

Expression Improvements

This page captures ideas for improving expression handling over the course of [2.4.x](#) development.

- [Literal Expressions](#)
 - [Acceptance Tests](#)
 - [Literal Evaluation Options](#)
 - [Alternative 0: Current Implementation](#)
 - [Alternative 1: Use Apache Commons ConvertUtils](#)
 - [Alternative 2: Roll our own Interface](#)
- [Property Expressions](#)
 - [Acceptance Test](#)
 - [Property Evaluation Options](#)
- [Evaluation Context](#)
 - [Acceptance Tests](#)
 - [API Alternatives](#)
 - [Context Push / Class Context Alternative \(GeoAPI Change\)](#)
 - [Context Pull / Value Object Alternative \(GeoAPI Change\)](#)
 - [Context Pull / Lazy Value Alternative \(Geotools Change\)](#)
 - [Evaluation API Options](#)
 - [Context Push Code Examples](#)
 - [Context Pull Code Example](#)
- [Geometry Filter](#)
 - [Acceptance Tests](#)
 - [Geometry Filter Alternatives](#)
 - [Geometry Converter to JTS](#)
 - [SpatialStrategy](#)

Filter Brainstorming at Refractor:

- [Filter-Brainstorming](#) (2006_11_08)

Literal Expressions

Literal expressions (and the Filter/Expression) system are "semi-typed". Literal values can be represented as plain text or an xml fragment; but they are interpreted based on context.

This problem breaks down into two:

- Simple – an "Extension point" needs to be process string literals into a variety of contexts (like date, or date range)
- Complex – this depends on an more capable XML parser, on the bright side the Literal will be a real POJO (or a ComplexAttribute) and special evaluation code will not be needed.

Requirements:

- support for more then one format (ie Date)
- need to consider printing as well (when making requests to external services)

Gabriel has mentioned in email a Jakarta Commons package suitable for handling the String constants. See <http://jira.codehaus.org/browse/GEOT-602> for more background.

Acceptance Tests

```
<Literal>1234</Literal> <!-- used as a
string -->
<Literal>1234</Literal> <!-- used as an
integer -->
<Literal>1234.56</Literal> <!-- used as an
integer 1235-->
<Literal>123E456</Literal> <!-- used as
string -->
<Literal>123E456</Literal> <!-- used as
integer -->
<Literal>2006/11/15</Literal> <!-- used as a
date -->
<Literal>06/11/2006</Literal>
10:04:06:37<!-- used as a timestamp -->
<Literal>>true <!-- used as Boolean -->
<Literal>1</Literal> <!-- used as Boolean
-->
```

Need examples of time range

Literal Evaluation Options

The first thing to note is that handling Literals and Context may be the same problem:

- For Literal: we need a String literal interpreted according to context
- For Context: we need an Expression result interpreted according to context

If we are "Eager" we need to ensure that we handle Literals correctly at parse time, if we are lazy we can put this off to evaluation time. Give our uses cases (handling the same literal as both an Integer and String) my money is on the lazy approach.

Alternative 0: Current Implementation

🚫 Not acceptable

The current implementation(s):

AbstractExpression - number() and friends

- uses constructor reflection (search for a constructor that takes a string parameter)
AttributeType.parse(Object)
- is type and context aware, not effectively used during filter encoding

Alternative 1: Use Apache Commons ConvertUtils

⊖ Not implemented

With this approach, we simply register instances of Converter and things will be happy.

ConvertUtils is somewhat limited:

- It only allows a single converter to be registered under a class (so more than one Date format cannot be used)
- It only does parsing, not encoding

We have reports of the implementation being based on singletons and generally not being suitable.

Alternative 2: Roll our own Interface

⊕ Implemented

This approach also has the benefit of being able to wrap around ConvertUtils, or any other library that performs literal parsing, for instance *java.beans.PropertyEditor*. The interface might look something like:

```
interface ConverterFactory {
    Converter createConverter( Object source,
Class target, Hints hints;
}
interface Converter {
    boolean canConvert( Object source, Class
target );
    Object convert( Object source, Class
target );
}
```

Creating an implementation based on ConvertUtils:

```

class CommonsConverterFactory implements
ConverterFactory {
    Converter createConverter( Object source,
Class target, Hints hints ) {
        return new CommonsConverter(
CommonsUtils.lookup( target ) );
    }
}

```

And one based on PropertyEditorManager:

```

class CommonsConverter implements Converter
{
    Converter createConverter( Object source,
Class target, Hints hints ) {
        return new PropertyEditorConverter(
PropertyEditorManager.findEditor( type ) );
    }
}

```

Questions:

- ❓ When dealing with a Date object and more then one Converter is available how can one choose between them?
- ℹ️ If using the extension point, the first you get. The only way to counter act this is for the client to create a factory directly, or supply some kind of hint, liks for instance a date format.

Property Expressions

Property expressions need to be able to extract information from a number of sources:

- Feature
- Bean (ie POJOs)
- Maps & Lists
- Metadata

And they need to implement a subset of XPath (as documented by the Filter specification.

JXPath provides us with both needs.

Acceptance Test

Easy:

```
<Property>name</Property> <!-- access named
attribute -->
<Property>@gml.id</Property> <!-- access to
feature id -->
<Property>foo[2]</Property> <!-- index into
an array -->
<Property>date/year</Property> <!-- from
feature to pojo -->
```

Hard:

```
<Property>foo[bar]</Property> <!-- index
into an array dynamically -->
<sld:Label>
  Speed: <Property>SPEED</Property>
<Property>UNIT</Property>
</sld:Label>
```

FeatureCollection:

```
<Property>*/SPEED</Property> <!-- SPEED for
all contents -->
```

Property Evaluation Options

 This option was implemented

Here are the links to JXPath:

- <http://jakarta.apache.org/commons/jxpath/>
- <http://jakarta.apache.org/commons/jxpath/users-guide.html#Customizing%20JXPath>

And we will need something like the following to hook it up:

```
interface PropertyAccessor {
    /**
     * Retrived indicated content from obj.
     * <p>
     * may be collection as dictated by
xpath specification
     * </p>
     * @return result of xpath query
     */
    public Object get( Object obj, String
xpath );
    public void set( Object obj, String
xpath, Object value);
}
```

And of course a Factory...

```
interface PropertyAccessorFactory {
    boolean canHandle( Class type, String
xpath ); // note: Filter.class is a type
    PropertyHandler create( Map hints );
}
```

Full XPath support is only one test .. so lets try for the following implementations:

- SimpleFeaturePropertyAccessor – recognizes "name" as Feature attribute
- FeaturePropertyAccessor - extends the above with support for "@gml.id" as FeatureID
- BeanPropertyAccessor – Implementation that recognizes "name" as Bean property
- XPathPropertyAccesor – Implementation using JXPath out of the box with bean and collections support
- XPathFeatureAccessor – Implementation using JXPath configured to navigate feature model

Tradeoffs:

- pros: works, can engage PropertyHandler based on type and xpath
- cons: xpath is not the be all and end all, review CSW2.0 to look for other accessors

Evaluation Context

We also need to evaluate our expression into the "context" it is expected. As an example when making a query against a datastore with an Integer "size" attribute, an expression will need to be sure to provide an Integer.

Acceptance Tests

```
<function name=sin> <!-- sin( double ):  
double -->  
  <div>  
    <literal>PI</literal> <!-- constant -->  
    <literal>2</literal> <!-- double  
context -->  
  </div>  
</function>
```

```
<PropertyEqualTo>  
  
<PropertyName>integerProperty</PropertyName>  
<!-- integer property -->  
  <Add> <!-- addition needs to  
evaluate as integer -->  
    <Literal>2.0</Literal>  
    <Literal>4.0</Literal>  
  </Add>  
</PropertyEqualTo>
```

API Alternatives

Here are a couple options for providing "context" to our existing filter evaluation interfaces.

Context Push / Class Context Alternative (GeoAPI Change)

+ This option was implemented

Add an additional method to Expression which takes a Class parameter:

```
interface Expression {
    Object evaluate( Object object );
    Object evaluate( Object object, Class
context );
}
```

The addition being the *Class hint* parameter, which as it states is a hint to the expression about which type of object to return.

This requires each and every expression (including all functions) to be able to type morph.

Context Pull / Value Object Alternative (GeoAPI Change)

- This option not implemented

We can have the evaluate method return a Value Object (basically a value wrapper) rather than an explicit value:

```
interface Expression {
    Object evaluate( Object object ); //
"natural value"
    Value eval( Object object );
}
```

We can then use Value to "unpack" the value into the type required for the context:

```
interface ExpressionValue {
    Object value( Class context);
    void setValue( Object );
    Object getValue();
}
```

This approach matches the problem in that it let's us put off typing until it is needed (the same value may be used as a "comparable, string, and interger"), but it is not very cool for those expecting a real value out of evualte(Object), breaking out another method is an option.

Context Pull / Lazy Value Alternative (Geotools Change)

➖ This option incomplete

To pull this off, expressions will now behave lazily, holding onto just a String, which can be a simple value, or something more complicated like an xml fragement.

As described in <http://jira.codehaus.org/browse/GEOT-602>, break out an extension point.

Cannot recommend the use of a String internally, Style objects like Stroke often use both "#FFFFFF" and Color.WHITE in time critical code). Since this approach is similar, I would recommend simply making use of the above ExpressionValue interface internally for both Cache and Morph.

Evaulation API Options

Since type morphing is mostly covered in the preceeding section on literal I am going to focus on the API differences between context push and context pull.

Context Push Code Examples

Let's start by isolating the type morphing into a single utility class for implementators:

```
class ExpressionValue {
    ExpressionValue( Object value );
    ExpressionValue( double value );
    ...
    Object getValue( Class type );
    void putValue( Object value );
    ...
    double doubleValue();
    void putValue( double value );
    ...
    String toLiteral(); // may produce XML
    fragment if complex
    String toLiteral(URI format);
}
```

Putting together the proposed API with ExpressionValue we have:

```

class Sin implements Function {
    ...
    ...
    Object evaluate( Object object ){
        return eval( object ).getValue(
Double.class );
    }
    Object evaluate( Object object, Class
context){
        ExpressionValue theta = new
ExpressionValue( param.evaulate( object ) );
        ExpressionValue result = new
ExpressionValue( Math.sin(
theta.doubleValue() ) );

        return result.getValue( context );
    }
}

```

Tradeoffs:

- Pro: works, easy to read
- Con: no ability to support Caching
- Con: not sure class will always provide enough context, consider Date Literals perhaps additional URI hint is needed?

Context Pull Code Example

Here is what changing Expression to use ExpressionValue would look like:

```

class Eval extends Expression {
    ExpressionValue eval( Object value );
}

```

Putting this together we have:

```
class Sin implements Function, Eval {
    ...
    ...
    Object evaluate( Object object ){
        return eval( object ).getValue(
Double.class );
    }
    ExpressionValue eval( Object object ){
        ExpressionValue theta = param.eval(
object );
        return new ExpressionValue( Math.sin(
theta.doubleValue() ) );
    }
}
```

Tradeoffs:

- Pro: ExpressionValue available to those that need it, may allow support for URI based date format selection etc...
- Pro: Useful Cache for uses like Stroke (where an expression is used as both "#FFFFFF" and Color.WHITE)
- Con: Another class visible to end users (never a good thing)

Geometry Filter

As we get more and more implementations of Geometry (beyond the basic JTS) it would be nice to keep the filter love flowing.

This is a similar problem to that delth with by Property Expression (in that we need to be sure we have an implementation on hand to deal with the provided content. This time rather than simple access we need to be able to:

- Apply a Filter to the provided "Geometry"
- Morph between Geometry implementations

The morph part is well covered by the [#Evaluation Context](#), an initial implementation based on WKT can morph from everything into JTS Geometry - where the functions work. More advanced implementations like Oracle SDO Geometry, or the PostGIS Geometry classes, can take the WKB route and not lose accuracy.

The Applying a Filter part can be handled using the same stratagy as Property Expression a small interface that does

what is needed.

Acceptance Tests

```
..list of filter functions...
```

All of the above examples need to work with the cross product of:

- JTS Geometry
- GeoAPI Geometry
- Oracle Geometry
- PostGIS Geometry

Geometry Filter Alternatives

Geometry Converter to JTS

Using a WKT (or WKB) Converter to allow the Evaluation Context solution to convert to JTS would let the existing Spatial Filter Implementations work out of the box with no further effort. They would "pull" the Geometry implementation into a JTS Geometry and then proceed from there.

```
// no new code example needed - covered by  
our current needs
```

 Please note reprojection is not covered as part of this conversion "pull"; although it could be considered part of the process (our pull method simply takes a class but that can be changed).

SpatialStrategy

Letting the Geometry implementations exist at the same level as JTS (and with greater range of expression would be a good thing.

```

interface SpatialStrategy {
    Class forGeometryType(); // return
XXXX.class
    boolean contains( Object a, Object b); //
both a and b have been pulled into
XXXX.class
    boolean within( Object a, Object b ); //
etc...
    ...
}

```

Example SpatialStrategy implementation:

```

interface JTSSpatialStrategy {
    Class forGeometryType(){ return
Geometry.class; )
    boolean contains( Object a, Object b){
        Geometry aGeom = (Geometry) a;
        Geometry aGeom = (Geometry) a;
        return aGeom.contains( bGeom );
    }
    boolean within( Object a, Object b ){
        ....
    }
    ...
}

```

Our implementations would need to look up a Spatial Strategy using the factory dance as usual. A SpatialStrategy implementation for each Geometry implementation would be quick to implement and allow for optimized performance. Adding in WKB Converters would let implementations work together without data loss.

```

class ContainsUmberImpl implements Contains
{
    ...
    public void evualte( Obj obj, Class
type){
        Object geomA = param1.evualate( obj
); // get back some kind of Geom
        SpatialStratagy spatial =
ff.acquireSpatialStratagy( geomA.getClass()
);
        if( spatial == null ){ // try to
JTS
            spatial =
ff.acquireSpatialStratagy( Geometry.class );
            geomA = param1.evualate( obj,
Geometry.class ); // get back some kind
        }
        Object geomB = param2.evualate(
obj, spatial.forGeometryType());

        return spatial.contains( geomA,
geomB );
    }
    ...
}

```