Tupelo Kernel API's

Tupelo has a modular, broker architecture providing a consistent set of data and metadata operations for a variety of heterogeneous storage and retrieval implementations. The purpose of this design is to enable interoperability between these underlying implementations.

There are several ways in which Tupelo's data and metadata model differ from relational databases, filesystems, and object stores, and it is important to understand these in order to use the kernel API's effectively:

- All entities are globally identified with URI's. Local identifiers such as pathnames, XPaths, or primary keys are not used to to identify data objects, or the entities described by metadata.
- Tupelo manages data and metadata. Data items are binary large objects (BLOB's) addressed via URI's. Metadata is in the form of RDF statements, which can describe any entity, including real-world entities and abstract concepts (i.e., it is not limited to describing files or other digital objects).
- All access to data and metadata is accomplished via Operators, which are performed by Contexts. Context implementations are responsible for mapping the abstract operations described by Operators to the underlying implementation they represent. A Context may delegate an Operator to another Context. Contexts can also transform Operators before and after performing them.
- Tupelo can be extended by developing new Context implementations, and by developing new Operators.

Operator architecture and core operators

The Tupelo kernel defines two kinds of operators:

1. Data operators, which provide the ability to read, write, and delete binary large objects (BLOB's)
2. Metadata operators, which provide the ability to assert and retract RDF statements, and search for RDF statements matching complex query patterns. Some Context implementations provide complete support for SPARQL; others provide support for a subset of SPARQL.

Operators are stateful objects. When a Context performs an operator, it modifies the state of the object (rather than returning a value) to include any results. Dispatching operators to Context implementations is done dynamically, rather than statically. When an operator is performed, the Tupelo kernel selects the most specific implementation based on the runtime class of the Operator. This allows for "filtering" behavior where a parent Context can handle all Operators in a certain way (e.g., logging or tracing) without losing the ability to later dispatch the Operator to the correct implementation in some delegate Context. However, this strategy also results in some performance and maintenance issues, and so may be retired in future versions of Tupelo.

Example

This example Context implementation allows files to be read from a local filesystem, by mapping BLOB URI's with a certain prefix to pathnames:

```
package org.tupeloproject.kernel;

import java.io.FileInputStream;
import java.io.FileNotFoundException;

public class ExampleContext extends Context {
    public void doPerform(BlobFetcher bf) throws OperatorException {
        String pathname = bf.getUri().toString().replaceFirst("http://foo.bar.baz/quux", "/usr/share/data
/fbbq");
        try {
            bf.setInputStream(new FileInputStream(pathname));
        } catch (FileNotFoundException e) {
            throw new NotFoundException("no file found for URI "+bf.getUri(), e);
        }
    }
}
```

And here's how it would be used to read a file:

```
ExampleContext ec = new ExampleContext();
BlobFetcher bf = new BlobFetcher();
bf.setSubject(Resource.uriRef("http://foo.bar.baz/quux/some/file.txt"));
ec.perform(bf);
InputStream is = bf.getInputStream();
...
```