Bring Back FeatureCollection

Related pages:

- org.geotools.data interface scratchpad

Disscussion goes here:

Bring back FeatureCollection!

Abstract:
This is from an email entilted 'Bring back the FeatureCollection!' In it Chris Holmes argues that a way our of our results/reader mess is to make use of this nice FeatureCollection construct we came up with, and then promptly relegated to memory only. Note you should have your coffee before reading this.

...

Thinking in the shower this morning I decided to take a look at our FeatureCollection/Iterator/Reader/Results mess.

The main thought is that FeatureResults was created to give a very similar construct to a FeatureCollection (though streaming), and perhaps we could just take it back in that direction.

Right now FeatureCollection is sort of a code word for an in-memory collection, we went to featureResults to ease the transition to streaming. But I'm thinking we should just take back FeatureCollection. Have DefaultFeatureCollection become MemoryFeatureCollection, and change DefaultFeatureResults to DefaultFeatureCollection.

Ok, on further thought it would need to be DefaultTypedFeatureCollection. The main (useful) thing that FeatureResults does that FeatureCollections do not is return the FeatureType. So we introduce the notion of a TypedFeatureCollection (something IanS and I thought about but never implemented). The TypedFeatureCollection would only allow Features that validated against the FeatureType.

Of course this gets into interesting questions, if the default is to allow multiple types in the feature collection, or demand a new interface for that. And then there are also interesting questions of combining two TypedFeatureCollections. A construct like that would be *very* useful for GML production, as you could get at all the FeatureType info directly from a single FeatureCollection construct.

Ok, I'm rambling into further implications, to bring this back to the positive aspects of this decision, FeatureReader and FeatureIterator have similar parallel structures . Reader will return the FeatureType. And it additionally has the close() method. But note that our FeatureIterator does *not* implement java.util.Iterator, so we can add the close method. So we can also introduce a TypedFeatureIterator, that will return the FeatureType, and FeatureReader can simply implement that as well.

One thing to note is that FeatureCollection has quite a few more methods than FeatureResults, since it extends Collection. But one thing to note is that most operations are optional in the Collection interface. The only none optional ones are contains, containsAll, equals, hashCode, isEmpty, iterator, size() (which is actually FeatureResults.getCount(), and toArray(). All of which could be implemented fairly easily.

DefaultFeatureCollection could change to take a Query and a FeatureSource as it's constructor, in short it would just expand on DefaultFeatureResults.

Doing this also gives us a nice interface for random access, in that a FeatureList would implement list, and provide get(int index), indexOf(), and lastIndexOf(). So FeatureSources that support random access could return List implementations, and clients could then check to see if they got a list. Or there could be an explicit getList(), which would build a random access index for an unordered collection. Check out feature/FeatureIndex.java, for some more interesting ideas that IanS had.

Granted we would probably need a TypedList, could possibly be done nicely by making a Typed interface, which just has a method FeatureType getFeatureType() (or FeatureType[] getFeatureTypes(), should decide what the default behaviour is).

Another thought is that *all* FeatureCollections would be typed, at least in some way. In this case the in memory FeatureCollection (or any FeatureCollection that allows any FeatureType), would make use of the ancestor stuff that we have implemented, all in it would be of type Feature (I'm not exactly sure how this works, but I'm thinking along the lines of the SLD stuff, that all must descend from Feature, no?). This gives a nice little constraint on what a FeatureCollection is vs. a normal collection . it must contain features. So the getSchema method would always return at least a Feature FeatureType. I think to get this to work well we'd have to introduce some type of OrFeatureType . that is a validation mechanism that can check against more than a single featureType. Perhaps a Schema class, of which a featureType represents a single instance of a feature. But a Schema could also have multiple FeatureTypes. Each Schema class could have a getFeatureTypes() call, and a FeatureTypeSchema (with just the one), could return itself. I think the key (and potentially only) method of a Schema would be the validate() method. FeatureType does not currently have a validate method, but DefaultAttributeType.Feature does, I think perhaps we should again contemplate FeatureType implementing AttributeType. Or at least have both descend from a common Schema class.

(Skip this section if you're having any trouble with all of this and read it over later)
If we have both from a common schema class then this also generates some more interesting questions. Like what we do about MultiAttributeTypes. Because a MultiAttributeType is in some ways a FeatureCollection (especially if FeatureCollections offer a validate method). For a quick review, a MultiAttributeType was my construct to basically handle a list. Never really been used (but I will when we get to joins), but it's for when there is more than one object in an Attribute, think maxOccurs greater than one in XML . you can put a bunch of little objects in there. In the easy to grasp case you'd have more than one sub features, literally a TypedFeatureCollection . all would be the same type, each in the list would have to validate against the same FeatureType. But they also could be any sort of objects, they could be strings (think all the names of people living at an address). This is a list of objects that must validate against something in common (all must be strings). So maybe we could have a ValidatingCollection, of which FeatureCollection is a subType. Ok, my ideas are straying here, but one thing to think about is maybe putting in hooks for more complex validations . we already have this field length thing, which is fairly meaningless. I believe David was going to do some work on this for his schema stuff, basically it'd be nice to have in the central api a way to express more complex validations that we don't currently support).
(end skip section)

Of course, doing this does beg some more interesting questions, like what if we actually implemented the Collection add and remove. I mean, if the FeatureSource backing it is a FeatureStore, then an add or remove could easily update the backend data format. Of course, this obviously gets messy, when we take into account locking and transactions and also the CollectionListener code we have (though I don't think it's really been used, and actually could potentially give us a way out of this problem).

The one question that remains is backwards compatibility, if we can pull it off. I'm hoping that re-using a few constructs we already have can help ease this pain. But the main point is that we should take things back to this great paradigm that IanS brought to the fore, of really making use of the Collection framework. It makes it far, far easier for java programmers to get acquainted, they can just see that getFeatures() returns a Collection, and they instantly know what to do with it, and then from there can learn the more optimized ways of doing things.

```
FeatureCollection myFeatures =
dataStore.getFeatureSource('roads').getFeatures();
Feature myFeature = myFeatures.features().next();
```

Also, a potentially interesting idea for FileDataStores . having it

implement FeatureSource? Or rather maybe call it SingleDataStore? It just might be cool if you could say

SingleDataStore store = new ShapefileDataStore(myUrl);
Feature myFeature = store.getFeatures().features().next();

(semi(un)related thoughts, just to get people started on what I'm going to write up next)
Ok, one last thought for everyone to ponder . the file and directory datastore question brings this up. Basically the thought of a multi-datastore, what happens when you combine two datastores? It could still easily implement DataStore, just with more featureTypes, right? When we get into joins and views it does start to beg the question of what a DataStore actually is. Especially when you add datarespository and discovery stuff (is that stuff gone now? I haven't been able to update my repository, and it's still all over my DataStore). I don't have any coherent thoughts on all this, but we should really think about what a DataStore is, and I think we should maybe split it up into more interfaces, instead of the current direction of adding ever more methods, so that each interface represents clearly what the DataStore is, and what you can do with it. Perhaps something like PhysicalDataStore, which FileDataStore and JDBCDataStore extend, and then a parallel VirtualDataStore, that is the result of combinations of DataStores (or perhaps that can also return 'views'). And/or a MultiDataStore, which DataRepository could then extend, and add the locking functionality. Discovery could then extend DataRepository or something. Speaking of which I would still appreciate a write up of the 'discovery' functionality that Refractions needs, and what their metadata plans/needs are, as I never got that explanation in the IRC from David. At the very least I'd like to be sure we are on the same page of what metadata means.
(end semi(un)related thoughts)

Ok, enough for today. Next up for me I will examine some of the questions I raise above, especially the implications of FeatureViews (my initial thoughts are actually in the direction of taking out QueryAs and only giving people As access in FeatureSources), and how to go about decoupling Features from the data sources they are defined in (mappings and joins and whatnot). I think it'd be really cool to take James's idea of defining common FeatureTypes even farther, so you could define your FeatureSource by the FeatureType you wanted, and supply it with an appropriate mapping object (and an object on what should be mapped from, the join to perform, the table to select from, the bypassSql statement, ect.)