NewAntFeaturesInDetail XmlNamespaceSupport

Ant 1.6 introduces support for XML namespaces. This page tries to show what this means, how it can be used, and where potential problems exist in the current implementation (as of 1.6 beta 3).

History

All releases of Ant prior to Ant 1.6 do not support XML namespaces. No support basically implies two things here:

- Element names correspond to the "qname" of the tags, which is usually the same as the local name. But if the build file writer uses colons in names of defined tasks/types, those become part of the element name. Turning on namespace support gives colon-separated prefixes in tag names a special meaning, and thus build files using colons in user-defined tasks and types will break.
- Attributes with the names 'xmlns' and 'xmlns:<prefix>' are not treated specially, which means that custom tasks and types have actually been able to use such attributes as parameter names. Again, such tasks/types are going to break when namespace support is enabled on the parser.

Use of colons in element names has been discouraged in the past IIRC, and using any attribute starting with "xml" is actually strongly discouraged by the XML spec to reserve such names for future use.

Motivation

In build files using a lot of custom and third-party tasks, it is easy to get into name conflicts. When individual types are defined, the build file writer can do some name-spacing manually (for example, using "tomcat-deploy" instead of just "deploy"). But when defining whole libraries of types using the <typedef> 'resource' attribute, the build file writer has no chance to override or even prefix the names supplied by the library.

Assigning Namespaces

Adding a 'prefix' attribute to <typedef> might have been enough, but XML already has a well-known method for name-spacing. Thus, instead of adding a 'prefix' attribute, the <typedef> and <taskdef> tasks get a 'uri' attribute, which stores the URI of the XML namespace with which the type should be associated:

```
<typedef resource="org/example/tasks.properties" uri="http://example.org/tasks"/>
<my:task xmlns:my="http://example.org/tasks">
  ...
</my:task>
```

As the above example demonstrates, the namespace URI needs to be specified at least twice: one time as the value of the 'uri' attribute, and another time to actually map the namespace to occurrences of elements from that namespace, by using the 'xmlns' attribute. This mapping can happen at any level in the build file:

```
<project name="test" xmlns:my="http://example.org/tasks">
  <typedef resource="org/example/tasks.properties" uri="http://example.org/tasks"/>
  <my:task>
    ...
  </my:task>
</project>
```

Use of a namespace prefix is of course optional. Therefore the example could also look like this:

```
<project name="test">
  <typedef resource="org/example/tasks.properties" uri="http://example.org/tasks"/>
  <task xmlns="http://example.org/tasks">
    ...
  </task>
</project>
```

Here, the namespace is set as the default namespace for the <task> element and all its descendants.

Namespaces and Nested Elements

Almost always in Ant 1.6, elements nested inside a namespaced element have the same namespace as their parent. So if 'task' in the example above allowed a nested 'config' element, the build file snippet would look like this:

```
<typedef resource="org/example/tasks.properties" uri="http://example.org/tasks"/>
<my:task xmlns:my="http://example.org/tasks">
  <my:config a="foo" b="bar"/>
  ...
</my:task>
```

If the element allows or requires a lot of nested elements, the prefix needs to be used for every nested element. Making the namespace the default can reduce the verbosity of the script:

```
<typedef resource="org/example/tasks.properties" uri="http://example.org/tasks"/>
<task xmlns="http://example.org/tasks">
  <config a="foo" b="bar"/>
  ...
</task>
```

Namespaces and Attributes

Attributes are only used to configure the element they belong to if:

- they have no namespace (note that the default namespace does **not** apply to attributes)
- they are in the same namespace as the element they belong to

Other attributes are simply ignored.

This means that both:

```
<my:task xmlns:my="http://example.org/tasks">
  <my:config a="foo" b="bar"/>
  ...
</my:task>
```

and

```
<my:task xmlns:my="http://example.org/tasks">
  <my:config my:a="foo" my:b="bar"/>
  ...
</my:task>
```

result in the parameters "a" and "b" being used as parameters to configure the nested "config" element.

It also means that you can use attributes from other namespaces to markup the build file with extra meta data, such as RDF and XML-Schema (whether that's a good thing or not). The same is not true for elements from unknown namespaces, which result in a error.

Mixing Elements from Different Namespaces

Now comes the difficult part: elements from different namespaces can be woven together under certain circumstances. This has a lot to do with the NewAnt FeaturesInDetail NewIntrospectionRules: Ant types and tasks are now free to accept arbritrary named types as nested elements, as long as the concrete type implements the interface expected by the task/type. The most obvious example for this is the <condition> task, which supports various nested conditions, all of which extend the interface `Condition`. To integrate a custom condition in Ant, you can now simply <typedef> the condition, and then use it anywhere where conditions are allowed (assuming the containing element has a generic `add(Condition)` or `addConfigured(Condition)` method):

```
<typedef resource="org/example/conditions.properties" uri="http://example.org/conditions"/>
<condition property="prop" xmlns="http://example.org/conditions">
  <and>
    <available file="bla.txt"/>
    <my:condition a="foo"/>
  </and>
</condition>
```

In Ant 1.6, this feature cannot be used as much as we'd all like to: a lot of code has not yet been adapted to the new introspection rules, and elements like the builtin Ant conditions and selectors are not really types in 1.6. This is expected to change in Ant 1.7.

Namespaces and Antlib

The new NewAntFeaturesInDetail AntLib feature is also very much integrated with the namespace support in Ant 1.6. Basically, you can "import" Antlibs simply by using a special scheme for the namespace URI: the `antlib`
scheme, which expects the package name in which a special `antlib.xml`
file is located.

More TBD

Namespaces and DynamicConfigurator

TBD