoding is the same as the parent document, implementations are free to re-encode XML obtained from the RSS (e.g. converting from UTF-8 to UTF-16) as suits their purposes.

Implementation Notes

The following outlines techniques that will enable the elements to work correctly across implementations based on the XML Infoset, DOM trees, SAX

events, RDF, or direct manipulation of the text the XML is persisted to. All of the following SHOULD be done, but none is stuff that MUST be done. The author notes that the ease in fulfilling each of these varies depending on the technologies used.

For RSS Producers

1. Encode in UTF-8, but use character references for characters with positions above U+007F (e.g. for non-ASCII characters). This ensures interoperability with parsers which don't parse UTF-16 (yes they aren't real XML parsers then, but these do exist!) and even with naïve implementations that assume US-ASCII or ISO 8859-1).
2. Always place relevant namespace information on the <reqv:title> and <reqv:description> elements, even if this means duplication (see below) to make it easier to detect what sort of XML is used.
3. If it is vital that namespace information be retained then put the declaration directly onto the elements enclosed, so that a simple cut-and-paste operation on the contents will retain the namespaces.
    4. In cases where the namespace may cause confusion place it on the enclosing <reqv:title> or <reqv:description> element only. E.g To conform with one of the XHTML1.0 DTDs the namespace must only be declared on the root <html> element. Further most browsers still don't accept a namespace prefix for HTML elements, hence to encode a fragment one should use:

```
<reqv:title rdf:parseType="Literal"
xmlns="http://www.w3.org/1999/xhtml">
   <p>A description.</p>
</reqv:title>
```

        This ensures that tree-based parsers have the correct namespace information but text-based parsers doing a copy-and-paste technique will have a fragment that works well when inserted into a HTML document.

5. Don't use any entity references, since your <!DOCTYPE> will be needed to process them naïve cut-and-paste operations may fail.

For RSS consumers

1. Always check the namespace of the elements you receive, don't just assume it's HTML (actually that one is a MUST rather than a SHOULD).
2. The basic procedure for obtaining content from these elements is:
    1. Look for instances of the relevant element (say your currently rendering the description, so look for <reqv:description>) when you encounter it examine the namespace of the elements to ascertain if you can make use of it. In the case of XML without namespaces (xmlns="") further heuristics may be needed to determin the type of XML in question.
    2. If you can't use it repeat the above step, otherwise you have your description, stop looking for <reqv:description>.
    3. If you fail to find an appropriate <reqv:description> then render the <description>, treating it as plain text.
3. You are free to perform any operation on the XML that retains the infoset information, e.g. Canonicalisation, Exclusive Canonicalisation, changing namespace prefixes etc. If passing the XML to another component for further processing you can encode it using any character set. Such decisions should be made in the context of what you need from this XML once you've received it. You may also remove any namespace context that isn't use, but which is inherited from the containing document.

RDF Schema

The following schema is embedded into this document (along with some other metadata). Note that it defines a circular subPropertyOf relationship between the elements defined in this document and their equivalents in the core RSS1.0 module. The effect of this is to cause an RDFS closure to produce the same graphs from the contents of the elements defined here as if those contents were in the respective RSS elements, and vice versa.

```
<rdf:RDF
            xmlns:rdf="http://www.w3.org/1999/02/22-rdf-
syntax-ns#"
            xmlns:rdfs="http://www.w3.org/2000/01/rdf-
schema#"
            xml:lang="en">
            <rdf:Property
rdf:about="http://purl.org/rss/1.0/modules/richequiv/des
cription"
            rdfs:label="Description"
            rdfs:comment="A rich XML description.">
```

```
            <rdfs:subPropertyOf>
                        <rdf:Description
rdf:about="http://purl.org/rss/1.0/description">
            <rdfs:subPropertyOf
rdf:resource="http://purl.org/rss/1.0/modules/richequiv/
description" />
                        </rdf:Description>
            </rdfs:subPropertyOf>
            <rdfs:isDefinedBy
rdf:resource="http://purl.org/rss/1.0/modules/richequiv/
" />
            </rdf:Property>
            <rdf:Property
rdf:about="http://purl.org/rss/1.0/modules/richequiv/tit
le"
            rdfs:label="Title"
            rdfs:comment="An XML descriptive title.">
            <rdfs:subPropertyOf>
                        <rdf:Description
rdf:about="http://purl.org/rss/1.0/title">
            <rdfs:subPropertyOf
rdf:resource="http://purl.org/rss/1.0/modules/richequiv/
title" />
                        </rdf:Description>
            </rdfs:subPropertyOf>
            <rdfs:isDefinedBy
rdf:resource="http://purl.org/rss/1.0/modules/richequiv/
" />
            </rdf:Property>
            </rdf:RDF>
```

Security Considerations

The security issues of this module are far-reaching and by no means trivial. It is worth noting however that all of these concerns also apply to the double-encoding technique, with the added danger that because it is not defined by any standard or specification there is nowhere to engage with these issues. Note also that the applicability and severity of these issues will vary according to other factors. For instance applications which send HTML to a browser need to be particularly careful if the browser considers the HTML to be from a "local" source, as it may trust this source and hence use a more lax security model.

1. When receiving unknown XML formats do not attempt to render them. Apart from XML formats explicitly described as having no namespace name (xmlns="") one should assume the XML is of a format you have no

use for and not attempt to guess further from heuristics or naïvely passing it to a browser. Many modern browsers accept many forms of XML, some of which are "active" and may contain malware, and at least one of which you don't know about!

2. Check the XML you receive for potentially dangerous elements. In HTML these elements are <script>, <object>, <applet> and the non-standard <embed> and <xml>, and any element that would cause the access of a URI beginning with "javascript:" "vbscript:" or "data:". There are cases where such elements are safe, but validation should work on a "default to secure" basis - i.e. rather than use the element unless something else indicates it is unsafe you should drop the element unless something else proves to your program that it is safe (whether because your program has clever analysis of what the element is doing, or because you trust the source).
3. When attempting to validate XML for dangerous content (see above) make sure that your validation occurs after processing the text's character set. Much software exists that will erroneously misinterpret certain illegal UTF-8 values as legal values, for example they may treat a byte of value 0xC0 followed by a byte of value 0xBC as a UTF-8 encoding of U+003C (a less-than character). This means that you could search the string for "<script" and fail to find it, hence letting it through to a browser which may "fix" the UTF-8 and go on to execute the script. The solution is to either fix such illegal UTF-8 encodings first, or else to throw an error when an illegal UTF-8 sequence is found.

Appendices

The following are for information only, and are not normative.

Appendix A: Example

The following example uses the module to provide HTML equivalents of the title and description of the channel and items. In the case of the channel title an SVG image is also provided.

```
<?xml version="1.0"?>

            <rdf:RDF
            xmlns:rdf="http://www.w3.org/1999/02/22-rdf-
syntax-ns#"

xmlns:reqv="http://purl.org/rss/1.0/modules/richequiv/"
            xmlns="http://purl.org/rss/1.0/">
            <channel
rdf:about="http://www.example.com/feed.rss">
            <title>Example.com</title>
            <link>http://www.example.com/</link>
            <reqv:title rdf:parseType="Literal"
                xmlns="http://www.w3.org/1999/xhtml">
                <h1>Example.com</h1>
                </reqv:title>
                <reqv:title rdf:parseType="Literal"
                xmlns="http://www.w3.org/2000/svg"

xmlns:xlink="http://www.w3.org/1999/xlink">
                <svg
                    xmlns="http://www.w3.org/2000/svg"

xmlns:xlink="http://www.w3.org/1999/xlink"
                    viewBox="0 0 176 44"
                    preserveAspectRatio="xMidYMid">
                    <a
xlink:href="http://www.example.com">
                                        <ellipse
                    style="fill: blue; stroke: green;"
                    cx="88" cy="22"
                    rx="84" ry="18"/>
                                        <text style="font-
family: arial, helvetica, sans-serif;
                    font-size: 10.00;
                    font-weight: bold;
```

```
                    fill: red;"
                    x="60" y="25">Example.com</text>
                    </a>
                    </svg>
                </reqv:title>
            <description>
                        The Hyphothetical Portal&#8482;
            </description>
            <reqv:description rdf:parseType="Literal"
xmlns="http://www.w3.org/1999/xhtml">
                <hr />
                    <p>The Hyphothetical
Portal&#8482;</p>
                </reqv:description>
            <items>
                        <rdf:Seq>
            <rdf:li
resource="http://www.example.com/item1.html"/>
            <rdf:li
resource="http://www.example.com/item2.html"/>
                        </rdf:Seq>
            </items>
            </channel>
            <item
rdf:about="http://www.example.com/item1.html">
            <title>First Example Item</title>
            <reqv:title rdf:parseType="Literal"
xmlns="http://www.w3.org/1999/xhtml">
                <h2>First Example Item</h2>
                </reqv:title>

<link>http://www.example.com/item1.html</link>
            <description>
                        Our first example Item.
            </description>
            <reqv:description rdf:parseType="Literal"
xmlns="http://www.w3.org/1999/xhtml">
                <p>Our 1<sup>st</sup> example Item.</p>
                </reqv:description>
            </item>
            <item
rdf:about="http://www.example.com/item2.html">
            <title>Second Example Item</title>

<link>http://www.example.com/item2.html</link>
            <reqv:title rdf:parseType="Literal"
xmlns="http://www.w3.org/1999/xhtml">
                <h3>Second Example Item</h3>
```

```
                </reqv:title>
            <description>
```

```
                        Our second example Item.
            </description>
            <reqv:description rdf:parseType="Literal"
xmlns="http://www.w3.org/1999/xhtml">
                <p>Our 2<sup>nd</sup> example Item.</p>
                </reqv:description>
            </item>
            </rdf:RDF>
```

Appendix B: Use with XML Inclusions.

With large pieces of XML there would be an obvious advantage in stating a URI from which the XML could be downloaded. There are a few possible approaches one could take with this, but they each have disadvantages.
One would be to allow the use of rdf:resource on the elements defined above.
Another would be to create new elements for this purpose. However both of these approaches would lose there equivalency with the RSS elements.
In addition referencing a fragment, as opposed to an entire document, has implentation difficulties.

Because of this no such mechanism is provided. The author does note however that it is a perfectly valid interpretation of the specification above to use the elements defined here to contain an <include> element as defined in XML Inclusions (XInclude) Version 1.0.

A parser choosing to process such an item (identified by the namespace name of http://www.w3.org/2001/XInclude) should be capable of determining if it can handle the fragment referenced as soon as such information is available. In effect this would be the same as processing the XInclude and then deciding whether to process the new element(s), however that would not be the most efficient way of carrying out such an action.

The task is probably daunting, and could likely require updates as XInclude is only at Candidate Recommendation stage and some questions remain open. However with the assistance of a tool or library for handling XInclude it could be a very powerful addition to an RSS parser.

Appendix C: Arguments Against Double-Encoding

The following list is probably not complete:

- It's simply not in any of the Specs. Implementing a specification should not require mind-reading.
- It is error-prone, especially if truncation occurs in transit (which does happen with some RSS producers) causing such errors as unclosed elements and incomplete tags.
- It ties RSS to HTML as it's only possible mechanism for rendering. This is limiting at best. Strong links between web technologies tend to hamper their development. As an example, the growth of HTML itself is partly due to the fact that it is not strongly tied to any of the technologies that are often used with it (HTTP, GIF, JPEG, CSS, Java, Javascript etc. all have "meeting points" with HTML, but all have uses not related to HTML, and vice-versa).
- While rendering HTML is easy if the task can be farmed out to an already existing browser, it is extremely complicated for any other implementation. This limits the available options to someone planning to implement HTML. And creates a "right way" of doing something which may not suit a particular programmer or particular programming task.
- It limits the ability of the parser to decide on how something should be displayed, which hence ceases to be the perogative of the person actually reading it.
- It puts the decision on how HTML should be produced at a point in the process where little is known about the browser that is displaying it. Conversely if HTML is produced by the parser it may be able to do so with knowledge of bugs and features of the browser being used, and optomise for that.
- It complicates the task of searching for content that may produce security breeches.
- It doesn't display well in RSS parsers that don't use the convention.
- Probably the most compelling argument is this: It is simply impossible to tell with 100% certainty when the convention is being used and when it isn't. Even the cleverest heuristics can only tell if someone **might** have been using it. As such failures to operate correctly are guaranteed.