Declarative Data Publication with DaCHS

Markus Demleitner

Universität Heidelberg, Zentrum für Astronomie, Heidelberg, Germany
msdemlei@ari.uni-heidelberg.de

Abstract. I discuss how the DaCHS data publication package tries to ensure long-time viability of data publications, in particular to the Virtual Observatory. This happens first and formost by introducing a declarative layer ("state the problem, not the solution") in the entire publication chain from ingestion to service operation to registration. I show some examples for the constructs making up the declarative layer, as well as examples for where declarative techniques were found hard to apply. I also mention some lessons learned where having a more consicously declarative mindset would have led to clearer names and perhaps even structures. A final section investigates the effects of declarative techniques by obtaining empirical estimates for how well they insolated data centre operators from having to do large updates during a major platform change, in this case the migration from Python 2 to Python 3.

1. Introduction

When publishing scientific data, one has to

- extract information from bytestreams supplied by the data providers (for instance FITS or CSV, often severely lacking in metadata and more often than not flawed in several respects) into malleable, normally relational, well-documented representations,
- offer standard (query) interfaces over the these representations,
- produce suitable bytestreams again to respond to users' requests,
- produce standard metadata for the data collections and disseminate it ("registration").

Each of these steps can be treated *imperatively*, i.e., by code that instructs the computer what to do in which sequence. However, that approach is dangerous when data centres grow from just one data collection to more and more of them. This is because many procedures are *almost*, but not quite, the same from one resource to the next. It is then rather tempting to simply copy the imperative code and apply the slight modifications inline.

This is quick and simple at first, but will inevitably lead to a situation in which it is almost impossible to make changes because their impact on all the small, subtly different procedures is impossible to assess, and, perhaps worse, it is extremely hard to track which bug fixes need to propagate where.

Declarative techniques revolve about the idea of stating the problem and letting the computer work out the steps to solve it. They often lead to more compact specifications, and they allow data publishers to isolate whatever descriptions they need per individual resource against changes in the underlying tooling or the external world. This is crucial for sustainable data publication because a data centre may have hundreds of thousands of data collections. Any change that would require changes to even a moderate part of these data collections will quickly become prohibitively cumbersome.

This contribution describes declarative techniques as offered by a the integrated Virtual Observatory server package DaCHS (Demleitner 2018; Demleitner et al. 2014). None of what I report here is particularly original; all non-trivial data centres employ similar techniques. Still, I think it is worthwhile to give these techniques names, and to state where they are and are perhaps not helpful.

In the rest of this article, I will first say a few more words about DaCHS. I will then discuss some examples for where it employs declarative techniques – and where it does not. Section 4 then discusses how well these stood up to a rather major upheaval of the software.

2. DaCHS

DaCHS is a software package for publishing astronomical data developed since 2007, mainly within the German Astronomical Virtual Observatory. It is currently in use by about 50 data publishers worldwide. It includes an ingestion component, pervasive metadata management, implementations of the major VO protocols including a publishing registry, as well as a network server exposing these. All components are fed by a single source of metadata per resource[1], the resource descriptor (RD), where "metadata" is to be understood in a rather general sense. A typcial RD contains (conventionally, but not necessarily, in this sequence):

- the global VOResource metadata, which is an extension of Dublin Core (title, description, creators, various curation dates, instruments, etc.)
- table metadata, in particular comprensive column metadata (descriptions, units, UCDs, types, etc.)
- ingestion rules, i.e., instructions for how to turn the artefacts delivered by the data providers into standards-compliant representations of dataset metadata (for nonstandard datasets, standards-compliant formats can be generated, too, but that is generally done within services); this includes specifications on where to parse from, how to parse (via "grammars"), and rules for how to map the parse results – generally, sequence of string-string mappings – into tables
- service definitions which, for instance for datalink, can be rather involved
- regression tests, consisting of recipes to build URIs and assertions about the responses to expect from them.

[1] "Resource" here by and large is a data collection, which in turn is best thought of as "astronomical data sharing a common set of metadata".

In a typical DaCHS deployment, there are between a few and several dozen such RDs, usually one per published database schema. The RDs should be considered source code, and should be kept under version control; the VESPA project (Erard et al. 2021) for instance, requires its contributors to have the RDs for their services in a central version control system. Against that, data files should not normally be version controlled by publishers; after all, the publishers are normally not supposed to alter them.

3. Declarative Techniques

To better flesh out what we mean by "declarative" and to show where being declarative is straightforward and beneficial and where it may not be, in this section we show snippets from RDs illustrating certain DaCHS features; we follow the conventional sequence of elements outlined above.

3.1. Global Metadata

By its nature, defining metadata of the Dublin Core-type is declarative: You assign a title or, say, authors to a record, and that is almost it. However, even in this field there are complications, in particular when you have sequences of complex metadata items. Consider VOResource (Plante et al. 2018), the VO's main metadata model used to describe resources in the Registry. In VOResource, creators – each of which can have a name, identifiers, and a logo –, are of this type of metadata. DaCHS offers a shortcut to declare them concisely:

```
<meta name="creator">Bailer-Jones, C.A.L.;
  Rybizki, J.; Fouesneau, M.; Mantelet, G.;
  Andrae, R.</meta>
```

In VOResource, this turns into

```
  <creator>
    <name>Bailer-Jones, C.A.L.</name>
  </creator>
  <creator>
    <name>Rybizki, J.</name>
  </creator>...
```

Writing imperative code to produce XML like this is no longer trivial. Specifying such metadata in code would certainly become a migration hurdle once metadata formats change or new metadata formats need to be supported. With declarative metadata, the data descriptions can in all likelihood be kept stable, and changes are only required in one place rather than potentially thousands of places.

3.2. Table Metadata

Central for table metadata is the list of columns together with their metadata. In DaCHS, this is declared like this:

```
<column name="err_flux" type="real"
  unit="mJy" ucd="stat.error;phot.flux"
```

```
  tablehead="Err(flux)"
  description="Error in mean flux in this band"
  verbLevel="15"/>
```

These declarations are useful during ingestion (for instance, to do automatic type conversion), during query processing (e.g., to have rich metadata in responses), while producing Registry records (e.g., to generate metadata for service parameters), and so forth.

While it is unlikely that metadata of this type will be built using imperative code, some pieces of metadata do have operational effects. For instance, when producing HTML tables, DaCHS will turn columns with a UCD of *meta.bib.bibcode* into links into ADS/SciX if it considers the column content sufficiently bibcode-like; in other output formats this piece of metadata has no effect at all, and bibcodes are handed through as is.

Table metadata, however, can be far more involved than just column lists. For instance, to ensure certain properties on a table (e.g., "mappable to Obscore"; "has a spatial index"), DaCHS uses mixins. A table suitable for SIAP (i.e., the simple VO protocol for searching for images) and Obscore (a table schema for observational data) querying would look like this:

```
<table id="images"...>
  <mixin>//siap#pgs</mixin>
  <mixin
    collectionName="'Maidanak Lenses'"
    facilityName="'Maidanak Observatory'"
    targetName="object"
    targetClass="'GravLens'"
    expTime="exposure"
    createDIDIndex="true"
  >//obscore#publishSIAP</mixin>
```

The first mixin makes sure the table contains all columns necessary in a SIAP (version 1) response. The second builds a map of this table to obscore and causes DaCHS to include whatever metadata the present table contains in the data centre's obscore view. This requires some extra metadata, declared in mixin attributes.

The example also shows how easy it is to unnecessarily think imperatively; consider the attribute `createDIDIndex="true"` in the publishSIAP mixin. That the attribute name is a verb phrase is a telltale sign that one is asking the publisher to tell the system what to do rather than what the problem is. In other words, one is being un-declarative. That is not always wrong, but often a sign of an unintuitive design.

This is the case in this particular example, too. The background is that there are dataset identifiers (DIDs) – conceptually, a primary key for data products in the VO – in the images table, and to keep queries against the obscore view fast, nontrivial tables will need an index on them. In many cases, the original tables will have such an index, but sometimes – as for this table – the DIDs are computed, and hence an index over these computed values needs to be created.

After these considerations, calling the attribute, perhaps, `computedDID` would have been clearer. An added benefit of stating the problem and not the solution is that the attribute name would still make sense when computed DIDs required further special treatment.

3.3. Grammars

The first step of data ingestion is to take the byte streams coming from the data providers and turn them into some more structured form. In the case of DaCHS, that more structured form is an iterator over dictionaries mapping string-valued keys to string values (where DaCHS will keep typed values for source formats already yielding these, as in the case of FITS binary tables).

In the simplest case, one could obtain dictionaries of FITS header keys like this:

```
<fitsProdGrammar/>
```

Much more typically, grammars have parameters and rowfilters, as in:

```
<fitsProdGrammar qnd="True">
  <maxHeaderBlocks>80</maxHeaderBlocks>
  <rowfilter procDef="//products#define">
    <bind key="table">"\schema.iris"</bind>
    <bind key="accref">\inputRelativePath{True}</bind>
  </rowfilter>
</fitsProdGrammar>
```

The rowfilter technically is a procedure, and indeed they often have more of a "do-this" (i.e., imperative) character than I would like. Here, even the name ("define") actually is a verb.

As above, it would be preferable not only from a declarative point of view to say "X has the property Y". Many rowfilters actually try to have a UI like that, including the rather common `//products#define`. Without considerations for backwards compatiblity, I would certainly rename it.

Grammar parameters, as it were, the statements of the problem, can be fairly rich. As an accessible example, the statement of the problem of parsing a column-by-column defined ASCII table would be stated like this (instead of writing slicing code):

```
<columnGrammar gunzip="True">
  <colDefs>
    ogle:1-6
    npoint: 8-10
    pmra_mas: 11-17 [...]
  </colDefs>
</columnGrammar>
```

3.4. Declarative Mapping Rules

In DaCHS ingestion, mapping is when the machine takes the string-string mappings coming out of the grammars and bends them into nice, relational, typed data. In DaCHS, that could be through map elements:

```
<map key="ssa_dateObs" source="obsdate" nullExcs="ValueError"/>
```

This means, roughly: "the column `ssa_dateObs` (with type and other metadata as above) is to be filled from the obsdate source field. During the automatic conversion, I expect ValueErrors, and they indicate missing data." Consider for a moment that

this declaration replaces quite a lot of default type conversion and exception handling code already; and while the element name `map` has a slightly suspicious imperative twang to it, this is still a purely declarative problem statement: "obsdate needs to fill ssa_dateObs; computer, do what's necessary and make sure you do the transforms necessary based on what metadata you have".

Based on the *column* metadata, DaCHS will infer many such mappings itself if the operator tells it to, either by stating that grammar keys match column names (using an idmaps element) or by just defining pairs of grammar keys and column names (using a simplemaps element).

To give another example where a quick design ended up needlessly implying an action where what actually happens is a declaration is in this snippet:

```
 <apply procDef="//ssap#fill-plainlocation">
   <bind key="ra">@ra</bind>
   <bind key="dec">@dec</bind>
   <bind key="aperture">1/3600.</bind>
</apply>
```

What this really says is: In this source table, the observed coordinates are in to separate columns. Also, the observation's aperture has been an arcsecond. The declarative nature of this construct would become much clearer with a noun phrase as its name; `spatial-coverage-source`, perhaps.

3.5. Non-declarative Ingestion Rules

Early on, I thought about some clever way to "declare" transformations to be performed on ingestion. However, there are just too many transformations and, frankly, hacks one has to apply to incoming data. A declarative apparatus to try and cope with them would probably end up at least as complex as Prolog and presumably almost as fragile as any imperative Python code. Hence, DaCHS admits Python expressions in mapping rules:

```
<map key="target_comment"
  >"(from %s) %s"%(@tfrom, @tcomment)</map>
<map key="offset_v">@offsets_v if @offsets else None</map>
<map key="mag_arr">[(NaN if v==99. else v)
  for v in [@mag1, @mag2, @mag3, @mag4, @mag5, @mag6, @mag7]]
</map>
```

Actually, with custom apply-s, operators are not restricted to expressions; they can use the full power of Python statements. Against using Python as a configuration language throughout, howerver, at least the imperative code is isolated in well-known positions.

3.6. Declarative Service Definition

In DaCHS, a service brings together

- a core that takes parameters, does a computation, and returns tables – for instance, an ssapCore would know about the various quirks of translating (parsed) SSAP (a parameter-based query protocol for spectra) query parameters into database constrataints, as well as the table structure and extra metadata that SSAP requires

- with one or more renderers; these take and produce bytestreams. For instance, the form renderer parses from what browsers produce from web forms and outputs HTML tables, whereas the ssap.xml renderer parses parameters in the SSAP syntax and produces SSAP-compliant VOTables.

Service elements are also where publications to the Registry (or to some local service roster) are specified. For instance:

```
<service id="svc_mrs" allowed="form,ssap.xml">
  <meta name="title">LAMOST DR6 Medium Resolution Spectra</meta>

  <publish render="ssap.xml" sets="ivo_managed"/>
  <publish render="form" sets="ivo_managed,local" service="mrs_web"/>

  <ssapCore queriedTable="ssa_mrs">
    <FEED source="//ssap#hcd_condDescs"/>
    <condDesc buildFrom="coadd"/>
  </ssapCore>
</service>
```

The main configuration item here is service parameters and their processing. For standard protocols like SSAP there are pre-configured packs of parameters that are pulled in through a macro facility (which for reasons of space I cannot discuss here). This is what the FEED element effects. DaCHS knows how to build appropriate HTTP parameters for most sorts of database columns (and adapt them for various parameter styles). That is what one asks for with the `buildFrom` attribute on `condDesc` elements.

3.7. Non-declarative Regression tests

Each RD should come with regression tests for the services it describes. While the URIs are still specified more or less declaratively, for the assertions DaCHS relies on pyunit-derived Python code.

```
<regTest title="LAMOST 6 LRS original FITS">
  <url>sdl_lrs/static/6/101026.fits</url>
  <code>
    self.assertHasStrings("SIMPLE  =",
      "MAG1    =                17.58")
  </code>
</regTest>
```

This came at a price. Of the changes to RDs that were necessary during the migration to Python 3 discussed in the next section, a wide majority concerned the regression tests. Would a declarative way to say what the example says (“expect these strings”) have helped? To some extent, probably. But on closer inspection it has to be admitted that many of the changes to the regression tests were actually due to subtle behavioural changes (as in “different serialisation or the last few bits of floats”); it is thus likely that possible savings by more declarative assertions in terms of lines touched would have been a small factor at best.

4. Software Empiricism

In 2019, we finally ported DaCHS from Python 2 to Python 3, which entailed significant changes in the underlying web framework, too.

To find out how successful the programme to have resource metadata as declarative as possible was in isolating service operators from changes to the software and network environment, I have tried to estimate the effort by counting how many lines had to change in the resource descriptors.

We still keep these under subversion version control[2]. It is hence relatively easy to obtain a diff over the migration period, which for us is revisions 7052 through 7190. That yields 7661 lines of context diff. I manually annotated this diff[3] with letters to find:

- 161 lines of change in the declarative parts of RDs, templates, and the like (*R* in the first column).
- 325 lines of change in the procedural parts of RDs (mostly regression tests; *r* in the first column)
- 1496 lines of change in ancillary python modules (e.g., validators, specialised grammars, calibration scripts, custom services; *p* in the first column).

All unannotated lines are either context or changes unrelated to the migration.

Turning these numbers into the more useful metric "percentage of lines touched" is not straightforward because one line touched normally produces two lines of context diff and because it is not entirely trivial to determine how many lines of the three categories are actually in the source code repository. However, within a factor of two, we can state that 100+200 (the r and R changes, corrected for context diffs) lines had to change for the Python3 port within $\sim 70'000$ lines of RD material (which is $> 80\%$ declarative).

About 800 lines were touched within $\sim 30'000$ lines in operator-level Python code.

In comparison, the core DaCHS software port to Python 3 is almost 50'000 lines of context diff (of currently 110'000 source lines).

5. Conclusion

The Python 3 port of DaCHS and the data centre metadata required changes in, by proportion of lines touched, approximately:

- about 0.2% of declarative RD specifications
- about 1% of imperative RD material
- about 2% of the operator-level, service-associated imperative code

---

[2] `http://svn.ari.uni-heidelberg.de/svn/gavo/hdinputs/`; make sure you use the http version, as the https one requires credentials.

[3] The result is found at `https://docs.g-vo.org/dachs2-migration.diff`

- about 20% of the core DaCHS code (mainly due to a framework change, bytes vs. strings, quite a bit less the syntax changes)

That imperative material within RDs was more stable than imperative code in modules was because most of it talks to DaCHS APIs we could keep stable, and very little of it uses constructs with interesting syntax; similarly, operator-level code was significantly easier to port than core code because again it was somewhat isolated from underlying changes by DaCHS and normally is structurally simpler (there are exceptions, of course). But then DaCHS itself was relatively hard to port, in particular because the core code had to deal with strings-vs-bytes in many places, and the framework change resulted in many changes in the http-facing parts.

Still, without the declarative techniques used in the implementation of the actual services, the Python 3 migration of the GAVO data centre would have been impossible with the resources we had.

References

Demleitner, M. 2018, DaCHS: Data Center Helper Suite, Astrophysics Source Code Library. 1804.005

Demleitner, M., Neves, M. C., Rothmaier, F., & Wambsganss, J. 2014, Astronomy and Computing, 7, 27. 1408.5733

Erard, S., Cecconi, B., Le Sidaner, P., Rossi, A. P., Tomasik, L., Ivanovski, S., Schmitt, B., André, N., Trompet, L., Scherf, M., Hueso, R., Demleitner, M., Manaud, N., Taylor, M., et al. 2021, in 5th Planetary Data Workshop & Planetary Science Informatics & Analytics, vol. 2549 of LPI Contributions, 7073

Plante, R., Demleitner, M., Benson, K., Graham, M., Greene, G., Harrison, P., Lemson, G., Linde, T., & Rixon, G. 2018, VOResource: an XML Encoding Schema for Resource Metadata Version 1.1, IVOA Recommendation 25 June 2018