ContextGuidedIngest

Note: This is a technical brief, and presumes knowledge of DSpace software architecture and internals. End-user documentation is forthcoming.

Introduction

The variety, nature, and complexity of pathways along which content travels to get into the repository has greatly increased since DSpace was first launched, yet the basic platform mechanisms to support ingest have changed little to none. Originally, DSpace ingest was seen foremost as a content author's web submission process, comprising a fixed, multipage set of forms, followed by an optional, but also fixed, set of workflow stages that largely just replayed the submission steps for other actors. Since this sequence of operations was suspendable and restartable (thus beyond what a http session could manage), it was necessary to persist state information relevant to the ongoing submission or workflow object. Architecturally, this need was addressed by creating content wrapper objects (viz. WorkspaceItem and WorkflowItem) which held - in addition to a reference to the item being submitted - various specific properties used in the ingest process (last page reached, destination collection, etc). Since the wrappers are first-class objects in the data model, they belong to the DB schema, and the content API.

However, as new means of ingest were added, and existing processes were made more flexible and configurable, the brittleness of the simple wrapper approach becomes clear. For example, when ingest is via SWORD (i.e. no web submission form), how should properties like 'isPublishedBefore' associated with the 'initial questions' be set? Or suppose we want to add a *new* question to the initial questions to capture a content format type, where do we store it? Wrapper properties are both context-specific and non-extensible (hard-coded), with the undesirable result that to change them for customization purposes we must alter the wrapper classes themselves. And, as just noted above, this means changing the content API itself - not an outcome that good software design should necessitate.

CGI ('Context-Guided-Ingest') is an attempt the remedy this problem, and improve ingest functionality generally, by providing a small set of service APIs and implementations that constitute more flexible and extensible equivalents to the content API wrappers. To ensure compatibility with previous work, as well as enhance general system modularity, CGI will be offered as a separate **add-on** or **module** and require no changes to the content API. Of course to utilize these services, new application code will have to be written. Over time, if CGI services prove useful, it may make sense to begin to abandon or deprecate the wrapper approach, but this will not be an immediate requirement of using CGI.

The Model

Concepts

To best describe the services, we will introduce a few terms with specific meaning in the domain model of CGI. First, an **IngestContext** (or merely **context** when contextually clear 😉) means a container of persistent state with a determinate life-cycle associated with a *specific* DSpace item. One could imagine other contexts (for Collections, etc), but since Items are the units of ingest, the range of the IngestContexts is restricted to them. An **attribute** is a simple name/value pair that belongs to the persistent state of the context. Generally, the life-cycle of an IngestContext begins whenever an API call asserting an attribute on a DSpace item occurs, and ends whenever the item is installed in the repository. The intent is that the context will *live* for the duration of the ingest, and then be discarded. This is very important to understand, that the IngestContext does not represent any kind of permanent 'extended item metadata' facility, although it can of course hold metadata for the item. Finally, an **IngestResource** (again, usually just **resource**), is an instance of any class that can be utilized to perform ingest functions (in UI, workflow, etc) that is *not* specific to an item. Examples of resources might be metadata templates, input forms, submission steps, curation task sets, etc. CGI places no restrictions on what could count as a resource.

In broad strokes then, CGI provides the following service APIs (and clients):

- IngestContext attribute management for items during ingest (ingest code)
- IngestResource mapping: attaching lookup keys to resources (repository or collection managers, typically)

Combining these 2 services allows application code to select appropriate resources based on context attributes: thereby having context help guide ingest.

Service APIs

To make the preceding descriptions a little more concrete, but also to illustrate how simple they might be, here are some prototype APIs.
NB: these are for illustrative purposes only, and are subject to change or outright abandonment.
ResourceMapService

```
// map key to instance of resource Class resClass identified by resId
public void map(String key, Class resClass, String resId);

// remove said mapping
public void unmap(String key, Class resClass, String resId);

// return all keys mapped to resource
public Set<String> keySet(Class resClass, String resId);
```

IngestService

```
// set an attribute in the IngestContext for item
public void setAttribute(int itemId, String name, String value);

// get an attribute from the IngestContext for item
public void getAttribute(int itemId, String name);

// remove Ingest context
public void clear(int itemId);

// obtain the resource we need
public <T> T findResource(int itemId, Class<T> resClass);
```

Finding Resources

The observant reader will notice that the IngestService *findResource* method returns a resource (of a given type) for an item without any additional parameters or information. How is this possible? On what basis is the selection made? CGI assumes that, in general, each resource type bears a similar relationship with each of its items, and thus that it can be expressed in a rule or formula. For reasons that will become apparent, these rules are known as **expressions** in a small grammar called *RCL* (Resource Composition Language). In most cases, RCL just amounts to a very simple template. For example, suppose we wanted to express the rule that DSpace currently uses in selecting an input form for configurable submission. We would write the property containing the expression as:

```
org.dspace.app.util.DCInputSet=collection:?,collection:default
```

That is, the resource class name is the property key, and the value is the RCL expression. The expression means: select the resource mapped to the key formed by the current ingest context attribute 'collection' (if any), otherwise look for a mapping with the key 'collection:default'. More will be said about RCL when we discuss resource composition. We can already begin to see additional flexibility:

```
org.dspace.app.util.DCInputSet=format:?,collection:default
```

By this small change, we have now effectively implemented format-type based submission (provided we have mapped appropriate resources to format types, and have code that captures the format type and sets a context attribute).
These properties - one for each ingest resource type we wish to define - can live in standard configuration locations, e.g.

```
{dspace.dir}/config/modules/cgi.cfg
```

It is the responsibility of the agent configuring RCL expressions to ensure that a default resource is always available, if this is the desired behavior; CGI will happily allow an expression to evaluate to null.

Composing Resources

We alluded to the fact that the rule language used to select resources from resource keys was called a **composition** language. To briefly elucidate, some resource types have a natural (de)composability or combinability that CGI can usefully exploit. Let us take metadata templates as an example. Currently in DSpace, one can (optionally) define one template for each collection. If defined, this template is *applied* by assigning the metadata values in the template to new items submitted to that collection. Conceptually, the set of values making up the template could be a combination (sum) of other templates, say one that is used for the collection, and one that is used for a given content type. RCL will give us the ability to do this declaratively:

```
org.dspace.app.util.MetadataTemplate=format:?+collection:?
```

This expression would combine the format template with the collection template to deliver a single resource. Not all resources are as naturally composable as metadata templates (after all, DC by definition allows repeatable values), but the flexibility it offers tempts one to reimplement them to be so.

The Implementation

For clients of the CGI services, knowledge of the APIs and configuration properties discussed above should be sufficient to obtain the primary benefits of the facility: it makes no difference how they are implemented. But for the curious, the following discussion details some of the key decisions and trade-offs that could be encountered in creating an acceptable, performant, and sustainable CGI implementation. And even the base service implementation raises numerous questions about user interfaces (esp. to the ResourceMapping service, but possibly also the ContextService) that might invoke them, which the base CGI does not address directly at all.

Persistence

Perhaps the biggest challenge concerns how persistence can be managed. Recall that the IngestContexts, ResourceMaps, etc are all persistent, as well as the Resource objects themselves. The latter already mostly exist, and have persistent serializations in XML files (input-forms.xml and that crowd), so we will split the topic into two parts: context and mapping persistence, and resource persistence.

Context and Mapping Persistence

Given that IngestContexts may be read and written to frequently, most file-based serializations (e.g. XML) do not seem attractive. The obvious alternative is the RDBMS, but we recall that CGI is an add-on, so we are reluctant to graft tables or columns to the standard DSpace *database_schema.sql*. Worse still, we have no Oracle licenses, (or expertise), etc. so cannot ensure that our database logic is portable across vendors. In fact, there is no existing practice for add-ons using DB tables at all. Is there a way forward? Fortunately, considerable work has been done in the area of virtualizing access to SQL data sources since DSpace was first written, and industry-standard, widely supported, performant, open source tools exist. The current Java enterprise standard is JPA (Java Persistence API), which we can use as the persistence layer for CGI data. These so-called 'ORM' (object-relational mapping) tools combine Java native OO semantics with SQL constructs (e.g. query languages) to provide familiar but powerful programming idioms. Add to this convenient Java 5-style annotations to POJOs, and the result is concise but readable database code. As an illustration, here is a complete implementation of the IngestContext service class (remember again, this is merely demo code):

```
@Entity
@NamedQuery(name="ic.findByItem",
            query="SELECT c FROM IngestContext c WHERE c.itemId = :itemId")
public class IngestContext {
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;
    private int itemId;
    @ElementCollection
    private Map<String, String> attributes = new HashMap<String, String>();

    public IngestContext() {}

    public String getAttribute(String name) {
        return attributes.get(name);
    }

    public void setAttribute(String name, String value) {
        attributes.put(name, value);
    }
}
```

The JPA persistence provider (e.g. Hibernate) code will worry about creating database schema, ensuring the correct SQL dialect for every vendor, etc. - responsibilities we are quite content to delegate.

Resource Persistence

The situation with Ingest Resources is quite different, as noted above: current DSpace practice (which basically all derives from configurable submission) is to encode resource data in an XML file, which is parsed at runtime (typically by an affiliated 'Reader' helper class) to create immutable (read-only) access objects. In this sense, there really is no resource persistence problem, since XML disk files are quite persistent. Rather, resource **access** is the real problem: the CGI implementation provides resource objects, and would prefer a uniform way of accessing them. In fact, it is interesting to note that the same XML files typically **also** contain what CGI would call a resource map: that is, a set of values mapped to resource instances. These facts suggest a number of strategies for working with XML-based resources. We summarize each below, noting some costs and benefits. But it is quite important to understand first, that these strategies can be combined opportunistically, and second, that any strategy can be revisited or overturned as new developer resources or time permits. Past experience suggests that constraints on developer time and availability will require low-barrier strategies to be adopted initially, with more robust ones pursued later.

Grandfathering

In this case, we do (almost) nothing to the resource implementation itself. The existing Readers are used to fetch the objects identified by the service. In this case, the main cost is loss of uniform access: for each such resource, CGI we have to have hard-coded knowledge of how to obtain it. Presumably this will not be intolerable for a small number of resource types. And at worst, today DSpace has perhaps four to six resource types (metadata templates, input forms, submission configuration, curation task set, etc). But even if resources are grandfathered, the map data co-resident in the XML files cannot be. One would have to parse and load this data into resource mapping persistent objects, and synchronization of changes would be clumsy.

Shimming (DAOs)

If the expectation is that there will likely always be heterogeneous resource persistence solutions (some XML files, some DB-resident, some flat file, etc), we could still achieve CGI uniform resource access at the cost of adding a layer of indirection, a shim, or interface for resources. Essentially, this would entail a new family of DAO-like objects that could abstract the actual means of obtaining resource data (i.e. parsing an XML file, vs a database query). The CGI service code would only work with these DAO surrogates. The cost here is a fair amount of glue code, which could in the end be discarded if a single resource persistence method is adopted.

Peer Replacement (JPA)

Finally, we could imagine beginning to replace each current resource type with an equivalent (a *peer*) whose state is persisted in the database in the same way as other CGI data - i.e. managed via JPA. This is architecturally fairly *pure* in that each resource would be a simple POJO but would be persistent in a portable way. It would, however, have far more upfront costs than any of the other strategies because:

- we could not re-use much if any of the XML-based code, instead having to re-implement the resource class
- we would have to provide conversion services so that the XML-resident data was migrated to the new DB tables
- since the resource data only now lives in a database, we would likely also have to write a UI for editing (probably a different UI for each type, in fact)

Still, if we could eventually achieve this state for every resource type, it would answer the wishes of quite a few DSpace users who want UI editable input forms, etc

User Interface

It will have been observed that nothing has been said about end user interactions with CGI services or functionality. In part, this is due to the fact that CGI is really a set of infrastructure services for application code, not the direct end-user. But it clearly implies that users in some way will have to manage the CGI service data, which primarily is the resource mappings. Also, for each resource type that is implemented as a JPA managed bean (see peer replacement above), we must provide some administrative UI (or other tools), since obviously we cannot require raw RDBMS administration to manage those resources and their mappings. This requirement is an additional significant cost to be factored into the calculation about which near and long-term strategy can be pursued. It is realistic to suppose that for some significant period, the old (XML editing) and new (DB-backed, UI edited) methods will co-exist.