

Dry Run at DataAccess Story

Introduction

Over the years a number of efforts wanted to provide complex feature support with different levels of success, and lead to the definition of the new GeoAPI interfaces and an implementation sitting as an unsupported module for some time now.

The GeoTools data access layer has moved from its own Feature/Type system to the GeoAPI SimpleFeature/Type interfaces.

GeoAPI SimpleFeature/Type are extensions to the more generic Feature/Type interfaces, that provide for convenient assumptions over the Feature structures being flat.

Now we *need* to be able of dealing with complex features, generally coming from WFS instances such as the [USGS Framework Web Feature Services](#), as well as to finally give GeoServer a transition path to serve complex features.

And we need to do it in a way that leverages all the knowledge put through the years in the current GeoTools infrastructure and allows a smooth integration of `Feature`, both for new code to be built upon it, as for existing code (both internally and library client wise) that want to transition to support complex Features.

I'm trying to figure out a way to smoothly introduce `Feature` with no detriment of all the work spent in `SimpleFeature` up till now. That is, enable complex feature support from now on, with no even need to deprecate all the `SimpleFeature` stuff, as that's working well and serving a bunch of use cases, yet enabling for new developments to leverage the use of both new complex-capable data stores as well as the existing simple-feature ones through the more generic `Feature/FeatureType`.

The big blocker factor is the lack of an appropriate **data access** API. All the GeoTools code that deals with features is settled up in terms of `SimpleFeature` and its assumptions. So the current `DataStore` API. During the work on the community-schema modules, the main glitches found with the `DataStore` API are:

- String is not enough to represent a `FeatureType` name, a qualified name is needed
- `TypeName` and `Feature` name **are not** the same thing, though a common miss-assumption for `SimpleFeatures`

Goals / API

Naming

At a `Source` level: Identification should be handled with with a qualified name rather than a `TypeName`. We are getting too many collisions. For the GeoAPI `Feature` to be located correctly we need to get rid of the idea that `TypeName == FeatureName` (eg, `"topp:states" != "topp:states_Type"`. They may even be in different namespaces!)

SimpleFeature/Type backwards compatibility, out of the box usage as the general case

We want current code to keep backwards compatible. This means the library won't break all the code written with the `SimpleFeature` assumptions. And we want all the format drivers written in simple terms to be used as the more general `Feature/Type` out of the box.

Approach(es)

There were good debate, see the following [email thread](#)

Basically in term of approaches to cover the above goals we managed this three possibilities:
Debate lead to people asking for actual code examples so the three were spiked:

1. Generics + DataStore superclass
 - code: <http://svn.geotools.org/geotools/trunk/spike/jdeolive/api>
 - example: <http://svn.geotools.org/geotools/trunk/spike/jdeolive/example>
2. Generics + FeatureSource Hierarchy
 - code: <http://svn.geotools.org/geotools/trunk/spike/gabriel/api-hierarchy>
 - example: package org.geotools.data.sample
3. Nogenerics
 - code: <http://svn.geotools.org/geotools/trunk/spike/gabriel/api-nogenerics>
 - example: package org.geotools.data.sample
4. Hybrid
(no spike, see bellow)

Generics + DataStore superclass

Introduce a superclass for `DataStore` and parametrize `FeatureSource`, `FeatureCollection` etc, based on the type and content it serves (`<FeatureType, Feature>` vs `SimpleFeatureType, SimpleFeature`).

- 1) cleaner abstract api
- 2) does not incur in naming conflicts (no need to find good names beyond the ones already in use and a superclass for `DataStore`)
- 3) introduces only one more interface, a pull up of `DataStore`. This allows for the separation of generic `Feature` capable datastores and `SimpleFeature`-only capable ones.
- 4) will break some existing client code when upgrading to geotools 2.5 due to the runtime erasing of generics. Yet, its easily fixable with a regular expression search and replace.

Generics + FeatureSource Hierarchy

Introduce a full hierarchy by pulling up parametrized versions of `DataStore`, `FeatureSource`, `FeatureCollection` but keep the current interfaces with no generics (ie, `DataStore` extends `DataAccess<SimpleFeatureType, SimpleFeature>`, `FeatureSource` extends `Source<SimpleFeatureType, SimpleFeature>`, etc).

- 1) cleaner concrete api for the `DataStore` case (ie, `SimpleFeature`)
- 2) Incurs in naming hell since the good ones are already taken
- 3) introduces a full layer of abstraction over `DataStore`, `FeatureSource/Store/Locking`, `FeatureReader/Writer`, `FeatureCollection`
- 4) Breaks less/no existing code. Only requires the addition of `Name` vs. `String` (ie. `DataStore.getNames():List<Name>` vs `DataStore.getTypeNames():String[]`)

Nogenerics

Introduce a full hierarchy by pulling up **non parametrized** versions of `DataStore`, `FeatureSource`, `FeatureCollection` and keep the current interfaces as they are, relying on Java5 return type narrowing to specialize the return types for the `SimpleFeature/Type` case, and pay the cost of new overloaded methods in the subclasses.

ie, `DataStore` implementations get two new methods each, which are overloaded versions of the originals:




```
interface DataStore{
    /** @since 2.4 */
    public void createSchema(SimpleFeatureType
featureType)
    /** @since 2.5 */
    public void createSchema(FeatureType
featureType)

    /** @since 2.4 */
    public void updateSchema(String typeName,
SimpleFeatureType featureType)
    /** @since 2.5 */
    public void updateSchema(String typeName,
FeatureType featureType)
}
```

```
interface FeatureStore{
    /** @since 2.5 */
    public Set<FeatureId>
addFeatures(FeatureCollection collection)
    /** @since 2.4 */
    public Set<FeatureId>
addFeatures(SimpleFeatureCollection
collection)

    /** @since 2.4 */
    public void setFeatures(FeatureReader
reader)
    /** @since 2.5 */
    public void setFeatures(Reader reader)
}
```



Hybrid

In this approach we try for a hybrid of (1) and (2) above.

- Use a simple super class (either one super class, or two) it does not matter
- Use generics only for the Query objects (and any other method parameters we need)

Original idea: if we use one super class:

- Data<FQ extends FeatureQuery>: is for generic Feature
- DataStore extends Data<Query>: is for SimpleFeature

The idea is nice, problem being that once you get to code you find yourself being forced to parametrize, at least, Query, FeatureCollection and FeatureReader, the last two ideally being present only in FeatureStore, but forced to propagate back to FeatureSource and DataStore to avoid type safety warnings.