Development notes for GAVO DaCHS

Author: Markus Demleitner
Email: gavo@ari.uni-heidelberg.de

Contents

Some of this is severely out of date.

Package Layout

The following rules should be followed as regards subpackages of gavo in order to keep the modules dependency graph manageable (and facilitate factoring out libraries).

- Each functionality block is in a subpackage, the ___init___ for which contains the main functions, classes, etc., of the sub-package interface most clients will be concerned with. Clients needing special tricks may still import individual modules (but then they're in much larger danger of breaking). What's in ___init___ should be considered "public interface" and hence changed very carefully if at all.
- Within each subpackage, *no* module imports the sub-package, i.e., a module in base never says "from gavo import base"
- A subpackage may have a module common, containing objects that multiple modules within that subpackage requires. common may *not* import any module from the subpackage, but may be imported from all of them. No rules wrt importing modules from the same subpackage exist of other modules. Just apply common sense here to avoid circular imports.
- Don't use `import *`. It interferes with our static checking. For relative imports, we will probably be slowly migrating towards `from . import` over the current absolute imports (`from gavo.base import...`).
- There is a hierarchy of subpackages, where subpackages lower in the hierarchy may not import anything from the higher or equal levels, but only from lower levels. This hierarchy currently looks like this: imp [<] utils < stc < (votable, adql) < base < dm < rscdef < grammars < formats < rsc < svcs < registry < protocols < web < rscdesc < (helpers, user) utils should never assume anything from imp is present, i.e., it may *attempt* to import from there, but it should not fail hard if the import doesn't work. Of course, concrete functions (e.g., from utils.fitstools) won't work if the base libraries are not present.

Getting Table Metadata and Querying Tables

The preferred way to do simple queries against tables in DaCHS these days is:

(a) get the table metadata:

```
td = base.resolveCrossId("resdir/q#mytable")
```

(b) use the td's `doSimpleQuery(selectClause, fragments, params)` method to get dicts of rows; all arguments are optional and default to pulling all; `selectClause` is just a list of column names:

```
for row in td.doSimpleQuery(["col1", "col2"],
    "col1<%(lim1)s AND col2=%(foo)s",
    {'lim1': 23, 'foo': 42}):
  print row
```

When you need explicit connection management or want to do more complex operations, use the context managers (typically `getTableConn` when querying, `getWritableAdminConn` when writing) using a context manager:

```
with base.getTableConn() as conn:
  for row in conn.queryToDicts(myComplexQuery,
      {'arg1': 23, 'pat': 'M32'}):
    ...
```

In contrast to `doSimpleQuery`, this will do case folding of the select list items.

In "user" code, get these symbols from `api` instead of from `base`.

Error handling, logging

Exception classes

It is the goal that all errors that can be triggered from the web or from within resource descriptors yield sensible error messages with, if possible, information on the location of the error. Also, major operations changing the content of the database should be loggable with time and, probably, user information.

The core of error processing is utils.excs. All "sensible" exceptions (i.e., MemoryErrors and software bugs excepted) should be instances of gavo.excs.Error. However, upwards from base you should always raise exceptions from base; all ("public") exception types from utils.excs are available there (i.e., raise base.NotFoundError(...) rather than utils.excs.NotFoundError(...)).

The base class takes a hint argument at construction that should give additional information on how to fix the problem that gave rise to the exception. All

exception constructor arguments except the first one must always be keyword arguments as a simple hack to allow pickling the excepitons.

When defining new exceptions, if there is structured information (e.g., line numbers, keys, and the like), always keep the information separate and use the __str__ method of the exception to construct something humans want to see. All built-in exceptions should accept a hint keyword.

The events subsystem

All proper DC code (i.e. above base) should do user interaction through base.ui.notify<something>. In base and below, you can use utils.sendUIEvent, but this should be reserved for weird circumstances; code so far down should't normally need to do user interaction or similar.

The <something> can be various things. base.events defines a class EventDispatcher (an instance of which then becomes base.ui) that defines the notify<something> methods. The docstrings there explain what you're supposed to pass, and they explain what observers get.

base.events itself does very little with the events, and in particular it does not do any user interaction -- the idea is that I may yet want to have Tkinter interfaces or whatever, and they should have a fair chance to control the user interaction of a program.

The actual action on events is done by observers; these are ususally defined in `user`, and some can be selected from the `gavo` command line. For convenience, you should derive your Observer classes from base.ObserverBase. This lets you stuff like:

```
from gavo.base import ObserverBase, listensTo

class PlainUI(ObserverBase):
  @listensTo("NewSource")
  def announceNewSource(self, srcString):
    print "Starting %s"%srcString
```

However, you can also just handle single events by saying things like:

```
from gavo import base

def handleNewSource(srcToken):
  pass

base.ui.subscribeNewSource(handleNewSource)
```

Most logging is done in user.logui; if you want logging, say:

```
from gavo.user import logui
logui.LoggingUI(base.ui)
```

Catching exceptions

In the DC software, is is frequently desirable to ignore the first rule of exception handling, viz., leave them alone as much as possible. Instead, we often map exceptions to DC-internal exceptions (this is very relevant for everything leading up to ValidationErrors, since they are used in user interaction on the web interface). However, to make the original exception information available for debugging or problem fixing, whenever you "translate" an exception, have `base.ui.notifyExceptionMutation(newException)` called. This should arrange logging the exception to the error log (although of course that's up to the observer selected).

The convenient way to do this is to call `ui.logOldExc(exc)`:

```
raise base.ui.logOldExc(GavoError(...))
```

LoggingUI only logs the information on old exceptions when base.DEBUG is true. You can set this from your code, or by passing the `--debug` option to gavo.

Testing

In an installed checkout of DaCHS, you can go to the `tests` subdirectory and run:

```
python runAllTests.py
```

for a fairly extensive set of unit tests.

This uses some management of test scaffolds; when something is severely wrong, generating these scaffolds can fail and the execution of the suite will stop. I'm not decided whether to regard that as a bug or a feature, but I'll not fix it any time soon. So, if this bites you, find out why resource generation fails and fix it.

XSD validation

XML Schema is a pain all around, and given that we don't want to hit W3C and IVOA with requests for schema files every time someone needs schema validation (which includes RD validation and unit tests), DaCHS goes to some lengths to use its own schema files.

The standard way these days is the LXML-based validator from gavo.helpers.testtricks; the rocket science part of this is to make LXML use the plethora of schema files we have locally.

"Locally" here means in the `schemata` subdirectory of the distribution. When you add a schema there that should be available in validation, you also need to add the filename to `gavo.testtricks.VO_SCHEMATA` (background: we keep some schema files in gavo/schema that the validator should not be bothered with; still, we should probably just pull in `*.xsd` at some point).

With this, run `gavo admin xsdVal` to XSD-validate a VO file.

If uncertain, you can also fall back so a xerces-based validator we used to use. Here, you first need xerces itself, which on debian is the package `libxerces2-java`, and you'll also need `libxerces2-java-doc` (really). These packages will dump the necessary jars (`xercesImpl.jar` and `xmlParserAPIs.jar`) in `/usr/share/java`; if you have a different setup, you'll need to edit `xsdclasspath` in `tests/test_data/test-gavorc` within your checkout and curse java's classpath construct.

You can then manually build the legacy validator by going to `schemata` subdirectory and saying `python makeValidator.py`. This will use leave the validator as `xsdval.class` in the cache directory (/var/gavo/cache by default).

To run this legacy validator, you could change:

```
getXSDErrors = getXSDErrorsLXML
```

in gavo.helpers.testtricks to

```
getXSDErrors = getXSDErrorsXerces
```

– but it's probably better to just run this one-off with a commandline like:

```
classpath=/usr/share/doc/libxerces2-java-doc/examples/xercesSamples.jar:/usr/share/java/xercesIm
java -cp `gavo config cacheDir`/xsdval.class:$classpath xsdval -n -v -s -f to_validate.xml
```

or similar.

Setting package installs up for testing

The debian package does not contain unit tests. If you want to nevertheless run them, check out the release corresponding to your package from http://svn.ari.uni-heidelberg.de/svn/gavo/python/tags/. Again, see the `tests` subdirectory in your checkout.

Test framework

All unit tests must import gavo.helpers.testhelpers before importing anything else from the gavo namespace. This is because testhelpers sets up a test environment in ~/gavo_test (set in tests/test_data/test-gavorc). To make this work reliably, it must manipulate the normal way configuration files are read.

helpers.testhelpers needs a dachstest database for which the current user is a superuser. It will create it provided you're a DB superuser with ident authentication (see install to figure out how to set this up).

There are doctests in modules (though fewer than I'd like), and pyunit- and trial-based tests in `<project root>/tests`. `tests/runAllTests.py` takes care of locating and executing them all.

In addition to setting up the test environment, testhelpers provides (check out the source) some useful helper functions (like `getTestRD`), the `VerboseTest` class adding test resources and some assertions to the normal `unittest.TestCase`. Do *not* import it in production code. Test-like functionality interesting to production code should go to `helpers.testtricks`.

`testhelpers.main` is useful after an `if __name__=='__main__'` in test modules. Pass a default test class, and you can call the module without arguments (in which case it will run all tests), with a single argument (that will be interpreted as a method prefix to locate tests on the default TestCase) or with two arguments (a TestCase name and a method prefix to find the methods to be run). All pyunit-based tests use this main.

`testhelpers.main` evaluates the `TEST_VERBOSITY` environment variable. With `TEST_VERBOSITY=2`, you'll see the test names as they are executed.

Regression testing of data

For certain kinds of data, unit testing is useful, too. Since it's always possible that server code changes may break such tests, it makes sense to run those unit tests at each commit. Therefore, `tests/runAllTests.py` has a facility to pick up such tests from directories named in $GAVO_INPUTS (the "real" one, not the fake test one) in the `__tests/__unitpaths`. It will pick up tests from there just as it picks them up from tests.

Such data-based tests (typically) must run "out of tree", i.e., in the actual server environment where the resources expected by the service tested are. To keep testhelper from fudging the environment, set the environment variable `GAVO_OOTTEST` to anythign before importing testhelpers. This is conveniently done in python, like this:

```
import os
os.environ["GAVO_OOTTEST"] = "dontcare"

from gavo.helpers import testhelpers
```

pyflakes

Not really testing, but static code checking using pyflakes should regularly be done, and result in no warnings eventually (right now, more annotations are required).

We have added a simple ignoring facility in our pyflakes driver, `tests/flake_all.py`:

- To ignore (not check) an entire file, add, preferably near the top, a line like:

  ```
  # Not checked by pyflakes: (reason)
  ```

  Please always give a reason so people can tell whether it has gone away and the file should now be included in the checks.

- To ignore a single error, add a comment like:

  ```
  #noflake: (rationale)
  ```

  to the line reported by pyflakes.

Also note that flake_all hardcodes that modules from imp are not checked.

Test Plan

(This is somewhat specific to Markus' setup; something similar is recommended for everyone, though)

Before every commit, do:

- start a local server
- go to `$checkout/tests`
- `python flake_all.py` (which does some static code checking)
- `python runAllTests.py` (which arranges for doctests, pyunit tests, trial tests, and data unit tests to be run)
- run `gavo val -tv ALL` (which, apart from validating the RDs, also runs the RD-defined regression tests against the server running locally)

- go to `$checkout`
- run `svn status` to make sure no files are left not in version control or explicitely ignored

After a checkout on the production server, do:

- `gavo test -t bigserver -u http://dc.g-vo.org/ ALL` (which runs all tests defined in the local RDs, even for the production server, against the production server; this does what it's supposed to da as the repo for the RDs is the same on development and production).

Configuration

DaCHS has far too many configuration hooks: gavo.rc, defaultmeta.txt, the database profiles, vanitynames.txt, userconfig.rd, as well as locally-overridden system RDs and templates. At least defaultmeta.txt was a mistake, as was probably vanitynames.txt. We should be working on getting rid of it.

New configuration should preferably go into userconfig.rd, while there's always going to be room for gavo.rc, too.

Configuration items for `userconfig.rd` typically are going to be STREAMs. To provide fallbacks for those if the user hasn't defined any, there's `//userconfig`, which also serves as built-in documentation for what's there. As an identifier is resolved in `//userconfig`, the system first looks in a `etc/userconfig.rd` and then, even if that file exists (but has no element with the id in question), in `//userconfig`.

When using the elements, always use the canonical abbreviation for userconfig, `%`, as in `<FEED source="%#registry-interfacerecords"/>`.

Structures

Resource description within the DC works via instances of base.Structure. These parse themselves from XML strings, do validation, etc. All compound RD elements correspond to a structure class (well, almost; meta is an exception).

A structure instance has the following callbacks:

- `completeElement(ctx)` -- called when the element's closing tag is encountered, used to fill in computed defaults. `ctx` is a parse context that you can use to, e.g. resolve XML ids.

- `validate()` -- called after completeElement, used to raise errors if some gross ("syntactic") mistakes are in the element
- `onElementComplete()` -- called after validate, i.e., elementCompleted can rely on seeing a "valid" structure

In addition, attributes can define `onParentCompleted` methods. These are called after onElementCompleted of the parent element is run when the attribute value is different from its default. They receive the new attribute value as the single argument. Maybe this is bad design.

This processing is done automatically when parsing elements from XML. When building elements manually, you should call the structure's finishElement method when done to arrange for these methods being called.

If you override these methods, make sure you call the methods of the superclass. Since we might, at some point, want mixins to be able to define validators etc, use super()-based superclass calling, through `_completeElementNext(cls, ctx)`, `_validateNext(cls)`, and `_onElementCompleteNext(cls)`.

The `user.docgen` module makes documentation out of these structures. There are several catches. One of the more striking is that element names in the *entire* DaCHS code must be unique, since docgen generates section heading from those names and actually checks that these headings are unique; hence, only one (essentially randomly selected) of two identically-named elements would be documented, and parent links would both point there.

Since there are cases when that limitation is a real pain (e.g., the publish element of services and data), there's a workaround: you can set a `docName_` class attribute on a structure that contains the name used for the documentation. See `rscdef.common.Registration` for an example.

Metadata

"Open" metadata (as opposed to the attributes of columns and the like) is kept in a `meta_` structure added by `base.meta.MetaMixin`. You should probably not access that attribute directly if at all possible since the current implementation is incredibly messy and liable to change.

For this kind of metadata, a simple inheritance exists. MetaMixins have a `setMetaParent` method that declares another structure as the current's meta parent. Any request for metadata that cannot be satisfied from self will then be propagated up to this parent (unless propagation is suppressed). Usually, parents will call their children's setMetaParent methods.

The metdata is organized in a tree with `MetaItem‘‘s as nodes. Each MetaItem contains one or more children that are instances ‘‘MetaValue` (or more specialized classes). A MetaValue in turn can have more MetaItem children.

Getting Metadata

Metadata are accessed by name (or "key", if you will).

The `getMeta(key, ...)->MetaItem` method usually follows the inheritance hierarchy up, meaning that if a meta item is not found in the current instance, it will ask its parent for that item, and so on. If no parent is known, the meta information contained in the configuration will be consulted. If all fails, a default is returned (which is set via a keyword argument that again defaults to None) or, if the raiseOnFail keyword argument evaluates to true, a gavo.NoMetaKey exception is raised.

If you require metadata exactly for the item you are querying, call getMeta(key, propagate=False).

getMeta will raise a gavo.MetaCardError when there is more than one matching meta item. For these, you will usually use a builder, which will usually be a subclass of meta.metaBuilder. web.common.HtmlMetaBuilder is an example of how such a thing may look like, for simple cases you may get by using ModelBasedBulder (see the registry code for examples). This really is too messy and needs to be replaced by something smarter.

The builders are passed to a MetaMixin's buildRepr(metakey, builder) method that returns whatever the builder's getResult method returns.

Setting Metadata

You can programmatically set metadata on any metadata container by calling its method `addMeta(key, value)`, where both key and value are (unicode-compatible) strings. You can build any hierarchy in this way, provided you stick with typeless meta values or can do with the default types. Those are set by key in meta._typesForKeys.

To build sequences, call addMeta repeatedly. To have a sequence of containers, call addMeta with None or an empty string as value, like this:

m.addMeta("p.q", "x") m.addMeta("p.r", "y") m.addMeta("p", None) m.addMeta("p.q", "u") m.addMeta("p.r", "v")

More complex structures require direct construction of MetaValues. Use the makeMetaValue factory for this. This function takes a value (default empty), and possibly a key and/or type arguments. All additional arguments depend on the meta type desired. These are documented in the reference manual.

The type argument selects an entry in the meta._typesForKeys table that specifies that, e.g., _related meta items always are links. You can also give the type directly (which overrides any specification through a key).

This can look like this:

m.addMeta("info", meta.makeMetaValue("content", type="info",
infoName="someInfo", infoValue="GIVEN"))

Memoization

The base.caches module should be the central point for all kinds of memoization/caching tasks; in particular, if you use base.caches, your caches will automatically be cleared on `gavo serve reload`. To keep dependencies and risks of recursive imports low, it is the providing modules' responsibility to register caching functions. The idea is that, e.g., rscdesc wants a cache of resource descriptors. Therefore, it says:

```
base.caches.makeCache("getRD", getRD)
```

Clients then say:

```
base.caches.getRD(id).
```

This mechanism for now is restricted to items that come with a unique id (the argument). It would be easy to extend this to multiple-argument functions, but I don't think that's a good idea -- the "identities" of the cached objects should be kept simple.

No provision is made to prevent accidental overwriting of function names.

Profiling

If you want to profile server actions, try a script like this:

```
"""
Make a profile of server responses.

Call as

trial --profile createProfile.py
"""

import sys

from gavo import api
from gavo.web import dispatcher

sys.path.append("/home/msdemlei/gavo/trunk/tests")
```

```
import trialhelpers


class ProfileThis(trialhelpers.RenderTest):
  renderer = dispatcher.ArchiveService()

  def testOneService(self):
    self.assertGETHasStrings("/ppmx/res/ppmx/scs/form",
      {"hscs_pos": "12 2", "hscs_sr": "20.0"},
      ["PPMX"])
```

After running, you can use pstats on the file profile.data.

To profile actually running DaCHS operations, use the --profile-to <profile file> option of the gavo program. For the server, you must make sure in cleanly exists in order to have meaningful stats. Do this by accessing /test/exit on a debug server.

Debugging

For anything outside of the server, lines like:

```
import code; code.interact(local=locals())
```

(for getting a python prompt to look around and introspect, e.g., using `dir()`; exit to let the program continue) and:

```
import pdb;pdb.set_trace()
```

(which dumps you into the debugger and lets you single-step, etc) are your friends.

When you want to inspect what's going on within the server, in particular when something only manifests itself after a long time, you may want to have a look at twisted's manhole; quite a bit easier, however, is to use the debug/q rd that you can get from http://svn.ari.uni-heidelberg.de/svn/gavo/hdinputs/debug and adapt it to your needs.

The idea here is that within q.rd#1 you create customDFs or customRFs exposing what you're interested in. You can then use those in res/page1.html. You can edit both files "live", they will both be reloaded as necessary.

Debugging memory leaks

Sometimes one is careless and leaves a reference somewhere, perhaps in an RD. Since this really only matters in the server, such situations are particularly insidious to debug. To help there, there's some scaffolding in web.root.

To activate things, you set `MEM_DEBUG` to `True`. Down in `locateChild` of `ArchiveService`, there's code like:

```
if MEM_DEBUG:
        from gavo.utils import codetricks
        import gc
        gr = gc.get_referrers
        if hasattr(base, "getNewStructs"):
                ns = base.getNewStructs()
                print ">>>>>> new structs:", len(ns)
```

What this lets you do is see when new structs are left somewhere in DaCHS' guts. What you do when such a thing happens is higher magic. I've found it helps to put something like a mini-memory debugger right into that handler. There's a rough one in testtricks, so you could put in something like:

```
    if len(ns)==147:
            from gavo.helpers import testtricks
            ob = ns[0]
            del ns
testtricks.debugReferenceChain(ob)
```

after the print (of course, this only makes sense if you're running `gavo serve debug`, as the actual server detaches from its tty). This lets you go through the objects referring to the first struct left over by hitting Return.

Enter anything to follow the (inverse) reference, except that a d will drop you in the debugger and x will continue normal execution. Do this until you see where the reference comes from. Just be aware that many references are harmless -- in particular, this function will hold a reference to the object in question, so you'll need some experience to figure out where to look.

Delimited SQL identifiers

Although it may look like it, we do not really support delimited identifiers (DIs) as column names (and not at all as table names). I happen to regard them as an SQL misfeature and really only want to keep them out of my software.

However, TAP forces me to deal with them at least superficially. That means that using them elsewhere will lead to lots of mysterious error messages from

inside of DaCHS's bowels. There still should not be any remote exploits possible when using them.

Here's the deal on them:

They are represented as `utils.misctricks.QuotedName` objects. These QuotedNames have some methods to control the impact the partial support for delimited identifiers has on the rest of the software. In particular, when you stringify them, they result in string ready for inclusion into SQL (i.e., hopefully properly escaped). The hash to the name, i.e., there are no implied quotes, and, unfortunately, hash(di)!=hash(str(di)).

The DC software right now assumes DIs are ASCII only (what do the standards people say?).

The one real painful thing is the representation of result rows with DIs -- I did not want to have lots of these ugly QuotedNames in the result rows, so they end up as SQL-escaped strings when used as keys. This is extra sad since in this way for a DI column foo, rec[QName("foo")] raises a KeyError. To work around this, fields have a key attribute, and rec[f.key] should never bomb.

Grammars

Grammars are DaCHS' means of turning some external data to rowdicts, i.e., dictionaries that map grammar keys to values that are usually strings. They are fed to rowmakers to come up with rows suitable for ingestion (or formatting).

A grammar consists of a Grammar object, which is a structure inheriting from grammars.Grammar. It contains all the "configuration" (e.g., rules). Grammars have a parse method receiving some kind of source token (typically, a file name). You will normally not need to override it.

The real action happens in the row iterator, which is declared in the rowIterator class attribute of the grammar. Row iterators should inherit from grammars.RowIterator.

TODO: yieldsTyped, rowfilters, sourceFields, targetData

Do not import modules from the grammars subpackage directly. Instead, use rscdef.getGrammar with the name of the grammar you want. If you define a new grammar, add a line in rscdef.builtingrammars.grammarRegistry. To inspect what grammars are available, consult the keys from rscdef.grammarRegistry.

Procedures

To embed actual (python) code into RDs, you should use the infrastructure given in rscdef.procdef. It basically leads up to `ProcApp`, which is what's usually embedded in RDs.

`ProcApp` inherits from `ProcDef`, a procedure definition. Such a definition gives some (python) code that is executed when the procedure is applied. To set up the execution environment of this code, there's the definition's setup child.

The setup contains code and parameters. The code is executed to set up the namespace that the procedure will run in; it is thus executed once -- at construction -- per procedure. The parameters allow configuration of the procedure. This is the place to do relatively expensive operations like I/O or imports.

For example, `//procs#resolveObject` creates the resolver in its setup code; this happens only once per creation of the embedding RD:

```
<procDef type="apply" id="resolveObject">
  <setup>
    <par key="ignoreUnknowns">True</par>
    <par key="identifier" late="True"/>
    <code>
      from gavo.protocols import simbadinterface
      resolver = simbadinterface.Sesame(saveNew=True)
    </code>
  </setup>
  <doc>...</doc>
   <code>
    ra, dec = None, None
    try:
      ra, dec = resolver.getPositionFor(identifier)
    except KeyError:
      if not ignoreUnknowns:
        raise base.Error("resolveObject could not resolve object"
          " %s."%identifier)
    vars["simbadAlpha"] = ra
    vars["simbadDelta"] = dec
  </code>
</procDef>
```

The setup definition introduced two parameters. One is ignoreUnknowns, which is "immediate" and just lets the code see a name ignoreUnknowns. As with all `par` elements, the content of the element is a python expression providing a default.

The other parameter, identifier, is a "late" identifier. This means that it is evaluated on each application of the procedure, much like a function argument. These are just translated into assignments at the top of the function body,

which means that everything available in the procedure code is available; e.g., for rowmaker procedures (i.e., type="apply"), you can access `vars` here.

Taken together, late and immediate `par` allow for all kinds of configuration of procedures. This is particularly convenient together with macros.

To actually execute the code, you need some kind of procedure application. These always inherit from procdef.ProcApp and add bindings. The `bind` element lets you give python expressions for all names defined using `par` in the `setup` child of the `ProcDef` given in the `procDef` attribute. You can also define just a procedure application without a procDef by giving `setup` and `code`.

Procedure application have "types" -- these give where they can be used. In particular, the type determines the signature of the python callable that the procedure application is compiled into. `procdef.ProcApp` has no type, and thus is "abstract"; it should never be a child factory of any `StructAttribute`.

Instead, inherit from it and give

- `name_` -- the element name, as always in structures. This is "apply" for rowmaker applys, "rowfilter" for grammar rowfilters, etc
- `formalArgs` -- a python argument list that gives the arguments of the callable a ProcApp of this type is compiled into. Thus, this defines the signature.
- `requiredType` -- a type name that specifies what kind of ProcDef the application will accept. This will in general be the same as `name_`. None would mean accept all, which probably is useless.

So, all you need to do to define a new sort of ProcApp is write something like:

```
class EmbeddedIterator(rscdef.ProcApp):
  name_ = "iterator"
  formalArgs = "self"
```

(of course, here, documentation as to what the code is supposed to do is particularly important, so don't leave out the docstring when actually doing anything.

Then, you could have:

```
_iterator = base.StructAttribute("iterator", default=base.Undefined,
 childFactory=EmbeddedIterator,
 description="Code yielding row dictionaries", copyable=True)
```

in some structure. To produce something you can execute, then say:

```
theIterator = self.iterator.compile()
for row in theIterator(self):
  print row
```

or somesuch.

Schema updates

If you need to change the on-disk schema, you must provide an updater in gavo.user.upgrade. See the docstring on Upgrade on what you can and should do in there.

The `version` attribute of your new upgrader must be the value of `upgrader.CURRENT_SCHEMAVERSION` (defined near the top) when you start working. After you have defined your upgrader, increase CURRENT_SCHEMAVERSION by one.

At Heidelberg, the next step would be to try the upgrader using:

```
testgavo upgrade
```

The effects should be visible in the `dachstest` database.

If you follow the rules, upgrade should be atomic, i.e., either the upgrade succeeds or the database is untouched, letting operators downgrade and continue operations until a problem is figured out.

If, on development, you notice your instructions have not had the desired effect, things are more difficult. You can, in principle, re-run the entire procedure by executing something like:

```
update dc.metastore set value=5 where key='schemaversion';
```

to re-set the schema version to the initial value, but that doesn't really help when changes are already in the database. During development, it pays to keep a quick backup of current database files so it's easy to entirely roll back if the upgrade hasn't quite worked.

Javascript

While it's our goal to let people operate the web-based part of DaCHS without javascript enabled, it's ok if fancier functionality depends on javascript.

After some hesitation, we decided to use the jquery javascript library (we used to have MochiKit but left that when we wanted nice in-browser plotting; so, if you still see MochiKit somewhere, please disregard). We also include some of jquery-ui.

We keep all javascript in "full" source form (in resources/web/js). DaCHS performs on-the-fly minimisaton (unless [web]jsSource is False).

For development of that, it's much more convenient if the stuff that gets served out is in source. To enable that, set `[web]jsSource` to true. This needs actual code support; right now this only works for files served out in commonhead. You need to restart the server for the setting to take effect.

gavo.js

The commonhead renderer that's applied to almost all pages pulls in the javascript from resources/web/js/gavo.js. This includes some utility functions in the global namespace (and some that should be moved elsewhere). In particular, it contains quite a bit of ugly mess for managing the output formats.

Here's a discussion of some features that may be interesting to template authors.

Built-in templating

There's a very plain templating engine in javascript included, using an idea due to John Resig, http://ejohn.org/. According to this, you define a template in your HTML as a script of type text/html:

```
<script type="text/html" id="tmpl_authorHeader">
  <li>
    <a class="arrow-e"
      onclick="toggleAuthorResources(this)" name="$author"/>
      $author ($nummatch)
  </li>
</script>
```

The $varName parts can then be filled – properly HTML-escaped – by calling:

```
renderTemplate("tmpl_authorHeader", {
  author: 'Thor, A. U',
  nummatch: 8})
```

Currently, filling variables is the only thing the engine knows how to do.

Fairly Simple Tabs

There's built-in javascript and CSS for switching tabs. The tabs require Javascript, so you'll usually want to hide them from non-JS-browsers. Thus, to define the tabs, do something along the lines of:

```
<script type="text/html" id="tabbar_store">
<ul id="tabset_tabs">
  <li class="selected"><a name="by-title">By Title</a></li>
  <li><a name="by-subject">By Subject</a></li>
  <li><a name="by-author">By Author</a></li>
</ul>
</script>

<p id="tab_placeholder" style="border:2pt dashed #bb9999;padding: 0.5ex">
  Enable Javascript for more choices.</p>
```

Note how the tab headings are within a elements that have a name – it's this name that lends identity to them. You could have hrefs for better non-javascript fallback if you have the tabs without javascript; remove the href attributes when you have javascript active, though.

Then, in your javascript, say:

```
$(document).ready(function() {
  $("#tab_placeholder").replaceWith(
    $(document.getElementById("tabbar_store").innerHTML));
  $("#tabset_tabs li").bind("click", makeTabCallback({
    'by-subject': func1,
    'by-author': func2,
    'by-title': func3,
  }));
}
```

(or do something equivalent, if you don't like the innerHTML here). The functions in the dictionary passed to `makeTabCallback` must then work on the container below the tabs. Here's CSS you could base the container css on:

```
position: relative;
background-color: #EAEBEE;
margin-top: 0px;
min-height:70ex;
```

The CSS that styles the tabs is in `resources/web/css/gavo_dc.css`, the images necessary in `resources/web/img`.

samp.js

This is Mark Taylor's samp.js, checked out from https://github.com/astrojs/sampjs.git.

jquery.flot.js

That's the plotting code. We got it from http://www.flotcharts.org/

Building jquery-gavo.js

- Go to a temporary directory
- Go to http://jqueryui.com/download and make yourself a jquery-ui archive. Currently, we want all the core, plus draggable and resizable (plus probably Dialog, though that's not used yet). Do not select any theme.
- Unzip the file, go to the js subdirectory of the distribution
- concatenate external/jquery/jquery.js and jquery-ui.js go jquery-gavo.js
- copy jquery-gavo.js to gavo/resources/web/js

Whatever CSS is necessary for jquery should for now go into gavo_dc.css – I've simply been pasting in jquery-ui.css (no idea what "-structure" is).

Stuff in gavo.imp

gavo.imp has some external dependencies of DaCHS. Shortly after release 1.0, many were dropped in favour of their packaged/native counterparts (argparse, pyparsing...). What's currently left is:

- formal -- the framework for HTML form widgets DaCHS uses. This stuff is long abandoned now by upstream. We'll probably keep it until at least version 2.0 because parts of formal leak into DaCHS' interface
- rjsmin -- Debian packaged, and the Debian version will be picked up automatically if installed (so, this should not go into the Debian package)

Different Database Backends

A request we get fairly regularly is to make DaCHS work with database engines other than Postgres, with MySQL and Oracle being the most popular alternatives for external requests and SQLite something we personally would like to see for ease of deployment.

The short answer to all this: It's tricky. You might get away with using foreign data wrappers in some cases; a group at Paris Observatory reports fairly good results with them.

Here's the longer answer: DaCHS does a lot of inspection of the database, while at the same time worrying about different access levels, reconnection on database restarts, and similar; it also creates extension types. We are not aware of any abstraction layer that would let us keep all this code generic, and that's why we let DaCHS slide into a fairly deep entanglement with psycopg2 and Postgres.

Seeing such an entanglement reduces the scope of DaCHS, we'd certainly help pulling it out of the entanglement. We probably won't do it ourselves. Here's a list of things that would need to be done for un-entanglement, that's probably somewhat incomplete and also contains some project mines (innocuous-looking things that blow up into a lot of refactoring once you step on them):

(a) separate what's specific to postgresql+psycopg2 from sqlsupport, put that into a module (backend_postgres, say), devise some sort of dispatcher to backends, and have, to work out things, a second backend, that would then contain different implementations for tableExists, indexExists, and so on. Actually, throwing out some cruft from sqlsupport that should have gone ages ago would be a good thing, too.

(b) figure out what other hidden dependencies exist; the most worrisome part probably is the extension types DaCHS uses and registers as well as the pgSphere interface; this is built into typesystems and used left and right. If there's no way to hide DB-specific differences, there'll have to be some major redesign. Also, DaCHS implicitely assumes TEXT in the database is cheap. If that's not true of a DB (and I think in Oracle TEXT can't be properly indexed) and you'll want much more VARCHARs and similar, minor adjustments might be in order.

(c) The ADQL translator would need to get another "morpher" (the thing that turns ADQL parse trees into the language of the backend database) That's already forseen, but figuring out how to enable maximum reuse of code between the different morphers might take some thought. Also, again the question of spherical geometry in the backend will have to be looked at.

(d) Some mixins directly depend on postgres features (//scs#q3cindex is an obvious example). I believe it'd be ok to say "well, don't use these on non-Postgres", and we'd provide similar things for the other DBs. But that would make RDs non-portable, which I don't like too much either.

(e) The C boosters generate material for Postgres binary copy. Obviously, one would need to figure out the analogon on other databases (which may not be well-documented; I had to check the Postgres source for some details, too) and then split up boosterskel.c into generic and postgres-specific parts. Or there'd be no support for C boosters on different databases, which might not be unreasonable, either.

Implementing Protocols

TBD

You should add the protocol mixin(s) in user.docgen.PUBLIC_MIXINS so they get included in the documentation; likewise if you need apply and/or rowfilters, amend PUBLIC_APPLYS or PUBLIC_ROWFILTERS.

Writing Documentation

Documentation on DaCHS is maintained in ReStructuredText format with some minor extensions (see below). While there's documentation in the tarball and the main SVN, in order to encourage external contributions (including, but not restricted to, typo fixes and the like) the main copy now is at https://github.com/chbrandt/dachs-doc.git.

When authoring, you can use some extra RST features (the price: stock rst2pdf and friends don't work properly; use `gavo gendoc latex` or `gavo gendoc html`, or a special sphinx configuration). These include:

- dachsref: Give a reference documentation heading (as in The //obscore#publishSIAP Mixin), and you'll get a link there.
- dachsdoc: Works like normal links, including explicit targets, but it prepends the root URL of the DaCHS documentation
- bibcode: Adds an ADS link to a bibcode

Random Stuff

Tracing imports

Sometimes it's nice to see what gets imported when. Futzing with PEP 302-style import hooks is a pain, and indeed a simple shell line produces more useful output than naive hooks:

```
strace gavo imp -h 2>&1 | grep 'open' | grep -v ENOENT | grep -v "pyc" | sed -e 's/.*"\(.*\)".*/
```

matplotlib

To use matplotlib and pyplot within renderers or some other server context, use the following import pattern:

```
import matplotlib
matplotlib.use("Agg")
from matplotlib import pyplot
```

It is crucial that the use("Agg") happens before the import of pyplot. If you fail to do this properly, your code will fail complaining about missing DISPLAYs.

I *guess* we'll soon properly depend on matplotlib and to that initialization in a good place in utils, but don't hold your breath.