GAVO DC Software Reference Documentation

Author: Markus Demleitner
Email: gavo@ari.uni-heidelberg.de
Date: 2018-10-02

Contents

Resource Descriptor Element Reference **14**

Element apply 16
Element bind 17
Element column 18
Element columnRef 20
Element condDesc 21
Element coverage 22
Element customDF 23
Element customRF 23
Element data 24
Element DEFAULTS 26
Element dm 26
Element EDIT 26
Element events 27
Element execute 27
Element foreignKey 28

Element group 29
Element httpUpload 30
Element ignoreOn 31
Element ignoreSources 31
Element index 32
Element inputKey 33
Element job 36
Element lateEvents 38
Element macDef 38
Element make 38
Element map 39
Element mixinDef 40
Element mixinPar 42
Element option 42
Element outputField 43
Element outputTable 45
Element par 48
Element param 49
Element paramRef 51
Element phraseMaker 52
Element procDef 53
Element processEarly 54
Element processLate 55
Element PRUNE 56
Element publish (data) 57
Element publish 57

Element regSuite 58
Element regTest 59
Element resource 60
Element resRec 62
Element rowmaker 63
Element script 64
Element service 65
Element setup 67
Element sources 68
Element stc 69
Element table 69
Element updater 71
Element url 72
Element values 73
Element var 75

Active Tags **75**

Element FEED 75
Element LFEED 76
Element LOOP 77
Element NXSTREAM 78
Element STREAM 79

Grammars Available **79**

Element binaryGrammar 79

Element binaryRecordDef 80

Element cdfHeaderGrammar 81

Element columnGrammar 82

Element contextGrammar 84

Element csvGrammar 85

Element customGrammar 87

Element dictlistGrammar 88

Element directGrammar 89

Element embeddedGrammar 90

Element fitsProdGrammar 92

Element fitsTableGrammar 94

Element freeREGrammar 95

Element iterator 96

Element keyValueGrammar 97

Element mapKeys 99

Element mySQLDumpGrammar 99

Element nullGrammar 101

Element odbcGrammar 102

Element pargetter 103

Element pdsGrammar 104

Element reGrammar 106

Element rowfilter 108

Element rowsetGrammar 109

Element sourceFields 110

Element transparentGrammar 112

Element voTableGrammar 113

Cores Available **114**

Element adqlCore 114

Element coreProc 114

Element customCore 115

Element dataFormatter 116

Element dataFunction 118

Element datalinkCore 119

Element dbCore 121

Element debugCore 122

Element descriptorGenerator 123

Element fancyQueryCore 124

Element fixedQueryCore 125

Element inputTable 126

Element metaMaker 127

Element nullCore 129

Element productCore 130

Element pythonCore 131

Element registryCore 132

Element scsCore 133

Element siapCutoutCore 134

Element ssapCore 135

Element tapCore 137

Element uploadCore 137

Predefined Macros **138**

Macro RSTcc0 . . . 138

Macro RSTccby . . . 139

Macro RSTccbysa . . . 139

Macro RSTservicelink . . . 139

Macro RSTtable . . . 140

Macro colNames . . . 140

Macro curtable . . . 140

Macro decapitalize . . . 141

Macro dlMetaURI . . . 141

Macro docField . . . 141

Macro fullDLURL . . . 142

Macro fullPath . . . 142

Macro getConfig . . . 142

Macro getParam . . . 143

Macro inputRelativePath . . . 143

Macro inputSize . . . 144

Macro internallink . . . 144

Macro lastSourceElements . . . 144

Macro magicEmpty . . . 145

Macro metaString . . . 145

Macro nameForUCD . . . 146

Macro nameForUCDs . . . 146

Macro property . . . 146

Macro qName . . . 147

Macro quote . . . 147

Macro rdId . . . 147
Macro rdIdDotted . . . 148
Macro rootlessPath . . . 148
Macro rowsMade . . . 148
Macro rowsProcessed . . . 149
Macro schema . . . 149
Macro sourceCDate . . . 149
Macro sourceDate . . . 150
Macro splitPreviewPath . . . 150
Macro sqlquote . . . 151
Macro srcstem . . . 151
Macro standardPreviewPath . . . 151
Macro standardPubDID . . . 152
Macro tablename . . . 152
Macro tablesForTAP . . . 153
Macro test . . . 153
Macro today . . . 153
Macro upper . . . 154
Macro urlquote . . . 154

Mixins **154**

The //epntap2#localfile-2_0 Mixin . . . 155
The //epntap2#table-2_0 Mixin . . . 155
The //obscore#publish Mixin . . . 156
The //obscore#publishSIAP Mixin . . . 158
The //obscore#publishSSAPHCD Mixin . . . 161
The //obscore#publishSSAPMIXC Mixin . . . 164

The //products#table Mixin 167
The //scs#positions Mixin 167
The //scs#q3cindex Mixin 168
The //siap#pgs Mixin 168
The //slap#basic Mixin 169
The //ssap#hcd Mixin 169
The //ssap#mixc Mixin 171
The //ssap#sdm-instance Mixin 172
The //ssap#simpleCoverage Mixin 173

Triggers **173**

Element and 173
Element keyIs 174
Element keyMissing 174
Element keyNull 175
Element keyPresent 175
Element not 175

Renderers Available **176**

The admin Renderer 176
The api Renderer 176
The availability Renderer 177
The capabilities Renderer 177
The coverage Renderer 177
The custom Renderer 177
The dlasync Renderer 178
The dlget Renderer 178

The dlmeta Renderer 178
The docform Renderer 178
The edition Renderer 179
The examples Renderer 179
The external Renderer 180
The fixed Renderer 180
The form Renderer 180
The get Renderer 181
The howtocite Renderer 181
The info Renderer 181
The logout Renderer 181
The mimg.jpeg Renderer 181
The mupload Renderer 182
The pubreg.xml Renderer 182
The qp Renderer 182
The rdinfo Renderer 182
The scs.xml Renderer 183
The siap.xml Renderer 183
The siap2.xml Renderer 184
The slap.xml Renderer 184
The soap Renderer 184
The ssap.xml Renderer 185
The static Renderer 185
The tableMetadata Renderer 186
The tableinfo Renderer 186
The tablenote Renderer 186

The tap Renderer 186
The upload Renderer 187
The uws.xml Renderer 187
The volatile Renderer 187

Predefined Procedures **187**
Procedures available for rowmaker/parmaker apply 187
Procedures available for grammar rowfilters 200
Procedures available for datalink cores 202

Predefined Streams **206**
Datalink-related Streams 206
Other Streams 208

Data Descriptors **209**
Updating Data Descriptors 210

Metadata **212**
Inputing Metadata 212
Meta inheritance 217
Meta formats 217
Macros in Meta Elements 218
Typed Meta Elements 218
Metadata in Standard Renderers 227
RMI-Style Metadata 229
Coverage Metadata 230

Display Hints **233**

Data Model Annotation **234**

Annotation Using SIL 234

GeoJSON annotation 236

DaCHS' Service Interface **238**

Core Args 239

Table-based cores 240

Output tables 245

Writing Custom Cores **246**

Defining a Custom Core 247

Giving the Core Functionality 249

Database Options 252

Python Cores instead of Custom Cores 252

Regression Testing **253**

Introduction 253

Writing Regression Tests 254

RegTest URLs 255

RegTest Tests 258

Running Tests 260

Examples 261

Datalink and SODA **263**

Integrating Datalink Services 265

Making Datalinks 266

Defining Processing Services 267

General Notes on Processing Services 270

Descriptor Generators . . . 271
Meta Makers . . . 273
Metadata Error Messages . . . 275
Data Functions . . . 276
Data Formatters . . . 278
Registry Matters . . . 279
Datalinks as Product URLs . . . 280
SDM compliant tables . . . 282

Product Previews **283**

Custom UWSes **285**

Custom Pages **286**

Manufacturing Spectra **289**

Making SDM Tables . . . 289

Echelle Spectra **291**

Table . . . 291

Supporting getData **291**

Adapting Obscore **292**

Writing Custom Grammars **294**

Dispatching Grammars . . . 296

Functions Available for Row Makers **297**

Scripting **304**

SQL scripts . . . 304

Python scripts . . . 304

Script types . . . 305

Examples . . . 305

ReStructuredText **306**

Code in DaCHS **307**

Importing modules . . . 307

The DaCHS API . . . 308

System Tables **337**

dc.authors . . . 337

dc.datalinkjobs . . . 338

dc.groups . . . 339

dc.interfaces . . . 339

dc.metastore . . . 340

dc.products . . . 340

dc.res_dependencies . . . 341

dc.resources . . . 341

dc.resources_join . . . 342

dc.sets . . . 343

dc.subjects . . . 344

dc.subjects_join . . . 344

dc.tablemeta . . . 345

dc.users . . . 346

ivoa.ObsCore . . . 346

ivoa._obscoresources 348
ivoa.emptyobscore 348
tap_schema.columns 350
tap_schema.groups 351
tap_schema.key_columns 351
tap_schema.keys 352
tap_schema.schemas 352
tap_schema.supportedmodels 352
tap_schema.tables 353
tap_schema.tapjobs 353
uws.userjobs 354

Resource Descriptor Element Reference

The following (XML) elements are defined for resource descriptors. Some elements are polymorous (Grammars, Cores). See below for a reference on the respective real elements known to the software.

Each element description gives a general introduction to the element's use (complain if it's too technical; it's not unlikely that it is since these texts are actually the defining classes' docstrings).

Within RDs, element properties that can (but need not) be written in XML attributes, i.e., as a single string, are called "atomic". Their types are given in parentheses after the attribute name along with a default value.

In general, items defaulted to Undefined are mandatory. Failing to give a value will result in an error at RD parse time.

Within RD XML documents, you can (almost always) give atomic children either as XML attribute (`att="abc"`) or as child elements (`<att>abc</abc>`). Some of the "atomic" attributes actually contain lists of items. For those, you should normally write multiple child elements (`<att>val1</att><att>val2</att>`), although sometimes it's allowed to mash together the individual list items using a variety of separators.

Here are some short words about the types you may encounter, together with valid literals:

- boolean – these allow quite a number of literals; use `True` and `False` or `yes` and `no` and stick to your choice.
- unicode string – there may be additional syntactical limitations on those. See the explanation
- integer – only decimal integer literals are allowed
- id reference – these are references to items within XML documents; all elements within RDs can have an `id` attribute, which can then be used as an id reference. Additionally, you can reference elements in different RDs using <rd-id>#<id>. Note that DaCHS does not support forward references (i.e., references to items lexically behind the referencing element).
- list of id references – Lists of id references. The values could be mashed together with commas, but prefer multiple child elements.

There are also "Dict-like" attributes. These are built from XML like:

```
<d key="ab">val1</d>
<d key="cd">val2</d>
```

In addition to key, other (possibly more descriptive) attributes for the key within these mappings may also be allowed. In special circumstances (in particular with properties) it may be useful to add to a value:

```
<property key="brokencols">ab,cd</property>
<property key="brokencols" cumulative="True">,x</property>
```

will leave `ab,cd,x` in brokencols.

Many elements can also have "structure children". These correspond to compound things with attributes and possibly children of their own. The name given at the start of each description is irrelevant to the pure user; it's the attribute name you'd use when you have the corresponding python objects. For authoring XML, you use the name in the following link; thus, the phrase "colRefs (contains Element columnRef..." means you'd write `<columnRef...>`.

Here are some guidelines as to the naming of the attributes:

- Attributes giving keys into dictionaries or similar (e.g., column names) should always be named `key`
- Attributes giving references to some source of events or data should always be named `source`, never "src" or similar

- Attributes referencing generic things should always be called `ref`; of course, references to specific things like tables or services should indicate in their names what they are supposed to reference.

Also note that examples for the usage of almost everything mentioned here can be found in in the GAVO datacenter element reference.

Element apply

A code fragment to manipulate the result row (and possibly more).

Apply elements allow embedding python code in rowmakers.

The current input fields from the grammar (including the rowmaker's vars) are available in the vars dictionary and can be changed there. You can also add new keys.

You can add new keys for shipping out in the result dictionary.

The active rowmaker is available as parent. It is also used to expand macros.

The table that the rowmaker feeds to can be accessed as targetTable. You probably only want to change meta information here (e.g., warnings or infos).

As always in procApps, you can get the embedding RD as rd; this is useful to, e.g., resolve references using rd.getByRD, and specify resdir-relative file names using rd.getAbsPath.

May occur in Element rowmaker.

Atomic Children

- **code** (unicode string; defaults to <Not given/empty>) -- A python function body.
- **deprecated** (unicode string; defaults to None) -- A deprecation message. This will be shown if this procDef is being compiled.
- **doc** (unicode string; defaults to '') -- Human-readable docs for this proc (may be interpreted as restructured text).
- **name** (unicode string; defaults to <Not given/empty>) -- A name of the proc. ProcApps compute their (python) names to be somwhat random strings. Set a name manually to receive more easily decipherable error messages. If you do that, you have to care about name clashes yourself, though.

- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **procDef** (id reference; defaults to <Not given/empty>) -- Reference to the procedure definition to apply
- **type** (One of: iterator, pargetter, regTest, rowfilter, dataFunction, descriptorGenerator, metaMaker, phraseMaker, mixinProc, dataFormatter, sourceFields, apply, t_t; defaults to None) -- The type of the procedure definition. The procedure applications will in general require certain types of definitions.

Structure Children

- bindings (contains Element bind and may be repeated zero or more times) -- Values for parameters of the procedure definition
- setups (contains Element setup and may be repeated zero or more times) -- Setup of the namespace the function will run in

Element bind

A binding of a procedure definition parameter to a concrete value.

The value to set is contained in the binding body in the form of a python expression. The body must not be empty.

May occur in Element iterator, Element rowfilter, Element apply, Element job, Element processLate, Element dataFormatter, Element regTest, Element coreProc, Element dataFunction, Element sourceFields, Element metaMaker, Element phraseMaker, Element descriptorGenerator, Element processEarly, Element pargetter.

Atomic Children

- Character content of the element (defaulting to <Not given/empty>) -- The default for the parameter. The special value __NULL__ indicates a NULL (python None) as usual. An empty content means a non-preset parameter, which must be filled in applications. The magic value __EMPTY__ allows presetting an empty string.
- **description** (whitespace normalized unicode string; defaults to None) -- Some human-readable description of what the parameter is about

- **key** (unicode string; defaults to <Undefined>) -- The name of the parameter
- **late** (boolean; defaults to 'False') -- Bind the name not at setup time but at applying time. In rowmaker procedures, for example, this allows you to refer to variables like vars or rowIter in the bindings.

Element column

A database column.

Columns contain almost all metadata to describe a column in a database table or a VOTable (the exceptions are for column properties that may span several columns, most notably indices).

Note that the type system adopted by the DC software is a subset of postgres' type system. Thus when defining types, you have to specify basically SQL types. Types for other type systems (like VOTable, XSD, or the software-internal representation in python values) are inferred from them.

Columns can have delimited identifiers as names. Don't do this, it's no end of trouble. For this reason, however, you should not use name but rather key to programmatially obtain field's values from rows.

Properties evaluated:

- std -- set to 1 to tell the tap schema importer to have the column's std column in TAP_SCHEMA 1 (it's 0 otherwise).
- statisticsTarget -- an integer to be set as this column's statistics-gathering target. Set this to something between 100 and 10000 on postgres if you have large tables and columns with strongly non-uniform distributions. Set to -1 to revert to the system default. gavo imp -m will apply changes here.
- targetType -- for a column containing a URL, the media type of the resource pointed at. This is for producing extra annotation for Aladin and friends as per http://mail.ivoa.net/pipermail/dal/2018-May/008017.html
- targetTitle -- if you give targetType, use this to set the link title (defaults to "Link").

May occur in Element table.

Atomic Children

- **description** (whitespace normalized unicode string; defaults to '') -- A short (one-line) description of the values in this column.
- **displayHint** (Display hint; defaults to '') -- Suggested presentation; the format is <kw>=<value>{,<kw>=<value>}, where what is interpreted depends on the output format. See, e.g., documentation on HTML renderers and the formatter child of outputFields.
- **fixup** (unicode string; defaults to None) -- A python expression the value of which will replace this column's value on database reads. Write a ____ to access the original value. You can use macros for the embedding table. This is for, e.g., simple URL generation (fixup="'internallink{/this/svc}'+____"). It will *only* kick in when tuples are deserialized from the database, i.e., *not* for values taken from tables in memory.
- **name** (a column name within an SQL table. These have to match the SQL regular_identifier production. In a desperate pinch, you can generate delimited identifiers (that can contain anything) by prefixing the name with 'quoted/'; defaults to <Undefined>) -- Name of the column
- **note** (unicode string; defaults to None) -- Reference to a note meta on this table explaining more about this column
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **required** (boolean; defaults to 'False') -- Record becomes invalid when this column is NULL
- **tablehead** (unicode string; defaults to None) -- Terse phrase to put into table headers for this column
- **type** (a type name; the internal type system is similar to SQL's with some restrictions and extensions. The known atomic types include: unicode, pql-float, text, spoly, char, raw, vexpr-mjd, boolean, file, smallint, vexpr-string, scircle, vexpr-float, vexpr- date, pql-string, smoc, real, spoint, pql-int, timestamp, pql-date, int4range, date, integer, box, pql-upload, double precision, sbox, bigint, time, bytea; defaults to 'real') -- datatype for the column (SQL-like type system)
- **ucd** (unicode string; defaults to '') -- UCD of the column
- **unit** (unicode string; defaults to '') -- Unit of the values

- **utype** (unicode string; defaults to None) -- utype for this column
- **verbLevel** (integer; defaults to '20') -- Minimal verbosity level at which to include this column
- **xtype** (unicode string; defaults to None) -- VOTable xtype giving the serialization form; you usually do *not* want to set this, as the xtypes actually used are computed from database type. DaCHS xtypes are only used for a few unsavoury, hopefully temporary, hacks

Structure Children

- values (contains Element values) -- Specification of legal values

Other Children

- **dmRoles** (read-only list of roles played by this column in DMs; defaults to []) -- Roles played by this column; cannot be asigned to.
- **meta** -- a piece of meta information, giving at least a name and some content. See Metadata on what is permitted here.
- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.
- **stc** (non-settable internally used value; defaults to None) -- Internally used STC information for this column (do not assign to unless instructed to do so)
- **stcUtype** (non-settable internally used value; defaults to None) -- Internally used STC information for this column (do not assign to)

Element columnRef

A reference from a group to a column within a table.

ColumnReferences do not support qualified references, i.e., you can only give simple names.

May occur in Element group.

Atomic Children

- **key** (unicode string; defaults to <Undefined>) -- The key (i.e., name) of the referenced column or param.
- **ucd** (unicode string; defaults to None) -- The UCD of the group
- **utype** (unicode string; defaults to None) -- A utype for the group

Element condDesc

A query specification for cores talking to the database.

CondDescs define inputs as a sequence of InputKeys (see Element InputKey). Internally, the values in the InputKeys can be translated to SQL.

May occur in Element scsCore, Element siapCutoutCore, Element resource, Element productCore, Element dbCore, Element fancyQueryCore, Element ssapCore.

Atomic Children

- **buildFrom** (id reference; defaults to None) -- A reference to a column or an InputKey to define this CondDesc
- **combining** (boolean; defaults to 'False') -- Allow some input keys to be missing when others are given? (you want this for pseudo- condDescs just collecting random input keys)
- **fixedSQL** (unicode string; defaults to None) -- Always insert this SQL statement into the query. Deprecated.
- **joiner** (unicode string; defaults to 'OR') -- When yielding multiple fragments, join them using this operator (probably the only thing besides OR is AND).
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **required** (boolean; defaults to 'False') -- Reject queries not filling the InputKeys of this CondDesc
- **silent** (boolean; defaults to 'False') -- Do not produce SQL from this CondDesc. This can be used to convey meta information to the core. However, in general, a service is a more appropriate place to deal with such information, and thus you should prefer service InputKeys to silent CondDescs.

Structure Children

- group (contains Element group) -- Group child input keys in the input table (primarily interesting for web forms, where this grouping is shown graphically; Set the style property to compact to have a one-line group there)
- inputKeys (contains Element inputKey and may be repeated zero or more times) -- One or more InputKeys defining the condition's input.
- phraseMaker (contains Element phraseMaker) -- Code to generate custom SQL from the input keys

Element coverage

The coverage of a resource.

For now, this is attached to the complete resource rather than the table, since this is where it sits in VOResource. DaCHS *could* be a bit more flexible, allowing different coverages per publish element. It is not right now, though.

Note: Technically, this will introduce or amend the coverage meta element. The information given here will be masked if you define a coverage meta on the service or table level. Just do not do that.

May occur in Element resource.

Atomic Children

- **spatial** (unicode string; defaults to <Not given/empty>) -- A MOC in ASCII representation giving the ICRS coverage of the resource
- **spectral** (A sequence of intervals (a space-separated pair of floats; defaults to u'[]') -- Interval(s) of spectral coverage, in meters of BARYCENTER vacuum wavelength.
- **temporal** (A sequence of intervals (a space-separated pair of floats; defaults to u'[]') -- Interval(s) of temporal coverage, in MJD (for TT BARYCENTER).

Structure Children

- updater (contains Element updater) -- Rules for automatic computation or updating of coverage information.

Element customDF

A custom data function for a service.

Custom data functions can be used to expose certain aspects of a service to Nevow templates. Thus, their definition usually only makes sense with custom templates, though you could, in principle, override built-in render functions.

In the data functions, you have the names ctx for nevow's context and data for whatever data the template passes to the renderer.

You can access the embedding service as service, the embedding RD as service.rd.

You can return arbitrary python objects -- whatever the render functions can deal with. You could, e.g., write:

```
<customDF name="now">
  return datetime.datetime.utcnow()
</customDF>
```

You also see a nevow context within the function. You can use that to access a query paramter `order` like this:

```
args = inevow.IRequest(ctx).args
sortOrder = args.get("order", ["authors"])
```

May occur in Element service.

Atomic Children

- Character content of the element (defaulting to '') -- Function body of the renderer; the arguments are named ctx and data.
- **name** (unicode string; defaults to <Undefined>) -- Name of the render function (use this in the n:render or n:data attribute in custom templates).

Element customRF

A custom render function for a service.

Custom render functions can be used to expose certain aspects of a service to Nevow templates. Thus, their definition usually only makes sense with custom templates, though you could, in principle, override built-in render functions.

In the render functions, you have the names ctx for nevow's context and data for whatever data the template passes to the renderer.

You can return anything that can be in a stan DOM. Usually, this will be a string. To return HTML, use the stan DOM available under the T namespace.

As an example, the following code returns the current data as a link:

```
return ctx.tag[T.a(href=data)[data]]
```

You can access the embedding service as service, the embedding RD as service.rd.

May occur in Element service.

Atomic Children

- Character content of the element (defaulting to '') -- Function body of the renderer; the arguments are named ctx and data.
- **name** (unicode string; defaults to <Undefined>) -- Name of the render function (use this in the n:render or n:data attribute in custom templates).

Element data

A description of how to process data from a given set of sources.

Data descriptors bring together a grammar, a source specification and "makes", each giving a table and a rowmaker to feed the table from the grammar output.

They are the "executable" parts of a resource descriptor. Their ids are used as arguments to gavoimp for partial imports.

May occur in Element resource.

Atomic Children

- **auto** (boolean; defaults to 'True') -- Import this data set if not explicitly mentioned on the command line?
- **dependents** (Zero or more unicode string-typed *recreateAfter* elements; defaults to u'[]') -- A data ID to recreate when this resource is remade; use # syntax to reference in other RDs.

- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **updating** (boolean; defaults to 'False') -- Keep existing tables on import? You usually want this False unless you have some kind of sources management, e.g., via a sources ignore specification.

Structure Children

- grammar (contains one of keyValueGrammar, cdfHeaderGrammar, directGrammar, dictlistGrammar, freeREGrammar, voTableGrammar, customGrammar, rowsetGrammar, fitsTableGrammar, csvGrammar, nullGrammar, odbcGrammar, fitsProdGrammar, contextGrammar, transparentGrammar, columnGrammar, embeddedGrammar, binaryGrammar, pdsGrammar, reGrammar, mySQLDumpGrammar) -- Grammar used to parse this data set.
- makes (contains Element make and may be repeated zero or more times) -- Specification of a target table and the rowmaker to feed them.
- params (contains Element param and may be repeated zero or more times) -- Param ("global columns") for this data (mostly for VOTable serialization).
- registration (contains Element publish (data)) -- A registration (to the VO registry) of this table or data collection.
- rowmakers (contains Element rowmaker and may be repeated zero or more times) -- Embedded build rules (preferably put rowmakers directly into make elements)
- sources (contains Element sources) -- Specification of sources that should be fed to the grammar.
- tables (contains Element table and may be repeated zero or more times) -- Embedded table definitions (usually, tables are defined toplevel)

Other Children

- **meta** -- a piece of meta information, giving at least a name and some content. See Metadata on what is permitted here.
- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Element DEFAULTS

Defaults for macros.

In STREAMs and NXSTREAMs, DEFAULTS let you specify values filled into macros when a FEED doesn't given them. Macro names are attribute names (or element names, if you insist), defaults are their values.

May occur in Element NXSTREAM, Element EDIT, Element STREAM, Element lateEvents, Element events.

Element dm

an annotation of a table in terms of data models.

The content of this element is a Simple Instance Language clause.

May occur in Element outputTable, Element table.

Atomic Children

- Character content of the element (defaulting to '') -- SIL (simple instance language) annotation.

Element EDIT

an event stream targeted at editing other structures.

When replaying a stream in the presence of EDITs, the elements are are continually checked against ref. If an element matches, the children of edit will be played back into it.

May occur in Element LFEED, Element LOOP, Element mixinDef, Element FEED.

Atomic Children

- **doc** (unicode string; defaults to None) -- A description of this stream (should be restructured text).
- **passivate** (unicode string; defaults to None) -- If set to True, do not expand active elements immediately in the body of these events (as in an NXSTREAM)
- **ref** (unicode string; defaults to <Undefined>) -- Destination of the edits, in the form elementName[<name or id>]

Structure Children

- DEFAULTS (contains Element DEFAULTS) -- A mapping giving defaults for macros expanded in this stream. Macros not defaulted will fail when not given in a FEED's attributes.

Element events

An event stream as a child of another element.

May occur in Element LFEED, Element LOOP, Element mixinDef, Element FEED.

Atomic Children

- **doc** (unicode string; defaults to None) -- A description of this stream (should be restructured text).
- **passivate** (unicode string; defaults to None) -- If set to True, do not expand active elements immediately in the body of these events (as in an NXSTREAM)

Structure Children

- DEFAULTS (contains Element DEFAULTS) -- A mapping giving defaults for macros expanded in this stream. Macros not defaulted will fail when not given in a FEED's attributes.

Element execute

a container for calling code.

This is a cron-like functionality. The jobs are run in separate threads, so they need to be thread-safe with respect to the rest of DaCHS. DaCHS serializes calls, though, so that your code should never run twice at the same time.

At least on CPython, you must make sure your code does not block with the GIL held; this is still in the server process. If you do daring things, fork off (note that you must not use any database connections you may have after forking, which means you can't safely use the RD passed in). See the docs on Element job.

Then testing/debugging such code, use `gavo admin execute rd#id` to immediately run the jobs.

May occur in Element resource.

Atomic Children

- **at** (Comma-separated list of strings; defaults to <Not given/empty>) -- One or more hour:minute pairs at which to run the code each day. This conflicts with every. Optionally, you can prefix each time by one of m<dom> or w<dow> for jobs only to be exectued at some day of the month or week, both counted from 1. So, 'm22 7:30, w3 15:02' would execute on the 22nd of each month at 7:30 UTC and on every wednesday at 15:02.
- **debug** (boolean; defaults to 'False') -- If true, on execution of external processes (span or spawnPython), the output will be accumulated and mailed to the administrator. Note that output of the actual cron job itself is not caught (it might turn up in serverStderr). You could use execDef.outputAccum.append(<stuff>) to have information from within the code included.
- **every** (integer; defaults to <Not given/empty>) -- Run the job roughly every this many seconds. This conflicts with at. Note that the first execution of such a job is after every/10 seconds, and that the timers start anew at every server restart. So, if you restart often, these jobs may run much more frequently or not at all if the interval is large. If every is smaller than zero, the job will be executed immediately when the RD is being loaded and is then run every abs(every) seconds
- **title** (unicode string; defaults to <Undefined>) -- Some descriptive title for the job; this is used in diagnostics.

Structure Children

- job (contains Element job) -- The code to run.

Other Children

- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Element foreignKey

A description of a foreign key relation between this table and another one.

May occur in Element outputTable, Element table.

Atomic Children

- **dest** (unicode string; defaults to <Not given/empty>) -- Comma- separated list of columns in the target table belonging to its key. No checks for their existence, uniqueness, etc. are done here. If not given, defaults to source.
- **inTable** (id reference; defaults to <Undefined>) -- Reference to the table the foreign key points to.
- **metaOnly** (boolean; defaults to 'False') -- Do not tell the database to actually create the foreign key, just declare it in the metadata. This is for when you want to document a relationship but don't want the DB to actually enforce this. This is typically a wise thing to do when you have, say a gigarecord of flux/density pairs and only several thousand metadata records -- you may want to update the latter without having to tear down the former.
- **source** (unicode string; defaults to <Undefined>) -- Comma- separated list of local columns corresponding to the foreign key. No sanity checks are performed here.

Element group

A group is a collection of columns, parameters and other groups with a dash of metadata.

Within a group, you can refer to columns or params of the enclosing table by their names. Nothing outside of the enclosing table can be part of a group.

Rather than referring to params, you can also embed them into a group; they will then *not* be present in the embedding table.

Groups may contain groups.

One application for this is grouping input keys for the form renderer. For such groups, you probably want to give the label property (and possibly cssClass).

May occur in Element inputTable, Element outputTable, Element table, Element condDesc.

Atomic Children

- **description** (whitespace normalized unicode string; defaults to None) -- A short (one-line) description of the group

- **name** (A name for a table or service parameter. These have to match `[A-Za-z_][A-Za-z0-9_]*$`.; defaults to None) -- Name of the column (must be SQL-valid for onDisk tables)
- **ucd** (unicode string; defaults to None) -- The UCD of the group
- **utype** (unicode string; defaults to None) -- A utype for the group

Structure Children

- columnRefs (contains Element columnRef and may be repeated zero or more times) -- References to table columns belonging to this group
- groups (contains an instance of the embedding element and may be repeated zero or more times) -- Sub-groups of this group (names are still referenced from the enclosing table)
- paramRefs (contains Element paramRef and may be repeated zero or more times) -- Names of table parameters belonging to this group
- params (contains Element param and may be repeated zero or more times) -- Immediate param elements for this group (use paramref to reference params defined in the parent table)

Other Children

- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Element httpUpload

An upload going with a URL.

May occur in Element url.

Atomic Children

- Character content of the element (defaulting to '') -- Inline data to be uploaded (conflicts with source)
- **fileName** (unicode string; defaults to None) -- Remote file name for the uploaded file.
- **name** (unicode string; defaults to <Undefined>) -- Name of the upload parameter
- **source** (unicode string; defaults to <Not given/empty>) -- Path to a file containing the data to be uploaded.

Element ignoreOn

A condition on a row that, if true, causes the row to be dropped.

Here, you can set bail to abort an import when the condition is met rather than just dropping the row.

May occur in Element voTableGrammar, Element rowmaker, Element reGrammar, Element contextGrammar, Element columnGrammar, Element cdfHeaderGrammar, Element fitsTableGrammar, Element rowsetGrammar, Element binaryGrammar, Element fitsProdGrammar, Element pdsGrammar, Element customGrammar, Element odbcGrammar, Element mySQLDumpGrammar, Element freeREGrammar, Element dictlistGrammar, Element keyValueGrammar, Element csvGrammar, Element embeddedGrammar, Element transparentGrammar, Element nullGrammar.

Atomic Children

- **bail** (boolean; defaults to 'False') -- Abort when condition is met?
- **name** (unicode string; defaults to 'unnamed') -- A name that should help the user figure out what trigger caused some condition to fire.

Structure Children

- triggers (contains any of and,keyPresent,keyNull,keyIs,keyMissing,not and may be repeated zero or more times) -- One or more conditions joined by an implicit logical or. See Triggers for information on what can stand here.

Element ignoreSources

A specification of sources to ignore.

Sources mentioned here are compared against the inputsDir-relative path of sources generated by sources (cf. Element sources). If there is a match, the corresponding source will not be processed.

You can get ignored files from various sources. If you give more than one source, the set of ignored files is the union of the the individual sets.

fromdbUpdating is a bit special in that the query must return UTC timestamps of the file's mtime during the last ingest in addition to the accrefs (see the tutorial for an example).

Macros are expanded in the RD.

May occur in Element sources.

Atomic Children

- **fromdb** (unicode string; defaults to None) -- A DB query to obtain a set of sources to ignore; the select clause must select exactly one column containing the source key. See also Using fromdb on ignoreSources
- **fromdbUpdating** (unicode string; defaults to None) -- A DB query to obtain a set of sources to ignore unless they the timestamp on disk is newer than what's returned. The query given must return pairs of accrefs and UTC timestamps of the last ingest. See also Using fromdbUpdating on ignoreSources
- **fromfile** (unicode string; defaults to None) -- A name of a file containing blacklisted source paths, one per line. Empty lines and lines beginning with a hash are ignored.
- **patterns** (Zero or more unicode string-typed *pattern* elements; defaults to u'[]') -- Shell patterns to ignore. Slashes are treated like any other character, i.e., patterns do not know about paths.

Element index

A description of an index in the database.

In real databases, indices may be fairly complex things; still, the most common usage here will be to just index a single column:

```
<index columns="my_col"/>
```

To index over functions, use the character content; parentheses are added by DaCHS, so don't have them in the content. An explicit specification of the index expression is also necessary to allow RE pattern matches using indices in character columns (outside of the C locale). That would be:

```
<index columns="uri">uri text_pattern_ops</index>
```

(you still want to give columns so the metadata engine is aware of the index). See section "Operator Classes and Operator Families" in the Postgres documentation for details.

For pgsphere-valued columns, you at the time of writing need to specify the method:

```
<index columns="coverage" method="GIST"/>
```

To define q3c indices, use the `//scs#q3cindex` mixin; if you're devious enough to require something more flexible, have a look at that mixin's definition.

If indexed columns take part in a DaCHS-defined view, DaCHS will not notice. You should still declare the indices so users will see them in the metadata; writing:

```
<index columns="col1, col2, col3"/>
```

is sufficent for that.

May occur in Element outputTable, Element table.

Atomic Children

- **cluster** (boolean; defaults to 'False') -- Cluster the table according to this index?
- **columns** (Comma-separated list of strings; defaults to '') -- Table columns taking part in the index (must be given even if there is an expression building the index and mention all columns taking part in the index generated by it
- Character content of the element (defaulting to '') -- Raw SQL specifying an expression the table should be indexed for. If not given, the expression will be generated from columns (which is what you usually want).
- **method** (unicode string; defaults to None) -- The indexing method, like an index type. In the 8.x, series of postgres, you need to set method=GIST for indices over pgsphere columns; otherwise, you should not need to worry about this.
- **name** (unicode string; defaults to <Undefined>) -- Name of the index. Defaults to something computed from columns; the name of the parent table will be prepended in the DB. The default will *not* work if you have multiple indices on one set of columns.

Element inputKey

A description of a piece of input.

Think of inputKeys as abstractions for input fields in forms, though they are used for services not actually exposing HTML forms as well.

Some of the DDL-type attributes (e.g., references) only make sense here if columns are being defined from the InputKey.

Properties evaluated:

- defaultForForm -- a value entered into form fields by default (be stingy with those; while it's nice to not have to set things presumably right for almost everyone, having to delete stuff you don't want over and over is really annoying).
- adaptToRenderer -- a true boolean literal here causes the param to be adapted for the renderer (e.g., float could become vizierexpr-float). You'll usually not want this, because the expressions are generally evaluated by the database, and the condDescs do the adaptation themselves. This is mainly for rare situations like file uploads in custom cores.
- notForRenderer -- a renderer name for which this inputKey is suppressed
- onlyForRenderer -- a renderer name for which this inputKey will be preserved; it will be dropped for all others.

May occur in Element inputTable, Element contextGrammar, Element condDesc, Element service, Element datalinkCore.

Atomic Children

- Character content of the element (defaulting to <Not given/empty>) -- The value of parameter. It is parsed according to the param's type using the default parser for the type VOTable tabledata.
- **description** (whitespace normalized unicode string; defaults to '') -- A short (one-line) description of the values in this column.
- **displayHint** (Display hint; defaults to '') -- Suggested presentation; the format is <kw>=<value>{,<kw>=<value>}, where what is interpreted depends on the output format. See, e.g., documentation on HTML renderers and the formatter child of outputFields.
- **fixup** (unicode string; defaults to None) -- A python expression the value of which will replace this column's value on database reads. Write a ____ to access the original value. You can use macros for the embedding table. This is for, e.g., simple URL generation (fixup="'internallink{/this/svc}'+____"). It will *only* kick in when tuples are deserialized from the database, i.e., *not* for values taken from tables in memory.
- **inputUnit** (unicode string; defaults to None) -- Override unit of the table column with this.
- **multiplicity** (unicode string; defaults to None) -- Set this to single to have an atomic value (chosen at random if multiple input values are given),

forced-single to have an atomic value and raise an exception if multiple values come in, or multiple to receive lists. On the form renderer, this is ignored, and the values are what nevow formal passes in. If not given, it is single unless there is a values element with options, in which case it's multiple.

- **name** (A name for a table or service parameter. These have to match `[A-Za-z_][A-Za-z0-9_]*$`.; defaults to <Undefined>) -- Name of the param
- **note** (unicode string; defaults to None) -- Reference to a note meta on this table explaining more about this column
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **required** (boolean; defaults to 'False') -- Record becomes invalid when this column is NULL
- **showItems** (integer; defaults to '3') -- Number of items to show at one time on selection widgets.
- **std** (boolean; defaults to 'False') -- Is this input key part of a standard interface for registry purposes?
- **tablehead** (unicode string; defaults to None) -- Terse phrase to put into table headers for this column
- **type** (a type name; the internal type system is similar to SQL's with some restrictions and extensions. The known atomic types include: unicode, pql-float, text, spoly, char, raw, vexpr-mjd, boolean, file, smallint, vexpr-string, scircle, vexpr-float, vexpr- date, pql-string, smoc, real, spoint, pql-int, timestamp, pql-date, int4range, date, integer, box, pql-upload, double precision, sbox, bigint, time, bytea; defaults to 'real') -- datatype for the column (SQL-like type system)
- **ucd** (unicode string; defaults to '') -- UCD of the column
- **unit** (unicode string; defaults to '') -- Unit of the values
- **utype** (unicode string; defaults to None) -- utype for this column
- **verbLevel** (integer; defaults to '20') -- Minimal verbosity level at which to include this column
- **widgetFactory** (unicode string; defaults to None) -- A python expression for a custom widget factory for this input, e.g., 'Hidden' or 'widgetFactory(TextArea, rows=15, cols=30)'

- **xtype** (unicode string; defaults to None) -- VOTable xtype giving the serialization form; you usually do *not* want to set this, as the xtypes actually used are computed from database type. DaCHS xtypes are only used for a few unsavoury, hopefully temporary, hacks

Structure Children

- values (contains Element values) -- Specification of legal values

Other Children

- **dmRoles** (read-only list of roles played by this column in DMs; defaults to []) -- Roles played by this column; cannot be asigned to.
- **meta** -- a piece of meta information, giving at least a name and some content. See Metadata on what is permitted here.
- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.
- **stc** (non-settable internally used value; defaults to None) -- Internally used STC information for this column (do not assign to unless instructed to do so)
- **stcUtype** (non-settable internally used value; defaults to None) -- Internally used STC information for this column (do not assign to)

Element job

Python code for use within execute.

The resource descriptor this runs at is available as rd, the execute definition (having such attributes as title, job, plus any properties given in the RD) as execDef.

Note that no I/O capturing takes place (that's impossible since in general the jobs run within the server). To have actual cron jobs, use `execDef.spawn(["cmd", "arg1"...])`. This will send a mail on failed execution and also raise a ReportableError in that case.

In the frequent use case of a resdir-relative python program, you can use the `execDef.spawnPython(modulePath)` function.

If you must stay within the server process, you can do something like:

```
mod, _ = utils.loadPythonModule(rd.getAbsPath("bin/coverageplot.py"))
mod.makePlot()
```

-- in that way, your code can sit safely within the resource directory and you still don't have to manipulate the module path.

May occur in Element execute.

Atomic Children

- **code** (unicode string; defaults to <Not given/empty>) -- A python function body.
- **deprecated** (unicode string; defaults to None) -- A deprecation message. This will be shown if this procDef is being compiled.
- **doc** (unicode string; defaults to '') -- Human-readable docs for this proc (may be interpreted as restructured text).
- **name** (unicode string; defaults to <Not given/empty>) -- A name of the proc. ProcApps compute their (python) names to be somwhat random strings. Set a name manually to receive more easily decipherable error messages. If you do that, you have to care about name clashes yourself, though.
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **procDef** (id reference; defaults to <Not given/empty>) -- Reference to the procedure definition to apply
- **type** (One of: iterator, pargetter, regTest, rowfilter, dataFunction, descriptorGenerator, metaMaker, phraseMaker, mixinProc, dataFormatter, sourceFields, apply, t_t; defaults to None) -- The type of the procedure definition. The procedure applications will in general require certain types of definitions.

Structure Children

- bindings (contains Element bind and may be repeated zero or more times) -- Values for parameters of the procedure definition
- setups (contains Element setup and may be repeated zero or more times) -- Setup of the namespace the function will run in

Element lateEvents

An event stream played back by a mixin when the substrate is being finalised (but before the early processing).

May occur in Element mixinDef.

Atomic Children

- **doc** (unicode string; defaults to None) -- A description of this stream (should be restructured text).
- **passivate** (unicode string; defaults to None) -- If set to True, do not expand active elements immediately in the body of these events (as in an NXSTREAM)

Structure Children

- DEFAULTS (contains Element DEFAULTS) -- A mapping giving defaults for macros expanded in this stream. Macros not defaulted will fail when not given in a FEED's attributes.

Element macDef

A macro definition within an RD.

The macro defined is available on the parent; macros are expanded within the parent (behaviour is undefined if you try a recursive expansion).

May occur in Element resource.

Atomic Children

- Character content of the element (defaulting to '') -- Replacement text of the macro
- **name** (unicode string; defaults to <Undefined>) -- Name the macro will be available as

Element make

A build recipe for tables belonging to a data descriptor.

All makes belonging to a DD will be processed in the order in which they appear in the file.

May occur in Element data.

Atomic Children

- **parmaker** (id reference; defaults to <Not given/empty>) -- The parmaker (i.e., mapping rules from grammar parameters to table parameters) for the table being made. You will usually not give a parmaker.
- **role** (unicode string; defaults to None) -- The role of the embedded table within the data set
- **rowSource** (One of: rows, parameters; defaults to 'rows') -- Source for the raw rows processed by this rowmaker.
- **rowmaker** (id reference; defaults to <Not given/empty>) -- The rowmaker (i.e., mapping rules from grammar keys to table columns) for the table being made.
- **table** (id reference; defaults to <Undefined>) -- Reference to the table to be embedded

Structure Children

- scripts (contains Element script and may be repeated zero or more times) -- Code snippets attached to this object. See Scripting .

Element map

A mapping rule.

To specify the source of a mapping, you can either

- grab a value from what's emitted by the grammar or defined using var via the source attribute. The value given for source is converted to a python value and stored.
- or give a python expression in the body. In that case, no further type conversion will be attempted.

If neither source or a body is given, map uses the key attribute as its source attribute.

The map rule generates a key/value pair in the result record.

May occur in Element rowmaker.

Atomic Children

- Character content of the element (defaulting to '') -- A python expression giving the value for key.
- **key** (unicode string; defaults to <Undefined>) -- Name of the column the value is to end up in.
- **nullExcs** (unicode string; defaults to <Not given/empty>) -- Exceptions that should be caught and cause the value to be NULL, separated by commas.
- **nullExpr** (unicode string; defaults to <Not given/empty>) -- A python expression for a value that is mapped to NULL (None). Equality is checked after building the value, so this expression has to be of the column type. Use map with the parseWithNull function to catch null values before type conversion.
- **source** (unicode string; defaults to None) -- Source key name to convert to column value (either a grammar key or a var).

Element mixinDef

A definition for a resource mixin.

Resource mixins are resource descriptor fragments typically rooted in tables (though it's conceivable that other structures could grow mixin attributes as well).

They are used to define and implement certain behaviours components of the DC software want to see:

- products want to be added into their table, and certain fields are required within tables describing products
- tables containing positions need some basic machinery to support scs.
- siap needs quite a bunch of fields

Mixins consist of events that are played back on the structure mixing in before anything else happens (much like original) and two procedure definitions, viz, processEarly and processLate. These can access the structure that has the mixin as substrate.

processEarly is called as part of the substrate's completeElement method. processLate is executed just before the parser exits. This is the place to fix up

anything that uses the table mixed in. Note, however, that you should be as conservative as possible here -- you should think of DC structures as immutable as long as possible.

Programmatically, you can check if a certain table mixes in something by calling its mixesIn method.

Recursive application of mixins, even to seperate objects, will deadlock.

May occur in Element resource.

Atomic Children

- **doc** (unicode string; defaults to None) -- Documentation for this mixin
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **reexpand** (boolean; defaults to 'False') -- Force re-expansion of macros; usually, when replaying, each string is only expanded once, mainly to avoid overly long backslash-fences. Set this to true to force further expansion.
- **source** (id reference; defaults to None) -- id of a stream to replay

Structure Children

- edits (contains Element EDIT and may be repeated zero or more times) -- Changes to be performed on the events played back.
- events (contains Element events) -- Events to be played back into the structure mixing this in at mixin time.
- lateEvents (contains Element lateEvents) -- Events to be played back into the structure mixing this in at completion time.
- pars (contains Element mixinPar and may be repeated zero or more times) -- Parameters available for this mixin.
- processEarly (contains Element processEarly) -- Code executed at element fixup.
- processLate (contains Element processLate) -- Code executed resource fixup.
- prunes (contains Element PRUNE and may be repeated zero or more times) -- Conditions for removing items from the playback stream.

Macros predefined here: Macro RSTservicelink, Macro RSTtable, Macro decapitalize, Macro getConfig, Macro internallink, Macro magicEmpty, Macro metaString, Macro quote, Macro rdId, Macro rdIdDotted, Macro schema, Macro sqlquote, Macro test, Macro today, Macro upper, Macro urlquote

Element mixinPar

A parameter definition for mixins.

The (optional) body provides a default for the parameter.

May occur in Element mixinDef.

Atomic Children

- Character content of the element (defaulting to <Not given/empty>) -- The default for the parameter. A __NULL__ here does not directly mean None/NULL, but since the content will frequently end up in attributes, it will ususally work as presetting None. An empty content means a non-preset parameter, which must be filled in applications. The magic value __EMPTY__ allows presetting an empty string.
- **description** (whitespace normalized unicode string; defaults to None) -- Some human-readable description of what the parameter is about
- **key** (unicode string; defaults to <Undefined>) -- The name of the parameter
- **late** (boolean; defaults to 'False') -- Bind the name not at setup time but at applying time. In rowmaker procedures, for example, this allows you to refer to variables like vars or rowIter in the bindings.

Element option

A value for enumerated columns.

For presentation purposes, an option can have a title, defaulting to the option's value.

May occur in Element values.

Atomic Children

- Character content of the element (defaulting to '') -- The value of the option; this is what is used in, e.g., queries and the like.
- **title** (unicode string; defaults to <Not given/empty>) -- A Label for presentation purposes; defaults to val.

Element outputField

A column for defining the output of a service.

It adds some attributes useful for rendering results, plus functionality specific to certain cores.

The optional formatter overrides the standard formatting code in HTML (which is based on units, ucds, and displayHints). You receive the item from the database as data and must return a string or nevow stan. In addition to the standard Functions available for row makers you have queryMeta and nevow's tags in T.

Here's an example for generating a link to another service using this facility:

```
<outputField name="more"
    select="array[centerAlpha,centerDelta] as more" tablehead="More"
    description="More exposures near the center of this plate">
  <formatter><![CDATA[
    return T.a(href=base.makeSitePath("/lswscans/res/positions/q/form?"
        "POS=%s,%s&SIZE=1&INTERSECT=OVERLAPS&cutoutSize=0.5"
        "&__nevow_form__=genForm"%tuple(data)
        ))["More"] ]]>
  </formatter>
</outputField>
```

Within the code, in addition do data, you see rd and queryMeta.

May occur in Element outputTable.

Atomic Children

- **description** (whitespace normalized unicode string; defaults to '') -- A short (one-line) description of the values in this column.
- **displayHint** (Display hint; defaults to '') -- Suggested presentation; the format is <kw>=<value>{,<kw>=<value>}, where what is interpreted depends on the output format. See, e.g., documentation on HTML renderers and the formatter child of outputFields.
- **fixup** (unicode string; defaults to None) -- A python expression the value of which will replace this column's value on database reads. Write a ____ to access the original value. You can use macros for the embedding table. This is for, e.g., simple URL generation (fixup="'internallink{/this/svc}'+____"). It will *only* kick in when tuples are deserialized from the database, i.e., *not* for values taken from tables in memory.

- **formatter** (unicode string; defaults to None) -- Function body to render this item to HTML.
- **name** (a column name within an SQL table. These have to match the SQL regular_identifier production. In a desperate pinch, you can generate delimited identifiers (that can contain anything) by prefixing the name with 'quoted/'; defaults to <Undefined>) -- Name of the column
- **note** (unicode string; defaults to None) -- Reference to a note meta on this table explaining more about this column
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **required** (boolean; defaults to 'False') -- Record becomes invalid when this column is NULL
- **select** (unicode string; defaults to <Undefined>) -- Use this SQL fragment rather than field name in the select list of a DB based core.
- **sets** (Comma-separated list of strings; defaults to '') -- Output sets this field should be included in; ALL includes the field in all output sets.
- **tablehead** (unicode string; defaults to None) -- Terse phrase to put into table headers for this column
- **type** (a type name; the internal type system is similar to SQL's with some restrictions and extensions. The known atomic types include: unicode, pql-float, text, spoly, char, raw, vexpr-mjd, boolean, file, smallint, vexpr-string, scircle, vexpr-float, vexpr- date, pql-string, smoc, real, spoint, pql-int, timestamp, pql-date, int4range, date, integer, box, pql-upload, double precision, sbox, bigint, time, bytea; defaults to 'real') -- datatype for the column (SQL-like type system)
- **ucd** (unicode string; defaults to '') -- UCD of the column
- **unit** (unicode string; defaults to '') -- Unit of the values
- **utype** (unicode string; defaults to None) -- utype for this column
- **verbLevel** (integer; defaults to '20') -- Minimal verbosity level at which to include this column
- **wantsRow** (boolean; defaults to None) -- Does formatter expect the entire row rather than the colum value only?

- **xtype** (unicode string; defaults to None) -- VOTable xtype giving the serialization form; you usually do *not* want to set this, as the xtypes actually used are computed from database type. DaCHS xtypes are only used for a few unsavoury, hopefully temporary, hacks

Structure Children

- values (contains Element values) -- Specification of legal values

Other Children

- **dmRoles** (read-only list of roles played by this column in DMs; defaults to []) -- Roles played by this column; cannot be asigned to.
- **meta** -- a piece of meta information, giving at least a name and some content. See Metadata on what is permitted here.
- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.
- **stc** (non-settable internally used value; defaults to None) -- Internally used STC information for this column (do not assign to unless instructed to do so)
- **stcUtype** (non-settable internally used value; defaults to None) -- Internally used STC information for this column (do not assign to)

Element outputTable

A table that has outputFields for columns.

Cores always have one of these, but they are implicitly defined by the underlying database tables in case of dbCores and such.

Services may define output tables to modify what is coming back fromt the core. Note that this usually only affects the output to web browsers. To use the output table also through VO protocols (and when producing VOTables, FITS files, and the like), you need to set the service's votableRespectsOutputTable property to True.

May occur in Element resource, Element service.

Atomic Children

- **adql** (boolean or 'hidden'; defaults to 'False') -- Should this table be available for ADQL queries? In addition to True/False, this can also be 'hidden' for tables readable from the TAP machinery but not published in the metadata; this is useful for, e.g., tables contributing to a published view. Warning: adql=hidden is incompatible with setting readProfiles manually.
- **allProfiles** (Comma separated list of profile names.; defaults to u'admin, msdemlei') -- A (comma separated) list of profile names through which the object can be written or administred (oh, and the default is not admin, msdemlei but is the value of [db]maintainers)
- **autoCols** (Comma-separated list of strings; defaults to '') -- Column names obtained from fromTable; you can use shell patterns into the output table's parent table (in a table core, that's the queried table; in a service, it's the core's output table) here.
- **dupePolicy** (One of: drop, check, overwrite, dropOld; defaults to 'check') -- Handle duplicate rows with identical primary keys manually by raising an error if existing and new rows are not identical (check), dropping the new one (drop), updating the old one (overwrite), or dropping the old one and inserting the new one (dropOld)?
- **forceUnique** (boolean; defaults to 'False') -- Enforce dupe policy for primary key (see dupePolicy)?
- A mixin reference, typically to support certain protocol. See Mixins.
- **namePath** (id reference; defaults to None) -- Reference to an element tried to satisfy requests for names in id references of this element's children.
- **nrows** (integer; defaults to None) -- Approximate number of rows in this table (usually, you want to use dachs limits to fill this out; write <nrows>0</nrows> to enable that).
- **onDisk** (boolean; defaults to 'False') -- Table in the database rather than in memory?
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **primary** (Comma-separated list of strings; defaults to '') -- Comma separated names of columns making up the primary key.

- **readProfiles** (Comma separated list of profile names.; defaults to u'trustedquery') -- A (comma separated) list of profile names through which the object can be read.
- **system** (boolean; defaults to 'False') -- Is this a system table? If it is, it will not be dropped on normal imports, and accesses to it will not be logged.
- **temporary** (boolean; defaults to 'False') -- If this is an onDisk table, make it temporary? This is mostly useful for custom cores and such.
- **verbLevel** (integer; defaults to None) -- Copy over columns from fromTable not more verbose than this.
- **viewStatement** (unicode string; defaults to None) -- A single SQL statement to create a view. Setting this makes this table a view. The statement will typically be something like CREATE VIEW \curtable AS (SELECT \colNames FROM...).

Structure Children

- columns (contains Element outputField and may be repeated zero or more times) -- Output fields for this table.
- dm (contains Element dm and may be repeated zero or more times) -- Annotations for data models.
- foreignKeys (contains Element foreignKey and may be repeated zero or more times) -- Foreign keys used in this table
- groups (contains Element group and may be repeated zero or more times) -- Groups for columns and params of this table
- indices (contains Element index and may be repeated zero or more times) -- Indices defined on this table
- params (contains Element param and may be repeated zero or more times) -- Param ("global columns") for this table.
- registration (contains Element publish (data)) -- A registration (to the VO registry) of this table or data collection.
- stc (contains Element stc and may be repeated zero or more times) -- STC-S definitions of coordinate systems.

Other Children

- **meta** -- a piece of meta information, giving at least a name and some content. See Metadata on what is permitted here.
- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Macros predefined here: Macro RSTservicelink, Macro RSTtable, Macro colNames, Macro curtable, Macro decapitalize, Macro getConfig, Macro getParam, Macro internallink, Macro magicEmpty, Macro metaString, Macro nameForUCD, Macro nameForUCDs, Macro qName, Macro quote, Macro rdId, Macro rdIdDotted, Macro schema, Macro sqlquote, Macro tablename, Macro test, Macro today, Macro upper, Macro urlquote

Element par

A parameter of a procedure definition.

Bodies of ProcPars are interpreted as python expressions, in which macros are expanded in the context of the procedure application's parent. If a body is empty, the parameter has no default and has to be filled by the procedure application.

May occur in Element setup.

Atomic Children

- Character content of the element (defaulting to <Not given/empty>) -- The default for the parameter. The special value ___NULL___ indicates a NULL (python None) as usual. An empty content means a non-preset parameter, which must be filled in applications. The magic value ___EMPTY___ allows presetting an empty string.
- **description** (whitespace normalized unicode string; defaults to None) -- Some human-readable description of what the parameter is about
- **key** (unicode string; defaults to <Undefined>) -- The name of the parameter
- **late** (boolean; defaults to 'False') -- Bind the name not at setup time but at applying time. In rowmaker procedures, for example, this allows you to refer to variables like vars or rowIter in the bindings.

Element param

A table parameter.

This is like a column, except that it conceptually applies to all rows in the table. In VOTables, params will be rendered as PARAMs.

While we validate the values passed using the DC default parsers, at least the VOTable params will be literal copies of the string passed in.

You can obtain a parsed value from the value attribute.

Null value handling is a bit tricky with params. An empty param (like `<param name="x"/>`) is always NULL (None in python). In order to allow setting NULL even where syntactially something has to stand, we also turn any ___NULL___ to None.

For floats, NaN will also yield NULLs. For integers, you can also use

> <param name="x" type="integer"><values nullLiteral="-1"/>-1</params>

For arrays, floats, and strings, the interpretation of values is undefined. Following VOTable practice, we do not tell empty strings and NULLs apart; for internal usage, there is a little hack: ___EMPTY___ as literal does set an empty string. This is to allow defaulting of empty strings -- in VOTables, these cannot be distinguished from "true" NULLs.

May occur in Element group, Element outputTable, Element table, Element data.

Atomic Children

- Character content of the element (defaulting to <Not given/empty>) -- The value of parameter. It is parsed according to the param's type using the default parser for the type VOTable tabledata.
- **description** (whitespace normalized unicode string; defaults to '') -- A short (one-line) description of the values in this column.
- **displayHint** (Display hint; defaults to '') -- Suggested presentation; the format is <kw>=<value>{,<kw>=<value>}, where what is interpreted depends on the output format. See, e.g., documentation on HTML renderers and the formatter child of outputFields.

- **fixup** (unicode string; defaults to None) -- A python expression the value of which will replace this column's value on database reads. Write a ____ to access the original value. You can use macros for the embedding table. This is for, e.g., simple URL generation (fixup="'internallink{/this/svc}'+____"). It will *only* kick in when tuples are deserialized from the database, i.e., *not* for values taken from tables in memory.
- **name** (A name for a table or service parameter. These have to match `[A-Za-z_][A-Za-z0-9_]*$`.; defaults to <Undefined>) -- Name of the param
- **note** (unicode string; defaults to None) -- Reference to a note meta on this table explaining more about this column
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **required** (boolean; defaults to 'False') -- Record becomes invalid when this column is NULL
- **tablehead** (unicode string; defaults to None) -- Terse phrase to put into table headers for this column
- **type** (a type name; the internal type system is similar to SQL's with some restrictions and extensions. The known atomic types include: unicode, pql-float, text, spoly, char, raw, vexpr-mjd, boolean, file, smallint, vexpr-string, scircle, vexpr-float, vexpr- date, pql-string, smoc, real, spoint, pql-int, timestamp, pql-date, int4range, date, integer, box, pql-upload, double precision, sbox, bigint, time, bytea; defaults to 'real') -- datatype for the column (SQL-like type system)
- **ucd** (unicode string; defaults to '') -- UCD of the column
- **unit** (unicode string; defaults to '') -- Unit of the values
- **utype** (unicode string; defaults to None) -- utype for this column
- **verbLevel** (integer; defaults to '20') -- Minimal verbosity level at which to include this column
- **xtype** (unicode string; defaults to None) -- VOTable xtype giving the serialization form; you usually do *not* want to set this, as the xtypes actually used are computed from database type. DaCHS xtypes are only used for a few unsavoury, hopefully temporary, hacks

Structure Children

- values (contains Element values) -- Specification of legal values

Other Children

- **dmRoles** (read-only list of roles played by this column in DMs; defaults to []) -- Roles played by this column; cannot be asigned to.
- **meta** -- a piece of meta information, giving at least a name and some content. See Metadata on what is permitted here.
- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.
- **stc** (non-settable internally used value; defaults to None) -- Internally used STC information for this column (do not assign to unless instructed to do so)
- **stcUtype** (non-settable internally used value; defaults to None) -- Internally used STC information for this column (do not assign to)

Element paramRef

A reference from a group to a parameter within a table.

ParamReferences do not support qualified references, i.e., you can only give simple names.

Also note that programmatically, you usually want to resolve ParamReferences within the Table instance, not the table definition.

May occur in Element group.

Atomic Children

- **key** (unicode string; defaults to <Undefined>) -- The key (i.e., name) of the referenced column or param.
- **ucd** (unicode string; defaults to None) -- The UCD of the group
- **utype** (unicode string; defaults to None) -- A utype for the group

Element phraseMaker

A procedure application for generating SQL expressions from input keys.

PhraseMaker code must *yield* SQL fragments that can occur in WHERE clauses, i.e., boolean expressions (thus, they must be generator bodies). The clauses yielded by a single condDesc are combined with the joiner set in the containing CondDesc (default=OR).

The following names are available to them:

- inputKeys -- the list of input keys for the parent CondDesc
- inPars -- a dictionary mapping inputKey names to the values provided by the user
- outPars -- a dictionary that is later used as the parameter dictionary to the query.
- core -- the core to which this phrase maker's condDesc belongs

To get the standard SQL a single key would generate, say:

```
yield base.getSQLForField(inputKeys[0], inPars, outPars)
```

To insert some value into outPars, do not simply use some key into outParse, since, e.g., the condDesc might be used multiple times. Instead, use getSQLKey, maybe like this:

```
ik = inputKeys[0]
yield "%s BETWEEN %%(%s)s AND %%(%s)s"%(ik.name,
  base.getSQLKey(ik.name, inPars[ik.name]-10, outPars),
  base.getSQLKey(ik.name, inPars[ik.name]+10, outPars))
```

getSQLKey will make sure unique names in outPars are chosen and enters the values there.

May occur in Element condDesc.

Atomic Children

- **code** (unicode string; defaults to <Not given/empty>) -- A python function body.
- **deprecated** (unicode string; defaults to None) -- A deprecation message. This will be shown if this procDef is being compiled.

- **doc** (unicode string; defaults to '') -- Human-readable docs for this proc (may be interpreted as restructured text).
- **name** (unicode string; defaults to <Not given/empty>) -- A name of the proc. ProcApps compute their (python) names to be somwhat random strings. Set a name manually to receive more easily decipherable error messages. If you do that, you have to care about name clashes yourself, though.
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **procDef** (id reference; defaults to <Not given/empty>) -- Reference to the procedure definition to apply
- **type** (One of: iterator, pargetter, regTest, rowfilter, dataFunction, descriptorGenerator, metaMaker, phraseMaker, mixinProc, dataFormatter, sourceFields, apply, t_t; defaults to None) -- The type of the procedure definition. The procedure applications will in general require certain types of definitions.

Structure Children

- bindings (contains Element bind and may be repeated zero or more times) -- Values for parameters of the procedure definition
- setups (contains Element setup and may be repeated zero or more times) -- Setup of the namespace the function will run in

Element procDef

An embedded procedure.

Embedded procedures are python code fragments with some interface defined by their type. They can occur at various places (which is called procedure application generically), e.g., as row generators in grammars, as applys in rowmakers, or as SQL phrase makers in condDescs.

They consist of the actual actual code and, optionally, definitions like the namespace setup, configuration parameters, or a documentation.

The procedure applications compile into python functions with special global namespaces. The signatures of the functions are determined by the type attribute.

ProcDefs are referred to by procedure applications using their id.

May occur in Element resource.

Atomic Children

- **code** (unicode string; defaults to <Not given/empty>) -- A python function body.
- **deprecated** (unicode string; defaults to None) -- A deprecation message. This will be shown if this procDef is being compiled.
- **doc** (unicode string; defaults to '') -- Human-readable docs for this proc (may be interpreted as restructured text).
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **type** (One of: iterator, pargetter, regTest, rowfilter, dataFunction, descriptorGenerator, metaMaker, phraseMaker, mixinProc, dataFormatter, sourceFields, apply, t_t; defaults to None) -- The type of the procedure definition. The procedure applications will in general require certain types of definitions.

Structure Children

- setups (contains Element setup and may be repeated zero or more times) -- Setup of the namespace the function will run in

Element processEarly

A code fragment run by the mixin machinery when the structure being worked on is being finished.

Within processEarly, you can access:

- Access the structure the mixin is applied to as "substrate"
- The mixin parameters as "mixinPars"
- The parse context as "context"

(the context is particularly handy for context.resolveId)

May occur in Element mixinDef.

Atomic Children

- **code** (unicode string; defaults to <Not given/empty>) -- A python function body.
- **deprecated** (unicode string; defaults to None) -- A deprecation message. This will be shown if this procDef is being compiled.
- **doc** (unicode string; defaults to '') -- Human-readable docs for this proc (may be interpreted as restructured text).
- **name** (unicode string; defaults to <Not given/empty>) -- A name of the proc. ProcApps compute their (python) names to be somwhat random strings. Set a name manually to receive more easily decipherable error messages. If you do that, you have to care about name clashes yourself, though.
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **procDef** (id reference; defaults to <Not given/empty>) -- Reference to the procedure definition to apply
- **type** (One of: iterator, pargetter, regTest, rowfilter, dataFunction, descriptorGenerator, metaMaker, phraseMaker, mixinProc, dataFormatter, sourceFields, apply, t_t; defaults to None) -- The type of the procedure definition. The procedure applications will in general require certain types of definitions.

Structure Children

- bindings (contains Element bind and may be repeated zero or more times) -- Values for parameters of the procedure definition
- setups (contains Element setup and may be repeated zero or more times) -- Setup of the namespace the function will run in

Element processLate

A code fragment run by the mixin machinery when the parser parsing everything exits.

Access the structure mixed in as "substrate", the root structure of the whole parse tree as root, and the context that is just about finishing as context.

May occur in Element mixinDef.

Atomic Children

- **code** (unicode string; defaults to <Not given/empty>) -- A python function body.
- **deprecated** (unicode string; defaults to None) -- A deprecation message. This will be shown if this procDef is being compiled.
- **doc** (unicode string; defaults to '') -- Human-readable docs for this proc (may be interpreted as restructured text).
- **name** (unicode string; defaults to <Not given/empty>) -- A name of the proc. ProcApps compute their (python) names to be somwhat random strings. Set a name manually to receive more easily decipherable error messages. If you do that, you have to care about name clashes yourself, though.
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **procDef** (id reference; defaults to <Not given/empty>) -- Reference to the procedure definition to apply
- **type** (One of: iterator, pargetter, regTest, rowfilter, dataFunction, descriptorGenerator, metaMaker, phraseMaker, mixinProc, dataFormatter, sourceFields, apply, t_t; defaults to None) -- The type of the procedure definition. The procedure applications will in general require certain types of definitions.

Structure Children

- bindings (contains Element bind and may be repeated zero or more times) -- Values for parameters of the procedure definition
- setups (contains Element setup and may be repeated zero or more times) -- Setup of the namespace the function will run in

Element PRUNE

An active tag that lets you selectively delete children of the current object.

You give it regular expression-valued attributes; on the replay of the stream, matching items and their children will not be replayed.

If you give more than one attribute, the result will be a conjunction of the specified conditions.

This only works if the items to be matched are true XML attributes (i.e., not written as children).

May occur in Element LFEED, Element LOOP, Element mixinDef, Element FEED.

Element publish (data)

A request for registration of a data or table item.

This is much like publish for services, just for data and tables; since they have no renderers, you can only have one register element per such element.

Data registrations may refer to published services that make their data available.

May occur in Element outputTable, Element table, Element data.

Atomic Children

- **services** (list of id references (comma separated or in distinct elements); defaults to []) -- A DC-internal reference to a service that lets users query that within the data collection; tables with adql=True are automatically declared to be servedBy the TAP service.
- **sets** (Comma-separated list of strings; defaults to 'ivo_managed') -- A comma-separated list of sets this data will be published in. To publish data to the VO registry, just say ivo_managed here. Other sets probably don't make much sense right now. ivo_managed also is the default.

Other Children

- **meta** -- a piece of meta information, giving at least a name and some content. See Metadata on what is permitted here.

Element publish

A specification of how a service should be published.

This contains most of the metadata for what is an interface in registry speak.

May occur in Element service, Element resRec.

Atomic Children

- **auxiliary** (boolean; defaults to 'False') -- Auxiliary publications are for capabilities not intended to be picked up for all-VO queries, typically because they are already registered with other services. This is mostly used internally; you probably have no reason to touch it.
- **render** (unicode string; defaults to <Undefined>) -- The renderer the publication will point at.
- **service** (id reference; defaults to <Not given/empty>) -- Reference for a service actually implementing the capability corresponding to this publication. This is mainly when there is a vs:WebBrowser service accompanying a VO protocol service, and this other service should be published in the same resource record. See also the operator's guide.
- **sets** (Comma-separated list of strings; defaults to '') -- Comma- separated list of sets this service will be published in. Predefined are: local=publish on front page, ivo_managed=register with the VO registry. If you leave it empty, 'local' publication is assumed.

Other Children

- **meta** -- a piece of meta information, giving at least a name and some content. See Metadata on what is permitted here.

Element regSuite

A suite of regression tests.

May occur in Element resource.

Atomic Children

- **sequential** (boolean; defaults to 'False') -- Set to true if the individual tests need to be run in sequence.
- **title** (whitespace normalized unicode string; defaults to None) -- A short, human-readable phrase describing what this suite is about.

Structure Children

- tests (contains Element regTest and may be repeated zero or more times) -- Tests making up this suite

Element regTest

A regression test.

May occur in Element regSuite.

Atomic Children

- **code** (unicode string; defaults to <Not given/empty>) -- A python function body.
- **deprecated** (unicode string; defaults to None) -- A deprecation message. This will be shown if this procDef is being compiled.
- **doc** (unicode string; defaults to '') -- Human-readable docs for this proc (may be interpreted as restructured text).
- **name** (unicode string; defaults to <Not given/empty>) -- A name of the proc. ProcApps compute their (python) names to be somwhat random strings. Set a name manually to receive more easily decipherable error messages. If you do that, you have to care about name clashes yourself, though.
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **procDef** (id reference; defaults to <Not given/empty>) -- Reference to the procedure definition to apply
- **tags** (Comma-separated list of strings; defaults to '') -- A list of (free-form) tags for this test. Tagged tests are only run when the runner is constructed with at least one of the tags given. This is mainly for restricting tags to production or development servers.
- **title** (whitespace normalized unicode string; defaults to <Undefined>) -- A short, human-readable phrase describing what this test is exercising.
- **type** (One of: iterator, pargetter, regTest, rowfilter, dataFunction, descriptorGenerator, metaMaker, phraseMaker, mixinProc, dataFormatter, sourceFields, apply, t_t; defaults to None) -- The type of the procedure definition. The procedure applications will in general require certain types of definitions.

Structure Children

- bindings (contains Element bind and may be repeated zero or more times) -- Values for parameters of the procedure definition
- setups (contains Element setup and may be repeated zero or more times) -- Setup of the namespace the function will run in
- url (contains Element url) -- The source from which to fetch the test data.

Element resource

A resource descriptor.

RDs collect all information about how to parse a particular source (like a collection of FITS images, a catalogue, or whatever), about the database tables the data ends up in, and the services used to access them.

In DaCHS' RD XML serialisation, they correspond to the root element.

Atomic Children

- **allProfiles** (Comma separated list of profile names.; defaults to u'admin, msdemlei') -- A (comma separated) list of profile names through which the object can be written or administred (oh, and the default is not admin, msdemlei but is the value of [db]maintainers)
- **readProfiles** (Comma separated list of profile names.; defaults to u'trustedquery') -- A (comma separated) list of profile names through which the object can be read.
- **require** (unicode string; defaults to None) -- Import the named gavo module (for when you need something registred)
- **resdir** (unicode string; defaults to None) -- Base directory for source files and everything else belonging to the resource.
- **schema** (unicode string; defaults to <Undefined>) -- Database schema for tables defined here. Follow the rule 'one schema, one RD' if at all possible. If two RDs share the same schema, the must generate exactly the same permissions for that schema; this means, in particular, that if one has an ADQL-published table, so must the other. In a nutshell: one schema, one RD.

Structure Children

- condDescs (contains Element condDesc and may be repeated zero or more times) -- Global condition descriptors for later reference
- cores (contains any of siapCutoutCore,scsCore,pythonCore,registryCore,dbCore,fancyQueryCore,fixedQueryCore,adqlCore,debugCore,datalinkCore,uploadCore,productCore,tapCore,customCore,ssapCore,nullCore and may be repeated zero or more times) -- Cores available in this resource.
- coverage (contains Element coverage) -- STC coverage of this resource.
- dds (contains Element data and may be repeated zero or more times) -- Descriptors for the data generated and/or published within this resource.
- jobs (contains Element execute and may be repeated zero or more times) -- Jobs to be run while this RD is active.
- macDefs (contains Element macDef and may be repeated zero or more times) -- User-defined macros available on this RD
- mixdefs (contains Element mixinDef and may be repeated zero or more times) -- Mixin definitions (usually not for users)
- outputTables (contains Element outputTable and may be repeated zero or more times) -- Canned output tables for later reference.
- procDefs (contains Element procDef and may be repeated zero or more times) -- Procedure definintions (rowgens, rowmaker applys)
- resRecs (contains Element resRec and may be repeated zero or more times) -- Non-service resources for the IVOA registry. They will be published when gavo publish is run on the RD.
- rowmakers (contains Element rowmaker and may be repeated zero or more times) -- Transformations for going from grammars to tables. If specified in the RD, they must be referenced from make elements to become active.
- scripts (contains Element script and may be repeated zero or more times) -- Code snippets attached to this object. See Scripting .
- services (contains Element service and may be repeated zero or more times) -- Services exposing data from this resource.
- tables (contains Element table and may be repeated zero or more times) -- A table used or created by this resource
- tests (contains Element regSuite and may be repeated zero or more times) -- Suites of regression tests connected to this RD.

Other Children

- **meta** -- a piece of meta information, giving at least a name and some content. See Metadata on what is permitted here.
- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Macros predefined here: Macro RSTcc0, Macro RSTccby, Macro RSTccbysa, Macro RSTservicelink, Macro RSTtable, Macro decapitalize, Macro getConfig, Macro internallink, Macro magicEmpty, Macro metaString, Macro quote, Macro rdId, Macro rdIdDotted, Macro schema, Macro sqlquote, Macro test, Macro today, Macro upper, Macro urlquote

Element resRec

A resource for pure registration purposes.

A Resource without DaCHS defined behaviour. This can be Organizations or Instruments, but possibly also external services

All resources must either have an id (which is used in the construction of their IVOID), or you must give an identifier meta item.

You must further set the following meta items:

- resType specifying the kind of resource record. You should not use this element to build resource records for services or tables (use the normal elements, even if the actual resrouces are external to DaCHS). resType can be registry, organization, authority, deleted, or anything else for which registry.builders has a handling class.
- title
- subject(s)
- description
- referenceURL
- creationDate

Additional meta keys (e.g., accessURL for a registry) may be required depending on resType. See the registry section in the operator's guide.

ResRecs can also have publication children. These will be turned into the appropriate capabilities depending on the value of the render attribute.

May occur in Element resource.

Structure Children

- publications (contains Element publish and may be repeated zero or more times) -- Capabilities the record should have (this is empty for standards, organisations, instruments, etc.)

Other Children

- **meta** -- a piece of meta information, giving at least a name and some content. See Metadata on what is permitted here.

Macros predefined here: Macro RSTservicelink, Macro RSTtable, Macro decapitalize, Macro getConfig, Macro internallink, Macro magicEmpty, Macro metaString, Macro quote, Macro rdId, Macro rdIdDotted, Macro schema, Macro sqlquote, Macro test, Macro today, Macro upper, Macro urlquote

Element rowmaker

A definition of the mapping between grammar input and finished rows ready for shipout.

Rowmakers consist of variables, procedures and mappings. They result in a python callable doing the mapping.

RowmakerDefs double as macro packages for the expansion of various macros. The standard macros will need to be quoted, the rowmaker macros above yield python expressions.

Within map and var bodies as well as late apply pars and apply bodies, you can refer to the grammar input as vars["name"] or, shorter @name.

To add output keys, use map or, in apply bodies, add keys to the result dictionary.

May occur in Element resource, Element data.

Atomic Children

- **idmaps** (Comma-separated list of strings; defaults to '') -- List of column names that are just "mapped through" (like map with key only); you can use shell patterns to select multiple colums at once.
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.

- **simplemaps** (Comma-separated list of <identifer>:<identifier> pairs; defaults to None) -- Abbreviated notation for <map source>; each pair is destination:source

Structure Children

- apps (contains Element apply and may be repeated zero or more times) -- Procedure applications.
- ignoreOn (contains Element ignoreOn) -- Conditions on the input record coming from the grammar to cause the input record to be dropped by the rowmaker, i.e., for this specific table. If you need to drop a row for all tables being fed, use a trigger on the grammar.
- maps (contains Element map and may be repeated zero or more times) -- Mapping rules.
- vars (contains Element var and may be repeated zero or more times) -- Definitions of intermediate variables.

Macros predefined here: Macro RSTservicelink, Macro RSTtable, Macro decapitalize, Macro dlMetaURI, Macro docField, Macro fullPath, Macro getConfig, Macro inputRelativePath, Macro inputSize, Macro internallink, Macro lastSourceElements, Macro magicEmpty, Macro metaString, Macro property, Macro qName, Macro quote, Macro rdId, Macro rdIdDotted, Macro rootlessPath, Macro rowsMade, Macro rowsProcessed, Macro schema, Macro sourceCDate, Macro sourceDate, Macro sqlquote, Macro srcstem, Macro standardPubDID, Macro test, Macro today, Macro upper, Macro urlquote

Element script

A script, i.e., some executable item within a resource descriptor.

The content of scripts is given by their type -- usually, they are either python scripts or SQL with special rules for breaking the script into individual statements (which are basically like python's).

The special language AC_SQL is like SQL, but execution errors are ignored. This is not what you want for most data RDs (it's intended for housekeeping scripts).

See Scripting.

May occur in Element resource, Element make.

Atomic Children

- Character content of the element (defaulting to '') -- The script body.
- **lang** (One of: python, AC_SQL, SQL; defaults to <Undefined>) -- Language of the script.
- **name** (unicode string; defaults to 'anonymous') -- A human- consumable designation of the script.
- **notify** (boolean; defaults to 'True') -- Send out a notification when running this script.
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **type** (One of: postCreation, newSource, beforeDrop, sourceDone, preCreation, preImport, preIndex; defaults to <Undefined>) -- Point of time at which script is to run.

Element service

A service definition.

A service is a combination of a core and one or more renderers. They can be published, and they carry the metadata published into the VO.

You can set the defaultSort property on the service to a name of an output column to preselect a sort order. Note again that this will slow down responses for all but the smallest tables unless there is an index on the corresponding column.

Properties evaluated:

- defaultSort -- a key to sort on by default with the form renderer. This differs from the dbCore's sortKey in that this does not suppress the widget itself, it just sets a default for its value. Don't use this unless you have to; the combination of sort and limit can have disastrous effects on the run time of queries.
- votableRespectsOutputTable -- usually, VOTable output puts in all columns from the underlying database table with low enough verbLevel (essentially). When this property is "True" (case-sensitive), that's not done and only the service's output table is evaluated.

May occur in Element resource.

Atomic Children

- **allowed** (Comma-separated list of strings; defaults to '') -- Names of renderers allowed on this service; leave emtpy to allow the form renderer only.
- **core** (id reference; defaults to <Undefined>) -- The core that does the computations for this service. Instead of a reference, you can use an immediate element of some registred core.
- **customPage** (unicode string; defaults to None) -- resdir-relative path to custom page code. It is used by the 'custom' renderer
- **defaultRenderer** (unicode string; defaults to None) -- A name of a renderer used when none is provided in the URL (lets you have shorter URLs).
- **limitTo** (unicode string; defaults to None) -- Limit access to the group given; the empty default disables access control.
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.

Structure Children

- customDFs (contains Element customDF and may be repeated zero or more times) -- Custom data functions for use in custom templates.
- customRFs (contains Element customRF and may be repeated zero or more times) -- Custom render functions for use in custom templates.
- outputTable (contains Element outputTable) -- The output fields of this service.
- publications (contains Element publish and may be repeated zero or more times) -- Sets and renderers this service is published with.
- serviceKeys (contains Element inputKey and may be repeated zero or more times) -- Input widgets for processing by the service, e.g. output sets.

Other Children

- **meta** -- a piece of meta information, giving at least a name and some content. See Metadata on what is permitted here.

- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.
- **template** (mapping; the value is the element content, the key is in the 'key' (or, equivalently, key) attribute) -- Custom nevow templates for this service; use key "form" to replace the Form renderer's standard template. Start the path with two slashes to access system templates.

Macros predefined here: Macro RSTservicelink, Macro RSTtable, Macro decapitalize, Macro getConfig, Macro internallink, Macro magicEmpty, Macro metaString, Macro quote, Macro rdId, Macro rdIdDotted, Macro schema, Macro sqlquote, Macro tablesForTAP, Macro test, Macro today, Macro upper, Macro urlquote

Element setup

Prescriptions for setting up a namespace for a procedure application.

You can add names to this namespace you using par(ameter)s. If a parameter has no default and an procedure application does not provide them, an error is raised.

You can also add names by providing a code attribute containing a python function body in code. Within, the parameters are available. The procedure application's parent can be accessed as parent. All names you define in the code are available as globals to the procedure body.

Caution: Macros are expanded within the code; this means you need double backslashes if you want a single backslash in python code.

May occur in Element iterator, Element rowfilter, Element apply, Element procDef, Element job, Element processLate, Element dataFormatter, Element regTest, Element coreProc, Element dataFunction, Element sourceFields, Element metaMaker, Element phraseMaker, Element descriptorGenerator, Element processEarly, Element pargetter.

Atomic Children

- **codeFrags** (Zero or more unicode string-typed *code* elements; defaults to u'[]') -- Python function bodies setting globals for the function application. Macros are expanded in the context of the procedure's parent.
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.

Structure Children

- pars (contains Element par and may be repeated zero or more times) -- Names to add to the procedure's global namespace.

Element sources

A Specification of a data descriptor's inputs.

This will typcially be files taken from a file system. If so, DaCHS will, in each directory, process the files in alphabetical order. No guarantees are made as to the sequence directories are processed in.

Multiple patterns are processed in the order given in the RD.

May occur in Element data.

Atomic Children

- Character content of the element (defaulting to '') -- A single file name (this is for convenience)
- **items** (Zero or more unicode string-typed *item* elements; defaults to u'[]') -- String literals to pass to grammars. In contrast to patterns, they are not interpreted as file names but passed to the grammar verbatim. Normal grammars do not like this. It is mainly intended for use with custom or null grammars.
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **patterns** (Zero or more unicode string-typed *pattern* elements; defaults to u'[]') -- Paths to the source files. You can use shell patterns here.
- **recurse** (boolean; defaults to 'False') -- Search for pattern(s) recursively in their directory part(s)?

Structure Children

- ignoredSources (contains Element ignoreSources) -- Specification of sources that should not be processed although they match patterns. Typically used in update-type data descriptors.

Element stc

A definition of a space-time coordinate system using STC-S.

May occur in Element outputTable, Element table.

Atomic Children

- Character content of the element (defaulting to '') -- An STC-S string with column references (using quote syntax) instead of values

Element table

A definition of a table, both on-disk and internal.

Some attributes are ignored for in-memory tables, e.g., roles or adql.

Properties for tables:

- supportsModel -- a short name of a data model supported through this table (for TAPRegExt dataModel); you can give multiple names separated by commas.
- supportsModelURI -- a URI of a data model supported through this table. You can give multiple URIs separated by blanks.

If you give multiple data model names or URIs, the sequences of names and URIs must be identical (in particular, each name needs a URI).

May occur in Element resource, Element data.

Atomic Children

- **adql** (boolean or 'hidden'; defaults to 'False') -- Should this table be available for ADQL queries? In addition to True/False, this can also be 'hidden' for tables readable from the TAP machinery but not published in the metadata; this is useful for, e.g., tables contributing to a published view. Warning: adql=hidden is incompatible with setting readProfiles manually.
- **allProfiles** (Comma separated list of profile names.; defaults to u'admin, msdemlei') -- A (comma separated) list of profile names through which the object can be written or administred (oh, and the default is not admin, msdemlei but is the value of [db]maintainers)

- **dupePolicy** (One of: drop, check, overwrite, dropOld; defaults to 'check') -- Handle duplicate rows with identical primary keys manually by raising an error if existing and new rows are not identical (check), dropping the new one (drop), updating the old one (overwrite), or dropping the old one and inserting the new one (dropOld)?
- **forceUnique** (boolean; defaults to 'False') -- Enforce dupe policy for primary key (see dupePolicy)?
- A mixin reference, typically to support certain protocol. See Mixins.
- **namePath** (id reference; defaults to None) -- Reference to an element tried to satisfy requests for names in id references of this element's children.
- **nrows** (integer; defaults to None) -- Approximate number of rows in this table (usually, you want to use dachs limits to fill this out; write <nrows>0</nrows> to enable that).
- **onDisk** (boolean; defaults to 'False') -- Table in the database rather than in memory?
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **primary** (Comma-separated list of strings; defaults to '') -- Comma separated names of columns making up the primary key.
- **readProfiles** (Comma separated list of profile names.; defaults to u'trustedquery') -- A (comma separated) list of profile names through which the object can be read.
- **system** (boolean; defaults to 'False') -- Is this a system table? If it is, it will not be dropped on normal imports, and accesses to it will not be logged.
- **temporary** (boolean; defaults to 'False') -- If this is an onDisk table, make it temporary? This is mostly useful for custom cores and such.
- **viewStatement** (unicode string; defaults to None) -- A single SQL statement to create a view. Setting this makes this table a view. The statement will typically be something like CREATE VIEW \curtable AS (SELECT \colNames FROM...).

Structure Children

- columns (contains Element column and may be repeated zero or more times) -- Columns making up this table.
- dm (contains Element dm and may be repeated zero or more times) -- Annotations for data models.
- foreignKeys (contains Element foreignKey and may be repeated zero or more times) -- Foreign keys used in this table
- groups (contains Element group and may be repeated zero or more times) -- Groups for columns and params of this table
- indices (contains Element index and may be repeated zero or more times) -- Indices defined on this table
- params (contains Element param and may be repeated zero or more times) -- Param ("global columns") for this table.
- registration (contains Element publish (data)) -- A registration (to the VO registry) of this table or data collection.
- stc (contains Element stc and may be repeated zero or more times) -- STC-S definitions of coordinate systems.

Other Children

- **meta** -- a piece of meta information, giving at least a name and some content. See Metadata on what is permitted here.
- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Macros predefined here: Macro RSTservicelink, Macro RSTtable, Macro colNames, Macro curtable, Macro decapitalize, Macro getConfig, Macro getParam, Macro internallink, Macro magicEmpty, Macro metaString, Macro nameForUCD, Macro nameForUCDs, Macro qName, Macro quote, Macro rdId, Macro rdIdDotted, Macro schema, Macro sqlquote, Macro tablename, Macro test, Macro today, Macro upper, Macro urlquote

Element updater

Information on where and how to update a piece of coverage information.

May occur in Element coverage.

Atomic Children

- **mocOrder** (integer; defaults to '6') -- Maximal HEALpix order to use in coverage MOCs (6 is about a degree resolution, each additional point doubles resolution).
- **sourceTable** (id reference; defaults to <Not given/empty>) -- A table from which to compute coverage by default.
- **spaceTable** (id reference; defaults to <Not given/empty>) -- A table from which to compute spatial coverage (overrides sourceTable).
- **spectralTable** (id reference; defaults to <Not given/empty>) -- A table from which to compute spectral coverage (overrides sourceTable)
- **timeTable** (id reference; defaults to <Not given/empty>) -- A table from which to compute temporal coverage (overrides sourceTable)

Element url

A source document for a regression test.

As string URLs, they specify where to get data from, but the additionally let you specify uploads, authentication, headers and http methods, while at the same time saving you manual escaping of parameters.

The bodies is the path to run the test against. This is interpreted as relative to the RD if there's no leading slash, relative to the server if there's a leading slash, and absolute if there's a scheme.

The attributes are translated to parameters, except for a few pre-defined names. If you actually need those as URL parameters, should at us and we'll provide some way of escaping these.

We don't actually parse the URLs coming in here. GET parameters are appended with a & if there's a ? in the existing URL, with a ? if not. Again, shout if this is too dumb for you (but urlparse really isn't all that robust either...)

May occur in Element regTest.

Atomic Children

- Character content of the element (defaulting to '') -- Base for URL generation; embedded whitespace will be removed, so you're free to break those whereever you like.

- **httpAuthKey** (unicode string; defaults to <Not given/empty>) -- A key into ~/.gavo/test.creds to find a user/password pair for this request.
- **httpHonorRedirects** (boolean; defaults to 'False') -- Follow 30x redirects instead of just using status, headers, and payload of the initial request.
- **httpMethod** (unicode string; defaults to 'GET') -- Request method; usually one of GET or POST
- **parSet** (One of: TAP, form; defaults to <Not given/empty>) -- Preselect a default parameter set; form gives what our framework adds to form queries.
- **postPayload** (unicode string; defaults to <Not given/empty>) -- Path to a file containing material that should go with a POST request (conflicts with additional parameters).

Structure Children

- uploads (contains Element httpUpload and may be repeated zero or more times) -- HTTP uploads to add to request (must have httpMethod="POST")

Other Children

- **value** (mapping; the value is the element content, the key is in the 'key' (or, equivalently, key) attribute) -- Additional HTTP headers to pass.
- (ignore)

Element values

Information on a column's values, in particular its domain.

This is quite like the values element in a VOTable. In particular, to accomodate VOTable usage, we require nullLiteral to be a valid literal for the parent's type.

Note that DaCHS does not validate for contraints from values on table import. This is mainly because before `gavo values` has run, values may not represent the new dataset in semiautomatic values.

With HTTP parameters, values validation does take place (but again, that's mostly not too helpful because there are query languages sitting in between most of the time).

Hence, the main utility of values is metadata declaration, both in the form render (where they become placeholders) and in datalink (where they are communicated as VOTable values).

May occur in Element param, Element column, Element inputKey, Element outputField.

Atomic Children

- **caseless** (boolean; defaults to 'False') -- When validating, ignore the case of string values. For non-string types, behaviour is undefined (i.e., DaCHS is going to spit on you).
- **default** (unicode string; defaults to None) -- A default value (currently only used for options).
- **fromdb** (unicode string; defaults to None) -- A query fragment returning just one column to fill options from (will add to options if some are given). Do not write SELECT or anything, just the column name and the where clause.
- **max** (unicode string; defaults to None) -- Maximum acceptable value as a datatype literal
- **min** (unicode string; defaults to None) -- Minimum acceptable value as a datatype literal
- **multiOk** (boolean; defaults to 'False') -- Deprecated, use multiplicity=multiple on input keys instead.
- **nullLiteral** (unicode string; defaults to None) -- An appropriate value representing a NULL for this column in VOTables and similar places. You usually should only set it for integer types and chars. Note that rowmakers make no use of this nullLiteral, i.e., you can and should choose null values independently of your source. Again, for reals, floats and (mostly) text you probably do not want to do this.
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.

Structure Children

- options (contains Element option and may be repeated zero or more times) -- List of acceptable values (if set)

Element var

A definition of a rowmaker variable.

It consists of a name and a python expression, including function calls. The variables are entered into the input row coming from the grammar.

var elements are evaluated before apply elements, in the sequence they are in the RD. You can refer to keys defined by vars already evaluated in the usual @key manner.

May occur in Element rowmaker.

Atomic Children

- Character content of the element (defaulting to '') -- A python expression giving the value for key.
- **key** (unicode string; defaults to <Undefined>) -- Name of the column the value is to end up in.
- **nullExcs** (unicode string; defaults to <Not given/empty>) -- Exceptions that should be caught and cause the value to be NULL, separated by commas.
- **nullExpr** (unicode string; defaults to <Not given/empty>) -- A python expression for a value that is mapped to NULL (None). Equality is checked after building the value, so this expression has to be of the column type. Use map with the parseWithNull function to catch null values before type conversion.
- **source** (unicode string; defaults to None) -- Source key name to convert to column value (either a grammar key or a var).

Active Tags

The following tags are "active", which means that they do not directly contribute to the RD parsed. Instead they define, replay, or edit streams of elements.

Element FEED

An active tag that takes an event stream and replays the events, possibly filling variables.

This element supports arbitrary attributes with unicode values. These values are available as macros for replayed values.

Atomic Children

- **reexpand** (boolean; defaults to 'False') -- Force re-expansion of macros; usually, when replaying, each string is only expanded once, mainly to avoid overly long backslash-fences. Set this to true to force further expansion.
- **source** (id reference; defaults to None) -- id of a stream to replay

Structure Children

- edits (contains Element EDIT and may be repeated zero or more times) -- Changes to be performed on the events played back.
- events (contains Element events) -- Alternatively to source, an XML fragment to be replayed
- prunes (contains Element PRUNE and may be repeated zero or more times) -- Conditions for removing items from the playback stream.

Macros predefined here: Macro RSTservicelink, Macro RSTtable, Macro decapitalize, Macro getConfig, Macro internallink, Macro magicEmpty, Macro metaString, Macro quote, Macro rdId, Macro rdIdDotted, Macro schema, Macro sqlquote, Macro test, Macro today, Macro upper, Macro urlquote

Element LFEED

A ReplayedEventStream that does not expand active tag macros.

You only want this when embedding a stream into another stream that could want to expand the embedded macros.

Atomic Children

- **reexpand** (boolean; defaults to 'False') -- Force re-expansion of macros; usually, when replaying, each string is only expanded once, mainly to avoid overly long backslash-fences. Set this to true to force further expansion.
- **source** (id reference; defaults to None) -- id of a stream to replay

Structure Children

- edits (contains Element EDIT and may be repeated zero or more times) -- Changes to be performed on the events played back.
- events (contains Element events) -- Alternatively to source, an XML fragment to be replayed
- prunes (contains Element PRUNE and may be repeated zero or more times) -- Conditions for removing items from the playback stream.

Macros predefined here: Macro RSTservicelink, Macro RSTtable, Macro decapitalize, Macro getConfig, Macro internallink, Macro magicEmpty, Macro metaString, Macro quote, Macro rdId, Macro rdIdDotted, Macro schema, Macro sqlquote, Macro test, Macro today, Macro upper, Macro urlquote

Element LOOP

An active tag that replays a feed several times, each time with different values.

Atomic Children

- **codeItems** (unicode string; defaults to None) -- A python generator body that yields dictionaries that are then used as loop items. You can access the parse context as the context variable in these code snippets.
- **csvItems** (unicode string; defaults to None) -- The items to loop over, in CSV-with-labels format.
- **listItems** (unicode string; defaults to None) -- The items to loop over, as space-separated single items. Each item will show up once, as 'item' macro.
- **reexpand** (boolean; defaults to 'False') -- Force re-expansion of macros; usually, when replaying, each string is only expanded once, mainly to avoid overly long backslash-fences. Set this to true to force further expansion.
- **source** (id reference; defaults to None) -- id of a stream to replay

Structure Children

- edits (contains Element EDIT and may be repeated zero or more times) -- Changes to be performed on the events played back.
- events (contains Element events) -- Alternatively to source, an XML fragment to be replayed

- prunes (contains Element PRUNE and may be repeated zero or more times) -- Conditions for removing items from the playback stream.

Macros predefined here: Macro RSTservicelink, Macro RSTtable, Macro decapitalize, Macro getConfig, Macro internallink, Macro magicEmpty, Macro metaString, Macro quote, Macro rdId, Macro rdIdDotted, Macro schema, Macro sqlquote, Macro test, Macro today, Macro upper, Macro urlquote

Element NXSTREAM

An event stream that records events, not expanding active tags.

Normal event streams expand embedded active tags in place. This is frequently what you want, but it means that you cannot, e.g., fill in loop variables through stream macros.

With non-expanded streams, you can do that:

```
<NXSTREAM id="cols">
  <LOOP listItems="\stuff">
    <events>
      <column name="\item"/>
    </events>
  </LOOP>
</NXSTREAM>
<table id="foo">
  <FEED source="cols" stuff="x y"/>
</table>
```

Note that the normal innermost-only rule for macro expansions within active tags does not apply for NXSTREAMS. Macros expanded by a replayed NXSTREAM will be re-expanded by the next active tag that sees them (this is allow embedded active tags to use macros; you need to double-escape macros for them, of course).

Atomic Children

- **doc** (unicode string; defaults to None) -- A description of this stream (should be restructured text).

Structure Children

- DEFAULTS (contains Element DEFAULTS) -- A mapping giving defaults for macros expanded in this stream. Macros not defaulted will fail when not given in a FEED's attributes.

Element STREAM

An active tag that records events as they come in.

Their only direct effect is to leave a trace in the parser's id map. The resulting event stream can be played back later.

Atomic Children

- **doc** (unicode string; defaults to None) -- A description of this stream (should be restructured text).

Structure Children

- DEFAULTS (contains Element DEFAULTS) -- A mapping giving defaults for macros expanded in this stream. Macros not defaulted will fail when not given in a FEED's attributes.

Grammars Available

The following elements are all grammar related. All grammar elements can occur in data descriptors.

Element binaryGrammar

A grammar that builds rowdicts from binary data.

The grammar expects the input to be in fixed-length records. the actual specification of the fields is done via a binaryRecordDef element.

Atomic Children

- **armor** (One of: fortran; defaults to None) -- Record armoring; by default it's None meaning the data was dumped to the file sequentially. Set it to fortran for fortran unformatted files (4 byte length before and after the payload).
- **enc** (unicode string; defaults to None) -- Encoding of strings coming in from source.
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **skipBytes** (integer; defaults to '0') -- Number of bytes to skip before parsing records.

Structure Children

- fieldDefs (contains Element binaryRecordDef) -- Definition of the record.
- ignoreOn (contains Element ignoreOn) -- Conditions for ignoring certain input records. These triggers drop an input record entirely. If you feed multiple tables and just want to drop a row from a specific table, you can use ignoreOn in a rowmaker.
- rowfilters (contains Element rowfilter and may be repeated zero or more times) -- Row filters for this grammar.
- sourceFields (contains Element sourceFields) -- Code returning a dictionary of values added to all returned rows.

Other Children

- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Macros predefined here: Macro RSTservicelink, Macro RSTtable, Macro colNames, Macro decapitalize, Macro dlMetaURI, Macro fullDLURL, Macro getConfig, Macro inputRelativePath, Macro inputSize, Macro internallink, Macro lastSourceElements, Macro magicEmpty, Macro metaString, Macro property, Macro quote, Macro rdId, Macro rdIdDotted, Macro rootlessPath, Macro rowsProcessed, Macro schema, Macro sourceDate, Macro splitPreviewPath, Macro sqlquote, Macro srcstem, Macro standardPreviewPath, Macro test, Macro today, Macro upper, Macro urlquote

Element binaryRecordDef

A definition of a binary record.

A binary records consists of a number of binary fields, each of which is defined by a name and a format code. The format codes supported here are a subset of what python's struct module supports. The widths given below are for big, little, and packed binfmts. For native (which is the default), it depends on your platform.

- <number>s -- <number> characters making up a string
- b,B -- signed and unsigned byte (8 bit)
- h,H -- signed and unsigned short (16 bit)

- i,I -- signed and unsigned int (32 bit)
- q,Q -- signed and unsigned long (64 bit)
- f,d -- float and double.

The content of this element gives the record structure in the format <name>(<code>){<whitespace><name>(<code>)} where <name> is a c-style identifier.

May occur in Element binaryGrammar.

Atomic Children

- **binfmt** (One of: big, little, packed, native; defaults to 'native') -- Binary format of the input data; big and little stand for msb first and lsb first, and packed is like native except no alignment takes place.
- Character content of the element (defaulting to '') -- The enumeration of the record fields.

Element cdfHeaderGrammar

A grammar that returns the header dictionary of a CDF file (global attributes).

This grammar yields a single dictionary per file, which corresponds to the global attributes. The values in this dictionary may have complex structure; in particular, sequences are returned as lists.

To use this grammar, additional software is required that (by 2014) is not packaged for Debian. See https://pythonhosted.org/SpacePy/install.html for installation instructions. Note that you must install the CDF library itself as described further down on that page; the default installation instructions do not install the library in a public place, so if you use these, you'll have to set CDF_LIB to the right value, too, before running dachs imp.

Atomic Children

- **autoAtomize** (boolean; defaults to 'False') -- Unpack 1-element lists to their first value.
- **enc** (unicode string; defaults to None) -- Encoding of strings coming in from source.
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.

Structure Children

- ignoreOn (contains Element ignoreOn) -- Conditions for ignoring certain input records. These triggers drop an input record entirely. If you feed multiple tables and just want to drop a row from a specific table, you can use ignoreOn in a rowmaker.
- mapKeys (contains Element mapKeys) -- Prescription for how to map labels keys to grammar dictionary keys
- rowfilters (contains Element rowfilter and may be repeated zero or more times) -- Row filters for this grammar.
- sourceFields (contains Element sourceFields) -- Code returning a dictionary of values added to all returned rows.

Other Children

- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Macros predefined here: Macro RSTservicelink, Macro RSTtable, Macro colNames, Macro decapitalize, Macro dlMetaURI, Macro fullDLURL, Macro getConfig, Macro inputRelativePath, Macro inputSize, Macro internallink, Macro lastSourceElements, Macro magicEmpty, Macro metaString, Macro property, Macro quote, Macro rdId, Macro rdIdDotted, Macro rootlessPath, Macro rowsProcessed, Macro schema, Macro sourceDate, Macro splitPreviewPath, Macro sqlquote, Macro srcstem, Macro standardPreviewPath, Macro test, Macro today, Macro upper, Macro urlquote

Element columnGrammar

A grammar that builds rowdicts out of character index ranges.

This works by using the colRanges attribute like <col key="mag">12-16</col>, which will take the characters 12 through 16 inclusive from each input line to build the input column mag.

As a shortcut, you can also use the colDefs attribute; it contains a string of the form {<key>:<range>}, i.e., a whitespace-separated list of colon-separated items of key and range as accepted by cols, e.g.:

```
<colDefs>
  a: 3-4
  _u: 7
</colDefs>
```

Atomic Children

- **colDefs** (unicode string; defaults to None) -- Shortcut way of defining cols
- **commentIntroducer** (unicode string; defaults to <Not given/empty>) -- A character sequence that, when found at the beginning of a line makes this line ignored
- **enc** (unicode string; defaults to None) -- Encoding of strings coming in from source.
- **gunzip** (boolean; defaults to 'False') -- Unzip sources while reading? (Deprecated, use preFilter='zcat')
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **preFilter** (unicode string; defaults to None) -- Shell command to pipe the input through before passing it on to the grammar. Classical examples include zcat or bzcat, but you can commit arbitrary shell atrocities here.
- **topIgnoredLines** (integer; defaults to '0') -- Skip this many lines at the top of each source file.

Structure Children

- ignoreOn (contains Element ignoreOn) -- Conditions for ignoring certain input records. These triggers drop an input record entirely. If you feed multiple tables and just want to drop a row from a specific table, you can use ignoreOn in a rowmaker.
- rowfilters (contains Element rowfilter and may be repeated zero or more times) -- Row filters for this grammar.
- sourceFields (contains Element sourceFields) -- Code returning a dictionary of values added to all returned rows.

Other Children

- **col** (mapping; the value is the element content, the key is in the 'key' (or, equivalently, key) attribute) -- Mapping of source keys to column ranges.
- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Macros predefined here: Macro RSTservicelink, Macro RSTtable, Macro colNames, Macro decapitalize, Macro dlMetaURI, Macro fullDLURL, Macro getConfig, Macro inputRelativePath, Macro inputSize, Macro internallink, Macro lastSourceElements, Macro magicEmpty, Macro metaString, Macro property, Macro quote, Macro rdId, Macro rdIdDotted, Macro rootlessPath, Macro rowsProcessed, Macro schema, Macro sourceDate, Macro splitPreviewPath, Macro sqlquote, Macro srcstem, Macro standardPreviewPath, Macro test, Macro today, Macro upper, Macro urlquote

Element contextGrammar

A grammar for web inputs.

The source tokens for context grammars are dictionaries; these are either typed dictionaries from nevow formal, where the values usually are atomic, or, preferably, the dictionaries of lists from request.args.

ContextGrammars never yield rows, so they're probably fairly useless in normal cirumstances.

In normal usage, they just yield a single parameter row, corresponding to the source dictionary possibly completed with defaults, where non-requried input keys get None defaults where not given. Missing required parameters yield errors.

This parameter row honors the multiplicity specification, i.e., single or forced-single are just values, multiple are lists. The content are *parsed* values (using the InputKeys' parsers).

Since most VO protocols require case-insensitive matching of parameter names, matching of input key names and the keys of the input dictionary is attempted first literally, then disregarding case.

Atomic Children

- **enc** (unicode string; defaults to None) -- Encoding of strings coming in from source.
- **inputTD** (id reference; defaults to <Not given/empty>) -- The input table from which to take the input keys
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.

Structure Children

- ignoreOn (contains Element ignoreOn) -- Conditions for ignoring certain input records. These triggers drop an input record entirely. If you feed multiple tables and just want to drop a row from a specific table, you can use ignoreOn in a rowmaker.
- inputKeys (contains Element inputKey and may be repeated zero or more times) -- Extra input keys not defined in the inputTD. This is used when services want extra input processed by them rather than their core.
- rowfilters (contains Element rowfilter and may be repeated zero or more times) -- Row filters for this grammar.
- sourceFields (contains Element sourceFields) -- Code returning a dictionary of values added to all returned rows.

Other Children

- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Macros predefined here: Macro RSTservicelink, Macro RSTtable, Macro colNames, Macro decapitalize, Macro dlMetaURI, Macro fullDLURL, Macro getConfig, Macro inputRelativePath, Macro inputSize, Macro internallink, Macro lastSourceElements, Macro magicEmpty, Macro metaString, Macro property, Macro quote, Macro rdId, Macro rdIdDotted, Macro rootlessPath, Macro rowsProcessed, Macro schema, Macro sourceDate, Macro splitPreviewPath, Macro sqlquote, Macro srcstem, Macro standardPreviewPath, Macro test, Macro today, Macro upper, Macro urlquote

Element csvGrammar

A grammar that uses python's csv module to parse files.

Note that these grammars by default interpret the first line of the input file as the column names. When your files don't follow that convention, you *must* give names (as in `names='raj2000, dej2000, magV'`), or you'll lose the first line and have silly column names.

CSVGrammars currently do not support non-ASCII inputs. Contact the authors if you need that.

If data is left after filling the defind keys, it is available under the NOTASSIGNED key.

Atomic Children

- **delimiter** (unicode string; defaults to ',') -- CSV delimiter
- **enc** (unicode string; defaults to None) -- Encoding of strings coming in from source.
- **gunzip** (boolean; defaults to 'False') -- Unzip sources while reading? (Deprecated, use preFilter='zcat')
- **names** (Comma-separated list of strings; defaults to None) -- Names for the parsed fields, in sequence of the comma separated values. The default is to read the field names from the first line of the csv file. You can use macros here, e.g., \colNames{someTable}.
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **preFilter** (unicode string; defaults to None) -- Shell command to pipe the input through before passing it on to the grammar. Classical examples include zcat or bzcat, but you can commit arbitrary shell atrocities here.
- **strip** (boolean; defaults to 'False') -- If True, whitespace immediately following a delimiter is ignored.
- **topIgnoredLines** (integer; defaults to '0') -- Skip this many lines at the top of each source file.

Structure Children

- ignoreOn (contains Element ignoreOn) -- Conditions for ignoring certain input records. These triggers drop an input record entirely. If you feed multiple tables and just want to drop a row from a specific table, you can use ignoreOn in a rowmaker.
- mapKeys (contains Element mapKeys) -- Prescription for how to map header keys to grammar dictionary keys
- rowfilters (contains Element rowfilter and may be repeated zero or more times) -- Row filters for this grammar.
- sourceFields (contains Element sourceFields) -- Code returning a dictionary of values added to all returned rows.

Other Children

- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Macros predefined here: Macro RSTservicelink, Macro RSTtable, Macro colNames, Macro decapitalize, Macro dlMetaURI, Macro fullDLURL, Macro getConfig, Macro inputRelativePath, Macro inputSize, Macro internallink, Macro lastSourceElements, Macro magicEmpty, Macro metaString, Macro property, Macro quote, Macro rdId, Macro rdIdDotted, Macro rootlessPath, Macro rowsProcessed, Macro schema, Macro sourceDate, Macro splitPreviewPath, Macro sqlquote, Macro srcstem, Macro standardPreviewPath, Macro test, Macro today, Macro upper, Macro urlquote

Element customGrammar

A Grammar with a user-defined row iterator taken from a module.

See the Writing Custom Grammars (in the reference manual) for details.

Atomic Children

- **enc** (unicode string; defaults to None) -- Encoding of strings coming in from source.
- **isDispatching** (boolean; defaults to 'False') -- Is this a dispatching grammar (i.e., does the row iterator return pairs of role, row rather than only rows)?
- **module** (unicode string; defaults to <Undefined>) -- Path to module containing your row iterator.
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.

Structure Children

- ignoreOn (contains Element ignoreOn) -- Conditions for ignoring certain input records. These triggers drop an input record entirely. If you feed multiple tables and just want to drop a row from a specific table, you can use ignoreOn in a rowmaker.

- rowfilters (contains Element rowfilter and may be repeated zero or more times) -- Row filters for this grammar.
- sourceFields (contains Element sourceFields) -- Code returning a dictionary of values added to all returned rows.

Other Children

- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Macros predefined here: Macro RSTservicelink, Macro RSTtable, Macro colNames, Macro decapitalize, Macro dlMetaURI, Macro fullDLURL, Macro getConfig, Macro inputRelativePath, Macro inputSize, Macro internallink, Macro lastSourceElements, Macro magicEmpty, Macro metaString, Macro property, Macro quote, Macro rdId, Macro rdIdDotted, Macro rootlessPath, Macro rowsProcessed, Macro schema, Macro sourceDate, Macro splitPreviewPath, Macro sqlquote, Macro srcstem, Macro standardPreviewPath, Macro test, Macro today, Macro upper, Macro urlquote

Element dictlistGrammar

A grammar that "parses" from lists of dicts.

Actually, it will just return the dicts as they are passed. This is mostly useful internally, though it might come in handy in custom code.

Atomic Children

- **asPars** (boolean; defaults to 'False') -- Just return the first item of the list as parameters row and exit?
- **enc** (unicode string; defaults to None) -- Encoding of strings coming in from source.
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.

Structure Children

- ignoreOn (contains Element ignoreOn) -- Conditions for ignoring certain input records. These triggers drop an input record entirely. If you feed multiple tables and just want to drop a row from a specific table, you can use ignoreOn in a rowmaker.
- rowfilters (contains Element rowfilter and may be repeated zero or more times) -- Row filters for this grammar.
- sourceFields (contains Element sourceFields) -- Code returning a dictionary of values added to all returned rows.

Other Children

- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Macros predefined here: Macro RSTservicelink, Macro RSTtable, Macro colNames, Macro decapitalize, Macro dlMetaURI, Macro fullDLURL, Macro getConfig, Macro inputRelativePath, Macro inputSize, Macro internallink, Macro lastSourceElements, Macro magicEmpty, Macro metaString, Macro property, Macro quote, Macro rdId, Macro rdIdDotted, Macro rootlessPath, Macro rowsProcessed, Macro schema, Macro sourceDate, Macro splitPreviewPath, Macro sqlquote, Macro srcstem, Macro standardPreviewPath, Macro test, Macro today, Macro upper, Macro urlquote

Element directGrammar

A user-defined external grammar.

See the separate document on user-defined code on more on direct grammars.

You will almost always use these in connection with C code generated by `dachs mkboost`.

Atomic Children

- **autoNull** (unicode string; defaults to None) -- Use this string as general NULL value (when reading from plain text).
- **cBooster** (unicode string; defaults to <Undefined>) -- resdir- relative path to the booster C source.

- **customFlags** (unicode string; defaults to '') -- Pass these flags to the C compiler when building the booster.
- **extension** (integer; defaults to '1') -- For FITS table boosters, get the table from this extension.
- **gzippedInput** (boolean; defaults to 'False') -- Pipe gzip before booster? (will not work for FITS)
- **ignoreBadRecords** (boolean; defaults to 'False') -- Let booster ignore invalid records?
- **preFilter** (unicode string; defaults to None) -- Pipe input through this program before handing it to the booster; this string is shell-expanded (will not work for FITS).
- **recordSize** (integer; defaults to '4000') -- For bin boosters, read this many bytes to make up a record; for line-based boosters, this is the maximum length of an input line.
- **splitChar** (unicode string; defaults to '|') -- For split boosters, use this as the separator.
- **type** (One of: bin, fits, col, split; defaults to 'col') -- Make code for a booster parsing by column indices (col), by splitting along separators (split), by reading fixed-length binary records (bin), for from FITS binary tables (fits).

Structure Children

- mapKeys (contains Element mapKeys) -- For a FITS booster, map DB table column names to FITS column names (e.g., if the FITS table name flx is to end up in the DB column flux, say flux:flx).

Element embeddedGrammar

A Grammar defined by a code application.

To define this grammar, write a ProcApp iterator leading to code yielding row dictionaries. The grammar input is available as self.sourceToken; for normal grammars within data elements, that would be a fully qualified file name.

Grammars can also return one "parameter" dictionary per source (the input to a make's parmaker). In an embedded grammar, you can define a pargetter to do that. It works like the iterator, except that it returns a single dictionary rather than yielding several of them.

This could look like this, when the grammar input is some iterable:

```
<embeddedGrammar>
  <iterator>
    <setup>
      <code>
        testData = "a"*1024
      </code>
    </setup>
    <code>
      for i in self.sourceToken:
        yield {'index': i, 'data': testData}
    </code>
  </iterator>
</embeddedGrammar>
```

Atomic Children

- **enc** (unicode string; defaults to None) -- Encoding of strings coming in from source.
- **isDispatching** (boolean; defaults to 'False') -- Is this a dispatching grammar (i.e., does the row iterator return pairs of role, row rather than only rows)?
- **notify** (boolean; defaults to 'False') -- Enable notification of begin/end of processing (as for other grammars; embedded grammars often have odd source tokens for which you don't want that).
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.

Structure Children

- ignoreOn (contains Element ignoreOn) -- Conditions for ignoring certain input records. These triggers drop an input record entirely. If you feed multiple tables and just want to drop a row from a specific table, you can use ignoreOn in a rowmaker.
- iterator (contains Element iterator) -- Code yielding row dictionaries
- pargetter (contains Element pargetter) -- Code returning a parameter dictionary
- rowfilters (contains Element rowfilter and may be repeated zero or more times) -- Row filters for this grammar.
- sourceFields (contains Element sourceFields) -- Code returning a dictionary of values added to all returned rows.

Other Children

- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Macros predefined here: Macro RSTservicelink, Macro RSTtable, Macro colNames, Macro decapitalize, Macro dlMetaURI, Macro fullDLURL, Macro getConfig, Macro inputRelativePath, Macro inputSize, Macro internallink, Macro lastSourceElements, Macro magicEmpty, Macro metaString, Macro property, Macro quote, Macro rdId, Macro rdIdDotted, Macro rootlessPath, Macro rowsProcessed, Macro schema, Macro sourceDate, Macro splitPreviewPath, Macro sqlquote, Macro srcstem, Macro standardPreviewPath, Macro test, Macro today, Macro upper, Macro urlquote

Element fitsProdGrammar

A grammar that returns FITS-headers as dictionaries.

This is the grammar you want when one FITS file corresponds to one row in the destination table.

The keywords of the grammar record are the cards in the primary header (or some other hdu using the same-named attribute). "-" in keywords is replaced with an underscore for easier @-referencing. You can use a mapKeys element to effect further name cosmetics.

This grammar should handle compressed FITS images transparently if set qnd="False". This means that you will essentially get the headers from the second extension for those even if you left hdu="0".

The original header is preserved as the value of the header_ key. This is mainly intended for use WCS use, as in `wcs.WCS(@header_)`.

If you have more complex structures in your FITS files, you can get access to the pyfits HDU using the hdusField attribute. With `hdusField="_H"`, you could say things like `@_H[1].data[10][0]` to get the first data item in the tenth row in the second HDU.

Atomic Children

- **enc** (unicode string; defaults to None) -- Encoding of strings coming in from source.
- **hdu** (integer; defaults to '0') -- Take the header from this HDU. You must say qnd='False' for this to take effect.

- **hdusField** (unicode string; defaults to None) -- If set, the complete pyfits HDU list for the FITS file is returned in this grammar field.
- **maxHeaderBlocks** (integer; defaults to '40') -- Stop looking for FITS END cards and raise an error after this many blocks. You may need to raise this for people dumping obscene amounts of data or history into headers.
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **qnd** (boolean; defaults to 'True') -- Use a hack to read the FITS header more quickly. This only works for the primary HDU

Structure Children

- ignoreOn (contains Element ignoreOn) -- Conditions for ignoring certain input records. These triggers drop an input record entirely. If you feed multiple tables and just want to drop a row from a specific table, you can use ignoreOn in a rowmaker.
- mapKeys (contains Element mapKeys) -- Prescription for how to map header keys to grammar dictionary keys
- rowfilters (contains Element rowfilter and may be repeated zero or more times) -- Row filters for this grammar.
- sourceFields (contains Element sourceFields) -- Code returning a dictionary of values added to all returned rows.

Other Children

- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Macros predefined here: Macro RSTservicelink, Macro RSTtable, Macro colNames, Macro decapitalize, Macro dlMetaURI, Macro fullDLURL, Macro getConfig, Macro inputRelativePath, Macro inputSize, Macro internallink, Macro lastSourceElements, Macro magicEmpty, Macro metaString, Macro property, Macro quote, Macro rdId, Macro rdIdDotted, Macro rootlessPath, Macro rowsProcessed, Macro schema, Macro sourceDate, Macro splitPreviewPath, Macro sqlquote, Macro srcstem, Macro standardPreviewPath, Macro test, Macro today, Macro upper, Macro urlquote

Element fitsTableGrammar

A grammar parsing from FITS tables.

fitsTableGrammar result in typed records, i.e., values normally come in the types they are supposed to have. Of course, that won't work for datetimes, STC-S regions, and the like.

The keys of the result dictionaries are simpily the names given in the FITS.

Atomic Children

- **enc** (unicode string; defaults to None) -- Encoding of strings coming in from source.
- **hdu** (integer; defaults to '1') -- Take the data from this extension (primary=0). Tabular data typically resides in the first extension.
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.

Structure Children

- ignoreOn (contains Element ignoreOn) -- Conditions for ignoring certain input records. These triggers drop an input record entirely. If you feed multiple tables and just want to drop a row from a specific table, you can use ignoreOn in a rowmaker.
- rowfilters (contains Element rowfilter and may be repeated zero or more times) -- Row filters for this grammar.
- sourceFields (contains Element sourceFields) -- Code returning a dictionary of values added to all returned rows.

Other Children

- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Macros predefined here: Macro RSTservicelink, Macro RSTtable, Macro colNames, Macro decapitalize, Macro dlMetaURI, Macro fullDLURL, Macro getConfig, Macro inputRelativePath, Macro inputSize, Macro internallink, Macro

lastSourceElements, Macro magicEmpty, Macro metaString, Macro property, Macro quote, Macro rdId, Macro rdIdDotted, Macro rootlessPath, Macro rowsProcessed, Macro schema, Macro sourceDate, Macro splitPreviewPath, Macro sqlquote, Macro srcstem, Macro standardPreviewPath, Macro test, Macro today, Macro upper, Macro urlquote

Element freeREGrammar

A grammar allowing "free" regular expressions to parse a document.

Basically, you give a rowProduction to match individual records in the document. All matches of rowProduction will then be matched with parseRE, which in turn must have named groups. The dictionary from named groups to their matches makes up the input row.

For writing the parseRE, we recommend writing an element, using a CDATA construct, and taking advantage of python's "verbose" regular expressions. Here's an example:

```
<parseRE><![CDATA[(?xsm)^name::(?P<name>.*)
  ^query::(?P<query>.*)
  ^description::(?P<description>.*)\.\.
]]></parseRE>
```

Atomic Children

- **enc** (unicode string; defaults to None) -- Encoding of strings coming in from source.
- **ignoreJunk** (boolean; defaults to 'False') -- Ignore everything outside of the row production
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **parseRE** (unicode string; defaults to <Undefined>) -- RE containing named groups matching a record
- **rowProduction** (unicode string; defaults to '(?m)^.+$\n') -- RE matching a complete record.
- **stripTokens** (boolean; defaults to 'False') -- Strip whitespace from result tokens?

Structure Children

- ignoreOn (contains Element ignoreOn) -- Conditions for ignoring certain input records. These triggers drop an input record entirely. If you feed multiple tables and just want to drop a row from a specific table, you can use ignoreOn in a rowmaker.
- rowfilters (contains Element rowfilter and may be repeated zero or more times) -- Row filters for this grammar.
- sourceFields (contains Element sourceFields) -- Code returning a dictionary of values added to all returned rows.

Other Children

- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Macros predefined here: Macro RSTservicelink, Macro RSTtable, Macro colNames, Macro decapitalize, Macro dlMetaURI, Macro fullDLURL, Macro getConfig, Macro inputRelativePath, Macro inputSize, Macro internallink, Macro lastSourceElements, Macro magicEmpty, Macro metaString, Macro property, Macro quote, Macro rdId, Macro rdIdDotted, Macro rootlessPath, Macro rowsProcessed, Macro schema, Macro sourceDate, Macro splitPreviewPath, Macro sqlquote, Macro srcstem, Macro standardPreviewPath, Macro test, Macro today, Macro upper, Macro urlquote

Element iterator

A definition of an iterator of a grammar.

The code defined here becomes the _iterRows method of a grammar.common.RowIterator class. This means that you can access self.grammar (the parent grammar; you can use this to transmit properties from the RD to your function) and self.sourceToken (whatever gets passed to parse()).

May occur in Element embeddedGrammar.

Atomic Children

- **code** (unicode string; defaults to <Not given/empty>) -- A python function body.

- **deprecated** (unicode string; defaults to None) -- A deprecation message. This will be shown if this procDef is being compiled.
- **doc** (unicode string; defaults to '') -- Human-readable docs for this proc (may be interpreted as restructured text).
- **name** (unicode string; defaults to <Not given/empty>) -- A name of the proc. ProcApps compute their (python) names to be somwhat random strings. Set a name manually to receive more easily decipherable error messages. If you do that, you have to care about name clashes yourself, though.
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **procDef** (id reference; defaults to <Not given/empty>) -- Reference to the procedure definition to apply
- **type** (One of: iterator, pargetter, regTest, rowfilter, dataFunction, descriptorGenerator, metaMaker, phraseMaker, mixinProc, dataFormatter, sourceFields, apply, t_t; defaults to None) -- The type of the procedure definition. The procedure applications will in general require certain types of definitions.

Structure Children

- bindings (contains Element bind and may be repeated zero or more times) -- Values for parameters of the procedure definition
- setups (contains Element setup and may be repeated zero or more times) -- Setup of the namespace the function will run in

Element keyValueGrammar

A grammar to parse key-value pairs from files.

The default assumes one pair per line, with # comments and = as separating character.

yieldPairs makes the grammar return an empty docdict and {"key":, "value":} rowdicts.

Whitespace around key and value is ignored.

Atomic Children

- **commentPattern** (unicode string; defaults to '(?m)#.*') -- A regular expression describing comments.
- **enc** (unicode string; defaults to None) -- Encoding of strings coming in from source.
- **kvSeparators** (unicode string; defaults to ':=') -- Characters accepted as separators between key and value
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **pairSeparators** (unicode string; defaults to 'n') -- Characters accepted as separators between pairs
- **yieldPairs** (boolean; defaults to 'False') -- Yield key-value pairs instead of complete records?

Structure Children

- ignoreOn (contains Element ignoreOn) -- Conditions for ignoring certain input records. These triggers drop an input record entirely. If you feed multiple tables and just want to drop a row from a specific table, you can use ignoreOn in a rowmaker.
- mapKeys (contains Element mapKeys) -- Mappings to rename the keys coming from the source files. Use this, in particular, if the keys are not valid python identifiers.
- rowfilters (contains Element rowfilter and may be repeated zero or more times) -- Row filters for this grammar.
- sourceFields (contains Element sourceFields) -- Code returning a dictionary of values added to all returned rows.

Other Children

- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Macros predefined here: Macro RSTservicelink, Macro RSTtable, Macro colNames, Macro decapitalize, Macro dlMetaURI, Macro fullDLURL, Macro getConfig, Macro inputRelativePath, Macro inputSize, Macro internallink, Macro lastSourceElements, Macro magicEmpty, Macro metaString, Macro property, Macro quote, Macro rdId, Macro rdIdDotted, Macro rootlessPath, Macro rowsProcessed, Macro schema, Macro sourceDate, Macro splitPreviewPath, Macro sqlquote, Macro srcstem, Macro standardPreviewPath, Macro test, Macro today, Macro upper, Macro urlquote

Element mapKeys

Mapping of names, specified in long or short forms.

mapKeys is necessary in grammars like keyValueGrammar or fitsProdGrammar. In these, the source files themselves give key names. Within the GAVO DC, keys are required to be valid python identifiers (i.e., match `[A-Za-z\_][A-Za-z\_0-9]*`). If keys coming in do not have this form, mapping can force proper names.

mapKeys could also be used to make incoming names more suitable for matching with shell patterns (like in rowmaker idmaps).

May occur in Element cdfHeaderGrammar, Element directGrammar, Element fitsProdGrammar, Element pdsGrammar, Element keyValueGrammar, Element csvGrammar.

Atomic Children

- Character content of the element (defaulting to '') -- Simple mappings in the form<dest>:<src>{,<dest>:<src>}

Other Children

- **map** (mapping; the key is the element content, the value is in the 'key' (or, equivalently, dest) attribute) -- Map source names given in content to the name given in dest.

Element mySQLDumpGrammar

A grammar pulling information from MySQL dump files.

WARNING: This is a quick hack. If you want/need it, please contact the authors.

At this point this is nothing but an ugly RE mess with lots of assumptions about the dump file that's easily fooled. Also, the entire dump file will be pulled into memory.

Since grammar semantics cannot do anything else, this will always only iterate over a single table. This currently is fixed to the first, but it's conceivable to make that selectable.

Database NULLs are already translated into Nones.

In other words: It might do for simple cases. If you have something else, improve this or complain to the authors.

Atomic Children

- **enc** (unicode string; defaults to None) -- Encoding of strings coming in from source.
- **gunzip** (boolean; defaults to 'False') -- Unzip sources while reading? (Deprecated, use preFilter='zcat')
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **preFilter** (unicode string; defaults to None) -- Shell command to pipe the input through before passing it on to the grammar. Classical examples include zcat or bzcat, but you can commit arbitrary shell atrocities here.

Structure Children

- ignoreOn (contains Element ignoreOn) -- Conditions for ignoring certain input records. These triggers drop an input record entirely. If you feed multiple tables and just want to drop a row from a specific table, you can use ignoreOn in a rowmaker.
- rowfilters (contains Element rowfilter and may be repeated zero or more times) -- Row filters for this grammar.
- sourceFields (contains Element sourceFields) -- Code returning a dictionary of values added to all returned rows.

Other Children

- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Macros predefined here: Macro RSTservicelink, Macro RSTtable, Macro colNames, Macro decapitalize, Macro dlMetaURI, Macro fullDLURL, Macro getConfig, Macro inputRelativePath, Macro inputSize, Macro internallink, Macro lastSourceElements, Macro magicEmpty, Macro metaString, Macro property, Macro quote, Macro rdId, Macro rdIdDotted, Macro rootlessPath, Macro rowsProcessed, Macro schema, Macro sourceDate, Macro splitPreviewPath, Macro sqlquote, Macro srcstem, Macro standardPreviewPath, Macro test, Macro today, Macro upper, Macro urlquote

Element nullGrammar

A grammar that never returns any rows.

Atomic Children

- **enc** (unicode string; defaults to None) -- Encoding of strings coming in from source.
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.

Structure Children

- ignoreOn (contains Element ignoreOn) -- Conditions for ignoring certain input records. These triggers drop an input record entirely. If you feed multiple tables and just want to drop a row from a specific table, you can use ignoreOn in a rowmaker.
- rowfilters (contains Element rowfilter and may be repeated zero or more times) -- Row filters for this grammar.
- sourceFields (contains Element sourceFields) -- Code returning a dictionary of values added to all returned rows.

Other Children

- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Macros predefined here: Macro RSTservicelink, Macro RSTtable, Macro colNames, Macro decapitalize, Macro dlMetaURI, Macro fullDLURL, Macro getConfig, Macro inputRelativePath, Macro inputSize, Macro internallink, Macro lastSourceElements, Macro magicEmpty, Macro metaString, Macro property, Macro quote, Macro rdId, Macro rdIdDotted, Macro rootlessPath, Macro rowsProcessed, Macro schema, Macro sourceDate, Macro splitPreviewPath, Macro sqlquote, Macro srcstem, Macro standardPreviewPath, Macro test, Macro today, Macro upper, Macro urlquote

Element odbcGrammar

A grammar that feeds from a remote database.

This works as a sort of poor man's foreign data wrapper: you pull data from a remote database now and then, mogrifying it into whatever format you want locally.

This expects files containing pyodbc connection strings as sources, so you'll normally just have one source. Having the credentials externally helps keeping RDs using this safe for public version control.

An example for an ODBC connection string:

```
DRIVER={SQL Server};SERVER=localhost;DATABASE=testdb;UID=me;PWD=pass
```

See also http://www.connectionstrings.com/

This will only work if pyodbc (debian: python-pyodbc) is installed. Additionally, you will have to install the odbc driver corresponding to your source database (e.g., odbc-postgresql).

Atomic Children

- **enc** (unicode string; defaults to None) -- Encoding of strings coming in from source.
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.

- **query** (unicode string; defaults to None) -- The query to run on the remote server. The keys of the grammar will be the names of the result columns.

Structure Children

- ignoreOn (contains Element ignoreOn) -- Conditions for ignoring certain input records. These triggers drop an input record entirely. If you feed multiple tables and just want to drop a row from a specific table, you can use ignoreOn in a rowmaker.
- rowfilters (contains Element rowfilter and may be repeated zero or more times) -- Row filters for this grammar.
- sourceFields (contains Element sourceFields) -- Code returning a dictionary of values added to all returned rows.

Other Children

- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Macros predefined here: Macro RSTservicelink, Macro RSTtable, Macro colNames, Macro decapitalize, Macro dlMetaURI, Macro fullDLURL, Macro getConfig, Macro inputRelativePath, Macro inputSize, Macro internallink, Macro lastSourceElements, Macro magicEmpty, Macro metaString, Macro property, Macro quote, Macro rdId, Macro rdIdDotted, Macro rootlessPath, Macro rowsProcessed, Macro schema, Macro sourceDate, Macro splitPreviewPath, Macro sqlquote, Macro srcstem, Macro standardPreviewPath, Macro test, Macro today, Macro upper, Macro urlquote

Element pargetter

A definition of the parameter getter of an embedded grammar.

The code defined here becomes the getParameters method of the generated row iterator. This means that the dictionary returned here becomes the input to a parmaker.

If you don't define it, the parameter dict will be empty.

Like the iterators, pargetters see the current source token as self.sourceToken, and the grammar as self.grammar.

May occur in Element embeddedGrammar.

Atomic Children

- **code** (unicode string; defaults to <Not given/empty>) -- A python function body.
- **deprecated** (unicode string; defaults to None) -- A deprecation message. This will be shown if this procDef is being compiled.
- **doc** (unicode string; defaults to '') -- Human-readable docs for this proc (may be interpreted as restructured text).
- **name** (unicode string; defaults to <Not given/empty>) -- A name of the proc. ProcApps compute their (python) names to be somwhat random strings. Set a name manually to receive more easily decipherable error messages. If you do that, you have to care about name clashes yourself, though.
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **procDef** (id reference; defaults to <Not given/empty>) -- Reference to the procedure definition to apply
- **type** (One of: iterator, pargetter, regTest, rowfilter, dataFunction, descriptorGenerator, metaMaker, phraseMaker, mixinProc, dataFormatter, sourceFields, apply, t_t; defaults to None) -- The type of the procedure definition. The procedure applications will in general require certain types of definitions.

Structure Children

- bindings (contains Element bind and may be repeated zero or more times) -- Values for parameters of the procedure definition
- setups (contains Element setup and may be repeated zero or more times) -- Setup of the namespace the function will run in

Element pdsGrammar

A grammar that returns labels of PDS documents as rowdicts.

PDS is the file format of the Planetary Data System; the labels are quite like, but not quite like FITS headers.

Extra care needs to be taken here since the values in the rawdicts can be complex objects (e.g., other labels). It's likely that you will need constructs like `@IMAGE["KEY"]`.

Current versions of PyPDS also don't parse the values. This is particularly insiduous because general strings are marked with " in PDS. When mapping those, you'll probably want a @KEY.strip('"').

You'll need PyPDS to use this; there's no Debian package for that yet, so you'll have to do a source install from git://github.com/RyanBalfanz/PyPDS.git

Atomic Children

- **enc** (unicode string; defaults to None) -- Encoding of strings coming in from source.
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.

Structure Children

- ignoreOn (contains Element ignoreOn) -- Conditions for ignoring certain input records. These triggers drop an input record entirely. If you feed multiple tables and just want to drop a row from a specific table, you can use ignoreOn in a rowmaker.
- mapKeys (contains Element mapKeys) -- Prescription for how to map labels keys to grammar dictionary keys
- rowfilters (contains Element rowfilter and may be repeated zero or more times) -- Row filters for this grammar.
- sourceFields (contains Element sourceFields) -- Code returning a dictionary of values added to all returned rows.

Other Children

- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Macros predefined here: Macro RSTservicelink, Macro RSTtable, Macro colNames, Macro decapitalize, Macro dlMetaURI, Macro fullDLURL, Macro getConfig, Macro inputRelativePath, Macro inputSize, Macro internallink, Macro lastSourceElements, Macro magicEmpty, Macro metaString, Macro property, Macro quote, Macro rdId, Macro rdIdDotted, Macro rootlessPath, Macro rowsProcessed, Macro schema, Macro sourceDate, Macro splitPreviewPath, Macro sqlquote, Macro srcstem, Macro standardPreviewPath, Macro test, Macro today, Macro upper, Macro urlquote

Element reGrammar

A grammar that builds rowdicts from records and fields specified via REs separating them.

There is also a simple facility for "cleaning up" records. This can be used to remove standard shell-like comments; use `recordCleaner="(?:#.*)?(.*)"`.

Atomic Children

- **commentPat** (unicode string; defaults to None) -- RE inter-record material to be ignored (note: make this match the entire comment, or you'll get random mess from partly-matched comments. Use '(?m)^#.*$' for beginning-of-line hash-comments.
- **enc** (unicode string; defaults to None) -- Encoding of strings coming in from source.
- **fieldSep** (unicode string; defaults to '\s+') -- RE for separating two fields in a record.
- **gunzip** (boolean; defaults to 'False') -- Unzip sources while reading? (Deprecated, use preFilter='zcat')
- **lax** (boolean; defaults to 'False') -- allow more or less fields in source records than there are names
- **names** (Comma-separated list of strings; defaults to '') -- Names for the parsed fields, in matching sequence. You can use macros here, e.g., \colNames{someTable}.
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **preFilter** (unicode string; defaults to None) -- Shell command to pipe the input through before passing it on to the grammar. Classical examples include zcat or bzcat, but you can commit arbitrary shell atrocities here.

- **recordCleaner** (unicode string; defaults to None) -- A regular expression matched against each record. The matched groups in this RE are joined by blanks and used as the new pattern. This can be used for simple cleaning jobs; However, records not matching recordCleaner are rejected.
- **recordSep** (unicode string; defaults to 'n') -- RE for separating two records in the source.
- **stopPat** (unicode string; defaults to None) -- Stop parsing when a record *matches* this RE (this is for skipping non-data footers
- **topIgnoredLines** (integer; defaults to '0') -- Skip this many lines at the top of each source file.

Structure Children

- ignoreOn (contains Element ignoreOn) -- Conditions for ignoring certain input records. These triggers drop an input record entirely. If you feed multiple tables and just want to drop a row from a specific table, you can use ignoreOn in a rowmaker.
- rowfilters (contains Element rowfilter and may be repeated zero or more times) -- Row filters for this grammar.
- sourceFields (contains Element sourceFields) -- Code returning a dictionary of values added to all returned rows.

Other Children

- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Macros predefined here: Macro RSTservicelink, Macro RSTtable, Macro colNames, Macro decapitalize, Macro dlMetaURI, Macro fullDLURL, Macro getConfig, Macro inputRelativePath, Macro inputSize, Macro internallink, Macro lastSourceElements, Macro magicEmpty, Macro metaString, Macro property, Macro quote, Macro rdId, Macro rdIdDotted, Macro rootlessPath, Macro rowsProcessed, Macro schema, Macro sourceDate, Macro splitPreviewPath, Macro sqlquote, Macro srcstem, Macro standardPreviewPath, Macro test, Macro today, Macro upper, Macro urlquote

Element rowfilter

A generator for rows coming from a grammar.

Rowfilters receive rows (i.e., dictionaries) as yielded by a grammar under the name row. Additionally, the embedding row iterator is available under the name rowIter.

Macros are expanded within the embedding grammar.

The procedure definition *must* result in a generator, i.e., there must be at least one yield; in general, this will typically be a `yield row`, but a rowfilter may swallow or create as many rows as desired.

If you forget to have a yield in the rowfilter source, you'll get a "NoneType is not iterable" error that's a bit hard to understand.

Here, you can only access whatever comes from the grammar. You can access grammar keys in late parameters as row[key] or, if key is like an identifier, as @key.

May occur in Element voTableGrammar, Element reGrammar, Element contextGrammar, Element columnGrammar, Element cdfHeaderGrammar, Element fitsTableGrammar, Element rowsetGrammar, Element binaryGrammar, Element fitsProdGrammar, Element pdsGrammar, Element customGrammar, Element odbcGrammar, Element mySQLDumpGrammar, Element freeREGrammar, Element dictlistGrammar, Element keyValueGrammar, Element csvGrammar, Element embeddedGrammar, Element transparentGrammar, Element nullGrammar.

Atomic Children

- **code** (unicode string; defaults to <Not given/empty>) -- A python function body.
- **deprecated** (unicode string; defaults to None) -- A deprecation message. This will be shown if this procDef is being compiled.
- **doc** (unicode string; defaults to '') -- Human-readable docs for this proc (may be interpreted as restructured text).
- **name** (unicode string; defaults to <Not given/empty>) -- A name of the proc. ProcApps compute their (python) names to be somwhat random strings. Set a name manually to receive more easily decipherable error messages. If you do that, you have to care about name clashes yourself, though.

- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **procDef** (id reference; defaults to <Not given/empty>) -- Reference to the procedure definition to apply
- **type** (One of: iterator, pargetter, regTest, rowfilter, dataFunction, descriptorGenerator, metaMaker, phraseMaker, mixinProc, dataFormatter, sourceFields, apply, t_t; defaults to None) -- The type of the procedure definition. The procedure applications will in general require certain types of definitions.

Structure Children

- bindings (contains Element bind and may be repeated zero or more times) -- Values for parameters of the procedure definition
- setups (contains Element setup and may be repeated zero or more times) -- Setup of the namespace the function will run in

Element rowsetGrammar

A grammar handling sequences of tuples.

To add semantics to the field, it must know the "schema" of the data. This is defined via the table it is supposed to get the input from.

This grammar probably is only useful for internal purposes.

Atomic Children

- **enc** (unicode string; defaults to None) -- Encoding of strings coming in from source.
- **fieldsFrom** (id reference; defaults to <Undefined>) -- the table defining the columns in the tuples.
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.

Structure Children

- ignoreOn (contains Element ignoreOn) -- Conditions for ignoring certain input records. These triggers drop an input record entirely. If you feed multiple tables and just want to drop a row from a specific table, you can use ignoreOn in a rowmaker.
- rowfilters (contains Element rowfilter and may be repeated zero or more times) -- Row filters for this grammar.
- sourceFields (contains Element sourceFields) -- Code returning a dictionary of values added to all returned rows.

Other Children

- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Macros predefined here: Macro RSTservicelink, Macro RSTtable, Macro colNames, Macro decapitalize, Macro dlMetaURI, Macro fullDLURL, Macro getConfig, Macro inputRelativePath, Macro inputSize, Macro internallink, Macro lastSourceElements, Macro magicEmpty, Macro metaString, Macro property, Macro quote, Macro rdId, Macro rdIdDotted, Macro rootlessPath, Macro rowsProcessed, Macro schema, Macro sourceDate, Macro splitPreviewPath, Macro sqlquote, Macro srcstem, Macro standardPreviewPath, Macro test, Macro today, Macro upper, Macro urlquote

Element sourceFields

A procedure application that returns a dictionary added to all incoming rows.

Use this to programmatically provide information that can be computed once but that is then added to all rows coming from a single source, usually a file. This could be useful to add information on the source of a record or the like.

The code must return a dictionary. The source that is about to be parsed is passed in as sourceToken. When parsing from files, this simply is the file name. The data the rows will be delivered to is available as "data", which is useful for adding or retrieving meta information.

May occur in Element voTableGrammar, Element reGrammar, Element contextGrammar, Element columnGrammar, Element cdfHeaderGrammar, Element fitsTableGrammar, Element rowsetGrammar, Element binaryGrammar, Element

fitsProdGrammar, Element pdsGrammar, Element customGrammar, Element odbcGrammar, Element mySQLDumpGrammar, Element freeREGrammar, Element dictlistGrammar, Element keyValueGrammar, Element csvGrammar, Element embeddedGrammar, Element transparentGrammar, Element nullGrammar.

Atomic Children

- **code** (unicode string; defaults to <Not given/empty>) -- A python function body.
- **deprecated** (unicode string; defaults to None) -- A deprecation message. This will be shown if this procDef is being compiled.
- **doc** (unicode string; defaults to '') -- Human-readable docs for this proc (may be interpreted as restructured text).
- **name** (unicode string; defaults to <Not given/empty>) -- A name of the proc. ProcApps compute their (python) names to be somwhat random strings. Set a name manually to receive more easily decipherable error messages. If you do that, you have to care about name clashes yourself, though.
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **procDef** (id reference; defaults to <Not given/empty>) -- Reference to the procedure definition to apply
- **type** (One of: iterator, pargetter, regTest, rowfilter, dataFunction, descriptorGenerator, metaMaker, phraseMaker, mixinProc, dataFormatter, sourceFields, apply, t_t; defaults to None) -- The type of the procedure definition. The procedure applications will in general require certain types of definitions.

Structure Children

- bindings (contains Element bind and may be repeated zero or more times) -- Values for parameters of the procedure definition
- setups (contains Element setup and may be repeated zero or more times) -- Setup of the namespace the function will run in

Element transparentGrammar

A grammar that returns its sourceToken as the row iterator.

This only makes sense in extreme situations and never without custom code. If you're not sure you need this, you don't want to know about it.

Atomic Children

- **enc** (unicode string; defaults to None) -- Encoding of strings coming in from source.
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.

Structure Children

- ignoreOn (contains Element ignoreOn) -- Conditions for ignoring certain input records. These triggers drop an input record entirely. If you feed multiple tables and just want to drop a row from a specific table, you can use ignoreOn in a rowmaker.
- rowfilters (contains Element rowfilter and may be repeated zero or more times) -- Row filters for this grammar.
- sourceFields (contains Element sourceFields) -- Code returning a dictionary of values added to all returned rows.

Other Children

- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Macros predefined here: Macro RSTservicelink, Macro RSTtable, Macro colNames, Macro decapitalize, Macro dlMetaURI, Macro fullDLURL, Macro getConfig, Macro inputRelativePath, Macro inputSize, Macro internallink, Macro lastSourceElements, Macro magicEmpty, Macro metaString, Macro property, Macro quote, Macro rdId, Macro rdIdDotted, Macro rootlessPath, Macro rowsProcessed, Macro schema, Macro sourceDate, Macro splitPreviewPath, Macro sqlquote, Macro srcstem, Macro standardPreviewPath, Macro test, Macro today, Macro upper, Macro urlquote

Element voTableGrammar

A grammar parsing from VOTables.

Currently, the PARAM fields are ignored, only the data rows are returned.

voTableGrammars result in typed records, i.e., values normally come in the types they are supposed to have.

Atomic Children

- **enc** (unicode string; defaults to None) -- Encoding of strings coming in from source.
- **gunzip** (boolean; defaults to 'False') -- Unzip sources while reading?
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.

Structure Children

- ignoreOn (contains Element ignoreOn) -- Conditions for ignoring certain input records. These triggers drop an input record entirely. If you feed multiple tables and just want to drop a row from a specific table, you can use ignoreOn in a rowmaker.
- rowfilters (contains Element rowfilter and may be repeated zero or more times) -- Row filters for this grammar.
- sourceFields (contains Element sourceFields) -- Code returning a dictionary of values added to all returned rows.

Other Children

- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Macros predefined here: Macro RSTservicelink, Macro RSTtable, Macro colNames, Macro decapitalize, Macro dlMetaURI, Macro fullDLURL, Macro getConfig, Macro inputRelativePath, Macro inputSize, Macro internallink, Macro lastSourceElements, Macro magicEmpty, Macro metaString, Macro property, Macro quote, Macro rdId, Macro rdIdDotted, Macro rootlessPath, Macro rowsProcessed, Macro schema, Macro sourceDate, Macro splitPreviewPath, Macro sqlquote, Macro srcstem, Macro standardPreviewPath, Macro test, Macro today, Macro upper, Macro urlquote

Cores Available

The following elements are related to cores. All cores can only occur toplevel, i.e. as direct children of resource descriptors. Cores are only useful with an id to make them referencable from services using that core.

Element adqlCore

A core taking an ADQL query from its query argument and returning the result of that query in a standard table.

Since the columns returned depend on the query, the outputTable of an ADQL core must not be defined.

Atomic Children

- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.

Structure Children

- inputTable (contains Element inputTable) -- Description of the input data
- outputTable (contains Element outputTable) -- Table describing what fields are available from this core

Other Children

- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Element coreProc

A definition of a pythonCore's functionalty.

This is a procApp complete with setup and code; you could inherit between these.

coreProcs see the embedding service, the input table passed, and the query metadata as service, inputTable, and queryMeta, respectively.

The core itself is available as self.

May occur in Element pythonCore.

Atomic Children

- **code** (unicode string; defaults to <Not given/empty>) -- A python function body.
- **deprecated** (unicode string; defaults to None) -- A deprecation message. This will be shown if this procDef is being compiled.
- **doc** (unicode string; defaults to '') -- Human-readable docs for this proc (may be interpreted as restructured text).
- **name** (unicode string; defaults to <Not given/empty>) -- A name of the proc. ProcApps compute their (python) names to be somwhat random strings. Set a name manually to receive more easily decipherable error messages. If you do that, you have to care about name clashes yourself, though.
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **procDef** (id reference; defaults to <Not given/empty>) -- Reference to the procedure definition to apply
- **type** (One of: iterator, pargetter, regTest, rowfilter, dataFunction, descriptorGenerator, metaMaker, phraseMaker, mixinProc, dataFormatter, sourceFields, apply, t_t; defaults to None) -- The type of the procedure definition. The procedure applications will in general require certain types of definitions.

Structure Children

- bindings (contains Element bind and may be repeated zero or more times) -- Values for parameters of the procedure definition
- setups (contains Element setup and may be repeated zero or more times) -- Setup of the namespace the function will run in

Element customCore

A wrapper around a core defined in a module.

This core lets you write your own cores in modules.

The module must define a class Core. When the custom core is encountered, this class will be instanciated and will be used instead of the CustomCore, so your code should probably inherit core.Core.

See Writing Custom Cores for details.

Atomic Children

- **module** (unicode string; defaults to <Undefined>) -- Path to the module containing the core definition.
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.

Structure Children

- inputTable (contains Element inputTable) -- Description of the input data
- outputTable (contains Element outputTable) -- Table describing what fields are available from this core

Other Children

- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Element dataFormatter

A procedure application that renders data in a processed service.

These play the role of the renderer, which for datalink is ususally trivial. They are supposed to take descriptor.data and return a pair of (mime-type, bytes), which is understood by most renderers.

When no dataFormatter is given for a core, it will return descriptor.data directly. This can work with the datalink renderer itself if descriptor.data will work as a nevow resource (i.e., has a renderHTTP method, as our usual products do). Consider, though, that renderHTTP runs in the main event loop and thus most not block for extended periods of time.

The following names are available to the code:

- descriptor -- whatever the DescriptorGenerator returned
- args -- all the arguments that came in from the web.

In addition to the usual names available to ProcApps, data formatters have:

- Page -- base class for resources with renderHTTP methods.
- IRequest -- the nevow interface to make Request objects with.
- File(path, type) -- if you just want to return a file on disk, pass its path and media type to File and return the result.
- TemporaryFile(path, type) -- as File, but the disk file is unlinked after use
- soda -- the protocols.soda module

May occur in Element datalinkCore.

Atomic Children

- **code** (unicode string; defaults to <Not given/empty>) -- A python function body.
- **deprecated** (unicode string; defaults to None) -- A deprecation message. This will be shown if this procDef is being compiled.
- **doc** (unicode string; defaults to '') -- Human-readable docs for this proc (may be interpreted as restructured text).
- **name** (unicode string; defaults to <Not given/empty>) -- A name of the proc. ProcApps compute their (python) names to be somwhat random strings. Set a name manually to receive more easily decipherable error messages. If you do that, you have to care about name clashes yourself, though.
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **procDef** (id reference; defaults to <Not given/empty>) -- Reference to the procedure definition to apply
- **type** (One of: iterator, pargetter, regTest, rowfilter, dataFunction, descriptorGenerator, metaMaker, phraseMaker, mixinProc, dataFormatter, sourceFields, apply, t_t; defaults to None) -- The type of the procedure definition. The procedure applications will in general require certain types of definitions.

Structure Children

- bindings (contains Element bind and may be repeated zero or more times) -- Values for parameters of the procedure definition
- setups (contains Element setup and may be repeated zero or more times) -- Setup of the namespace the function will run in

Element dataFunction

A procedure application that generates or modifies data in a processed data service.

All these operate on the data attribute of the product descriptor. The first data function plays a special role: It *must* set the data attribute (or raise some appropriate exception), or a server error will be returned to the client.

What is returned depends on the service, but typcially it's going to be a table or products.*Product instance.

Data functions can shortcut if it's evident that further data functions can only mess up (i.e., if the do something bad with the data attribute); you should not shortcut if you just *think* it makes no sense to further process your output.

To shortcut, raise either of FormatNow (falls though to the formatter, which is usually less useful) or DeliverNow (directly returns the data attribute; this can be used to return arbitrary chunks of data).

The following names are available to the code:

- descriptor -- whatever the DescriptorGenerator returned
- args -- all the arguments that came in from the web.

In addition to the usual names available to ProcApps, data functions have:

- FormatNow -- exception to raise to go directly to the formatter
- DeliverNow -- exception to raise to skip all further formatting and just deliver what's currently in descriptor.data
- File(path, type) -- if you just want to return a file on disk, pass its path and media type to File and assign the result to descriptor.data.
- TemporaryFile(path,type) -- as File, but the disk file is unlinked after use
- makeData -- the rsc.makeData function
- soda -- the protocols.soda module

May occur in Element datalinkCore.

Atomic Children

- **code** (unicode string; defaults to <Not given/empty>) -- A python function body.
- **deprecated** (unicode string; defaults to None) -- A deprecation message. This will be shown if this procDef is being compiled.
- **doc** (unicode string; defaults to '') -- Human-readable docs for this proc (may be interpreted as restructured text).
- **name** (unicode string; defaults to <Not given/empty>) -- A name of the proc. ProcApps compute their (python) names to be somwhat random strings. Set a name manually to receive more easily decipherable error messages. If you do that, you have to care about name clashes yourself, though.
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **procDef** (id reference; defaults to <Not given/empty>) -- Reference to the procedure definition to apply
- **type** (One of: iterator, pargetter, regTest, rowfilter, dataFunction, descriptorGenerator, metaMaker, phraseMaker, mixinProc, dataFormatter, sourceFields, apply, t_t; defaults to None) -- The type of the procedure definition. The procedure applications will in general require certain types of definitions.

Structure Children

- bindings (contains Element bind and may be repeated zero or more times) -- Values for parameters of the procedure definition
- setups (contains Element setup and may be repeated zero or more times) -- Setup of the namespace the function will run in

Element datalinkCore

A core for processing datalink and processed data requests.

The input table of this core is dynamically generated from its metaMakers; it makes no sense at all to try and override it.

See Datalink and SODA for more information.

In contrast to "normal" cores, one of these is made (and destroyed) for each datalink request coming in. This is because the interface of a datalink service depends on the request's value(s) of ID.

The datalink core can produce both its own metadata and data generated. It is the renderer's job to tell them apart.

Atomic Children

- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.

Structure Children

- dataFormatter (contains Element dataFormatter) -- Code that turns descriptor.data into a nevow resource or a mime, content pair. If not given, the renderer will be returned descriptor.data itself (which will probably not usually work).
- dataFunctions (contains Element dataFunction and may be repeated zero or more times) -- Code that generates of processes data for this core. The first of these plays a special role in that it must set descriptor.data, the others need not do anything at all.
- descriptorGenerator (contains Element descriptorGenerator) -- Code that takes a PUBDID and turns it into a product descriptor instance. If not given, //soda#fromStandardPubDID will be used.
- inputKeys (contains Element inputKey and may be repeated zero or more times) -- A parameter to one of the proc apps (data functions, formatters) active in this datalink core; no specific relation between input keys and procApps is supposed; all procApps are passed all argments. Conventionally, you will write the input keys in front of the proc apps that interpret them.
- inputTable (contains Element inputTable) -- Description of the input data
- metaMakers (contains Element metaMaker and may be repeated zero or more times) -- Code that takes a data descriptor and either updates input key options or yields related data.
- outputTable (contains Element outputTable) -- Table describing what fields are available from this core

Other Children

- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Element dbCore

A core performing database queries on one table or view.

DBCores ask the service for the desired output schema and adapt their output. The DBCore's output table, on the other hand, lists all fields available from the queried table.

Atomic Children

- **distinct** (boolean; defaults to 'False') -- Add a 'distinct' modifier to the query?
- **groupBy** (unicode string; defaults to None) -- A group by clause. You shouldn't generally need this, and if you use it, you must give an outputTable to your core.
- **limit** (integer; defaults to None) -- A pre-defined match limit (suppresses DB options widget).
- **namePath** (id reference; defaults to None) -- Id of an element that will be used to located names in id references. Defaults to the queriedTable's id.
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **queriedTable** (id reference; defaults to <Undefined>) -- A reference to the table this core queries.
- **sortKey** (unicode string; defaults to None) -- A pre-defined sort order (suppresses DB options widget). The sort key accepts multiple columns, separated by commas.

Structure Children

- condDescs (contains Element condDesc and may be repeated zero or more times) -- Descriptions of the SQL and input generating entities for this core; if not given, they will be generated from the table columns.
- inputTable (contains Element inputTable) -- Description of the input data
- outputTable (contains Element outputTable) -- Table describing what fields are available from this core

Other Children

- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Element debugCore

a core that returns its arguments stringified in a table.

You need to provide an external input tables for these.

Atomic Children

- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.

Structure Children

- inputTable (contains Element inputTable) -- Description of the input data
- outputTable (contains Element outputTable) -- Table describing what fields are available from this core

Other Children

- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Element descriptorGenerator

A procedure application for making product descriptors for PUBDIDs

Despite the name, a descriptor generator has to *return* (not yield) a descriptor instance. While this could be anything, it is recommended to derive custom classes from prodocols.datalink.ProductDescrpitor, which exposes essentially the columns from DaCHS' product table as attributes. This is what you get when you don't define a descriptor generator in your datalink core.

The following names are available to the code:

- pubDID -- the pubDID to be resolved
- args -- all the arguments that came in from the web (these should not ususally be necessary for making the descriptor and are completely unparsed at this point)
- FITSProductDescriptor -- the base class of FITS product descriptors
- DLFITSProductDescriptor -- the same, just for when the product table has a datalink.
- ProductDescriptor -- a base class for your own custom descriptors
- DatalinkFault -- use this when flagging failures
- soda -- contents of the soda module for convenience

If you made your pubDID using the `getStandardPubDID` rowmaker function, and you need no additional logic within the descriptor, the default (//soda#fromStandardPubDID) should do.

If you need to derive custom descriptor classes, you can see the base class under the name ProductDescriptor; there's also FITSProductDescriptor and DatalinkFault in each proc's namespace. If your Descriptor does not actually refer to something in the product table, it is likely that you want to set the descriptor's `suppressAutoLinks` attribute to False. This will stop DaCHS from attempting to add automatic #this and #preview links.

May occur in Element datalinkCore.

Atomic Children

- **code** (unicode string; defaults to <Not given/empty>) -- A python function body.

- **deprecated** (unicode string; defaults to None) -- A deprecation message. This will be shown if this procDef is being compiled.
- **doc** (unicode string; defaults to '') -- Human-readable docs for this proc (may be interpreted as restructured text).
- **name** (unicode string; defaults to <Not given/empty>) -- A name of the proc. ProcApps compute their (python) names to be somwhat random strings. Set a name manually to receive more easily decipherable error messages. If you do that, you have to care about name clashes yourself, though.
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **procDef** (id reference; defaults to <Not given/empty>) -- Reference to the procedure definition to apply
- **type** (One of: iterator, pargetter, regTest, rowfilter, dataFunction, descriptorGenerator, metaMaker, phraseMaker, mixinProc, dataFormatter, sourceFields, apply, t_t; defaults to None) -- The type of the procedure definition. The procedure applications will in general require certain types of definitions.

Structure Children

- bindings (contains Element bind and may be repeated zero or more times) -- Values for parameters of the procedure definition
- setups (contains Element setup and may be repeated zero or more times) -- Setup of the namespace the function will run in

Element fancyQueryCore

A core executing a pre-specified query with fancy conditions.

Unless you select *, you *must* define the outputTable here; Weird things will happen if you don't.

The queriedTable attribute is ignored.

Atomic Children

- **namePath** (id reference; defaults to None) -- Id of an element that will be used to located names in id references. Defaults to the queriedTable's id.
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **queriedTable** (id reference; defaults to <Undefined>) -- A reference to the table this core queries.
- **query** (unicode string; defaults to <Undefined>) -- The query to execute. It must contain exactly one %s where the generated where clause is to be inserted. Do not write WHERE yourself. All other percents must be escaped by doubling them.
- **timeout** (float; defaults to '5.0') -- Seconds until the query is aborted

Structure Children

- condDescs (contains Element condDesc and may be repeated zero or more times) -- Descriptions of the SQL and input generating entities for this core; if not given, they will be generated from the table columns.
- inputTable (contains Element inputTable) -- Description of the input data
- outputTable (contains Element outputTable) -- Table describing what fields are available from this core

Other Children

- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Element fixedQueryCore

A core executing a predefined query.

This usually is not what you want, unless you want to expose the current results of a specific query, e.g., for log or event data.

Atomic Children

- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **query** (unicode string; defaults to <Undefined>) -- The query to be executed. You must define the output fields in the core's output table. The query will be macro-expanded in the resource descriptor.
- **timeout** (float; defaults to '15.0') -- Seconds until the query is aborted
- **writable** (boolean; defaults to 'False') -- Use a writable DB connection?

Structure Children

- inputTable (contains Element inputTable) -- Description of the input data
- outputTable (contains Element outputTable) -- Table describing what fields are available from this core

Other Children

- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Element inputTable

an input for a core.

These aren't actually proper tables but actually just collection of the paramlike inputKeys. They serve as input declarations for cores and services (where services derive their inputTDs from the cores' ones by adapting them to the current renderer. Their main use is for the derivation of contextGrammars.

They can carry metadata, though, which is sometimes convenient when transporting information from the parameter parsers to the core.

For the typical dbCores (and friends), these are essentially never explicitly defined but rather derived from condDescs.

Do *not* read input values by using table.getParam. This will only give you one value when a parameter has been given multiple times. Instead, use the output

of the contextGrammar (inputParams in condDescs). Only there you will have the correct multiplicities.

May occur in Element scsCore, Element siapCutoutCore, Element customCore, Element nullCore, Element tapCore, Element productCore, Element adqlCore, Element pythonCore, Element registryCore, Element dbCore, Element fancyQueryCore, Element debugCore, Element datalinkCore, Element fixedQueryCore, Element uploadCore, Element ssapCore.

Atomic Children

- **exclusive** (boolean; defaults to 'False') -- If true, context grammars built from this will raise an error if contexts passed in have keys not defined by this table

Structure Children

- groups (contains Element group and may be repeated zero or more times) -- Groups of inputKeys (this is used for form UI formatting).
- inputKeys (contains Element inputKey and may be repeated zero or more times) -- Input parameters for this table.

Macros predefined here: Macro RSTservicelink, Macro RSTtable, Macro decapitalize, Macro getConfig, Macro internallink, Macro magicEmpty, Macro metaString, Macro quote, Macro rdId, Macro rdIdDotted, Macro schema, Macro sqlquote, Macro test, Macro today, Macro upper, Macro urlquote

Element metaMaker

A procedure application that generates metadata for datalink services.

The code must be generators (i.e., use yield statements) producing either svcs.InputKeys or protocols.datalink.LinkDef instances.

metaMaker see the data descriptor of the input data under the name descriptor.

The data attribute of the descriptor is always None for metaMakers, so you cannot use anything given there.

Within MetaMakers' code, you can access InputKey, Values, Option, and LinkDef without qualification, and there's the MS function to build structures. Hence, a metaMaker returning an InputKey could look like this:

```
<metaMaker>
  <code>
    yield MS(InputKey, name="format", type="text",
      description="Output format desired",
      values=MS(Values,
        options=[MS(Option, content_=descriptor.mime),
          MS(Option, content_="text/plain")]))
  </code>
</metaMaker>
```

(of course, you should give more metadata -- ucds, better description, etc) in production).

It's ok to yield None; this will suppress a Datalink and is convenient when some component further down figures out that a link doesn't exist (e.g., because a file isn't there). Note that in many cases, it's more helpful to client components to handle such situations by yielding a DatalinkFault.NotFoundFault.

In addition to the usual names available to ProcApps, meta makers have:

- MS -- function to make DaCHS structures
- InputKey -- the class to make for input parameters
- Values -- the class to make for input parameters' values attributes
- Options -- used by Values
- LinkDef -- a class to define further links within datalink services.
- DatalinkFault -- a container of datalink error generators
- soda -- the soda module.

May occur in Element datalinkCore.

Atomic Children

- **code** (unicode string; defaults to <Not given/empty>) -- A python function body.
- **deprecated** (unicode string; defaults to None) -- A deprecation message. This will be shown if this procDef is being compiled.
- **doc** (unicode string; defaults to '') -- Human-readable docs for this proc (may be interpreted as restructured text).

- **name** (unicode string; defaults to <Not given/empty>) -- A name of the proc. ProcApps compute their (python) names to be somwhat random strings. Set a name manually to receive more easily decipherable error messages. If you do that, you have to care about name clashes yourself, though.
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **procDef** (id reference; defaults to <Not given/empty>) -- Reference to the procedure definition to apply
- **type** (One of: iterator, pargetter, regTest, rowfilter, dataFunction, descriptorGenerator, metaMaker, phraseMaker, mixinProc, dataFormatter, sourceFields, apply, t_t; defaults to None) -- The type of the procedure definition. The procedure applications will in general require certain types of definitions.

Structure Children

- bindings (contains Element bind and may be repeated zero or more times) -- Values for parameters of the procedure definition
- setups (contains Element setup and may be repeated zero or more times) -- Setup of the namespace the function will run in

Element nullCore

A core always returning None.

This core will not work with the common renderers. It is really intended to go with coreless services (i.e. those in which the renderer computes everthing itself and never calls service.runX). As an example, the external renderer could go with this.

Atomic Children

- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.

Structure Children

- inputTable (contains Element inputTable) -- Description of the input data
- outputTable (contains Element outputTable) -- Table describing what fields are available from this core

Other Children

- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Element productCore

A core retrieving paths and/or data from the product table.

You will not usually mention this core in your RDs. It is mainly used internally to serve /getproduct queries.

It is instanciated from within //products.rd and relies on tables within that RD.

The input data consists of accref; you can use the string form of RAccrefs, and if you renderer wants, it can pass in ready-made RAccrefs. You can pass accrefs in through both an accref param and table rows.

The accref param is the normal way if you just want to retrieve a single image, the table case is for building tar files and such. There is one core instance in //products for each case.

The core returns a list of instances of a subclass of ProductBase above.

This core and its supporting machinery handles all the fancy product functionality (user autorisation, cutouts, ...).

Atomic Children

- **distinct** (boolean; defaults to 'False') -- Add a 'distinct' modifier to the query?
- **groupBy** (unicode string; defaults to None) -- A group by clause. You shouldn't generally need this, and if you use it, you must give an outputTable to your core.
- **limit** (integer; defaults to None) -- A pre-defined match limit (suppresses DB options widget).

- **namePath** (id reference; defaults to None) -- Id of an element that will be used to located names in id references. Defaults to the queriedTable's id.
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **queriedTable** (id reference; defaults to <Undefined>) -- A reference to the table this core queries.
- **sortKey** (unicode string; defaults to None) -- A pre-defined sort order (suppresses DB options widget). The sort key accepts multiple columns, separated by commas.

Structure Children

- condDescs (contains Element condDesc and may be repeated zero or more times) -- Descriptions of the SQL and input generating entities for this core; if not given, they will be generated from the table columns.
- inputTable (contains Element inputTable) -- Description of the input data
- outputTable (contains Element outputTable) -- Table describing what fields are available from this core

Other Children

- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Element pythonCore

A core doing computation using a piece of python.

See Python Cores instead of Custom Cores in the reference.

Atomic Children

- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.

Structure Children

- coreProc (contains Element coreProc) -- Code making the outputTable from the inputTable.
- inputTable (contains Element inputTable) -- Description of the input data
- outputTable (contains Element outputTable) -- Table describing what fields are available from this core

Other Children

- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Element registryCore

is a core processing OAI requests.

Its signature requires a single input key containing the complete args from the incoming request. This is necessary to satisfy the requirement of raising errors on duplicate arguments.

It returns an ElementTree.

This core is intended to work the the RegistryRenderer.

Atomic Children

- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.

Structure Children

- inputTable (contains Element inputTable) -- Description of the input data
- outputTable (contains Element outputTable) -- Table describing what fields are available from this core

Other Children

- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Element scsCore

A core performing cone searches.

This will, if it finds input parameters it can make out a position from, add a _r column giving the distance between the match center and the columns that a cone search will match against.

If any of the conditions for adding _r aren't met, this will silently degrade to a plain DBCore.

You will almost certainly want a:

```
<FEED source="//scs#coreDescs"/>
```

in the body of this (in addition to whatever other custom conditions you may have).

Atomic Children

- **distinct** (boolean; defaults to 'False') -- Add a 'distinct' modifier to the query?
- **groupBy** (unicode string; defaults to None) -- A group by clause. You shouldn't generally need this, and if you use it, you must give an output-Table to your core.
- **limit** (integer; defaults to None) -- A pre-defined match limit (suppresses DB options widget).
- **namePath** (id reference; defaults to None) -- Id of an element that will be used to located names in id references. Defaults to the queriedTable's id.
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **queriedTable** (id reference; defaults to <Undefined>) -- A reference to the table this core queries.
- **sortKey** (unicode string; defaults to None) -- A pre-defined sort order (suppresses DB options widget). The sort key accepts multiple columns, separated by commas.

Structure Children

- condDescs (contains Element condDesc and may be repeated zero or more times) -- Descriptions of the SQL and input generating entities for this core; if not given, they will be generated from the table columns.
- inputTable (contains Element inputTable) -- Description of the input data
- outputTable (contains Element outputTable) -- Table describing what fields are available from this core

Other Children

- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Element siapCutoutCore

A core doing SIAP plus cutouts.

It has, by default, an additional column specifying the desired size of the image to be retrieved. Based on this, the cutout core will tweak its output table such that references to cutout images will be retrieved.

The actual process of cutting out is performed by the product core and renderer.

Atomic Children

- **distinct** (boolean; defaults to 'False') -- Add a 'distinct' modifier to the query?
- **groupBy** (unicode string; defaults to None) -- A group by clause. You shouldn't generally need this, and if you use it, you must give an outputTable to your core.
- **limit** (integer; defaults to None) -- A pre-defined match limit (suppresses DB options widget).
- **namePath** (id reference; defaults to None) -- Id of an element that will be used to located names in id references. Defaults to the queriedTable's id.
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.

- **queriedTable** (id reference; defaults to <Undefined>) -- A reference to the table this core queries.
- **sortKey** (unicode string; defaults to None) -- A pre-defined sort order (suppresses DB options widget). The sort key accepts multiple columns, separated by commas.

Structure Children

- condDescs (contains Element condDesc and may be repeated zero or more times) -- Descriptions of the SQL and input generating entities for this core; if not given, they will be generated from the table columns.
- inputTable (contains Element inputTable) -- Description of the input data
- outputTable (contains Element outputTable) -- Table describing what fields are available from this core

Other Children

- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Element ssapCore

A core doing SSAP queries.

This core knows about metadata queries, version negotiation, and dispatches on REQUEST. Thus, it may return formatted XML data under certain circumstances.

Interpreted Properties:

- previews: If set to "auto", the core will automatically add a preview column and fill it with the URL of the products-based preview. Other values are not defined.

Atomic Children

- **distinct** (boolean; defaults to 'False') -- Add a 'distinct' modifier to the query?
- **groupBy** (unicode string; defaults to None) -- A group by clause. You shouldn't generally need this, and if you use it, you must give an outputTable to your core.
- **limit** (integer; defaults to None) -- A pre-defined match limit (suppresses DB options widget).
- **namePath** (id reference; defaults to None) -- Id of an element that will be used to located names in id references. Defaults to the queriedTable's id.
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.
- **queriedTable** (id reference; defaults to <Undefined>) -- A reference to the table this core queries.
- **sortKey** (unicode string; defaults to None) -- A pre-defined sort order (suppresses DB options widget). The sort key accepts multiple columns, separated by commas.

Structure Children

- condDescs (contains Element condDesc and may be repeated zero or more times) -- Descriptions of the SQL and input generating entities for this core; if not given, they will be generated from the table columns.
- inputTable (contains Element inputTable) -- Description of the input data
- outputTable (contains Element outputTable) -- Table describing what fields are available from this core

Other Children

- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Element tapCore

A core for the TAP renderer.

Right now, this is a no-op and not used by the renderer.

This will change as we move to regularise the TAP system.

Atomic Children

- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.

Structure Children

- inputTable (contains Element inputTable) -- Description of the input data
- outputTable (contains Element outputTable) -- Table describing what fields are available from this core

Other Children

- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Element uploadCore

A core handling uploads of files to the database.

It allows users to upload individual files into a special staging area (taken from the stagingDir property of the destination data descriptor) and causes these files to be parsed using destDD. Note that destDD *must* have `updating="True"` for this to work properly (it will otherwise drop the table on each update). If uploads are the only way updates into the table occur, source management is not necessary for these, though.

You can tell UploadCores to either insert or update the incoming data using the "mode" input key.

Atomic Children

- **destDD** (id reference; defaults to <Undefined>) -- Reference to the data we are uploading into. The destination must be an updating data descriptor.
- **original** (id reference; defaults to None) -- An id of an element to base the current one on. This provides a simple inheritance method. The general rules for advanced referencing in RDs apply.

Structure Children

- inputTable (contains Element inputTable) -- Description of the input data
- outputTable (contains Element outputTable) -- Table describing what fields are available from this core

Other Children

- **property** (mapping of user-defined keywords in the name attribute to string values) -- Properties (i.e., user-defined key-value pairs) for the element.

Predefined Macros

Macro expansions in DaCHS start with a backslash, arguments are given in curly braces. What macros are available depends on the element doing the expansion; regrettably, not all strings are expanded, and at this point it's not usually documented which are and which are not (though we hope DaCHS typically behaves "as expected"). If this bites you, complain to the authors and we promise we'll give fixing this a higher priority.

Macro RSTcc0

```
\RSTcc0{stuffDesignation}
```

expands to a declaration that stuffDesignation is available under CC-0.

This only works in reStructured text (though it's still almost readable as source).

Available in Element resource

Macro RSTccby

```
\RSTccby{stuffDesignation}
```

expands to a declaration that stuffDesignation is available under CC-BY.

This only works in reStructured text (though it's still almost readable as source).

Available in Element resource

Macro RSTccbysa

```
\RSTccbysa{stuffDesignation}
```

expands to a declaration that stuffDesignation is available under CC-BY-SA.

This only works in reStructured text (though it's still almost readable as source).

Available in Element resource

Macro RSTservicelink

```
\RSTservicelink{serviceId}{title=None}
```

a link to an internal service; id is <rdId>/<serviceId>/<renderer>, title, if given, is the anchor text.

The result is a link in the short form for restructured test.

Available in Element FEED, Element LFEED, Element LOOP, Element binaryGrammar, Element cdfHeaderGrammar, Element columnGrammar, Element contextGrammar, Element csvGrammar, Element customGrammar, Element dictlistGrammar, Element embeddedGrammar, Element fitsProdGrammar, Element fitsTableGrammar, Element freeREGrammar, Element inputTable, Element keyValueGrammar, Element mixinDef, Element mySQLDumpGrammar, Element nullGrammar, Element odbcGrammar, Element outputTable, Element pdsGrammar, Element reGrammar, Element resRec, Element resource, Element rowmaker, Element rowsetGrammar, Element service, Element table, Element transparentGrammar, Element voTableGrammar

Macro RSTtable

```
\RSTtable{tableName}
```

adds an reStructured test link to a tableName pointing to its table info.

Available in Element FEED, Element LFEED, Element LOOP, Element binaryGrammar, Element cdfHeaderGrammar, Element columnGrammar, Element contextGrammar, Element csvGrammar, Element customGrammar, Element dictlistGrammar, Element embeddedGrammar, Element fitsProdGrammar, Element fitsTableGrammar, Element freeREGrammar, Element inputTable, Element keyValueGrammar, Element mixinDef, Element mySQLDumpGrammar, Element nullGrammar, Element odbcGrammar, Element outputTable, Element pdsGrammar, Element reGrammar, Element resRec, Element resource, Element rowmaker, Element rowsetGrammar, Element service, Element table, Element transparentGrammar, Element voTableGrammar

Macro colNames

```
\colNames
```

returns an SQL-ready list of column names of this table.

Available in Element binaryGrammar, Element cdfHeaderGrammar, Element columnGrammar, Element contextGrammar, Element csvGrammar, Element customGrammar, Element dictlistGrammar, Element embeddedGrammar, Element fitsProdGrammar, Element fitsTableGrammar, Element freeREGrammar, Element keyValueGrammar, Element mySQLDumpGrammar, Element nullGrammar, Element odbcGrammar, Element outputTable, Element pdsGrammar, Element reGrammar, Element rowsetGrammar, Element table, Element transparentGrammar, Element voTableGrammar

Macro curtable

```
\curtable
```

returns the qualified name of the current table.

Available in Element outputTable, Element table

Macro decapitalize

```
\decapitalize{aString}
```

returns aString with the first character lowercased.

Available in Element FEED, Element LFEED, Element LOOP, Element binaryGrammar, Element cdfHeaderGrammar, Element columnGrammar, Element contextGrammar, Element csvGrammar, Element customGrammar, Element dictlistGrammar, Element embeddedGrammar, Element fitsProdGrammar, Element fitsTableGrammar, Element freeREGrammar, Element inputTable, Element keyValueGrammar, Element mixinDef, Element mySQLDumpGrammar, Element nullGrammar, Element odbcGrammar, Element outputTable, Element pdsGrammar, Element reGrammar, Element resRec, Element resource, Element rowmaker, Element rowsetGrammar, Element service, Element table, Element transparentGrammar, Element voTableGrammar

Macro dlMetaURI

```
\dlMetaURI{dlId}
```

returns a link to the datalink document for the current product.

This assumes you're assinging standard pubDIDs (see also standardPubDID, which is used by this).

dlId is the XML id of the datalink service, which is supposed to be in the sameRD as the rowmaker.

Available in Element binaryGrammar, Element cdfHeaderGrammar, Element columnGrammar, Element contextGrammar, Element csvGrammar, Element customGrammar, Element dictlistGrammar, Element embeddedGrammar, Element fitsProdGrammar, Element fitsTableGrammar, Element freeREGrammar, Element keyValueGrammar, Element mySQLDumpGrammar, Element nullGrammar, Element odbcGrammar, Element pdsGrammar, Element reGrammar, Element rowmaker, Element rowsetGrammar, Element transparentGrammar, Element voTableGrammar

Macro docField

```
\docField{name}
```

returns an expression giving the value of the column name in the document row.

Available in Element rowmaker

Macro fullDLURL

```
\fullDLURL{dlService}
```

returns a python expression giving a link to the full current data set retrieved through the datalink service.

You would write \fullDLURL{dlsvc} here, and the macro will expand into something like http://yourserver/currd/dlsvc/dlget?ID=ivo://whatever.

dlService is the id of the datalink service in the current RD.

This is intended for "virtual" data where the dataset is generated on the fly through datalink.

Available in Element binaryGrammar, Element cdfHeaderGrammar, Element columnGrammar, Element contextGrammar, Element csvGrammar, Element customGrammar, Element dictlistGrammar, Element embeddedGrammar, Element fitsProdGrammar, Element fitsTableGrammar, Element freeREGrammar, Element keyValueGrammar, Element mySQLDumpGrammar, Element nullGrammar, Element odbcGrammar, Element pdsGrammar, Element reGrammar, Element rowsetGrammar, Element transparentGrammar, Element voTableGrammar

Macro fullPath

```
\fullPath
```

returns an expression expanding to the full path of the current input file.

Available in Element rowmaker

Macro getConfig

```
\getConfig{section}{name=None}
```

the current value of configuration item {section}{name}.

You can also only give one argument to access settings from the general section.

Available in Element FEED, Element LFEED, Element LOOP, Element binaryGrammar, Element cdfHeaderGrammar, Element columnGrammar, Element contextGrammar, Element csvGrammar, Element customGrammar, Element

dictlistGrammar, Element embeddedGrammar, Element fitsProdGrammar, Element fitsTableGrammar, Element freeREGrammar, Element inputTable, Element keyValueGrammar, Element mixinDef, Element mySQLDumpGrammar, Element nullGrammar, Element odbcGrammar, Element outputTable, Element pdsGrammar, Element reGrammar, Element resRec, Element resource, Element rowmaker, Element rowsetGrammar, Element service, Element table, Element transparentGrammar, Element voTableGrammar

Macro getParam

```
\getParam{parName}{default=''}
```

returns the string representation of the parameter parName.

This is the parameter as given in the table definition. Any changes to an instance are not reflected here.

If the parameter named does not exist, an empty string is returned. NULLs/Nones are rendered as NULL; this is mainly a convenience for obscore-like applications and should not be exploited otherwise, since it's ugly and might change at some point.

If a default is given, it will be returned for both NULL and non-existing params.

Available in Element outputTable, Element table

Macro inputRelativePath

```
\inputRelativePath{liberalChars='True'}
```

returns an expression giving the current source's path relative to inputsDir

liberalChars can be a boolean literal (True, False, etc); if false, a value error is raised if characters that will result in trouble with the product mixin are within the result path.

In rowmakers fed by grammars with //products#define, better use @prodtblAccref.

Available in Element binaryGrammar, Element cdfHeaderGrammar, Element columnGrammar, Element contextGrammar, Element csvGrammar, Element customGrammar, Element dictlistGrammar, Element embeddedGrammar, Element fitsProdGrammar, Element fitsTableGrammar, Element freeREGrammar, Element keyValueGrammar, Element mySQLDumpGrammar, Element nullGrammar, Element odbcGrammar, Element pdsGrammar, Element reGrammar, Element rowmaker, Element rowsetGrammar, Element transparentGrammar, Element voTableGrammar

Macro inputSize

```
\inputSize
```

returns an expression giving the size of the current source.

Available in Element binaryGrammar, Element cdfHeaderGrammar, Element columnGrammar, Element contextGrammar, Element csvGrammar, Element customGrammar, Element dictlistGrammar, Element embeddedGrammar, Element fitsProdGrammar, Element fitsTableGrammar, Element freeREGrammar, Element keyValueGrammar, Element mySQLDumpGrammar, Element nullGrammar, Element odbcGrammar, Element pdsGrammar, Element reGrammar, Element rowmaker, Element rowsetGrammar, Element transparentGrammar, Element voTableGrammar

Macro internallink

```
\internallink{relPath}
```

an absolute URL from a path relative to the DC root.

Available in Element FEED, Element LFEED, Element LOOP, Element binaryGrammar, Element cdfHeaderGrammar, Element columnGrammar, Element contextGrammar, Element csvGrammar, Element customGrammar, Element dictlistGrammar, Element embeddedGrammar, Element fitsProdGrammar, Element fitsTableGrammar, Element freeREGrammar, Element inputTable, Element keyValueGrammar, Element mixinDef, Element mySQLDumpGrammar, Element nullGrammar, Element odbcGrammar, Element outputTable, Element pdsGrammar, Element reGrammar, Element resRec, Element resource, Element rowmaker, Element rowsetGrammar, Element service, Element table, Element transparentGrammar, Element voTableGrammar

Macro lastSourceElements

```
\lastSourceElements{numElements}
```

returns an expression calling rmkfuncs.lastSourceElements on the current input path.

Available in Element binaryGrammar, Element cdfHeaderGrammar, Element columnGrammar, Element contextGrammar, Element csvGrammar, Element customGrammar, Element dictlistGrammar, Element embeddedGrammar, Element fitsProdGrammar, Element fitsTableGrammar, Element freeREGrammar,

Element keyValueGrammar, Element mySQLDumpGrammar, Element nullGrammar, Element odbcGrammar, Element pdsGrammar, Element reGrammar, Element rowmaker, Element rowsetGrammar, Element transparentGrammar, Element voTableGrammar

Macro magicEmpty

```
\magicEmpty{val}
```

returns ___EMPTY___ if val is empty.

This is necessary when feeding possibly empty params from mixin parameters (don't worry if you don't understand this).

Available in Element FEED, Element LFEED, Element LOOP, Element binaryGrammar, Element cdfHeaderGrammar, Element columnGrammar, Element contextGrammar, Element csvGrammar, Element customGrammar, Element dictlistGrammar, Element embeddedGrammar, Element fitsProdGrammar, Element fitsTableGrammar, Element freeREGrammar, Element inputTable, Element keyValueGrammar, Element mixinDef, Element mySQLDumpGrammar, Element nullGrammar, Element odbcGrammar, Element outputTable, Element pdsGrammar, Element reGrammar, Element resRec, Element resource, Element rowmaker, Element rowsetGrammar, Element service, Element table, Element transparentGrammar, Element voTableGrammar

Macro metaString

```
\metaString{metaKey}{default=None}
```

the value of metaKey on the macro expander.

This will raise an error when the meta Key is not available unless you give a default.

Available in Element FEED, Element LFEED, Element LOOP, Element binaryGrammar, Element cdfHeaderGrammar, Element columnGrammar, Element contextGrammar, Element csvGrammar, Element customGrammar, Element dictlistGrammar, Element embeddedGrammar, Element fitsProdGrammar, Element fitsTableGrammar, Element freeREGrammar, Element inputTable, Element keyValueGrammar, Element mixinDef, Element mySQLDumpGrammar, Element nullGrammar, Element odbcGrammar, Element outputTable, Element pdsGrammar, Element reGrammar, Element resRec, Element resource, Element rowmaker, Element rowsetGrammar, Element service, Element table, Element transparentGrammar, Element voTableGrammar

Macro nameForUCD

```
\nameForUCD{ucd}
```

returns the (unique!) name of the field having ucd in this table.

If there is no or more than one field with the ucd in this table, we raise a ValueError.

Available in Element outputTable, Element table

Macro nameForUCDs

```
\nameForUCDs{ucds}
```

returns the (unique!) name of the field having one of ucds in this table.

Ucds is a selection of ucds separated by vertical bars (|). The rules for when this raises errors are so crazy you don't want to think about them. This really is only intended for cases where "old" and "new" standards are to be supported, like with pos.eq.*;meta.main and POS_EQ_*_MAIN.

If there is no or more than one field with the ucd in this table, we raise an exception.

Available in Element outputTable, Element table

Macro property

```
\property{propName}
```

returns an expression giving the value of the property propName on the current DD.

Available in Element binaryGrammar, Element cdfHeaderGrammar, Element columnGrammar, Element contextGrammar, Element csvGrammar, Element customGrammar, Element dictlistGrammar, Element embeddedGrammar, Element fitsProdGrammar, Element fitsTableGrammar, Element freeREGrammar, Element keyValueGrammar, Element mySQLDumpGrammar, Element nullGrammar, Element odbcGrammar, Element pdsGrammar, Element reGrammar, Element rowmaker, Element rowsetGrammar, Element transparentGrammar, Element voTableGrammar

Macro qName

```
\qName
```

returns the qName of the table we are currently parsing into.

Available in Element outputTable, Element rowmaker, Element table

Macro quote

```
\quote{arg}
```

returns the argument in quotes (with internal quotes backslash-escaped if necessary).

Available in Element FEED, Element LFEED, Element LOOP, Element binaryGrammar, Element cdfHeaderGrammar, Element columnGrammar, Element contextGrammar, Element csvGrammar, Element customGrammar, Element dictlistGrammar, Element embeddedGrammar, Element fitsProdGrammar, Element fitsTableGrammar, Element freeREGrammar, Element inputTable, Element keyValueGrammar, Element mixinDef, Element mySQLDumpGrammar, Element nullGrammar, Element odbcGrammar, Element outputTable, Element pdsGrammar, Element reGrammar, Element resRec, Element resource, Element rowmaker, Element rowsetGrammar, Element service, Element table, Element transparentGrammar, Element voTableGrammar

Macro rdId

```
\rdId
```

the identifier of the current resource descriptor.

Available in Element FEED, Element LFEED, Element LOOP, Element binaryGrammar, Element cdfHeaderGrammar, Element columnGrammar, Element contextGrammar, Element csvGrammar, Element customGrammar, Element dictlistGrammar, Element embeddedGrammar, Element fitsProdGrammar, Element fitsTableGrammar, Element freeREGrammar, Element inputTable, Element keyValueGrammar, Element mixinDef, Element mySQLDumpGrammar, Element nullGrammar, Element odbcGrammar, Element outputTable, Element pdsGrammar, Element reGrammar, Element resRec, Element resource, Element rowmaker, Element rowsetGrammar, Element service, Element table, Element transparentGrammar, Element voTableGrammar

Macro rdIdDotted

```
\rdIdDotted
```

the identifier for the current resource descriptor with slashes replaced with dots (so they work as the "host part" in URIs.

Available in Element FEED, Element LFEED, Element LOOP, Element binaryGrammar, Element cdfHeaderGrammar, Element columnGrammar, Element contextGrammar, Element csvGrammar, Element customGrammar, Element dictlistGrammar, Element embeddedGrammar, Element fitsProdGrammar, Element fitsTableGrammar, Element freeREGrammar, Element inputTable, Element keyValueGrammar, Element mixinDef, Element mySQLDumpGrammar, Element nullGrammar, Element odbcGrammar, Element outputTable, Element pdsGrammar, Element reGrammar, Element resRec, Element resource, Element rowmaker, Element rowsetGrammar, Element service, Element table, Element transparentGrammar, Element voTableGrammar

Macro rootlessPath

```
\rootlessPath
```

returns an expression giving the current source's path with the resource descriptor's root removed.

Available in Element binaryGrammar, Element cdfHeaderGrammar, Element columnGrammar, Element contextGrammar, Element csvGrammar, Element customGrammar, Element dictlistGrammar, Element embeddedGrammar, Element fitsProdGrammar, Element fitsTableGrammar, Element freeREGrammar, Element keyValueGrammar, Element mySQLDumpGrammar, Element nullGrammar, Element odbcGrammar, Element pdsGrammar, Element reGrammar, Element rowmaker, Element rowsetGrammar, Element transparentGrammar, Element voTableGrammar

Macro rowsMade

```
\rowsMade
```

returns an expression giving the number of records already returned by this row maker.

This number excludes failed and skipped rows.

Available in Element rowmaker

Macro rowsProcessed

```
\rowsProcessed
```

returns an expression giving the number of records already delivered by the grammar.

Available in Element binaryGrammar, Element cdfHeaderGrammar, Element columnGrammar, Element contextGrammar, Element csvGrammar, Element customGrammar, Element dictlistGrammar, Element embeddedGrammar, Element fitsProdGrammar, Element fitsTableGrammar, Element freeREGrammar, Element keyValueGrammar, Element mySQLDumpGrammar, Element nullGrammar, Element odbcGrammar, Element pdsGrammar, Element reGrammar, Element rowmaker, Element rowsetGrammar, Element transparentGrammar, Element voTableGrammar

Macro schema

```
\schema
```

the schema of the current resource descriptor.

Available in Element FEED, Element LFEED, Element LOOP, Element binaryGrammar, Element cdfHeaderGrammar, Element columnGrammar, Element contextGrammar, Element csvGrammar, Element customGrammar, Element dictlistGrammar, Element embeddedGrammar, Element fitsProdGrammar, Element fitsTableGrammar, Element freeREGrammar, Element inputTable, Element keyValueGrammar, Element mixinDef, Element mySQLDumpGrammar, Element nullGrammar, Element odbcGrammar, Element outputTable, Element pdsGrammar, Element reGrammar, Element resRec, Element resource, Element rowmaker, Element rowsetGrammar, Element service, Element table, Element transparentGrammar, Element voTableGrammar

Macro sourceCDate

```
\sourceCDate
```

returns an expression giving the timestamp for the create date of the current source.

Use dateTimeToJdn or dateTimeToMJD to turn this into JD or MJD (which is usually preferred in database tables). See also the sourceDate macro.

Available in Element rowmaker

Macro sourceDate

```
\sourceDate
```

returns an expression giving the timestamp of the current source.

This is a timestamp of the modification date; use dateTimeToJdn or dateTimeToMJD to turn this into JD or MJD (which is usually preferred in database tables). See also the sourceCDate macro.

Available in Element binaryGrammar, Element cdfHeaderGrammar, Element columnGrammar, Element contextGrammar, Element csvGrammar, Element customGrammar, Element dictlistGrammar, Element embeddedGrammar, Element fitsProdGrammar, Element fitsTableGrammar, Element freeREGrammar, Element keyValueGrammar, Element mySQLDumpGrammar, Element nullGrammar, Element odbcGrammar, Element pdsGrammar, Element reGrammar, Element rowmaker, Element rowsetGrammar, Element transparentGrammar, Element voTableGrammar

Macro splitPreviewPath

```
\splitPreviewPath{ext}
```

returns an expression for the split standard path for a custom preview.

As standardPreviewPath, except that the directory hierarchy of the data files will be reproduced in previews. For ext, you should typically pass the extension appropriate for the preview (like {.png} or {.jpeg}).

See the introduction to custom previews for details.

Available in Element binaryGrammar, Element cdfHeaderGrammar, Element columnGrammar, Element contextGrammar, Element csvGrammar, Element customGrammar, Element dictlistGrammar, Element embeddedGrammar, Element fitsProdGrammar, Element fitsTableGrammar, Element freeREGrammar, Element keyValueGrammar, Element mySQLDumpGrammar, Element nullGrammar, Element odbcGrammar, Element pdsGrammar, Element reGrammar, Element rowsetGrammar, Element transparentGrammar, Element voTableGrammar

Macro sqlquote

```
\sqlquote{arg}
```

returns the argument as a quoted string, unless it is 'NULL' or None, in which case just NULL is returned.

Available in Element FEED, Element LFEED, Element LOOP, Element binaryGrammar, Element cdfHeaderGrammar, Element columnGrammar, Element contextGrammar, Element csvGrammar, Element customGrammar, Element dictlistGrammar, Element embeddedGrammar, Element fitsProdGrammar, Element fitsTableGrammar, Element freeREGrammar, Element inputTable, Element keyValueGrammar, Element mixinDef, Element mySQLDumpGrammar, Element nullGrammar, Element odbcGrammar, Element outputTable, Element pdsGrammar, Element reGrammar, Element resRec, Element resource, Element rowmaker, Element rowsetGrammar, Element service, Element table, Element transparentGrammar, Element voTableGrammar

Macro srcstem

```
\srcstem
```

returns python code for the stem of the source file currently parsed in a rowmaker.

Example: if you're currently parsing /tmp/foo.bar.gz, the stem is foo.

Available in Element binaryGrammar, Element cdfHeaderGrammar, Element columnGrammar, Element contextGrammar, Element csvGrammar, Element customGrammar, Element dictlistGrammar, Element embeddedGrammar, Element fitsProdGrammar, Element fitsTableGrammar, Element freeREGrammar, Element keyValueGrammar, Element mySQLDumpGrammar, Element nullGrammar, Element odbcGrammar, Element pdsGrammar, Element reGrammar, Element rowmaker, Element rowsetGrammar, Element transparentGrammar, Element voTableGrammar

Macro standardPreviewPath

```
\standardPreviewPath
```

returns an expression for the standard path for a custom preview.

This consists of resdir, the name of the previewDir property on the embedding DD, and the flat name of the accref (which this macro assumes to see in its

namespace as accref; this is usually the case in //products#define, which is where this macro would typically be used).

As an alternative, there is the splitPreviewPath macro, which does not mogrify the file name. In particular, do not use standardPreviewPath when you have more than a few 1e4 files, as it will have all these files in a single, flat directory, and that can become a chore.

See the introduction to custom previews for details.

Available in Element binaryGrammar, Element cdfHeaderGrammar, Element columnGrammar, Element contextGrammar, Element csvGrammar, Element customGrammar, Element dictlistGrammar, Element embeddedGrammar, Element fitsProdGrammar, Element fitsTableGrammar, Element freeREGrammar, Element keyValueGrammar, Element mySQLDumpGrammar, Element nullGrammar, Element odbcGrammar, Element pdsGrammar, Element reGrammar, Element rowsetGrammar, Element transparentGrammar, Element voTableGrammar

Macro standardPubDID

```
\standardPubDID
```

returns the "standard publisher DID" for the current product.

The publisher dataset identifier (PubDID) is important in protocols like SSAP and obscore. If you use this macro, the PubDID will be your authority, the path compontent ~, and the current value of @prodtblAccref. It thus will only work where products#define (or a replacement) is in action. If it isn't, a normal function call getStandardPubDID(\\inputRelativePath) would be an obvious alternative.

You *can* of course define your PubDIDs in a different way.

Available in Element rowmaker

Macro tablename

```
\tablename
```

returns the unqualified name of the current table.

Available in Element outputTable, Element table

Macro tablesForTAP

```
\tablesForTAP
```

undocumented Available in Element service

Macro test

```
\test{*args}
```

always "test macro expansion".

Available in Element FEED, Element LFEED, Element LOOP, Element binaryGrammar, Element cdfHeaderGrammar, Element columnGrammar, Element contextGrammar, Element csvGrammar, Element customGrammar, Element dictlistGrammar, Element embeddedGrammar, Element fitsProdGrammar, Element fitsTableGrammar, Element freeREGrammar, Element inputTable, Element keyValueGrammar, Element mixinDef, Element mySQLDumpGrammar, Element nullGrammar, Element odbcGrammar, Element outputTable, Element pdsGrammar, Element reGrammar, Element resRec, Element resource, Element rowmaker, Element rowsetGrammar, Element service, Element table, Element transparentGrammar, Element voTableGrammar

Macro today

```
\today
```

today's date in ISO representation.

Available in Element FEED, Element LFEED, Element LOOP, Element binaryGrammar, Element cdfHeaderGrammar, Element columnGrammar, Element contextGrammar, Element csvGrammar, Element customGrammar, Element dictlistGrammar, Element embeddedGrammar, Element fitsProdGrammar, Element fitsTableGrammar, Element freeREGrammar, Element inputTable, Element keyValueGrammar, Element mixinDef, Element mySQLDumpGrammar, Element nullGrammar, Element odbcGrammar, Element outputTable, Element pdsGrammar, Element reGrammar, Element resRec, Element resource, Element rowmaker, Element rowsetGrammar, Element service, Element table, Element transparentGrammar, Element voTableGrammar

Macro upper

```
\upper{aString}
```

returns aString uppercased.

There's no guarantees for characters outside ASCII.

Available in Element FEED, Element LFEED, Element LOOP, Element binaryGrammar, Element cdfHeaderGrammar, Element columnGrammar, Element contextGrammar, Element csvGrammar, Element customGrammar, Element dictlistGrammar, Element embeddedGrammar, Element fitsProdGrammar, Element fitsTableGrammar, Element freeREGrammar, Element inputTable, Element keyValueGrammar, Element mixinDef, Element mySQLDumpGrammar, Element nullGrammar, Element odbcGrammar, Element outputTable, Element pdsGrammar, Element reGrammar, Element resRec, Element resource, Element rowmaker, Element rowsetGrammar, Element service, Element table, Element transparentGrammar, Element voTableGrammar

Macro urlquote

```
\urlquote{string}
```

wraps urllib.quote.

Available in Element FEED, Element LFEED, Element LOOP, Element binaryGrammar, Element cdfHeaderGrammar, Element columnGrammar, Element contextGrammar, Element csvGrammar, Element customGrammar, Element dictlistGrammar, Element embeddedGrammar, Element fitsProdGrammar, Element fitsTableGrammar, Element freeREGrammar, Element inputTable, Element keyValueGrammar, Element mixinDef, Element mySQLDumpGrammar, Element nullGrammar, Element odbcGrammar, Element outputTable, Element pdsGrammar, Element reGrammar, Element resRec, Element resource, Element rowmaker, Element rowsetGrammar, Element service, Element table, Element transparentGrammar, Element voTableGrammar

Mixins

Mixins ensure a certain functionality on a table. Typically, this is used to provide certain guaranteed fields to particular cores. For many mixins, there are predefined procedures (both rowmaker applys and grammar rowfilters) that should be used in grammars and/or rowmakers feeding the tables mixing in a given mixin.

The //epntap2#localfile-2_0 Mixin

Use this mixin if your epntap table is filled with local products (i.e., sources matches files on your hard disk that DaCHS should hand out itself). This will arrange for your products to be entered into the products table, and it will automatically compute file size, etc.

This wants a //products#define rowfilter in your grammar and a //epntap2#populate-localfile-2_0 apply in your rowmaker.

The //epntap2#table-2_0 Mixin

This mixin defines a table suitable for publication via the EPN-TAP protocol.

According to the standard definition, tables mixing this in should be called `epn_core`. The mixin already arranges for the table to be accessible by ADQL and be on disk.

This also causes the product table to be populated. This means that grammars feeding such tables need a //products#define row filter. At the very least, you need to say:

```
<rowfilter procDef="//products#define">
  <bind name="table">"\schema.epn_core"</bind>
</rowfilter>
```

If you absolutely cannot use //products#define, you will hve to manually provide the prodtblFsize (file size in *bytes*), prodtblAccref (product URL), and prodtblPreview (thumbnail image or None) keys in what's coming from your grammar.

Use the //epntap2#populate-2_0 apply in rowmakers feeding tables mixing this in.

This mixin has the following parameters:

Parameter *optional_columns* Space-separated list of names of optional columns to include. Column names available include access_url access_format access_estsize access_md5 thumbnail_url file_name species filter alt_target_name target_region feature_name bib_reference publisher spatial_coordinate_description spatial_origin time_origin time_scale

Parameter *spatial_frame_type* Flavour of the coordinate system. Since this determines the units of the coordinates columns, this must be set globally for the entire dataset. Values defined by EPN-TAP and understood by this mixin include celestial, body, cartesian, cylindrical, spherical, healpix.

The //obscore#publish Mixin

Publish this table to ObsTAP.

This means mapping or giving quite a bit of data from the present table to ObsCore rows. Internally, this information is converted to an SQL select statement used within a create view statement. In consequence, you must give *SQL* expressions in the parameter values; just naked column names from your input table are ok, of course. Most parameters are set to NULL or appropriate defaults for tables mixing in //products#table.

Since the mixin generates script elements, it cannot be used in untrusted RDs. The fact that you can enter raw SQL also means you will get ugly error messages if you give invalid parameters.

Some items are filled from product interface fields automatically. You must change these if you obscore-publish tables not mixin in products.

Note: you must say `dachs imp //obscore` before anything obscore-related will work.

This mixin has the following parameters:

Parameter *accessURL* defaults to `accref`; URL at which the product can be obtained. Leave as is for tables mixing in products.

Parameter *calibLevel* defaults to `0`; Calibration level of data, a number between 0 and 3; for details, see http://dc.g-vo.org/tableinfo/ivoa.obscore#note-calib

Parameter *collectionName* defaults to `'unnamed'`; A human-readable name for this collection. This should be short, so don't just use the resource title

Parameter *coverage* defaults to `NULL`; A polygon giving the spatial coverage of the data set; this must always be in ICRS. This is cast to an pgsphere spoly, which currently means that you have to provide an spoly (reference), too.

Parameter *creatorDID* defaults to `NULL`; Global identifier of the data set assigned by the creator. Leave NULL unless the creator actually assigned an IVO id herself.

Parameter *dec* defaults to `NULL`; Center Dec

Parameter *did* defaults to `$COMPUTE`; Global identifier of the data set. Leave $COMPUTE for tables mixing in products.

Parameter *emMax* defaults to NULL; Upper bound of wavelengths represented in the data set, in meters.

Parameter *emMin* defaults to NULL; Lower bound of wavelengths represented in the data set, in meters.

Parameter *emResPower* defaults to NULL; Spectral resolution as lambda/delta lambda

Parameter *emUCD* defaults to NULL; UCD of the spectral axis as defined by the spectrum DM, plus a few values defined in obscore 1.1 for Doppler axes

Parameter *emXel* defaults to NULL; Number of samples along the spectral axis

Parameter *expTime* defaults to NULL; Total time of event counting. This simply is tMax-tMin for simple exposures.

Parameter *facilityName* defaults to NULL; The institute or observatory at which the data was produced

Parameter *fov* defaults to NULL; Approximate diameter of region covered

Parameter *instrumentName* defaults to NULL; The instrument that produced the data

Parameter *mime* defaults to mime; The MIME type of the product file. Only touch if you do not mix in products.

Parameter *oUCD* defaults to NULL; UCD of the observable quantity, e.g., em.opt for wide-band optical frames.

Parameter *obsId* defaults to accref; Identifier of the data set. Only change this when you do not mix in products.

Parameter *polStates* defaults to NULL; List of polarization states present in the data; if you give something, use the convention of choosing the appropriate from {I Q U V RR LL RL LR XX YY XY YX POLI POLA} and write them *in alphabetical order* with / separators, e.g. /I/Q/XX/.

Parameter *polXel* defaults to NULL; Number of polarisation states in this product

Parameter *productSubtype* defaults to NULL; File subtype. Details pending

Parameter *productType* Data product type; one of image, cube, spectrum, sed, timeseries, visibility, event, or NULL if None of the above

Parameter *ra* defaults to NULL; Center RA

Parameter *sPixelScale* defaults to NULL; Size of a spatial pixel (in arcsec)

Parameter *sResolution* defaults to NULL; The (best) angular resolution within the data set, in arcsecs

Parameter *sXel1* defaults to NULL; Number of pixels along the first spatial axis

Parameter *sXel2* defaults to NULL; Number of pixels along the second spatial axis

Parameter *size* defaults to accsize/1024; The estimated size of the product in kilobytes. Only touch when you do not mix in products#table.

Parameter *tMax* defaults to NULL; MJD for the upper bound of times covered in the data set. See tMin

Parameter *tMin* defaults to NULL; MJD for the lower bound of times covered in the data set (e.g. start of exposure). Use ts_to_mjd(ts) to get this from a postgres timestamp.

Parameter *tResolution* defaults to NULL; Temporal resolution

Parameter *tXel* defaults to NULL; Number of samples along the time axis

Parameter *targetClass* defaults to NULL; Class of target object(s). You should take whatever you put here from http://simbad.u-strasbg.fr/guide/chF.htx

Parameter *targetName* defaults to NULL; Name of the target object.

Parameter *title* defaults to NULL; A human-readable title of the data set.

The //obscore#publishSIAP Mixin

Publish a PGS SIAP table to ObsTAP.

This works like //obscore#publish except some defaults apply that copy fields that work analoguously in SIAP and in ObsTAP.

For special situations, you can, of course, override any of the parameters, but most of them should already be all right. To find out what the parameters described as "preset for SIAP" mean, refer to //obscore#publish.

Note: you must say `dachs imp //obscore` before anything obscore-related will work.

This mixin has the following parameters:

Parameter *accessURL* defaults to `accref`; URL at which the product can be obtained. Leave as is for tables mixing in products.

Parameter *calibLevel* defaults to `0`; Calibration level of data, a number between 0 and 3; for details, see http://dc.g-vo.org/tableinfo/ivoa.obscore#note-calib

Parameter *collectionName* defaults to `'unnamed'`; A human-readable name for this collection. This should be short, so don't just use the resource title

Parameter *coverage* defaults to `coverage`; preset for SIAP

Parameter *creatorDID* defaults to `NULL`; Global identifier of the data set assigned by the creator. Leave NULL unless the creator actually assigned an IVO id herself.

Parameter *dec* defaults to `centerDelta`; preset for SIAP

Parameter *did* defaults to `$COMPUTE`; Global identifier of the data set. Leave $COMPUTE for tables mixing in products.

Parameter *emMax* defaults to `bandpassHi`; preset for SIAP

Parameter *emMin* defaults to `bandpassLo`; preset for SIAP

Parameter *emResPower* defaults to `NULL`; Spectral resolution as lambda/delta lambda

Parameter *emUCD* defaults to `NULL`; UCD of the spectral axis as defined by the spectrum DM, plus a few values defined in obscore 1.1 for Doppler axes

Parameter *emXel* defaults to `NULL`; Number of samples along the spectral axis

Parameter *expTime* defaults to `NULL`; Total time of event counting. This simply is tMax-tMin for simple exposures.

Parameter *facilityName* defaults to `NULL`; The institute or observatory at which the data was produced

Parameter *fov* defaults to `pixelScale[1]*pixelSize[1]`; preset for SIAP; we use the extent along the X axis as a very rough estimate for the size. If you can do better, by all means do.

Parameter *instrumentName* defaults to `instId`; The instrument that produced the data

Parameter *mime* defaults to `mime`; The MIME type of the product file. Only touch if you do not mix in products.

Parameter *oUCD* defaults to `'em.opt'`; preset for SIAP; fix if you either know more about the band of if your images are not in the optical.

Parameter *obsId* defaults to `accref`; Identifier of the data set. Only change this when you do not mix in products.

Parameter *polStates* defaults to `NULL`; List of polarization states present in the data; if you give something, use the convention of choosing the appropriate from {I Q U V RR LL RL LR XX YY XY YX POLI POLA} and write them *in alphabetical order* with / separators, e.g. /I/Q/XX/.

Parameter *polXel* defaults to `NULL`; Number of polarisation states in this product

Parameter *productSubtype* defaults to `NULL`; File subtype. Details pending

Parameter *productType* defaults to `'image'`; preset for SIAP

Parameter *ra* defaults to `centerAlpha`; preset for SIAP

Parameter *sPixelScale* defaults to `pixelScale[0]/3600`; preset for SIAP

Parameter *sResolution* defaults to `pixelScale[1]*3600`; preset for SIAP; this is just the pixel scale in one dimension. If that's seriously wrong or you have uncalibrated images in your collection, you may need to be more careful here.

Parameter *sXel1* defaults to `pixelSize[1]`; preset for SIAP

Parameter *sXel2* defaults to `pixelSize[2]`; preset for SIAP

Parameter *size* defaults to `accsize/1024`; The estimated size of the product in kilobytes. Only touch when you do not mix in products#table.

Parameter *tMax* defaults to `dateObs`; preset for SIAP; if you want, change this to end of observation as available.

Parameter *tMin* defaults to `dateObs`; preset for SIAP; if you want, change this to start of observation as available.

Parameter *tResolution* defaults to `NULL`; Temporal resolution

Parameter *tXel* defaults to `NULL`; Number of samples along the time axis

Parameter *targetClass* defaults to `NULL`; Class of target object(s). You should take whatever you put here from http://simbad.u-strasbg.fr/guide/chF.htx

Parameter *targetName* defaults to `NULL`; Name of the target object.

Parameter *title* defaults to `imageTitle`; preset for SIAP

The //obscore#publishSSAPHCD Mixin

Publish a table mixing in //ssap#hcd to ObsTAP. Since //ssap#hcd is deprecated, this should not be used in new RDs, either. For //ssap#mixc tables, use publishSSAPMIXC.

This works like the //obscore#publish mixin except some defaults apply that copy fields that work analoguously in SSAP and in ObsTAP.

The columns already set in SSAP are marked as UNDOCUMENTED in the parameter list below. For special situations, you can, of course, override any of the parameters. To find out what they actually mean, mean, refer to the //obscore#publish mixin.

Note that this mixin does *not* set coverage (obscore: s_region). This is because although we could make a circle from ssa_location and ssa_aperture, circles are not allowed in DaCHS' s_region (which has a fixed type of spoly). The recommended solution to still have s_region is to add (and index) a custom field in the ssa table and compute some sort of spolys for the coverage.

Note: you must say `dachs imp //obscore` before anything obscore-related will work.

This mixin has the following parameters:

Parameter *accessURL* defaults to `accref`; URL at which the product can be obtained. Leave as is for tables mixing in products.

Parameter *calibLevel* defaults to `0`; Calibration level of data, a number between 0 and 3; for details, see http://dc.g-vo.org/tableinfo/ivoa.obscore#note-calib

Parameter *collectionName*

defaults to `\sqlquote{\getParam{ssa_collection}{NULL}}`; UNDOCUMENTED

Parameter *coverage* defaults to `NULL`; Use ssa_region when the table also mixes in //ssap#simpleCoverage

Parameter *creatorDID*

defaults to `ssa_creatorDID`; UNDOCUMENTED

Parameter *dec*

defaults to `degrees(lat(ssa_location))`; UNDOCUMENTED

Parameter *did* defaults to `$COMPUTE`; Global identifier of the data set. Leave $COMPUTE for tables mixing in products.

Parameter *emMax*

defaults to `ssa_specend`; UNDOCUMENTED

Parameter *emMin*

defaults to `ssa_specstart`; UNDOCUMENTED

Parameter *emResPower* defaults to `NULL`; Spectral resolution as lambda/delta lambda

Parameter *emUCD*

defaults to `\sqlquote{\getParam{ssa_spectralucd}};
UNDOCUMENTED

Parameter *emXel* defaults to `NULL`; Number of samples along the spectral axis

Parameter *expTime*

defaults to `ssa_timeExt`; UNDOCUMENTED

Parameter *facilityName* defaults to `NULL`; The institute or observatory at which the data was produced

Parameter *fov*

defaults to `ssa_aperture`; UNDOCUMENTED

Parameter *instrumentName*

defaults to `\sqlquote{\getParam{ssa_instrument}{NULL}};
UNDOCUMENTED

Parameter *mime* defaults to `mime`; The MIME type of the product file. Only touch if you do not mix in products.

Parameter *oUCD*

defaults to `\sqlquote{\getParam{ssa_fluxucd}}; UNDOCUMENTED

Parameter *obsId* defaults to `accref`; Identifier of the data set. Only change this when you do not mix in products.

Parameter *polStates* defaults to `NULL`; List of polarization states present in the data; if you give something, use the convention of choosing the appropriate from {I Q U V RR LL RL LR XX YY XY YX POLI POLA} and write them *in alphabetical order* with / separators, e.g. /I/Q/XX/.

Parameter *polXel* defaults to NULL; Number of polarisation states in this product

Parameter *productSubtype* defaults to NULL; File subtype. Details pending

Parameter *productType*

defaults to `'spectrum'`; UNDOCUMENTED

Parameter *ra*

defaults to `degrees(long(ssa_location))`; UNDOCUMENTED

Parameter *sPixelScale* defaults to NULL; Size of a spatial pixel (in arcsec)

Parameter *sResolution*

defaults to `\getParam{ssa_spaceRes}{NULL}/3600.`; UNDOCUMENTED

Parameter *sXel1* defaults to NULL; Number of pixels along the first spatial axis

Parameter *sXel2* defaults to NULL; Number of pixels along the second spatial axis

Parameter *size* defaults to `accsize/1024`; The estimated size of the product in kilobytes. Only touch when you do not mix in products#table.

Parameter *tMax*

defaults to `ssa_dateObs+ssa_timeExt/43200.`; UNDOCUMENTED

Parameter *tMin*

defaults to `ssa_dateObs-ssa_timeExt/43200.`; UNDOCUMENTED

Parameter *tResolution* defaults to NULL; Temporal resolution

Parameter *tXel* defaults to NULL; Number of samples along the time axis

Parameter *targetClass*

defaults to `ssa_targclass`; UNDOCUMENTED

Parameter *targetName*

defaults to `ssa_targname`; UNDOCUMENTED

Parameter *title*

defaults to `ssa_dstitle`; UNDOCUMENTED

The //obscore#publishSSAPMIXC Mixin

Publish a table mixing in //ssap#mixc to ObsTAP.

This works like the //obscore#publish mixin except some defaults apply that copy fields that work analoguously in SSAP and in ObsTAP.

The columns already set in SSAP are marked as UNDOCUMENTED in the parameter list below. For special situations, you can, of course, override any of the parameters. To find out what they actually mean, mean, refer to the //obscore#publish mixin.

Note that this mixin does *not* set coverage (obscore: s_region). This is because although we could make a circle from ssa_location and ssa_aperture, circles are not allowed in DaCHS' s_region (which has a fixed type of spoly). The recommended solution to still have s_region is to add (and index) a custom field; the //ssap#simpleCoverage will do this.

Note: you must say `dachs imp //obscore` before anything obscore-related will work.

This mixin has the following parameters:

Parameter *accessURL* defaults to `accref`; URL at which the product can be obtained. Leave as is for tables mixing in products.

Parameter *calibLevel* defaults to `0`; Calibration level of data, a number between 0 and 3; for details, see http://dc.g-vo.org/tableinfo/ivoa.obscore#note-calib

Parameter *collectionName*

defaults to `ssa_collection`; UNDOCUMENTED

Parameter *coverage* defaults to `NULL`; Use ssa_region when the table also mixes in //ssap#simpleCoverage

Parameter *creatorDID*

defaults to `ssa_creatorDID`; UNDOCUMENTED

Parameter *dec*

defaults to `degrees(lat(ssa_location))`; UNDOCUMENTED

Parameter *did* defaults to `$COMPUTE`; Global identifier of the data set. Leave $COMPUTE for tables mixing in products.

Parameter *emMax*

defaults to `ssa_specend`; UNDOCUMENTED

Parameter *emMin*

defaults to `ssa_specstart`; UNDOCUMENTED

Parameter *emResPower*

defaults to `ssa_specstart/ssa_specres`; UNDOCUMENTED

Parameter *emUCD*

defaults to `\sqlquote{\getParam{ssa_spectralucd}}`; UNDOCUMENTED

Parameter *emXel*

defaults to `ssa_length`; UNDOCUMENTED

Parameter *expTime*

defaults to `ssa_timeExt`; UNDOCUMENTED

Parameter *facilityName*

defaults to `\sqlquote{\metaString{facility}}`; UNDOCUMENTED

Parameter *fov*

defaults to `ssa_aperture`; UNDOCUMENTED

Parameter *instrumentName*

defaults to `ssa_instrument`; UNDOCUMENTED

Parameter *mime* defaults to `mime`; The MIME type of the product file. Only touch if you do not mix in products.

Parameter *oUCD*

defaults to `\sqlquote{\getParam{ssa_fluxucd}}`; UNDOCUMENTED

Parameter *obsId* defaults to `accref`; Identifier of the data set. Only change this when you do not mix in products.

Parameter *polStates* defaults to `NULL`; List of polarization states present in the data; if you give something, use the convention of choosing the appropriate from {I Q U V RR LL RL LR XX YY XY YX POLI POLA} and write them *in alphabetical order* with / separators, e.g. /I/Q/XX/.

Parameter *polXel* defaults to `NULL`; Number of polarisation states in this product

Parameter *productSubtype* defaults to NULL; File subtype. Details pending

Parameter *productType*

defaults to `ssa_dstype`**;** UNDOCUMENTED

Parameter *ra*

defaults to `degrees(long(ssa_location))`**;** UNDOCUMENTED

Parameter *sPixelScale* defaults to NULL; Size of a spatial pixel (in arcsec)

Parameter *sResolution*

defaults to `\getParam{ssa_spaceRes}{NULL}/3600.`**;** UNDOCUMENTED

Parameter *sXel1* defaults to NULL; Number of pixels along the first spatial axis

Parameter *sXel2* defaults to NULL; Number of pixels along the second spatial axis

Parameter *size* defaults to `accsize/1024`; The estimated size of the product in kilobytes. Only touch when you do not mix in products#table.

Parameter *tMax*

defaults to `ssa_dateObs+ssa_timeExt/43200.`**;** UNDOCUMENTED

Parameter *tMin*

defaults to `ssa_dateObs-ssa_timeExt/43200.`**;** UNDOCUMENTED

Parameter *tResolution* defaults to NULL; Temporal resolution

Parameter *tXel* defaults to NULL; Number of samples along the time axis

Parameter *targetClass*

defaults to `ssa_targclass`**;** UNDOCUMENTED

Parameter *targetName*

defaults to `ssa_targname`**;** UNDOCUMENTED

Parameter *title*

defaults to `ssa_dstitle`**;** UNDOCUMENTED

The //products#table Mixin

A mixin for tables containing "products".

A "product" here is some kind of binary, typically a FITS file. The table receives the columns accref, accsize, owner, and embargo (which is defined in //products#prodcolUsertable).

By default, the accref is the path to the file relative to the inputs directory; this is also what /getproduct expects for local products. You can of course enter URLs to other places.

For local files, you are strongly encouraged to keep the accref URL- and shell-clean, the most important reason being your users' sanity. Another is that obscore in the current implementation does no URL escaping for local files. So, just don't use characters like like +, the ampersand, apostrophes and so on; the default accref parser will reject those anyway. Actually, try making do with alphanumerics, the underscore, the dash, and the dot, ok?

owner and embargo let you introduce access control. Embargo is a date at which the product will become publicly available. As long as this date is in the future, only authenticated users belonging to the *group* owner are allowed to access the product.

In addition, the mixin arranges for the products to be added to the system table products, which is important when delivering the files.

Tables mixing this in should be fed from grammars using the //products#define row filter.

The //scs#positions Mixin

A mixin adding standardized columns for equatorial positions to the table.

It consists of the fields alphaFloat, deltaFloat (float angles in degrees, J2000.0) and c_x, c_y, c_z (intersection of the radius vector to alphaFloat, deltaFloat with the unit sphere).

You will usually use it in conjunction with the //scs#eqFloat procDef that preparse these fields for you.

Thus, you could say:

```
<proc procDef="//scs#eqFloat">
  <arg name="alpha">alphaSrc</arg>
  <arg name="delta">deltaSrc</arg>
</proc>
```

Note, however, that it's usually much better to not mess with the table structure and handle positions using the q3cindex mixin.

The //scs#q3cindex Mixin

A mixin adding an index to the main equatorial positions.

This is what you usually want if your input data already has "sane" (i.e., ICRS or at least J2000) positions or you convert the positions manually.

You have to designate exactly one column with the ucds pos.eq.ra;meta.main pos.eq.dec;meta.main, respectively. These columns receive the positional index.

This will fail without the q3c extension to postgres.

The //siap#pgs Mixin

A table mixin for simple support of SIAP.

The columns added into the tables include

- (certain) FITS WCS headers
- imageTitle (interpolateString should come in handy for these)
- instId -- some id for the instrument used
- dateObs -- MJD of the "characteristic" observation time
- the bandpass* values. You're on your own with them...
- the values of the product mixin.
- mimetype -- the mime type of the product.
- the coverage, centerAlpha and centerDelta, nAxes, pixelSize, pixelScale, wcs* fields calculated by the computePGS macro.

(their definition is in the siap system RD)

Tables mixing in pgs can be used for SIAP querying and automatically mix in the products table mixin.

To feed these tables, use the //siap#computePGS and //siap#setMeta procs. Since you are dealing with products, you will also need the //products#define rowgen in your grammar.

The //slap#basic Mixin

This mixin is for tables serving SLAP services, i.e., tables with spectral lines. It does not contain all "optional" columns, hence the name basic. We'd do "advanced", too, if there's demand.

Use the //slap#fillBasic procDef to populate such tables.

The //ssap#hcd Mixin

Do not use this in new RDs. Use mixc instead.

This mixin is for "homogeneous" data collections, where homogeneous means that all values in hcd_outpars are constant for all datasets in the collection. This is usually the case if they all come from one instrument.

Rowmakers for tables using this mixin should use the //ssap#setMeta proc application.

Do not forget to call the //products#define row filter in grammars feeding tables mixing this in. At the very least, you need to say:

```
<rowfilter procDef="//products#define">
  <bind name="table">"mySchema.myTableName"</bind>
</rowfilter>
```

This mixin has the following parameters:

Parameter *collection* defaults to `__NULL__`; ivo id of the originating collection; ssa:DataID.Collection

Parameter *creationType* defaults to `__NULL__`; Process used to produce the data (zero or more of archival, cutout, filtered, mosaic, projection, spectralExtraction, catalogExtraction); ssa:DataID.CreationType

Parameter *creator* defaults to `__NULL__`; Creator designation; ssa:DataID.Creator

Parameter *dataSource* defaults to `__NULL__`; Generation type (typically, one survey, pointed, theory, custom, artificial); ssa:DataID.DataSource

Parameter *fluxCalibration* Type of flux calibration (one of ABSOLUTE, RELATIVE, NORMALIZED, or UNCALIBRATED); ssa:Char.FluxAxis.Calibration

Parameter *fluxSI* defaults to `__NULL__`; SI conversion factor for fluxes in the spectrum instance (not the SSA metadata) in Osuna-Salgado convention; ssa:Dataset.FluxSI (you probably want to leave this empty)

Parameter *fluxUCD* defaults to `phot.flux.density;em.wl`; ucd of the flux column, like phot.count, phot.flux.density, etc. Default is for flux over wavelength; ssa:Char.FluxAxis.Ucd

Parameter *fluxUnit* Flux unit used by the spectra and in SSA char metadata. This must be a VOUnit string (use a single blank if your spectrum is not calibrated).

Parameter *instrument* defaults to `__NULL__`; Instrument or code used to produce these datasets; ssa:DataID.Instrument

Parameter *publisher* defaults to `\metaString{publisherID}`; Publisher IVO (by default taken from the DC config); ssa:Curation.Publisher

Parameter *reference* defaults to `__NULL__`; URL or bibcode of a publication describing this data; ssa:Curation.Reference

Parameter *spectralCalibration* defaults to `__NULL__`; Type of wavelength Calibration (one of ABSOLUTE, RELATIVE, NORMALIZED, or UNCALIBRATED); ssa:Char.SpectralAxis.Calibration

Parameter *spectralResolution* defaults to `NaN`; Resolution on the spectral axis; you must give this as FWHM wavelength in meters here. Approximate as necessary; ssa:Char.SpectralAxis.Resolution

Parameter *spectralSI* defaults to `__NULL__`; SI conversion factor of frequency or wavelength in the spectrum instance (not the SSA metadata, they are all in meters); ssa:Dataset.SpectralSI (you probably want to leave this empty)

Parameter *spectralUCD* defaults to `em.wl`; ucd of the spectral column, like em.freq or em.energy; default is wavelength; ssa:Char.SpectralAxis.Ucd

Parameter *spectralUnit* Spectral unit used by the spectra (SSA char metadata always is wavelength in meters). This must be a VOUnit string (use a single blank if your spectrum is not calibrated).

Parameter *statFluxError* defaults to `__NULL__`; Statistical error in flux; ssa:Char.FluxAxis.Accuracy.StatError

Parameter *statSpaceError* defaults to `__NULL__`; Statistical error in position in degrees; ssa:Char.SpatialAxis.Accuracy.StatError

Parameter *statSpectError* defaults to `__NULL__`; Statistical error in wavelength (units of specralSI); ssa:Char.SpectralAxis.Accuracy.StatError

Parameter *sysFluxError* defaults to `__NULL__`; Systematic error in flux; ssa:Char.FluxAxis.Accuracy.SysError

Parameter *sysSpectError* defaults to `__NULL__`; Systematic error in wavelength (in m); ssa:Char.SpectralAxis.Accuracy.SysError

Parameter *timeSI* defaults to `__NULL__`; SI conversion factor for times in Osuna-Salgado convention; ssa:DataSet.TimeSI (you probably want to leave this empty)

The //ssap#mixc Mixin

This mixin provides the columns and params for a common SSA service.

Rowmakers for tables using this mixin should use the //ssap#setMeta and the //ssap#setMixcMeta proc applications.

There are some limitations to the variability; in particular, all spectra must have the same types of axes (i.e., frequency, wavelength, or energy) with identical units. If you don't have that, either leave the respective metadata empty or homogenize it before ingestion.

Do not forget to call the //products#define row filter in grammars feeding tables mixing this in. At the very least, you need to say:

```
<rowfilter procDef="//products#define">
  <bind name="table">"schema.table"</bind>
</rowfilter>
```

This mixin has the following parameters:

Parameter *fluxSI* defaults to `__NULL__`; SI conversion factor for fluxes in the spectrum instance (not the SSA metadata) in Osuna-Salgado convention; ssa:Dataset.FluxSI (you probably want to leave this empty)

Parameter *fluxUCD* defaults to `phot.flux.density;em.wl`; ucd of the flux column, like phot.count, phot.flux.density, etc. Default is for flux over wavelength; ssa:Char.FluxAxis.Ucd

Parameter *fluxUnit* Flux unit used by the spectra and in SSA char metadata. This must be a VOUnit string (use a single blank if your spectrum is not calibrated).

Parameter *spectralSI* defaults to `__NULL__`; SI conversion factor of frequency or wavelength in the spectrum instance (not the SSA metadata, they are all in meters); ssa:Dataset.SpectralSI (you probably want to leave this empty)

Parameter *spectralUCD* defaults to `em.wl`; ucd of the spectral column, like em.freq or em.energy; default is wavelength; ssa:Char.SpectralAxis.Ucd

Parameter *spectralUnit* Spectral unit used by the spectra (SSA char metadata always is wavelength in meters). This must be a VOUnit string (use a single blank if your spectrum is not calibrated).

Parameter *timeSI* defaults to `__NULL__`; SI conversion factor for times in Osuna-Salgado convention; ssa:DataSet.TimeSI (you probably want to leave this empty)

The //ssap#sdm-instance Mixin

This mixin is intended for tables that get serialized into documents conforming to the Spectral Data Model 1, specifically to VOTables

The input to such tables comes from ssa tables (hcd, in this case). Their columns (and params) are transformed into params here.

The mixin adds two columns (you could add more if, e.g., you had errors depending on the spectral or flux value), spectral (wavelength or the like) and flux. Their metadata is taken from the ssa fields where available (ssa_fluxucd as flux UCD, ssa_fluxunit etc).

This mixin in action could look like this:

```
<table id="instance" onDisk="False">
  <mixin ssaTable="spectra"
    fluxUnit="Jy"
    >//ssap#sdm-instance</mixin>
</table>
```

The mixin thus defines a gazillion of params. This will almost always be filled using //ssap#feedSSAToSDM as explaned in SDM compliant tables

This mixin has the following parameters:

Parameter *fluxDescription* defaults to `The dependent variable of this spectrum (see the ucd for its physical meaning)`; Description for the flux column

Parameter *spectralDescription* defaults to `The independent variable of this spectrum (see its ucd to figure out whether it's a wavelength, frequency, or energy)`; Description for the spectral column

Parameter *spectralUCDOverride* Force UCD of the spectral column (don't use this)

Parameter *spectralUnitOverride* Force unit of the spectral column (don't use this)

Parameter *ssaTable* The SSAP (HCD) instance table to take the params from

The //ssap#simpleCoverage Mixin

A mixin furnishes a table with an ssa_region column giving a polygonal coverage. For SSA, that's unnecessary, but it's highly recommended if you have data with positional and aperture data and will publish it via obscore, too (which in turn is highly recommended).

The column will be filled with a hexagon approximating the aperture by //ssap#setMeta, so usually you're set with this mixin. We also create an index for the ssa_region field.

To make it visible in obscore, however, you must bind the `coverage` mixin par of //obscore#publishSSAPHCD to `ssa_region`.

Triggers

In the context of the GAVO DC, triggers are conditions on rows -- either the raw rows emitted by grammars if they are used within grammars, or the rows about to be shipped to a table if they are used within tables. Triggers may be used recursively, i.e., triggers may contain more triggers. Child triggers are normally or-ed together.

Currently, there is one useful top-level trigger, the element ignoreOn. If an ignoreOn is triggered, the respective row is silently dropped (actually, you ignoreOn has a bail attribute that allows you to raise an error if the trigger is pulled; this is mainly for debugging).

The following triggers are defined:

Element and

A trigger that is true when all its children are true.

Atomic Children

- **name** (unicode string; defaults to 'unnamed') -- A name that should help the user figure out what trigger caused some condition to fire.

Structure Children

- triggers (contains any of and,keyPresent,keyNull,keyIs,keyMissing,not and may be repeated zero or more times) -- One or more conditions joined by an implicit logical or. See Triggers for information on what can stand here.

Element keyIs

A trigger firing when the value of key in row is equal to the value given.

Missing keys are always accepted. You can define an SQL type; value will then be interpreted as a literal for this type, and this literal's value will be compared against the key's value. This is only needed for grammars like fitsProductGrammar that actually yield typed values.

Atomic Children

- **key** (unicode string; defaults to <Undefined>) -- Key to check
- **name** (unicode string; defaults to 'unnamed') -- A name that should help the user figure out what trigger caused some condition to fire.
- **type** (unicode string; defaults to 'text') -- An SQL type the python equivalent of which the value should be converted to before checking.
- **value** (unicode string; defaults to <Undefined>) -- The string value to fire on.

Element keyMissing

A trigger firing if a certain key is missing in the dict.

This is equivalent to:

```
<not><keyPresent key="xy"/></not>
```

Atomic Children

- **key** (unicode string; defaults to <Undefined>) -- Key to check
- **name** (unicode string; defaults to 'unnamed') -- A name that should help the user figure out what trigger caused some condition to fire.

Element keyNull

A trigger firing if a certain key is missing or NULL/None

Atomic Children

- **key** (unicode string; defaults to <Undefined>) -- Key to check
- **name** (unicode string; defaults to 'unnamed') -- A name that should help the user figure out what trigger caused some condition to fire.

Element keyPresent

A trigger firing if a certain key is present in the dict.

Atomic Children

- **key** (unicode string; defaults to <Undefined>) -- Key to check
- **name** (unicode string; defaults to 'unnamed') -- A name that should help the user figure out what trigger caused some condition to fire.

Element not

A trigger that is false when its children, or-ed together, are true and vice versa.

Atomic Children

- **name** (unicode string; defaults to 'unnamed') -- A name that should help the user figure out what trigger caused some condition to fire.

Structure Children

- triggers (contains any of and,keyPresent,keyNull,keyIs,keyMissing,not and may be repeated zero or more times) -- One or more conditions joined by an implicit logical or. See Triggers for information on what can stand here.

Renderers Available

The following renderers are available for allowing and URL creation. The parameter style is relevant when adapting *condDescs'* or table based cores to renderers:

- With clear, parameters are just handed through
- With form, suitable parameters are turned into vizier-like expressions
- With pql, suitable parameters are turned into their PQL counterparts, letting you specify ranges and such.

Unchecked renderers can be applied to any service and need not be explicitly allowed by the service.

The admin Renderer

This renderer's parameter style is "clear".

A renderer allowing to block and/or reload services.

This renderer could really be attached to any service since it does not call it, but it usually lives on //services/overview. It will always require authentication.

It takes the id of the RD to administer from the path segments following the renderer name.

By virtue of builtin vanity, you can reach the admin renderer at /seffe, and thus you can access /seffe/foo/q to administer the foo/q RD.

The api Renderer

This renderer's parameter style is "dali".

A renderer that works like a VO standard renderer but that doesn't actually follow a given protocol.

Use this for improvised APIs. The default output format is a VOTable, and the errors come in VOSI VOTables. The renderer does, however, evaluate basic DALI parameters. You can declare that by including <FEED source="//pql#DALIPars"/> in your service.

These will return basic serice metadata if passed MAXREC=0.

The availability Renderer

This renderer's parameter style is "clear". This is an unchecked renderer.

A renderer for a VOSI availability endpoint.

An endpoint with this renderer is automatically registered for every service. The answers can be configured using the admin renderer.

The capabilities Renderer

This renderer's parameter style is "clear". This is an unchecked renderer.

A renderer for a VOSI capability endpoint.

An endpoint with this renderer is automatically registered for every service. The responses contain information on what renderers ("interfaces") are available for a service and what properties they have.

The coverage Renderer

This renderer's parameter style is "clear". This is an unchecked renderer.

A renderer returning various forms of a service's spatial coverage.

This will return a 404 if the service doesn't have a coverage.spatial meta (and will bomb out if that isn't a SMoc).

Based on the accept header, it will return a PNG if the client indicates it's interested in that or if it accepts text/html, in which case we assume it's a browser; otherwise, it will produce a MOC in FITS format.

The custom Renderer

This renderer's parameter style is "clear".

A renderer defined in a python module.

To define a custom renderer write a python module and define a class MainPage inheriting from gavo.web.ServiceBasedPage.

This class basically is a nevow resource, i.e., you can define docFactory, locateChild, renderHTTP, and so on.

To use it, you have to define a service with the resdir-relative path to the module in the customPage attribute and probably a nullCore. You also have to allow the custom renderer (but you may have other renderers, e.g., static).

If the custom page is for display in web browsers, define a class method isBrowseable(cls, service) returning true. This is for the generation of links like "use this service from your browser" only; it does not change the service's behaviour with your renderer.

There should really be a bit more docs on this, but alas, there's none as yet.

The dlasync Renderer

This renderer's parameter style is "pql".

A renderer for asynchronous datalink.

The dlget Renderer

This renderer's parameter style is "clear".

A renderer for data processing by datalink cores.

This must go together with a datalink core, nothing else will do.

This renderer will actually produce the processed data. It must be complemented by the dlmeta renderer which allows retrieving metadata.

The dlmeta Renderer

This renderer's parameter style is "clear".

A renderer for data processing by datalink cores.

This must go together with a datalink core, nothing else will do.

This renderer will return the links and services applicable to one or more pubDIDs.

See Datalink and SODA for more information.

The docform Renderer

This renderer's parameter style is "form".

A renderer displaying a form and delivering core's result as a document.

The core must return a pair of mime-type and content; on errors, the form is redisplayed.

This is mainly useful with custom cores doing weird things. This renderer will not work with dbBasedCores and similar.

The edition Renderer

This renderer's parameter style is "clear". This is an unchecked renderer.

A renderer representing a (tutorial-like) text document.

Not sure yet what I'll do when people actually call this; for now, the access URL must be given as metadata.

The examples Renderer

This renderer's parameter style is "clear". This is an unchecked renderer.

A renderer for examples for service usage.

This renderer formats _example meta items in its service. Its output is XHTML compliant to VOSI examples; clients can parse it to, for instance, fill forms for service operation or display examples to users.

The examples make use of RDFa to convey semantic markup. To see what kind of semantics is contained, try http://www.w3.org/2012/pyRdfa/Overview.html and feed it the example URL of your service.

The default content of _example is ReStructuredText, and really, not much else makes sense. An example for such a meta item can be viewed by executing `gavo admin dumpDF //userconfig`, in the tapexamples STREAM.

To support annotation of things within the example text, DaCHS defines several RST extensions, both interpreted text roles (used like `:role-name:‘content with blanks‘`) and custom directives (used to mark up blocks introduced by a single line like `.. directive-name ::` (the blanks before and after the directive name are significant).

Here's the custom interpreted text roles:

- *dl-id*: An publisher DID a service returns data for (used in datalink examples)
- *taptable*: A (fully qualified) table name a TAP example query is (particularly) relevant for; in HTML, this is also a link to the table description.
- *genparam*: A "generic parameter" as defined by DALI. The values of these have the form param(value), e.g., :genparam:‘POS(32,4)‘. Right now, not parantheses are allowed in the value. Complain if this bites you.

These are the custom directives:

- *tapquery*: The query discussed in a TAP example.

Examples for how to write TAP examples are in the userconfig.rd distributed with DaCHS. Examples for Datalink examples can be found in the GAVO RDs feros/q and califa/q3.

The external Renderer

This renderer's parameter style is "clear".

A renderer redirecting to an external resource.

These try to access an external publication on the parent service and ask it for an accessURL. If it doesn't define one, this will lead to a redirect loop.

In the DC, external renderers are mainly used for registration of third-party browser-based services.

The fixed Renderer

This renderer's parameter style is "clear".

A renderer that renders a single template.

Use something like `<template key="fixed">res/ft.html</template>` in the enclosing service to tell the fixed renderer where to get this template from.

In the template, you can fetch parameters from the URL using something like `<n:invisible n:data="parameter FOO" n:render="string"/>`; you can also define new render and data functions on the service using customRF and customDF.

This is, in particular, used for the data center's root page.

The fixed renderer is intended for non- or slowly changing content. It is annotated as cachable, which means that DaCHS will in general only render it once and then cache it. If the render functions change independently of the RD, use the volatile renderer.

Built-in services for such browser apps should go through the //run RD.

The form Renderer

This renderer's parameter style is "form".

The "normal" renderer within DaCHS for web-facing services.

It will display a form and allow outputs in various formats.

It also does error reporting as long as that is possible within the form.

The get Renderer

This renderer's parameter style is "clear".

The renderer used for delivering products.

This will only work with a ProductCore since the resulting data set has to contain products.Resources. Thus, you probably will not use this in user RDs.

The howtocite Renderer

This renderer's parameter style is "clear". This is an unchecked renderer.

A renderer that lets you format citation instructions.

The info Renderer

This renderer's parameter style is "clear". This is an unchecked renderer.

A renderer showing all kinds of metadata on a service.

This renderer produces the default referenceURL page. To change its appearance, override the serviceinfo.html template.

The logout Renderer

This renderer's parameter style is "clear". This is an unchecked renderer.

logs users out.

With a valid authorization header, this emits a 401 unauthorized, without one, it displays a logout page.

The mimg.jpeg Renderer

This renderer's parameter style is "form".

A machine version of the JpegRenderer -- no vizier expressions, hardcoded parameters, plain text errors.

This should not have been part of DaCHS proper. It will be removed.

The mupload Renderer

This renderer's parameter style is "form".

A renderer allowing for updates to individual records using file uploads.

The difference to Uploader is that no form-redisplay will be done. All errors are reported through HTTP response codes and text strings. It is likely that this renderer will change and/or go away.

The pubreg.xml Renderer

This renderer's parameter style is "clear".

A renderer that works with registry.oaiinter to provide an OAI-PMH interface.

The core is expected to return a stanxml tree.

The qp Renderer

This renderer's parameter style is "clear".

The Query Path renderer extracts a query argument from the query path.

Basically, whatever segments are left after the path to the renderer are taken and fed into the service. The service must cooperate by setting a queryField property which is the key the parameter is assigned to.

QPRenderers cannot do forms, of course, but they can nicely share a service with the form renderer.

To adjust the results' appreance, you can override resultline (for when there's just one result row) and resulttable (for when there is more than one result row) templates.

The rdinfo Renderer

This renderer's parameter style is "clear".

A renderer for displaying various properties about a resource descriptor.

This renderer could really be attached to any service since it does not call it, but it usually lives on //services/overview.

By virtue of builtin vanity, you can reach the rdinfo renderer at /browse, and thus you can access /browse/foo/q to view the RD infos. This is the form used by table registrations.

In addition to all services, this renderer also links tableinfos for all non-temporary, on-disk tables defined in the RD. When you actually want to hide some internal on-disk tables, you can set a property `internal` on the table (the value is ignored).

The scs.xml Renderer

This renderer's parameter style is "pql".

A renderer for the Simple Cone Search protocol.

These do their error signaling in the value attribute of an INFO child of RESOURCE.

You must set the following metadata items on services using this renderer if you want to register them:

- testQuery.ra, testQuery.dec -- A position for which an object is present within 0.001 degrees.

The siap.xml Renderer

This renderer's parameter style is "pql".

A renderer for a the Simple Image Access Protocol.

These have errors in the content of an info element, and they support metadata queries.

For registration, services using this renderer must set the following metadata items:

- sia.type -- one of Cutout, Mosaic, Atlas, Pointed, see SIAP spec

You should set the following metadata items:

- testQuery.pos.ra, testQuery.pos.dec -- RA and Dec for a query that yields at least one image
- testQuery.size.ra, testQuery.size.dec -- RoI extent for a query that yields at least one image.

You can set the following metadata items (there are defaults on them that basically communicate there are no reasonable limits on them):

- sia.maxQueryRegionSize.(long|lat)
- sia.maxImageExtent.(long|lat)
- sia.maxFileSize
- sia.maxRecord (default dalHardLimit global meta)

The siap2.xml Renderer

This renderer's parameter style is "dali".

A renderer for SIAPv2.

In general, if you want a SIAP2 service, you'll need something like the obscore view in the underlying table.

The slap.xml Renderer

This renderer's parameter style is "pql".

A renderer for the simple line access protocol SLAP.

For registration, you must set the following metadata on services using the slap.xml renderer:

There's two mandatory metadata items for these:

- slap.dataSource -- one of observational/astrophysical, observational/laboratory, or theoretical
- slap.testQuery -- parameters that lead to a non-empty response. The way things are written in DaCHS, MAXREC=1 should in general work.

The soap Renderer

This renderer's parameter style is "clear".

A renderer that receives and formats SOAP messages.

This is for remote procedure calls. In particular, the renderer takes care that you can obtain a WSDL definition of the service by appending ?wsdl to the access URL.

The ssap.xml Renderer

This renderer's parameter style is "pql".

A renderer for the simple spectral access protocol.

For registration, you must set the following metadata on services using the ssap.xml renderer:

- ssap.dataSource -- survey, pointed, custom, theory, artificial
- ssap.testQuery -- a query string that returns some data; REQUEST=queryData is added automatically

Other SSA metadata includes:

- ssap.creationType -- archival, cutout, filtered, mosaic, projection, spectralExtraction, catalogExtraction (defaults to archival)
- ssap.complianceLevel -- set to "query" when you don't deliver SDM compliant spectra; otherwise don't say anything, DaCHS will fill in the right value.

Properties supported by this renderer:

- datalink -- if present, this must be the id of a datalink service that can work with the pubDIDs in this table (don't use this any more, datalink is handled through table-level metadata now)
- defaultRequest -- by default, requests without a REQUEST parameter will be rejected. If you set defaultRequest to queryData, such requests will be processed as if REQUEST were given (which is of course sane but is a violation of the standard).

The static Renderer

This renderer's parameter style is "clear".

A renderer that just hands through files.

The standard operation here is to set a staticData property pointing to a resdir-relative directory used to serve files for. Indices for directories are created.

You can define a root resource by giving an indexFile property on the service. Note in particular that you can use an index file with an extension of shtml. This lets you use nevow templates, but since metadata will be taken from the global context, that's probably not terribly useful. You are probably looking for the fixed renderer if you find yourself needing this.

The tableMetadata Renderer

This renderer's parameter style is "clear". This is an unchecked renderer.

A renderer for a VOSI table metadata endpoint.

An endpoint with this renderer is automatically registered for every service. The responses contain information on the tables exposed by a given service.

The tableinfo Renderer

This renderer's parameter style is "clear". This is an unchecked renderer.

A renderer for displaying table information.

Since tables don't necessarily have associated services, this renderer cannot use a service to sit on. Instead, the table is being passed in as as an argument. There's a built-in vanity tableinfo that sits on //dc_tables#show using this renderer (it could really sit anywhere else).

The tablenote Renderer

This renderer's parameter style is "clear". This is an unchecked renderer.

A renderer for displaying table notes.

It takes a schema-qualified table name and a note tag in the segments.

This does not use the underlying service, so it could and will run on any service. However, you really should run it on __system__/dc_tables/show, and there's a built-in vanity name tablenote for this.

The tap Renderer

This renderer's parameter style is "clear".

A renderer speaking all of TAP (including sync, async, and VOSI).

Basically, this just dispatches to the sync and async resources.

The upload Renderer

This renderer's parameter style is "form".

A renderer allowing for updates to individual records using file upload.

This renderer exposes a form with a file widget. It is likely that the interface will change.

The uws.xml Renderer

This renderer's parameter style is "pql".

A renderer speaking UWS.

This is for asynchronous exection of larger jobs. Operators will normally use this together with a custom core or a python core.

See Custom UWSes for details.

The volatile Renderer

This renderer's parameter style is "clear".

A renderer rendering a single template with fast-changing results.

This is like the fixed renderer, except that the results are not cached.

Predefined Procedures

Procedures available for rowmaker/parmaker apply

//epntap2#populate-2_0

Sets metadata for an epntap data set, including its products definition.

The values are left in vars, so you need to do manual copying, e.g., using idmaps="*".

In some descriptions below, you will see ___replace_framed___. This means that the actual descriptions, units, and UCDs will depend on the value of spatial_frame_type in the //epntap2#table-2_0 mixin. After you have made a first (possibly severely incomplete) import of your table, you can see the actual metadata by opening http://localhost:8080/tableinfo/yourschema.epn_core.

Setup parameters for the procedure are:

Late parameter *c1_resol_max* defaults to `None`; Resolution in the first coordinate, upper limit

Late parameter *c1_resol_min* defaults to `None`; Resolution in the first coordinate, lower limit.

Late parameter *c1max* defaults to `None`; ___replace_framed___, upper limit

Late parameter *c1min* defaults to `None`; ___replace_framed___, lower limit.

Late parameter *c2_resol_max* defaults to `None`; Resolution in the second coordinate, upper limit

Late parameter *c2_resol_min* defaults to `None`; Resolution in the second coordinate, lower limit.

Late parameter *c2max* defaults to `None`; ___replace_framed___, upper limit

Late parameter *c2min* defaults to `None`; ___replace_framed___, lower limit.

Late parameter *c3_resol_max* defaults to `None`; Resolution in the third coordinate, upper limit

Late parameter *c3_resol_min* defaults to `None`; Resolution in the third coordinate, lower limit.

Late parameter *c3max* defaults to `None`; ___replace_framed___, upper limit

Late parameter *c3min* defaults to `None`; ___replace_framed___, lower limit.

Late parameter *creation_date* defaults to `None`; Date of first entry of this granule

Late parameter *dataproduct_type* defaults to `None`; The high-level organization of the data product, from enumerated list (e.g., 'im' for image, sp for spectrum)

Late parameter *emergence_max* defaults to `None`; Emergence angle during data acquisition, upper limit

Late parameter *emergence_min* defaults to `None`; Emergence angle during data acquisition, lower limit.

Late parameter *granule_gid* Common to granules of same type (e.g. same map projection, or geometry data products). Can be alphanumeric.

Late parameter *granule_uid* Internal table row index Unique ID in data service, also in v2. Can be alphanumeric.

Late parameter *incidence_max* defaults to `None`; Incidence angle (solar zenithal angle) during data acquisition, upper limit

Late parameter *incidence_min* defaults to `None`; Incidence angle (solar zenithal angle) during data acquisition, lower limit.

Late parameter *index_* defaults to `\rowsMade`; A numeric reference for the item. By default, this is just the row number. As this will (usually) change when new data is added, you should override it with some unique integer number specific to the data product when there is such a thing.

Late parameter *instrument_host_name* Name of the observatory or spacecraft that the observation originated from; for ground-based data, use IAU observatory codes, http://www.minorplanetcenter.net/iau/lists/ObsCodesF.html, for space-borne instruments use http://nssdc.gsfc.nasa.gov/nmc/

Late parameter *instrument_name* defaults to `None`; Service providers are invited to include multiple values for instrument_name, e.g., complete name + usual acronym. This will allow queries on either 'VISIBLE AND INFRARED THERMAL IMAGING SPECTROMETER' or VIRTIS to produce the same reply.

Late parameter *measurement_type* defaults to `None`; UCD(s) defining the data, with multiple entries separated by hash (#) characters.

Late parameter *modification_date* defaults to `None`; Date of last modification (used to handle mirroring)

Late parameter *obs_id* Associates granules derived from the same data (e.g. various representations/processing levels). Can be alphanumeric, may be the ID of original observation.

Late parameter *phase_max* defaults to `None`; Phase angle during data acquisition, upper limit

Late parameter *phase_min* defaults to `None`; Phase angle during data acquisition, lower limit.

Late parameter *processing_level* CODMAC calibration level; see the et_cal note http://dc.g-vo.org/tableinfo/titan.epn_core#note-et_cal for what values are defined here.

Late parameter *release_date* defaults to `None`; Start of public access period

Late parameter *s_region* defaults to `None`; A spatial footprint of a dataset located on a spherical coordinate system. Currently, this is fixed to be a spherical polygon (fill it with something like pgsphere.SPoly.fromDALI([@long1, @lat1, @long2, @lat2,...], all coordinates in degrees). You could use circles or MOCs here; contact the tool maintainers if you need that.

Late parameter *service_title* defaults to `None`; Title of resource (an acronym really, will be used to handle multiservice results)

Late parameter *spectral_range_max* defaults to `None`; Spectral range (frequency), upper limit

Late parameter *spectral_range_min* defaults to `None`; Spectral range (frequency), lower limit.

Late parameter *spectral_resolution_max* defaults to `None`; Sectral resolution, upper limit

Late parameter *spectral_resolution_min* defaults to `None`; Sectral resolution, lower limit.

Late parameter *spectral_sampling_step_max* defaults to `None`; spectral sampling step, upper limit

Late parameter *spectral_sampling_step_min* defaults to `None`; spectral sampling step, lower limit.

Late parameter *target_class* defaults to `"UNKNOWN"`; The type of the target; choose from asteroid, dwarf_planet, planet, satellite, comet, exoplanet, interplanetary_medium, ring, sample, sky, spacecraft, spacejunk, star

Late parameter *target_name* Name of the target object, preferably according to the official IAU nomenclature. As appropriate, take these from the exoplanet encyclopedia http://exoplanet.eu, the meteor catalog at http://www.lpi.usra.edu/meteor/, the catalog of stardust samples at http://curator.jsc.nasa.gov/stardust/catalog/

Late parameter *target_region* defaults to `None`; This is a complement to the target name to identify a substructure of the target that was being observed (e.g., Atmosphere, Surface). Take terms from them Spase dictionary at http://www.spase-group.org or the IVOA thesaurus.

Late parameter *time_exp_max* defaults to `None`; Integration time of the measurement, upper limit

Late parameter *time_exp_min* defaults to `None`; Integration time of the measurement, lower limit.

Late parameter *time_max* defaults to `None`; Acquisition stop time (in JD)

Late parameter *time_min* defaults to `None`; Acquisition start time (in JD)

Late parameter *time_sampling_step_max* defaults to `None`; Sampling time for measurements of dynamical phenomena, upper limit

Late parameter *time_sampling_step_min* defaults to `None`; Sampling time for measurements of dynamical phenomena, lower limit.

Late parameter *time_scale* defaults to `"UNKNOWN"`; Time scale used for the various times, as given by IVOA's STC data model. Choose from TT, TDB, TOG, TOB, TAI, UTC, GPS, UNKNOWN

//epntap2#populate-localfile-2_0

Use this apply when you use the //epntap2#localfile-2_0 mixin. This will only (properly) work when you use a //products#define rowfilter; if you have that, this will work without further configuration.

Setup parameters for the procedure are:

Late parameter *creation_date* defaults to `\sourceCDate`; A timestamp giving the dataset's creation time as a datetime object

//procs#dictMap

Maps input values through a dictionary.

The dictionary is given in its python form here. This apply only operates on the rawdict, i.e., the value in vars is changed, while nothing is changed in the rowdict.

Setup parameters for the procedure are:

Parameter *default* defaults to `KeyError`; Default value for missing keys (with this at the default, an error is raised)

Parameter *key* Name of the input key to map

Parameter *mapping* Python dictionary literal giving the mapping

//procs#fullQuery

runs a free query against the data base and enters the first result record into vars.

locals() will be passed as data, so you can define more bindings and refer to their keys in the query.

Setup parameters for the procedure are:

Parameter *errCol* defaults to `'<unknown>'`; a column name to use when raising a ValidationError on failure.

Parameter *query* an SQL query

//procs#mapValue

is an apply proc that translates values via a utils.NameMap

Destination may of course be the source field (though that messes up idempotency of macro expansion, which shouldn't usually hurt).

The format of the mapping file is:

```
<target key><tab><source keys>
```

where source keys is a whitespace-seperated list of values that should be mapped to target key (sorry the sequence's a bit unusual).

A source key must be encoded quoted-printable. This usually doesn't matter except when it contains whitespace (a blank becomes =20) or equal signs (which become =3D).

Here's an example application for a filter that's supposed to translate some botched object names:

```
<apply name="cleanObject" procDef="//procs#mapValue">
  <bind name="destination">"cleanedObject"</bind>
  <bind name="failuresMapThrough">True</bind>
  <bind name="value">@preObject</bind>
  <bind name="sourceName">"flashheros/res/namefixes.txt"</bind>
</apply>
```

The input could look like this, with a Tab char written as " <TAB> " for clarity:

```
alp Cyg <TAB> aCyg alphaCyg
Nova Cygni 1992 <TAB> Nova=20Cygni=20'92 Nova=20Cygni
```

Setup parameters for the procedure are:

Parameter *destination* name of the field the mapped value should be written into

Parameter *failuresAreNone* defaults to `False`; Rather than raise an error, yield NULL for values not in the mapping

Parameter *failuresMapThrough* defaults to `False`; Rather than raise an error, yield the input value if it is not in the mapping (this is for 'fix some'-like functions and only works when failureAreNone is False)

Parameter *logFailures* defaults to `False`; Log non-resolved names?

Parameter *sourceName* An inputsDir-relative path to the NameMap source file.

Late parameter *value* The value to be mapped.

//procs#resolveObject

Resolve identifiers to simbad positions.

It caches query results (positive as well as negative ones) in cacheDir. To avoid flooding simbad with repetetive requests, it raises an error if this directory is not writable.

It leaves J2000.0 positions as floats in the simbadAlpha and simbadDelta variables.

Setup parameters for the procedure are:

Late parameter *identifier* The identifier to be resolved.

Parameter *ignoreUnknowns* defaults to `True`; Return Nones for unknown objects? (if false, ValidationErrors will be raised)

Parameter *logUnknowns* defaults to `False`; Write unresolved object names to the info log

//procs#simpleSelect

Fill variables from a simple database query.

The idea is to obtain a set of values from the data base into some columns within vars (i.e., available for mapping) based on comparing a single input value against a database column. The query should always return exactly one row. If more rows are returned, the first one will be used (which makes the whole thing a bit of a gamble), if none are returned, a ValidationError is raised.

Setup parameters for the procedure are:

Parameter *assignments* mapping from database column names to vars column names, in the format {<db colname>:<vars name>}"

Parameter *column* the column to compare the input value against

Parameter *errCol*

defaults to `'<unknown>'`; UNDOCUMENTED

Parameter *table* name of the database table to query

Late parameter *val* UNDOCUMENTED

//siap#computePGS

Computes WCS information for SIA tables from FITS WCS keys.

It takes no arguments but expects WCS-like keywords in rowdict, i.e., CRVAL1, CRVAL2 (interpreted as float deg), CRPIX1, CRPIX2 (pixel corresponding to CRVAL1, CRVAL2), CUNIT1, CUNIT2 (pixel scale unit, we bail out if it isn't deg and assume deg when it's not present), CDn_n (the transformation matrix; substitutable by CDELTn), NAXISn (the image size).

Records without or with insufficient wcs keys are furnished with all-NULL wcs info if the missingIsError setup parameter is False, else they bomb out with a DataError (the default).

Setup parameters for the procedure are:

Parameter *missingIsError* defaults to `True`; Throw an exception when no WCS information can be located.

Parameter *naxis* defaults to `"1,2"`; Comma-separated list of integer axis indices (1=first) to be considered for WCS

//siap#getBandFromFilter

sets the bandpassId, bandpassUnit, bandpassRefval, bandpassHi, and bandpassLo from a set of standard band Ids.

The bandpass ids known are contained in a file supplied file that you should consult for supported values. Run gavo admin dumpDF data/filters.txt for details.

All values filled in here are in meters.

If this is used, it must run after //siap#setMeta since setMeta clobbers our result fields.

Setup parameters for the procedure are:

Parameter *sourceCol* defaults to `None`; Name of the column containing the filter name; leave at default None to take the band from result['bandpassId'], where such information would be left by siap#setMeta.

//siap#setMeta

sets siap meta *and* product table fields.

These fields are common to all SIAP implementations.

If you define the bandpasses yourself, do *not* change bandpassUnit and give all values in Meters. If you do change it, at least obscore would break, but probably more. For optical images, we recommend to fill out bandpassId and then let the //siap#getBandFromFilter apply compute the actual limits. If your band is not known, please supply the necessary information to the authors.

Do *not* use `idmaps="*"` when using this procDef; it writes directly into result, and you would be clobbering what it does.

Setup parameters for the procedure are:

Late parameter *bandpassHi* defaults to `None`; lower value of wavelength or frequency (you usually want to use //siap#getBandFromFilter to fill this).

Late parameter *bandpassId* defaults to `None`; a rough indicator of the bandpass, like Johnson bands

Late parameter *bandpassLo* defaults to `None`; upper value of the wavelength or frequency (you usually want to use //siap#getBandFromFilter to fill this).

Late parameter *bandpassRefval* defaults to `None`; characteristic frequency or wavelength of the exposure (you usually want to use //siap#getBandFromFilter to fill this).

Late parameter *bandpassUnit* defaults to `"m"`; The unit of the bandpassRefval and friends (just don't touch this)

Late parameter *dateObs* defaults to `None`; the midpoint of the observation; this can either be a datetime instance, or a float>1e6 (a julian date) or something else (which is then interpreted as an MJD)

Late parameter *instrument* defaults to `str(rd.getMeta("instrument"))`; a short identifier for the instrument used

Late parameter *pixflags* defaults to `None`; processing flags (C atlas image or cutout, F resampled, X computed without interpolation, Z pixel flux calibrated, V unspecified visualisation for presentation only)

Late parameter *refFrame* defaults to `'ICRS'`; reference frame of the coordinates (change at your peril)

Late parameter *title* defaults to `None`; image title. This should, in as few characters as possible, convey some idea what the image will show (e.g., instrument, object, bandpass

//slap#fillBasic

This apply is intended for rowmakers filling tables mixing in //slap#basic. It populates vars for all the columns in there; you'll normally want idmaps="*" with this apply.

For most of its parameters, it will take them for same-named vars, so you can slowly build up its arguments through var elements.

Setup parameters for the procedure are:

Late parameter *chemical_element* defaults to `@chemical_element`; Element that makes the transition. It's probably ok to dump molecule names in here, too.

Late parameter *final_level_energy* defaults to `@final_level_energy`; Energy of the final state

Late parameter *final_name* defaults to `@final_name`; Designation of the final state

Late parameter *id_status* defaults to `"identified"`; Identification status; this would be identified or unidentified plus possibly uncorrected (but read the SLAP spec for that).

Late parameter *initial_level_energy* defaults to `@initial_level_energy`; Energy of the initial state

Late parameter *initial_name* defaults to `@initial_name`; Designation of the initial state

Late parameter *linename* defaults to `@linename`; A brief designation for the line, like 'H alpha' or 'N III 992.973 A'.

Late parameter *pub* defaults to `@pub`; Publication this came from (use a bibcode).

Late parameter *wavelength* defaults to `@wavelength`; Wavelength of the transition in meters; this will typically be an expression like int(@wavelength)*1e-10

//ssap#feedSSAToSDM

feedSSAToSDM takes the current rowIterator's sourceToken and feeds it to the params of the current target. sourceTokens must be an SSA rowdict (as provided by the sdmCore). Futher, it takes the params from the sourceTable argument and feeds them to the params, too.

All this probably only makes sense in parmakers when making tables mixing in //ssap#sdm-instance in data children of sdmCores.

//ssap#setMeta

Sets metadata for an SSA data set, including its products definition.

The values are left in vars, so you need to do manual copying, e.g., using idmaps="*", or, if you need to be more specific, idmaps="ssa_*".

Setup parameters for the procedure are:

Late parameter *alpha* defaults to `None`; right ascension of target (ICRS degrees); ssa:Char.SpatialAxis.Coverage.Location.Value.C1

Late parameter *aperture* defaults to `None`; angular diameter of aperture (expected in degrees); ssa:Char.SpatialAxis.Coverage.Bounds.Extent

Late parameter *bandpass* defaults to `None`; bandpass (i.e., rough spectral location) of this dataset; ssa:DataID.Bandpass

Late parameter *cdate* defaults to `None`; date the file was created (or processed; optional); this must be either a string in ISO format, or you need to parse to a timestamp yourself; ssa:DataID.Date

Late parameter *creatorDID* defaults to `None`; id given by the creator (leave out if not applicable); ssa:DataID.CreatorDID

Late parameter *cversion* defaults to `None`; creator assigned version for this file (should be incremented when it is changed); ssa:DataID.Version

Late parameter *dateObs* defaults to `None`; observation midpoint (you can give a datetime, a string in iso format, a jd, or an mjd, the latter two being told apart by comparing against 1e6)

Late parameter *delta* defaults to `None`; declination of target (ICRS degrees); ssa:Char.SpatialAxis.Coverage.Location.Value.C2

Late parameter *dstitle* a title for the data set (e.g., instrument, filter, target in some short form; must be filled in); ssa:DataID.Title

Late parameter *length* defaults to `None`; Number of samples in the spectrum; ssa:Dataset.Length

Late parameter *pdate* defaults to `datetime.datetime.utcnow()`; date the file was last published (in general, the default is fine); ssa:Curation.Date

Late parameter *pubDID* Id provided by the publisher (i.e., you); this is an opaque string and must be given; ssa:Curation.PublisherDID

Late parameter *redshift* defaults to `None`; source redshift; ssa:Target.Redshift

Late parameter *snr* defaults to `None`; signal-to-noise ratio estimated for this dataset; ssa:Derived.SNR

Late parameter *specend* defaults to `None`; upper bound of wavelength interval (in meters); ssa:Char.SpectralAxis.Coverage.Bounds.Stop

Late parameter *specext* defaults to `None`; (ignored; only present for compatiblity, computed from specstart and specend)

Late parameter *specmid* defaults to `None`; (ignored; only present for compatiblity, computed from specstart and specend)

Late parameter *specstart* defaults to `None`; lower bound of wavelength interval (in meters); ssa:Char.SpectralAxis.Coverage.Bounds.Start

Late parameter *targclass* defaults to `None`; object class (star, QSO,...); ssa:Target.Class

Late parameter *targname* defaults to `None`; common name of the object observed; ssa:Target.Name

Late parameter *timeExt* defaults to `None`; exposure time (in seconds); ssa:Char.TimeAxis.Coverage.Bounds.Extent

//ssap#setMixcMeta

Sets metadata for an SSA data set from mixed sources. This will only work sensibly in cooperation with setMeta

As with setMeta, the values are left in vars; if you did as recommended with setMeta, you'll have this covered as well.

Setup parameters for the procedure are:

Late parameter *binSize* defaults to `None`; Bin size on the spectral axis in m

Late parameter *collection* defaults to `None`; IOVA id of the originating data collection (leave empty if you don't know what this is about)

Late parameter *creationType* defaults to `None`; Process used to produce the data (zero or more of archival, cutout, filtered, mosaic, projection, spectralExtraction, catalogExtraction, concatenated by commas); ssa:DataID.CreationType

Late parameter *creator* defaults to `"Take from RD"`; Creator/Author

Late parameter *dataSource* defaults to `None`; Generation type (typically, one survey, pointed, theory, custom, artificial); ssa:DataID.DataSource

Late parameter *dstype* defaults to `"spectrum"`; Type of data. The only defined value currently is Spectrum, but you may get away with TimeSeries; ssa:Dataset.Type

Late parameter *fluxCalib* defaults to `None`; Type of flux calibration (one of ABSOLUTE, RELATIVE, NORMALIZED, or UNCALIBRATED); ssa:Char.FluxAxis.Calibration

Late parameter *fluxStatError* defaults to `None`; Statistical error for flux in units of fluxUnit

Late parameter *fluxSysError* defaults to `None`; Systematic error for flux in units of fluxUnit

Late parameter *instrument* defaults to `"Take from RD"`; Instrument or code used to produce this dataset; ssa:DataID.Instrument

Late parameter *publisher* defaults to `"Take from RD"`; Publisher IVO; ssa:Curation.Publisher

Late parameter *reference* defaults to `"Take from RD"`; URL or bibcode of a publication describing this data.

Late parameter *specCalib* defaults to `None`; Type of wavelength Calibration (one of ABSOLUTE, RELATIVE, NORMALIZED, or UNCALIBRATED); ssa:Char.SpectralAxis.Calibration

Late parameter *specres* defaults to `None`; Resolution on the spectral axis; you must give this as FWHM wavelength in meters here. This will default to binSize if not given; ssa:Char.SpectralAxis.Resolution

Late parameter *spectStatError* defaults to `None`; Statistical error for the spectral coordinate in m

Late parameter *spectSysError* defaults to `None`; Systematic error for the spectral coordinate in m

Procedures available for grammar rowfilters

//procs#expandComma

A row generator that reads comma seperated values from a field and returns one row with a new field for each of them.

Setup parameters for the procedure are:

Parameter *destField* Name of the column the individual columns are written to

Parameter *srcField* Name of the column containing the full string

//procs#expandDates

is a row generator to expand time ranges.

The finished dates are left in destination as datetime.datetime instances

Setup parameters for the procedure are:

Parameter *dest* defaults to `'curTime'`; name of the column the time should appear in

Parameter *end* the end date(time)

Late parameter *hrInterval* defaults to `24`; difference between generated timestamps in hours

Parameter *start* the start date(time), as either a datetime object or a column ref

//procs#expandIntegers

A row processor that produces copies of rows based on integer indices.

The idea is that sometimes rows have specifications like "Star 10 through Star 100". These are a pain if untreated. A RowExpander could create 90 individual rows from this.

Setup parameters for the procedure are:

Parameter *endName* column containing the end value

Parameter *indName* name the counter should appear under

Parameter *startName* column containing the start value

//products#define

Enters the values defined by the product interface into a grammar's result.

See the documentation on the //products#table mixin. In short: you will always have to touch table (to the name of the table this row is managed in).

If you don't serve FITS images, you will also have to set mime. Use a media type like "image/jpeg" or "text/csv" here as appropriate. If not set, this defaults to "image/fits" (which is, we claim, suitable for cubes and certain spectra, too); for FITS binary tables, use application/fits.

Everything else is optional: You may want to set preview and preview_mime if DaCHS can't do previews of your stuff automatically. datalink is there if you have a datalink thing. What's left is for special situations.

This will create the keys prodblAccref, prodtblOwner, prodtblEmbargo, prodtblPath, prodtblFsize, prodtblTable, prodtblMime, prodtblPreview, prodtbleMime, and prodtblDatalink keys in rawdict -- you can refer to them in the usual @foo way, which is sometimes useful even outside products processing proper (in particular for prodtblAccref).

Setup parameters for the procedure are:

Late parameter *accref* defaults to `\inputRelativePath{False}`; an access reference (this ususally is the input-relative path; only file names well-behaved in URLs are accepted here by default for easier operation with ObsTAP)

Late parameter *datalink* defaults to `None`; id of a datalink service that understands this file's pubDID.

Late parameter *embargo* defaults to `None`; for proprietary data, the date the file will become public

Late parameter *fsize* defaults to `\inputSize`; the size of the input in bytes

Late parameter *mime* defaults to `'image/fits'`; MIME-type for the product

Late parameter *owner* defaults to `None`; for proprietary data, the owner as a gavo creds-created user

Late parameter *path* defaults to `\inputRelativePath{True}`; the inputs-relative path to the product file (change at your peril)

Late parameter *preview* defaults to `'AUTO'`; file path to a preview, dcc://rd.id/svcid id of a preview-enabled datalink service, None to disable previews, or 'AUTO' to make DaCHS guess.

Late parameter *preview_mime* defaults to `None`; MIME-type for the preview (if there is one).

Parameter *table* the table this product is managed in. You must fill this in, and don't forget the quotes.

Procedures available for datalink cores

//soda#fits_doWCSCutout

A fairly generic FITS cutout function.

It expects some special attributes in the descriptor to allow it to decode the arguments. These must be left behind by the metaMaker(s) creating the parameters.

This is axisNames, a dictionary mapping parameter names to the FITS axis numbers or the special names WCSLAT or WCSLONG. It also expects a skyWCS attribute, a wcs.WCS instance for spatial cutouts.

Finally, descriptor must have a list attribute slices, containing zero or more tuples of (fits axis, lowerPixel, upperPixel); this allows things like BAND to add their slices obtained from parameters in standard units.

The .data attribute must be a pyfits hduList, as generated by the fits_makeHDUList data function.

//soda#fits_formatHDUs

Formats pyfits HDUs into a FITS file.

This all works in memory, so for large FITS files you'd want something more streamlined.

//soda#fits_genDesc

A data function for SODA returning the a fits descriptor.

This has, in addition to the standard stuff, a hdr attribute containing the primary header as pyfits structure.

The functionality of this is in its setup, getFITSDescriptor. The intention is that customized DGs (e.g., fixing the header) can use this as an original.

Setup parameters for the procedure are:

Parameter *accrefPrefix* defaults to `None`; A prefix for the accrefs the parent SODA service works on. Calls on all other accrefs will be rejected with a 403 forbidden. You should always include a restriction like this when you make assumptions about the FITSes (e.g., what axes are available).

Parameter *descClass* defaults to `FITSProductDescriptor`; The descriptor class to use. The default is fine for vanilla FITS files, but when you deliver datalinks through the product table, you'll have to use DLFITSDescriptor here. Also, you can define a descriptor yourself in the setup (inherit from FITSDescriptor).

Parameter *qnd* defaults to `True`; Pass 0 or False here to not use DaCHS fast header reader here. This is necessary to properly handle compressed FITS images -- but it entails the risk that astropy magic will mogrify the header, and it may be dramatically slower in some circumstances.

//soda#fits_makeBANDMeta

Yields standard BAND params.

This adds lambdaToMeterFactor and lambdaAxis attributes to the descriptor for later use by fits_makeBANDSlice

Setup parameters for the procedure are:

Parameter *fitsAxis* defaults to `3`; FITS axis index (1-based) of the wavelength dimension

Parameter *wavelengthOverride* defaults to `None`; Override for the FITS unit given for the wavelength (for when it is botched or missing; leave at None for taking it from the header); this is a python literal.

//soda#fits_makeBANDSlice

Computes a cutout for the parameters added by makeBANDMeta.

This *must* sit in front of doWCSCutout.

This also reuses internal state added by makeBANDMeta, so this really only makes sense together with it.

//soda#fits_makeHDUList

An initial data function to construct a pyfits hduList and make that into a descriptor's data attribute.

This wants a descriptor as returned by fits_genDesc.

There's a hack here: this sets a dataIsPristine boolean on descriptor that's made false when one of the fits manipulators change something. If that's true by the time the formatter sees it, it will just push out the entire file. So, if you use this and insert your own data functions, make sure you set dataIsPristine accordingly.

Setup parameters for the procedure are:

Parameter *crop* defaults to `True`; Cut away everything but the primary HDU? (This is unconditionally suppresed for compressed FITSes and when operations are on a non-primary extension).

//soda#fits_makeWCSParams

A metaMaker that generates parameters allowing cutouts along the various WCS axes in physical coordinates.

This uses astropy.wcs for the spatial coordinates and tries to figure out what these are with some heuristics. For the remaining coordinates, it assumes all are basically 1D, and it sets up separate, manual transformations for them.

The metaMaker leaves an axisNames mapping in the descriptor. This is important for the fits_doWCSCutout, and replacement metaMakers must do the same.

The meta maker also creates a skyWCS attribute in the descriptor if successful, containing the spatial transformation only. All other transformations, if present, are in miscWCS, by a dict mapping axis labels to the fitstools.WCS1Trans instances.

If individual metadata in the header are wrong or to give better metadata, use axisMetaOverrides. This will not generate standard parameters for non-spatial axis (BAND and friends). There are other //soda streams for those.

Setup parameters for the procedure are:

Parameter *axisMetaOverrides* defaults to `{}`; A python dictionary mapping fits axis indices (1-based) to dictionaries of inputKey constructor arguments; for spatial axes, use the axis name instead of the axis index.

Parameter *stcs* defaults to `None`; A QSTC expression describing the STC structure of the parameters. This is currently ignored and will almost certainly look totally different when STC2 finally comes around. Meanwhile, don't bother.

//soda#fromStandardPubDID

A descriptor generator for SODA that builds a ProductDescriptor for PubDIDs that have been built by getStandardsPubDID (i.e., the path part of the IVOID is a tilde, with the products table accref as the query part).

Setup parameters for the procedure are:

Parameter *accrefPrefix* defaults to `None`; A prefix for the accrefs the parent SODA service works on. Calls on all other accrefs will be rejected with a 403 forbidden. You should always include a restriction like this when you make assumptions about the FITSes (e.g., what axes are available).

//soda#generateProduct

A data function for SODA that returns a product instance. You can restrict the mime type of the product requested so the following filters have a good idea what to expect.

Setup parameters for the procedure are:

Parameter *requireMimes* defaults to `frozenset()`; A set or sequence of mime type strings; when given, the data generator will bail out with ValidationError if the product mime is not among the mimes given.

//soda#sdm_genData

A data function for SODA returning a spectral data model compliant table that later data functions can then work on. As usual for generators, it uses the implicit PUBDID argument.

Setup parameters for the procedure are:

Parameter *builder* Full reference (like path/rdname#id) to a data element building the SDM instance table as its primary table.

//soda#sdm_genDesc

A data function for SODA returning the product row corresponding to a PubDID within an SSA table.

The descriptors generated have an ssaRow attribute containing the original row in the SSA table.

Setup parameters for the procedure are:

Late parameter *descriptorClass* defaults to `ssap.SSADescriptor`; The SSA descriptor class to use. You'll need to override this if the dc.products path doesn't actually lead to the file (see custom generators). This class must have an fromSSAResult constructor.

Parameter *ssaTD* Full reference (like path/rdname#id) to the SSA table the spectrum's PubDID can be found in.

//soda#trivialFormatter

The tivial formatter for SODA processed data -- it just returns descriptor.data, which will only work it it works as a nevow resource.

If you do not give any dataFormatter yourself in a SODA core, this is what will be used.

Predefined Streams

Streams are recorded RD elements that can be replayed into resource descriptors using the `FEED` active tag. They do, however, support macro expansion; if macros are expanded, you need to given them values in the FEED element (as attributes). What attributes are required should be mentioned in the following descriptions for those predefined streams within DaCHS that are intended for developer consumption.

Datalink-related Streams

//soda#sdm_plainfluxcalib

A stream inserting a data function and its metadata generator to do select flux calibrations in SDM data. This expects sdm_generate (or at least parameters.data as an SDM data instance) as the generating function within the SODA core.

Clients can select "RELATIVE" as FLUXCALIB, which does a normalization to max(flux)=1 here. Everything else is rejected right now.

This probably is more an example of how to write such a thing then genuinely useful.

//soda#sdm_cutout

A stream inserting a data function and its metaMaker to do cutouts in SDM data. This expects sdm_generate (or at least parameters.data as an SDM data instance) as the generating function within the SODA core.

The cutout limits are always given in meters, regardless of the spectrum's actual units (as in SSAP's BAND parameter).

//soda#sdm_format

A formatter for SDM data, together with its input key for FORMAT.

//soda#fits_genKindPar

This stream should be included in FITS-handling SODA services; it adds parameter and code to just retrieve the FITS header to the core.

For this to work as expected, it must be immediately before the formatter.

//soda#fits_genPixelPar

This stream should be included in FITS-handling SODA services; it add parameters and code to perform cut-outs along pixel coordinates.

//soda#fits_standardDLFuncs

Pulls in all "standard" SODA functions for FITSes, including cutouts and header retrieval.

You can give an stcs attribute (for fits_makeWCSParams); for this doesn't make sense because STCS cannot express the SODA parameter structure.

For cubes, you can give a spectralAxis attribute here containing the fits axis index (1..n) of the spectral axis. If you don't, no BAND cutout will be generated. If you do, you may want to fix wavelengthOverride (default is to take what the FITS says).

To work, this needs a descriptor generator; you probably want //soda#fits_genDesc here.

Defaults for macros used in this stream:

- spectralAxis: '0'
- stcs: ''
- wavelengthOverride: 'None'

//soda#fits_standardBANDCutout

Adds metadata and data function for one axis containing wavelengths.

(this could be extended to cover frequency and energy axes, I guess)

To use this, give the fits axis containing the spectral coordinate in the spectralAxis attribute; if needed, you can override the unit in wavelengthUnit (if the unit in the header is somehow bad or missing; don't use quotes here).

This *must* be included physically before fits_doWCSCutout. Otherwise, no cutout will be performed.

Defaults for macros used in this stream:

- spectralAxis: '0'
- wavelengthOverride: 'None'

Other Streams

//procs#license-cc0

Include this stream with a @what (a short phrase saying what is licensed) to make your resource licensed under Creative Commons-0 (a.k.a. public domain). This will generate the copyright, rights and rightsURI metadata items. It needn to live in the toplevel /resource element.

Example:

```
<FEED source="//procs#license-cc0" what="the HSOY catalogue"/>
```

//procs#license-cc-by

Include this stream with a @what (a short phrase saying what is licensed) to make your resource licensed under Creative Commons Attribution (CC-BY). This will generate the copyright, rights and rightsURI metadata items. It needs to live in the toplevel /resource element.

Example:

```
<FEED source="//procs#license-cc-by" what="the HSOY catalogue"/>
```

//procs#license-cc-by-sa

Include this stream with a @what (a short phrase saying what is licensed) to make your resource licensed under Creative Commons Attribution Share Alike (CC-BY-SA). This will generate the copyright, rights and rightsURI metadata items. It needs to live in the toplevel /resource element.

Example:

```
<FEED source="//procs#license-cc-by-sa" what="the HSOY catalogue"/>
```

//obscore#obscore-columns

The columns of a (standard) obscore table. This can be used to define a "native" obscore table (as opposed to the more usual mixins below that expose standard products via obscore.

Even if you are sure you want to do this, better ask again...

//ssap#hcd_condDescs

This stream defines the condDescs for an SSA service. It is designed to work with both the mixc and the (deprecated) hcd mixins.

//ssap#atomicCoords

A stream for form-based service's VOTables to include simple RA and Dec rather than normal ssa_location.

SSA services get that from the core and don't need this.

//echelle#ssacols

Additional columns for SSA metadata tables describing Echelle spectra.

//scs#coreDescs

This stream inserts three condDescs for SCS services on tables with pos.eq.(ra|dec).main columns; one producing the standard SCS RA, DEC, and SR parameters, another creating input fields for human consumption, and finally MAXREC.

Data Descriptors

Most basic information on data descriptors is contained in tutorial.html. The material here just covers some advanced topics.

Updating Data Descriptors

By default, `dachs imp` will try to drop all tables made by the data descriptors selected. For "growing" data, that is suboptimal, since typicaly just a few new datasets need to be added to the table, and re-ingesting everything else is just a waste of time and CPU.

To accomodate such situations, DaCHS allows to add an `updating="True"` attribute to a `data` element; updating DDs will create tables that do not exist but will not drop existing ones.

Using fromdb on ignoreSources

Updating DDs will still run like normal DDs and thus import everything matching the DD's `sources`. Thus, after the second import you would have duplicate records for sources that existed during the first import.

To avoid that, you (usually) need to ignore existing sources (see Element ignoreSources). In the typical case, where a dataset's accref is just the inputs-relative path to the dataset's source, that is easily accomplished through the `fromdb` attribute of `ignoreSources`; its value is a database query that returns the inputs-relative paths of sources to ignore.

Hence, unless you are playing games with the accrefs (in which case you are probably smart enough to figure out how to adapt the pattern), the following speficiation will exactly import all FITS files within the data subdirectory of the resdir that haven't been ingested into the `mydata` table during the last run, either because they've not been there or because there were skipped during an `import -c`:

```
<data id="import" updating="true">
  <sources pattern="data/*.fits">
    <ignoreSources fromdb="select accref from \schema.mydata"/>
  </sources>

  <fitsProdGrammar>
    <rowfilter procDef="//products#define">
      <bind key="table">"\schema.mydata"</bind>
    </rowfilter>
  </fitsProdGrammar>

  <make table="mydata">
    <!-- your rowmaker here -->
  </make>
</data>
```

Note that `fromdb` can be combined with `fromfiles` and `pattern`; whatever is specified in the latter two will always be ignored.

To completely re-import such a table – for instance after a table schema change or because the whole data collection has been re-processed –, just run `dachs drop` on the DD and run import as usual.

It is probably a good idea to occasionally run `dachs imp -I` on tables updated in this way to optimise the indices (a `REINDEX <tablename>` in a database shell will do, too).

Using fromdbUpdating on ignoreSources

Sometimes reprocessing happens quite frequently to a small subset of the datasets in a resource. In that case, it would again be a waste to tear down the entire thing just to update a handful of records.

For such situations, there is the `fromdbUpdating` attribute of `ignoreSources`. As with `fromdb`, this contains a database query, but in addition to the accref, this query has to return a timestamp. A source is then only ignored if this timestamp is not newer than the disk file's one. If that timestamp is the mtime of the file in the original import, the net effect is that files that have been modified since that import will be re-ingested.

There is a catch, though: You need to make sure that the record ingested previously is removed from the table. Typically, you can do that by defining accref as a primary key (if that's not possible because you are generating multiple records with the same accref, there is nothing wrong with using a compound primary key). This will, on an attempted overwrite, cause an `IntegrityError`, and you can configure DaCHS to turn this into an overwrite using the `table`'s `forceUnique` and `dupePolicy` attributes.

The following snippet illustrates the technique:

```
<table id="withdate" mixin="//products#table" onDisk="True"
    primary="accref"
    forceUnique="True"
    dupePolicy="overwrite">
  <column name="mtime" type="timestamp"
    ucd="time;meta.file"
    tablehead="Timestamp"
    description="Modification date of the source file."/>
  <!-- your other columns -->
</table>

<data id="import" updating="True">
  <sources pattern="data/*.fits">
    <ignoreSources
```

```
      fromdbUpdating="select accref, mtime from \schema.withdate"/>
  </sources>
  <fitsProdGrammar>
    <rowfilter procDef="//products#define">
      <bind key="table">"\schema.withdate"</bind>
    </rowfilter>
  </fitsProdGrammar>
  <make table="withdate">
    <rowmaker>
      <map key="mtime">datetime.datetime.utcfromtimestamp(
        os.path.getmtime(\fullPath))</map>
      <!-- other rowmaker rules -->
    </rowmaker>
  </make>
</data>
```

Again, this can be combined with the other attributes of `ignoreSources`; in effect, whatever is ignored from them is treated as if their modification dates were in the future.

Metadata

Various elements support the setting of metadata through meta elements. Metadata is used for conveying RMI-style metadata used in the VO registry. See [RMI] for an overview of those. We use the keys given in RMI, but there are some extensions discussed in RMI-style Metadata.

The other big use of meta information is for feeding templates. Those "local" keys should all start with an underscore. You are basically free to use those as you like and fetch them from your custom templates. The predefined templates already have some meta items built in, discussed in *Template Metadata*.

So, metadata is a key-value mapping. Keys may be compound like in RMI, i.e., they may consist of period-separated atoms, like publisher.address.email. There may be multiple items for each meta key.

Inputing Metadata

In RDs, there are two ways to define metadata: Meta elements and meta streams; the latter are also used in `defaultmeta.txt`.

Meta Elements

These look like normal XML elements and have a mandatory `name` attribute, a meta key relative to the element's root . The text content is taken as the meta value; child meta elements are legal.

An optional attribute for all meta elements is `format` (see Meta Formats).

Typed meta elements can have further attributes; these usually can also be given as meta children with the same name.

Usually, metadata is additive; add a key twice and you will have a sequence of two meta values. To remove previous content, prefix the meta name with a bang (!). Here is an example:

```
<resource>
  <!-- a simple piece of metadata -->
  <meta name="title">A Meta example</meta>

  <!-- repeat a meta thing for a sequence (caution: not everything
    is repeatable in all output formats -->
  <meta name="subject">Examples</meta>
  <meta name="subject">DaCHS</meta>

  <!-- Hierarchical meta can be set nested -->
  <meta name="creator">
    <meta name="name">Nations, U.N.</meta>
    <meta name="logo">http://un.org/logo.png</meta>
  </meta>
  <meta name="creator">
    <meta name="name">Neumann, A.E.</meta>
  </meta>

  <!-- @format lets you specify extra markup; make sure you
  have consistent initial indentation. -->
  <meta name=description" format="rst">
    This resource is used in the `DaCHS reference docs`_

    .. _DaCHS reference Docs: http://docs.g-vo.org/DaCHS
  </meta>

  <!-- you can contract "deeper" trees in paths -->
  <meta name="contact.email">gavo@ari.uni-heidelberg.de</meta>

  <!-- typed meta elements can have additional attributes -->
  <meta name="uses" ivoId="ivo://org.gavo.dc/DaCHS"
    >DaCHS server sortware</meta>

  <!-- To overwrite a key set before, prefix the name with a bang. -->
  <meta name="!title">An improved Meta example</meta>
</resource>
```

The resulting meta structure is like this:

```
+-- title
|     +---- "An improved Meta example
|
```

```
+-- subject
|     +---- "Examples
|     +---- "DaCHS
|
+-- creator
|     +----- name
|     |       +---- "Nations, U.N.
|     +----- logo
|     |       +---- "http://un.org/logo.png
+-- creator
|     +----- name
|             +---- "Neumann, A.E.
|
+-- description
|     +----- [formatted text, "This resource..."]
|
+-- contact
|      +----- email
|               +----- "gavo@ari.uni-heidelberg.de
|
+-- uses
       +----- "DaCHS server software
       +----- ivoId
                +----- "ivo://org.gavo.dc/DaCHS
```

Stream Metadata

In several places, most notably in the `defaultmeta.txt` file and in meta elements without a `name` attribute, you can give metadata as a "meta stream". This is just a sequence of lines containing pairs of <meta key> and <meta value>.

In addition, there are comments, empty lines, and continuations. Continuation lines work by ending a line with a backslash. The following line separator and all blanks and tabs following it are then ignored. Thus, the following two meta keys end up having identical values:

```
meta1: A contin\
  uation line needs \
    a blank if you wan\
t one.
meta2: A continuation line needs a blank if you want one
```

Note that whitespace behind a backslash prevents it from being a continuation character. That is, admittedly, a bit of a trap.

Other than their use as continuation characters, backslashes have no special meaning within meta streams as such. Within meta elements, however, macros are expanded after continuation line processing if the meta parent knows how to expand macros. This lets you write things like:

```
<meta>
  creationDate: \metaString{authority.creationDate}
  managingOrg:ivo://\getConfig{ivoa}{authority}
</meta>
```

Comments and empty lines are easy: Empty lines are allowed, and a comment is a line with a hash (#) as its first non-whitespace character. Both constructs are ignored, and you can even continue comments (though you should not).

A Pitfall with Sequential Nested Meta

The creator.name meta illustrates a pitfall with our metadata definition. Suppose you had more than one creator. What you'd want is a metadata structure like this:

```
+-- creator -- name (Arthur)
|
+-- creator -- name (Berta)
```

However, if you write:

```
creator.name: Arthur
creator.name: Berta
```

or, equivalently:

```
<meta name="creator.name">Arthur</meta>
<meta name="creator.name">Berta</meta>
```

by the above rules, you'll get this:

```
+-- creator -- name (Arthur)
       |
       +------ name (Berta)
```

i.e., one creator with two names.

To avoid this, make a new creator node in between, i.e., write:

```
creator.name: Arthur
creator:
creator.name: Berta
```

In the creator.name case, where this is so common, DaCHS provides a shortcut, which you should use as a default; if you set `creator` directly, DaCHS will expect a string of the form:

```
<author1>, <inits1> {; <authorn>, <initsn>}
```

(i.e., Last, I.-form separated by semicolons, as in "Foo, X.; Bar, Q.; et al") and split it up into the proper structure. You can mix the two notations, for instance if you want to set a logo on the first creator:

```
<meta name="creator">
  <meta name="name">Chandrasekhar, S.</meta>
  <meta name="logo">http://sit.in/chandra.png</meta>
</meta>
<meta name="creator">Copernicus, N.; Gallilei, G.</meta>
```

Meta information can have a complex tree structure. With meta streams, you can build trees by referencing dotted meta identifiers. If you specify meta information for an item that already exists, a sibling will be created. Thus, after:

```
creator.name: A. Author
creator:
creator.name: B. Buthor
```

there are two creator elements, each specifying a name meta. For the way creators are specified within VOResource, the following would be wrong:

```
creator.name: This is wrong.
creator.name: and will not work
```

-- you would have a single creator meta with two name metas, which is not allowed by VOResource.

If you write:

```
contact.address: 7 Miner's Way, Behind the Seven Mountains
contact.email: dwarfs@fairytale.fa
```

you have a single contact meta giving address and email.

Meta inheritance

When you query an element for metadata, it first sees if it has this metadata. If that is not the case, it will ask its meta parent. This usually is the embedding element. It wil again delegate the request to its parent, if it exists. If there is no parent, configured defaults are examined. These are taken from rootDir/etc/defaultmeta, where they are given as colon-separated key-value pairs, e.g.,

```
publisher: The GAVO DC team
publisherID: ivo://org.gavo.dc
contact.name: GAVO Data Center Team
contact.address: Moenchhofstrasse 12-14, D-69120 Heidelberg
contact.email: gavo@ari.uni-heidelberg.de
contact.telephone: ++49 6221 54 1837
creator.name: GAVO Data Center
creator.logo: http://vo.ari.uni-heidelberg.de/docs/GavoTiny.png
```

The effect is that you can give global titles, descriptions, etc. in the RD but override them in services, tables, etc. The configured defaults let you specify meta items that are probably constant for everything in your data center, though of course you can override these in your RD elements, too.

In HTML templates, missing meta usually is not an error. The corresponding elements are just left empty. In registry documents, missing meta may be an error.

Meta formats

Metadata must work in registry records as well as in HTML pages and possibly in other places. Thus, it should ideally be given in formats that can be sensibly transformed into the various formats.

The GAVO DC software knows four input formats:

literal The textual content of the element will not be touched. In HTML, it will end up in a div block of class literalmeta.

plain The textual content of the element will be whitespace-normalized, i.e., whitespace will be stripped from the start and the end, runs of blanks and tabs are replaced by a single blank, and empty lines translate into paragraphs. In HTML, these blocks com in plainmeta div elements.

rst The textual content of the element is interpreted as ReStructuredText. When requested as plain text, the ReStructuredText itself is returned, in HTML, the standard docutils rendering is returned.

raw The textual content of the element is not touched. It will be embedded into HTML directly. You can use this, probably together with CDATA sections, to embed HTML -- the other formats should not contain anything special to HTML (i.e., they should be PCDATA in XML lingo). While the software does not enforce this, raw content should not be used with RMI-type metadata. Only use it for items that will not be rendered outside of HTML templates.

Macros in Meta Elements

Macros will be expanded in meta items using the embedding element as macro processors (i.e., you can use the macros defined by this element).

Typed Meta Elements

While generally the DC software does not care what you put into meta items and views them all as strings, certain keys are treated specially. The following meta keys trigger some special behaviour:

_example A MetaValue to keep VOSI examples in.

All of these must have a title, which is also used to generate references.

These also are in reStructuredText by default, and changing that probably makes no sense at all, as these will always need interpreted text roles for proper markup.

Thus, the usual pattern here is:

```
<meta name="_example" title="An example for _example">
  See docs_

  .. _docs: http://docs.g-vo.org
</meta>
```

_news A meta value representing a "news" items.

The content is the body of the news. In addition, they have date, author, and role children. In plain text, you would write:

```
_news: Frobnicated the quux.
_news.author: MD
_news.date: 2009-03-06
_news.role: updated
```

In XML, you would usually write:

```
<meta name="_news" author="MD" date="2009-03-06">
  Frobnicated the quux.
</meta>
```

_news items become serialised into Registry records despite their leading underscores. role then becomes the date's role.

_related A meta value containing a link and optionally a title

In plain text, this would look like this:

```
_related:http://foo.bar
_related.title: The foo page
```

In XML, you can write:

```
<meta name="_related" title="The foo page"
  ivoId="ivo://bar.org/foo">http://foo.bar</meta>
```

or, if you prefer:

```
<meta name="_related">http://foo.bar
   <meta name="title">The foo page</meta></meta>
```

These values are used for _related (meaning "visible" links to other services).

For links within you data center, use the internallink macro, the argument of which the the "path" to a resource, i.e. RD path/service/renderer; we recommend to use the info renderer in such links as a rule. This would look like this:

```
<meta name="_related" title="Aspec SSAP"
  >\internallink{aspec/q/ssa/info}</meta>
```

cites A meta value containing an ivo-id and a name of a related resource.

These all are translated to relationship elements in VOResource renderings. These correspond to the terms in the official relationship vocabulary http://docs.g-vo.org/vocab-test/relationship_type. There, the camelCase terms are preferred, and for DaCHS meta, they are written with a lowercase initial.

Relationship metas should look like this:

```
servedBy: GAVO TAP service
servedBy.ivoId: ivo://org.gavo.dc
```

`servedBy` and `serviceFor` are somewhat special cases, as the service attribute of data publications automatically takes care of them; so, you shouldn't usually need to bother with these two manually.

continues A meta value containing an ivo-id and a name of a related resource.

These all are translated to relationship elements in VOResource renderings. These correspond to the terms in the official relationship vocabulary

http://docs.g-vo.org/vocab-test/relationship_type. There, the camelCase terms are preferred, and for DaCHS meta, they are written with a lowercase initial.

Relationship metas should look like this:

```
servedBy: GAVO TAP service
servedBy.ivoId: ivo://org.gavo.dc
```

`servedBy` and `serviceFor` are somewhat special cases, as the service attribute of data publications automatically takes care of them; so, you shouldn't usually need to bother with these two manually.

creator.logo A MetaValue corresponding to a small image.

These are rendered as little images in HTML. In XML meta, you can say:

```
<meta name="_somelogo" type="logo">http://foo.bar/quux.png</meta>
```

derivedFrom A meta value containing an ivo-id and a name of a related resource.

These all are translated to relationship elements in VOResource renderings. These correspond to the terms in the official relationship vocabulary http://docs.g-vo.org/vocab-test/relationship_type. There, the camelCase terms are preferred, and for DaCHS meta, they are written with a lowercase initial.

Relationship metas should look like this:

```
servedBy: GAVO TAP service
servedBy.ivoId: ivo://org.gavo.dc
```

`servedBy` and `serviceFor` are somewhat special cases, as the service attribute of data publications automatically takes care of them; so, you shouldn't usually need to bother with these two manually.

hasPart A meta value containing an ivo-id and a name of a related resource.

These all are translated to relationship elements in VOResource renderings. These correspond to the terms in the official relationship vocabulary http://docs.g-vo.org/vocab-test/relationship_type. There, the camelCase terms are preferred, and for DaCHS meta, they are written with a lowercase initial.

Relationship metas should look like this:

```
servedBy: GAVO TAP service
servedBy.ivoId: ivo://org.gavo.dc
```

`servedBy` and `serviceFor` are somewhat special cases, as the service attribute of data publications automatically takes care of them; so, you shouldn't usually need to bother with these two manually.

info A meta value for info items in VOTables.

In addition to the content (which should be rendered as the info element's text content), it contains an infoName and an infoValue.

They are only used internally in VOTable generation and might go away without notice.

isContinuedBy A meta value containing an ivo-id and a name of a related resource.

These all are translated to relationship elements in VOResource renderings. These correspond to the terms in the official relationship vocabulary http://docs.g-vo.org/vocab-test/relationship_type. There, the camelCase terms are preferred, and for DaCHS meta, they are written with a lowercase initial.

Relationship metas should look like this:

```
servedBy: GAVO TAP service
servedBy.ivoId: ivo://org.gavo.dc
```

`servedBy` and `serviceFor` are somewhat special cases, as the service attribute of data publications automatically takes care of them; so, you shouldn't usually need to bother with these two manually.

isDerivedFrom A meta value containing an ivo-id and a name of a related resource.

These all are translated to relationship elements in VOResource renderings. These correspond to the terms in the official relationship vocabulary http://docs.g-vo.org/vocab-test/relationship_type. There, the camelCase terms are preferred, and for DaCHS meta, they are written with a lowercase initial.

Relationship metas should look like this:

```
servedBy: GAVO TAP service
servedBy.ivoId: ivo://org.gavo.dc
```

`servedBy` and `serviceFor` are somewhat special cases, as the service attribute of data publications automatically takes care of them; so, you shouldn't usually need to bother with these two manually.

isIdenticalTo A meta value containing an ivo-id and a name of a related resource.

These all are translated to relationship elements in VOResource renderings. These correspond to the terms in the official relationship vocabulary http://docs.g-vo.org/vocab-test/relationship_type. There, the camelCase terms are preferred, and for DaCHS meta, they are written with a lowercase initial.

Relationship metas should look like this:

```
servedBy: GAVO TAP service
servedBy.ivoId: ivo://org.gavo.dc
```

`servedBy` and `serviceFor` are somewhat special cases, as the service attribute of data publications automatically takes care of them; so, you shouldn't usually need to bother with these two manually.

isNewVersionOf A meta value containing an ivo-id and a name of a related resource.

These all are translated to relationship elements in VOResource renderings. These correspond to the terms in the official relationship vocabulary http://docs.g-vo.org/vocab-test/relationship_type. There, the camelCase terms are preferred, and for DaCHS meta, they are written with a lowercase initial.

Relationship metas should look like this:

```
servedBy: GAVO TAP service
servedBy.ivoId: ivo://org.gavo.dc
```

`servedBy` and `serviceFor` are somewhat special cases, as the service attribute of data publications automatically takes care of them; so, you shouldn't usually need to bother with these two manually.

isPartOf A meta value containing an ivo-id and a name of a related resource.

These all are translated to relationship elements in VOResource renderings. These correspond to the terms in the official relationship vocabulary http://docs.g-vo.org/vocab-test/relationship_type. There, the camelCase terms are preferred, and for DaCHS meta, they are written with a lowercase initial.

Relationship metas should look like this:

```
servedBy: GAVO TAP service
servedBy.ivoId: ivo://org.gavo.dc
```

`servedBy` and `serviceFor` are somewhat special cases, as the service attribute of data publications automatically takes care of them; so, you shouldn't usually need to bother with these two manually.

isPreviousVersionOf A meta value containing an ivo-id and a name of a related resource.

These all are translated to relationship elements in VOResource renderings. These correspond to the terms in the official relationship vocabulary http://docs.g-vo.org/vocab-test/relationship_type. There, the camelCase terms are preferred, and for DaCHS meta, they are written with a lowercase initial.

Relationship metas should look like this:

```
servedBy: GAVO TAP service
servedBy.ivoId: ivo://org.gavo.dc
```

`servedBy` and `serviceFor` are somewhat special cases, as the service attribute of data publications automatically takes care of them; so, you shouldn't usually need to bother with these two manually.

isServedBy A meta value containing an ivo-id and a name of a related resource.

These all are translated to relationship elements in VOResource renderings. These correspond to the terms in the official relationship vocabulary http://docs.g-vo.org/vocab-test/relationship_type. There, the camelCase terms are preferred, and for DaCHS meta, they are written with a lowercase initial.

Relationship metas should look like this:

```
servedBy: GAVO TAP service
servedBy.ivoId: ivo://org.gavo.dc
```

`servedBy` and `serviceFor` are somewhat special cases, as the service attribute of data publications automatically takes care of them; so, you shouldn't usually need to bother with these two manually.

isServiceFor A meta value containing an ivo-id and a name of a related resource.

These all are translated to relationship elements in VOResource renderings. These correspond to the terms in the official relationship vocabulary http://docs.g-vo.org/vocab-test/relationship_type. There, the camelCase terms are preferred, and for DaCHS meta, they are written with a lowercase initial.

Relationship metas should look like this:

```
servedBy: GAVO TAP service
servedBy.ivoId: ivo://org.gavo.dc
```

`servedBy` and `serviceFor` are somewhat special cases, as the service attribute of data publications automatically takes care of them; so, you shouldn't usually need to bother with these two manually.

isSourceOf A meta value containing an ivo-id and a name of a related resource.

These all are translated to relationship elements in VOResource renderings. These correspond to the terms in the official relationship vocabulary http://docs.g-vo.org/vocab-test/relationship_type. There, the camelCase terms are preferred, and for DaCHS meta, they are written with a lowercase initial.

Relationship metas should look like this:

```
servedBy: GAVO TAP service
servedBy.ivoId: ivo://org.gavo.dc
```

`servedBy` and `serviceFor` are somewhat special cases, as the service attribute of data publications automatically takes care of them; so, you shouldn't usually need to bother with these two manually.

isSupplementTo A meta value containing an ivo-id and a name of a related resource.

These all are translated to relationship elements in VOResource renderings. These correspond to the terms in the official relationship vocabulary http://docs.g-vo.org/vocab-test/relationship_type. There, the camelCase terms are preferred, and for DaCHS meta, they are written with a lowercase initial.

Relationship metas should look like this:

```
servedBy: GAVO TAP service
servedBy.ivoId: ivo://org.gavo.dc
```

`servedBy` and `serviceFor` are somewhat special cases, as the service attribute of data publications automatically takes care of them; so, you shouldn't usually need to bother with these two manually.

isSupplementedBy A meta value containing an ivo-id and a name of a related resource.

These all are translated to relationship elements in VOResource renderings. These correspond to the terms in the official relationship vocabulary http://docs.g-vo.org/vocab-test/relationship_type. There, the camelCase terms are preferred, and for DaCHS meta, they are written with a lowercase initial.

Relationship metas should look like this:

```
servedBy: GAVO TAP service
servedBy.ivoId: ivo://org.gavo.dc
```

`servedBy` and `serviceFor` are somewhat special cases, as the service attribute of data publications automatically takes care of them; so, you shouldn't usually need to bother with these two manually.

logo A MetaValue corresponding to a small image.

These are rendered as little images in HTML. In XML meta, you can say:

```
<meta name="_somelogo" type="logo">http://foo.bar/quux.png</meta>
```

mirrorOf A meta value containing an ivo-id and a name of a related resource.

These all are translated to relationship elements in VOResource renderings. These correspond to the terms in the official relationship vocabulary

http://docs.g-vo.org/vocab-test/relationship_type. There, the camelCase terms are preferred, and for DaCHS meta, they are written with a lowercase initial.

Relationship metas should look like this:

```
servedBy: GAVO TAP service
servedBy.ivoId: ivo://org.gavo.dc
```

`servedBy` and `serviceFor` are somewhat special cases, as the service attribute of data publications automatically takes care of them; so, you shouldn't usually need to bother with these two manually.

note A meta value representing a "note" item.

This is like a footnote, typically on tables, and is rendered in table infos.

The content is the note body. In addition, you want a tag child that gives whatever the note is references as. We recommend numbers.

Contrary to other meta items, note content defaults to rstx format.

Typically, this works with a column's note attribute.

In XML, you would usually write:

```
<meta name="note" tag="1">
  Better ignore this.
</meta>
```

referenceURL A meta value containing a link and optionally a title

In plain text, this would look like this:

```
_related:http://foo.bar
_related.title: The foo page
```

In XML, you can write:

```
<meta name="_related" title="The foo page"
  ivoId="ivo://bar.org/foo">http://foo.bar</meta>
```

or, if you prefer:

```
<meta name="_related">http://foo.bar
   <meta name="title">The foo page</meta></meta>
```

These values are used for _related (meaning "visible" links to other services).

For links within you data center, use the internallink macro, the argument of which the the "path" to a resource, i.e. RD path/service/renderer; we recommend to use the info renderer in such links as a rule. This would look like this:

```
<meta name="_related" title="Aspec SSAP"
  >\internallink{aspec/q/ssa/info}</meta>
```

relatedTo A meta value containing an ivo-id and a name of a related resource.

These all are translated to relationship elements in VOResource renderings. These correspond to the terms in the official relationship vocabulary http://docs.g-vo.org/vocab-test/relationship_type. There, the camelCase terms are preferred, and for DaCHS meta, they are written with a lowercase initial.

Relationship metas should look like this:

```
servedBy: GAVO TAP service
servedBy.ivoId: ivo://org.gavo.dc
```

`servedBy` and `serviceFor` are somewhat special cases, as the service attribute of data publications automatically takes care of them; so, you shouldn't usually need to bother with these two manually.

servedBy A meta value containing an ivo-id and a name of a related resource.

These all are translated to relationship elements in VOResource renderings. These correspond to the terms in the official relationship vocabulary http://docs.g-vo.org/vocab-test/relationship_type. There, the camelCase terms are preferred, and for DaCHS meta, they are written with a lowercase initial.

Relationship metas should look like this:

```
servedBy: GAVO TAP service
servedBy.ivoId: ivo://org.gavo.dc
```

`servedBy` and `serviceFor` are somewhat special cases, as the service attribute of data publications automatically takes care of them; so, you shouldn't usually need to bother with these two manually.

serviceFor A meta value containing an ivo-id and a name of a related resource.

These all are translated to relationship elements in VOResource renderings. These correspond to the terms in the official relationship vocabulary http://docs.g-vo.org/vocab-test/relationship_type. There, the camelCase terms are preferred, and for DaCHS meta, they are written with a lowercase initial.

Relationship metas should look like this:

```
servedBy: GAVO TAP service
servedBy.ivoId: ivo://org.gavo.dc
```

`servedBy` and `serviceFor` are somewhat special cases, as the service attribute of data publications automatically takes care of them; so, you shouldn't usually need to bother with these two manually.

source A MetaValue that may contain bibcodes, which are rendered as links into ADS.

uses A meta value containing an ivo-id and a name of a related resource.

These all are translated to relationship elements in VOResource renderings. These correspond to the terms in the official relationship vocabulary http://docs.g-vo.org/vocab-test/relationship_type. There, the camelCase terms are preferred, and for DaCHS meta, they are written with a lowercase initial.

Relationship metas should look like this:

```
servedBy: GAVO TAP service
servedBy.ivoId: ivo://org.gavo.dc
```

`servedBy` and `serviceFor` are somewhat special cases, as the service attribute of data publications automatically takes care of them; so, you shouldn't usually need to bother with these two manually.

votlink A MetaValue serialized into VOTable links (or, ideally, analoguous constructs).

This exposes the various attributes of VOTable LINKs as href linkname, contentType, and role. You cannot set ID here; if this ever needs referencing, we'll need to think about it again. The href attribute is simply the content of our meta (since there's no link without href), and there's never any content in VOTable LINKs).

You could thus say:

```
votlink: http://docs.g-vo.org/DaCHS
votlink.role: doc
votlink.contentType: text/html
votlink.linkname: GAVO DaCHS documentation
```

Additionally, there is `creator`, which is really special (at least for now). When you set creator to a string, the string will be split at semicolons, and for each substring a creator item with the respective name is generated. This may sound complicated but really does about what your would expect when you write:

```
<meta name="creator">Last, J; First, B; Middle, I.</meta>
```

Metadata in Standard Renderers

Certain meta keys have a data center-internal interpretation, used in renderers or writers of certain formats. These keys should always start with an underscore. Among those are:

- _intro -- used by the standard HTML template for explanatory text above the seach form.
- _bottominfo -- used by the standard HTML template for explanatory text below the seach form.
- _copyright -- used by the standard HTML template for copyright-related information (there's also copyright in RMI; the one with the underscore is intended to be less formal).
- _related -- used in the standard HTML template for links to related services. As listed above, this is a link, i.e., you can give a title attribute.
- _longdoc -- used by the service info renderer for an explanatory piece of text of arbitrary length. This will usually be in ReStructuredText, and we recommend having the whole meta body in a CDATA section.
- _news -- news on the service. See above at Typed Meta Elements.
- _warning -- used by both the VOTable and the HTML table renderer. The content is rendered as some kind of warning. Unfortunately, there is no standard how to do this in VOTables. There is no telling if the info elements generated will show anywhere.
- _noresultwarning -- displayed by the default response template instead of an empty table (use it for things like "No Foobar data for your query")
- _type -- on Data instances, used by the VOTable writer to set the `type` attribute on `RESOURCE` elements (to either "results" or "meta"). Probably only useful internally.
- _plotOptions – typically set on services, this lets you configure the initial appearance of the javascript-based quick plot. The value must be a javascript dictionary literal (like `{"xselIndex": 2}`) unless you're trying CSS deviltry (which you could, using this meta; then again, if you can inject RDs, you probably don't need CSS attacks). Keys evaluated include:
  - xselIndex – 0-based index of the column plotted on the x-axis (default: 0)
  - yselIndex – 0-based index of the column plotted on the y-axis (default: length of the column list; that's "histogram on y)
  - usingIndex – 0-based index of the plotting style selector. For now, that's 0 for points and 1 for lines.

RMI-Style Metadata

For services (and other things) that are registred in the Registry, you must give certain metadata items (and you can give more), where we take their keys from [RMI]. We provide a explanatory leaflet for data providers. The most common keys -- used by the registry interface and in part by HTML and VOTable renderers -- include:

- title -- this should in general be given seperately on the resource, each table, and each service. In simple cases, though, you may get by by just having one global title on the resource and rely on metdata inheritance.
- shortName -- a string that should indicate what the service is in 16 characters or less.
- creationDate -- Use ISO format with time, UTC only, like this: 2007-10-04T12:00:00Z
- subject -- as noted in the explanatory leaflet, these should be taken from the IVOA Vocabulary Explorer.
- copyright -- freetext copyright notice.
- source -- bibcodes will be expanded to ADS links here.
- referenceURL -- again, a link, so you can give a title for presentation purposes. If you give no referenceURL, the service's info page will be used.
- dateUpdated -- an ISO date. Do not set this. This is determined from timestamps in DaCHS's state directory. There is also datetimeUpdated that you would have to keep in sync with dateUpdated if you were to change it.
- creator.name -- this should be the name of the "author" of the data set. See below for multiple creators. If you set this, you may want to override creator.logo as well.
- type – one of Other, Archive, Bibliography, Catalog, Journal, Library, Simulation, Survey, Transformation, Education, Outreach, EPOResource, Animation, Artwork, Background, BasicData, Historical, Photographic, Press, Organisation, Project, Registry – it's optional and we doubt its usefulness.
- contentLevel – addresse(s) of the data: Research, Amateur, General
- facility – no IVOA ids are supported here yet, but probably this should change.

- coverage – see the special section
- service-specific metadata (for SIA, SCS, etc.) – see the documentation of the respective cores.
- utype – tables (and possibly other items) can have utypes to signify their role in specific data models. For tables, this utype gets exported to the tap_schema.
- identifier – this is the IVOID of the resource, usually generated by DaCHS. Do not override this unless you know what you are doing (which at least means you know how to make DaCHS declare an authority and claim it). If you do override the identifier of a service that's already published, make sure you run `gavo admin makeDeletedRecord <previous identifier>` (before or after the `gavo pub` on the resource, or the registries will have two copies of your record, one of which will not be updated any more; and that would suck for Registry users.
- mirrorURL – add these on publication to declare mirrors for a service. Only do so if you actually manage the other service. If you list the service's own accessURL here, it will be filtered from this registry record; this is so you can use the same RD on the primary site and the mirror.

While you can set any of these in etc/defaultmeta.txt, the following items are usually set there:

- publisher
- publisherID
- contact.name
- contact.address
- contact.email
- contact.telephone

Coverage Metadata

Coverage metadata lets clients get a quick idea of where in space, time, and electromagnetic spectrum the data within a resource is. Obviously, this information is particularly important for resource discovery in registries.

Not all resources have coverages on all axes; a service validator, say, probably has no physical coverage at all, and a theoretical spectral service may just have meaningful spectral coverage.

There are two meta keys pertinent to coverage metadata:

- `coverage.waveband` – One of Radio, Millimeter, Infrared, Optical, UV, EUV, X-ray, Gamma-ray, and you can have multiple waveband specifications. As this information is quite regularly used in discovery you should make sure to define it if applicable.

- `coverage.regionOfRegard` – in essence, the "pixel size" of the service in degrees. If, for example, your service gives data on a lattice of sampling points, the typical distance of such points should be given here. You will not usually specify this unless your „pixel size" is significantly larger than about an arcsec.

The legacy `coverage.profile` meta key should not be used any more.

To give proper, numeric STC coverage, use the Element coverage.

It has three children, one each for the spatial, spectral, and temporal axes. For spectral and temporal, just add as many intervals as necessary. Do not worry about gaps in the temporal coverage: it is not necessary that the coverage is "tight"; as long as there is a reasonable expectation that data *could* be there, it's fine to declare coverage. Hence, for ground-based observations, there is no need to exclude intervals of daylight, bad weather, or even maintenance downtime.

Intervals are given as in VOTable tabledata, i.e., as two floating point numbers separated by whitespace. There are no (half-) open intervals – just use insanely small or large numbers if you really think you need them.

For spatial coverage, a single `spatial` element should be given. It has to contain a MOC in ASCII serialisation. Recent versions of Aladin can generate those, or you can write SQL queries to have them computed by sufficiently new versions of pgsphere. Most typically, you will use `updater` elements to fill spatial coverage (see below).

A complete coverage element would thus look like this:

```
<coverage>
  <spectral>3.8e-07 5.2e-07</spectral>
  <temporal>18867 27155</temporal>
  <spatial>
    4/2068
    5/8263,8268-8269,8271,8280,8323,8326,8329,9376,9378
    6/33045-33047,33049,33051,33069,33080-33081,33083,33104-33106,
      33112,33124-33126,33128-33130,33287,33289,33291,33297-33299,
      33313,33315,33323-33326,33328-33330,37416,37418,37536
  </spatial>
</coverage>
```

In general computing coverage is a tedious task. Hence, DaCHS has rules to compute it for many common cases (SSAP, SIAP, Obscore, catalogs with usable UCDs). Because coverage calculations can run for a long time, they are not performed online. Instead, DaCHS updates coverage elements when the operator runs `dachs limits`. In the simplest case, operators add:

```
<coverage>
  <updater sourceTable="data"/>
  <spectral/>
  <temporal/>
  <spatial/>
</coverage>
```

into an RD with a table named data. Currently, this must be lexically below the table element, but if this isn't fixed to allow the location of the coverage element near the rest of the metadata near the top of the RD, complain fiercely.

Operators then run `dachs limits q` (assuming the RD is called `q.rd`), and DaCHS will fill out the three coverage elements (in case you want to fix them: the heuristics it uses to do that are in gavo.user.info).

In this construction, DaCHS will overwrite any previous content in the coverage child elements. If you want to fill out some coverage items manually and have DaCHS only compute, say, the spatial coverage, don't give the `sourceTable` attribute (which essentially says: "grab as much coverage from the referenced table as you can") but rather the specialised `spaceTable`. This is particularly useful if you want to annotate "holes" in your temporal coverage. For instance, if your resource contains two fairly separate campaigns (which DaCHS does not currently realise automatically):

```
<coverage>
  <updater spaceTable="main"/>
  <spatial/>
  <temporal>45201 45409</temporal>
  <temporal>54888 55056</temporal>
</coverage>
```

Due to limitations of pgsphere, DaCHS does not currently take into account the size of the items in a database table. While that is probably all right for spectra and catalogs, for images this might lose significant coverage, as DaCHS only uses the centers of the images and just marks the containing healpix of the selected MOC order. The default MOC order is 6 (a resolution of about a degree). Until we properly deal with polygons, make sure to increase the MOC order to at least the order of magnitude of the images in an image service, like this:

```
<coverage>
 <updater sourceTable="main" mocOrder="4"/>
 <spatial/>
</coverage>
```

If you know your resource only contains relatively few but compact patches, you may also want to increase `mocOrder` (spatial resolution doubles when you increase mocOrder by one).

Display Hints

Display hints use an open vocabulary. As you add value formatters, you can evaluate any display hint you like. Display hints understood by the built-in value formatters include:

displayUnit use the value of this hint as the unit to display a value in.

nopreview if this key is present with any value, no HTML code to generate previews when mousing over a link will be generated.

sepChar a separation character for sexagesimal displays and the like.

sf "Significant figures" -- length of the mantissa for this column. Will probably be replaced by a column attribute analoguous to what VOTable does.

type a key that gives hints what to do with the column. Values currently understood include:

- **bar** display a numeric value as a bar of length value pixels.
- **bibcode** display the value as a link to an ADS bibcode query.
- **checkmark** in HTML tables, render this column as empty or checkmark depending on whether the value is false or true to python.
- **humanDate** display a timestamp value or a real number in either yr (julian year), d (JD, or MJD if DaCHS guesses it's mjd; that's unfortunately arcane still), or s (unix timestamp) as an ISO string.
- **humanDay** display a timestamp or date value as an ISO string without time.
- **humanTime** display values as h:m:s.
- **keephtml** lets you include raw HTML. In VOTables, tags are removed.
- **product** treats the value as a product key and expands it to a URL for the product (i.e., typically image). This is defined in protocols.products. This display hint is also used by, e.g., the tar format to identify which columns should contribute to the tar file.
- **dms** format a float as degree, minutes, seconds.

simbadlink formats a column consisting of alpha and delta as a link to query simbad. You can add a coneMins displayHint to specify the search radius.

hms force formatting of this column as a time (usually for RA).

url makes value a link in HTML tables. The anchor text will be the last element of the path part of the URL, or, if given, the value of the anchorText property of the column (which is for cases when you want a constant text like "Details"). If you need more control over the anchor text, use an outputField with a formatter.

imageURL makes value the src of an image. Add width to force a certain image size.

noxml if 'true' (exactly like this), do not include this column in VOTables.

Note that not any combination of display hints is correctly interpreted. The interpretation is greedy, and only one formatter at a time attempts to interpret display hints.

Data Model Annotation

In the VO, data models are used when simple, more or less linear annotation methods like UCDs do not provide sufficent expressive power. Or well, they should be used. As of early 2017, things are, admittedly, still a mess.

DaCHS lets you annotate your data in `dm` elements; the annotation will then be turned into standard VOTable annotation (when that's defined). Sometimes, the structured references provided by the DM annotation are useful elsewhere, too – the first actual use of this framework was the geojson serialisation discussed below.

We first discuss SIL, then its use in actual data models. At least skim over the next section – it sucks to discover the SIL grammar by trial and error.

Old-style STC annotation is not discussed here. If you still want to do it (and for now, you have to if you want any STC annotation – sigh), check out the terse discussion in the tutorial

Annotation Using SIL

Data model annotation in DaCHS is done using SIL, the Simple Instance Language. It essentially resembles JSON, but all delimiters not really necessary for our use case have been dropped, and type annotation has been added.

The elements of SIL are:

- Atomic Values. For SIL, everything is a string (it's a problem of DM validation to decide otherwise). When your string consists exclusively of alphanumerics and [._-], you can just write it in SIL. Otherwise, you must use double quotes. as in SQL, write two double quotes to include a literal double quote. So, valid literals in SIL are
  - `2.3e-3`
  - `red`
  - `"white and blue"`
  - `"""Yes,"" the computer said."`
  - `"could write (type:foo) {bar.  baz} here"` (elements of SIL are protected in quoted literals)

  Invalid literals include:
  - `http://www.g-vo.org` (: and / may not occur in literals)
  - `red, white and blue` (no blanks and commas)
  - `22"` (no single quotes)
- Plain Identifiers. These are C-like identifiers (a letter or an underscore optionally followed by letters, number or underscores).
- Comments. SIL comments are classical C-style comments (`/*...*/`). They don't nest yet, but they probably will at some point, so don't write `/*` within a comment.
- Object annotation. This is like a dictionary; only plain identifiers are allowed as keys. So, an object looks like this:

```
{ foo: bar
  longer: "This is a value with blanks in it"
}
```

  Note again that no commas or quotes around the keys are necessary (or even allowed).
- Sequences. This is like a list. Members can be atomic or objects, but they have to be homogenous (SIL doesn't enforce this by grammatical means, though. Here is an object with two sequences:

```
{ seq1: [3 4 5 "You thought these were numbers? They're strings!"]
  seq2: [
    { seq_index: 0 value: 3.3}
    { seq_index: 1 value: 0.3}
  ]
}
```

- References. The point of SIL is to say things about column and param instances. Both of them (and other dm instances, tables, and in principle anything else in RDs) can be referenced from within SIL. A reference starts with an @ and is then a normal DaCHS cross identifier (columns and params within a table can be referenced by name only, columns take precedence on name clashes). If you use odd characters in your RD names or in-RD identifiers, think again: only [._/#-] are allowed in such references. Here is an object with some valid references:

```
{ long: @raj2000  /* a column in the enclosing table */
  lat: @dej2000
  system: @//systems#icrs /* could be a dm instance in a
    DaCHS-global RD;, this does *not* exist yet */
  source: @supercat/q#main /* perhaps a table in another RD */
}
```

- Casting. You can (and sometimes have to) give explicit types in the SIL annotation. Types look like C-style casts. The root of a SIL annotation must always have a cast; that allows DaCHS to figure out what it is, which is essential for validation (and possibly inference of defaults and such). You can cast both single objects and sequences. Here's an example that actually validates for DaCHS' SIL (which the examples above wouldn't because they're missing the root annotation):

```
(testdm:testclass) {  /* cast on root: mandatory */
  attr1 {   /* no cast here; DaCHS can infer attr1's type if necessary */
    attr2: val
  }
  seq: (testdm:otherclass)[  /* Sequence cast: */
    {attr1: a}  /* all of these are now treated as testdm:otherclass */
    {attr1: b}
    {attr1: c}]}
```

GeoJSON annotation

To produce GeoJSON output (as supported by DaCHS' TAP implementation), DaCHS needs to know what the "geometry" in the sense of GeoJSON is. Furthermore, DaCHS keeps supporting declaring reference systems in the `crs` attribute, as the planetology community uses it.

The root class of the geojson DM is `geojson:FeatureCollection`. It has up to two attributes (*crs* and *feature*), closely following the GeoJSON structure itself. The geometry is defined in *feature*'s *geometry* attribute. All columns not used for geometry will end up in GeoJSON properties.

So, a complete GeoJSON annotation, in this case for an EPN-TAP table, could look like this:

```
<table>
   <dm>
    (geojson:FeatureCollection){
      crs: (geojson:CRS) {
          type: name
          properties: (geojson:CRSProperties) {
            name: "urn:x-invented:titan"}}}}
      feature: {
        geometry: {
          type: sepsimplex
          c1min: @c1min
          c2min: @c2min
          c1max: @c1max
          c2max: @c2max }}}
  </dm>

  <mixin
    spatial_frame_type="body"/>
</table>
```

Yes, the use *type* attributes is a bit of an abomination, but we wanted the structure to follow GeoJSON in spirit.

The *crs* attribute could also be of *type* `link`, in which case the *properties* would have attributes *href* and *type*; we're not aware of any applications of this in planetology, though. *crs* is optional (but standards-compliant GeoJSON clients will interpret your coordinates as WGS84 on Earth if you leave it out).

For *geometry*, several values for *type* are defined by DaCHS, depending on how the GeoJSON geometry should be constructed from the table. Currently defined types include (complain if you need something else, it's not hard to add):

- `sepcoo` – this is for a spherical point with separate columns for the two axes. This needs *latitude* and *longitude* attributes, like this:

```
<dm>
  (geojson:FeatureCollection){
    feature: {
      geometry: {
        type: sepcoo
        latitude: @lat
        longitude: @long }}}
</dm>
```

- `seppoly` – this constructs a spherical polygon out of column references. These have the form *c_n_m*, where m is 1 or 2, and n is counted from 1 up to the number of points. DaCHS will stop collecting points as soon as it doesn't find an expected key. If you find yourself using this, check your data model. An example:

```
<dm>
  (geojson:FeatureCollection){
    feature: {
      geometry: {
        type: seppoly /* a triangle of some kind */
        c1_1: @rb0
        c1_2: @rb1
        c2_1: @lb0
        c2_2: @lb1
        c3_1: @t0
        c3_2: @t1 }}}
</dm>
```

- `sepsimplex` – this constructs a spherical box-like thing from minimum and maximum values. It has *c[12](min|max)* keys as in EPN-TAP. As a matter of fact, a fairly typical annotation for EPN-TAP would be:

```
<dm>
  (geojson:FeatureCollection){
    feature: {
      geometry: {
        type: sepsimplex
        c1min: @c1min
        c2min: @c2min
        c1max: @c1max
        c2max: @c2max }}}
</dm>
```

- `geometry` – this constructs a geometry from a pgsphere column. Since GeoJSON doesn't have circles, only spoint and spoly columns can be used. They are referenced from the *value* key. For instance, obscore and friends could use:

```
<dm>
  (geojson:FeatureCollection) {
    feature:  {
        geometry:  {
          type: geometry
          value: @s_region }}}
  </dm>
```

DaCHS' Service Interface

Even though normal users should rarely be confronted with too many of the technical details of request processing in DaCHS, it helps to have a rough comprehension in order to understand several user-visible details.

In DaCHS' architecture, a service is essentially a combination of a core and a renderer. The core is what actually does the query or the computation, the renderer adapts input and outputs to what a protocol or interface expects. While a service always has exactly one core (could be a nullCore, though), it

can support more than one renderer, although the parameters in all renderers are, within reason, about the same, within reason.

However, parameters on a form interface will typcially be interpreted differently from a VO interface on the same core. For instance, ranges on the form interface are written as `1 .. 3` (VizieR compliance), on an SSA 1.x interface `1/3` ("PQL" prototype), and on a datalink dlget interface "1 2" (DALI 1.1 style). The extreme of what probably still makes sense is the core search core that replaces SCS's RA, DEC, and SR with an entirely different set of parameters perhaps better suited for interactive, browser-based usage.

Cores communicate their input interface by defining an input table, which is essentially a sequence of input keys, which in turn essentially work like params: in particular, they have all the standard metadata like units, ucds, etc. Input tables, contrary to what their name might suggest, have no rows. They can hold metadata, though, which is sometimes convenient to pass data between parameter parsers and the core.

When a request comes in, the service first determines the renderer responsible. It then requests an inputTable for that renderer from the core. The core, in turn, will map each inputKey in its inputTable through a renderer adaptor as returned from svcs.inputdef.getRendererAdaptor; this inspects the renderer.parameterStyle, which must be taken from the svcs.inputdef._RENDERER_ADAPTORS' keys (currently `form`, `pql`, `dali`). inputKeys have to have the adaptToRenderer property set to True to have them adapted. Most automatically generated inputKeys have that; where you manually define inputKeys, you would have to set the property manually if you want that behaviour (and know that you want it; outside of table-based cores, it is unikely that you do).

Core Args

The input table, together with the raw arguments coming from the client, is then used to build a `svcs.CoreArgs` instance, which in turn takes the set of input keys to build a context grammar. The core args have the underlying input table (with the input keys for the metadata) in the `inputTD` attribute, the parsed arguments in the dictionary `args`.

For each input key `args` maps its name to a value; context grammars are case-semisensitive, meaning that case in the HTTP parameter names is in general ignored, but if a parameter name matching case is found, it is preferred. Yes, ugly, but unfortunately the VO has started with case-insensitive parameter names. Sigh.

The values in `args` are a bit tricky:

- each raw parameter given must parse with a single inputKey's parse. For instance, if an inputKey is a real[2], it will be parsed as a float array.
- if no raw parameter is given for an input key, its value will be None.
- when an inputKey specifies multiplicity="multiple", the non-None value in the core args is a list. Each list item is something that came out of the inputKey's parser (i.e., it could be another list for array-valued parameters).
- when an inputKey specifies multiplicity="single", the value in the core args is a single value of whatever inputKey parses (or None for missing parameters). This is even true when a parameter has been given multiple times; while currently, the last parameter will win, we don't guarantee that.
- when an inputKey specifies multiplicity="force-single", DaCHS works as in the single case, except that multiple specification will lead to an error.
- when an inputKey does not specify multiplicity, DaCHS will infer the desired multiplicity from various hints; essentially, enumerated parameters (values/options given in some way) have multiplicity multiple, everything else multiplicity single. It is wise not to rely on this behaviour.

These rules are independent of the type of core and hold for pythonCores or whatever just as for the normal, table-based cores. For these (and they are what users are mostly concerned with), special rules and shortcuts apply, though.

Table-based cores

Conddescs and input keys: Defining the input parameters

You will usually deal with cores querying database tables – dbCore, ssapCore, etc. For these, you will not normally define an inputTable, as it is being generated by the software from condDescs.

To create simple constraints, just `buildFrom` the columns queried:

```
<condDesc buildFrom="myColumn"/>
```

(the names are resolved in the core's queried table). DaCHS will automatically adapt the concrete parameter style is adapted to the renderer – in the web interface, there are vizier-like expressions, in protocol interfaces, you get fields understanding expressions, either as in SSAP (for the *pql* parameter style) or as defined in DALI (the *dali* parameter style).

This will generate query fields that work against data as stored in the database, with some exceptions (columns containing MJDs will, for example, be turned into VizieR-like date expressions for web forms).

Since in HTML forms, astronomers often ask for odd units and then want to input them, too, DaCHS will also honor the displayUnit display hint for forms. for instance, if you wrote:

```
<table id="ex1">
  <column name="minDist"
    unit="deg"
    displayHint="displayUnit=arcsec"/>
  ...

<dbCore queriedTable="ex1">
  <condDesc buildFrom="minDist"/>
  ...
```

then the form renderer would declare the minDist column to take its values in arcsecs and do the necessary conversions, while minDist would properly work with degrees in SCS or TAP.

For object lists and similar, it is frequently desirable to give the possible values (unless there are too many of those; these will be translated to option lists in forms and to metadata items for protocol services and hence be user visible). In this case, you need to change the input key itself. You can do this by deriving the input key from the column and assign it to a condDesc, like this:

```
<condDesc>
   <inputKey original="source">
     <values fromdb="source from plc.data"/>
   </inputKey>
 </condDesc>
```

Use the `showItems="n"` attribute of input keys to determine how many items in the selector are shown at one time.

If you want your service to fail if a parameter is not given, declare the condDesc as required:

```
<condDesc buildFrom="myColumn" required="True"/>
```

(you can also declare individual an `inputKey` as `required`).

If, on the other hand, you want DaCHS to fill in a default if the user provides no value, give a default to the input key using the values child:

```
<condDesc>
  <inputKey original="radius">
    <values default="0.5"/>
  </inputKey>
<condDesc>
```

Sometimes a parameter shouldn't be defaulted in a protocol request (perhaps to satisfy an external contract), while the web interface should pre-fill a sensible choice. In that case, use the `defaultForForm` property:

```
<condDesc>
  <inputKey original="radius">
    <property key="defaultForForm">0.5</property>
  </inputKey>
<condDesc>
```

DaCHS will also interpret `min` and `max` attributes on the input keys (and the columns they are generated from) to generate input hints; that's a good way to fight the horror vacui users have when there's an input box and they have no idea what to put there. The best way to deal with this, however, is to not change the input keys but the columns themselves, as in:

```
<table id="ex1">
  <column name="mjd" type="double precision"
    ...>
    <values min="" max=""/>
...
<dbCore queriesTable="ex1">
  <condDesc buildFrom="mjd"/>
```

You will typically leave min and max empty and run:

```
dachs limits q#ex
```

when the table contents change; this will make DaCHS update the values in the RD itself.

Phrasemakers: Making custom queries

CondDescs will generate SQL adapted to the type of their input keys, which; as you can imagine, for cases like the VizieR expressions, that's not done in a couple of lines. However, there are times when you need custom behaviour. You can then give your conddescs a `phraseMaker`, a piece of python code generating a query and adding parameters:

```
<condDesc>
   <inputKey original="confirmed" multiplicity="single">
     <property name="adaptToRenderer">False</property>
   </inputKey>
   <phraseMaker>
     <code>
       if inPars.get(inputKeys[0].name, False):
         yield "confirmed IS NOT NULL"
     </code>
   </phraseMaker>
 </condDesc>
```

PhraseMakers work like other code embedded in RDs (and thus may have setup). `inPars` gives a dictionary of the input parameters as parsed by the inputDD according to multiplicity. `inputKeys` contains a sequence of the condDesc's inputKeys. By using their names as above, your code will not break if the parameters are renamed.

It is usually a good idea to set the property `adaptToRenderer` to False in such cases – you generally don't want DaCHS to use its standard rules for input key adaptation as discussion above because that will typically change what ends up in `inPars` and hence break your code for some renderers.

Note again that parameters not given will have the value None throughout. The will be present in inPars, though, so do *not* try things like `"myName" in inPars` – that's always true.

Phrase makers must yield zero or more SQL fragments; multiple SQL fragments are joined in conjunctions (i.e., end up in ANDed conditions in the WHERE clause). If you need to OR your fragments, you'll have to do that yourself. Use the `base.joinOperatorExpr(operator, operands)` for robustness to construct ORs.

Since you are dealing with raw SQL here, **never** include material from `inPars` directly in the query strings you return – this would immediately let people do SQL injections at least when the input key's type is text or similar. Instead, use the `getSQLKey` function as in this example:

```
<condDesc>
  <inputKey original="hdwl" multiplicity="single"/>
  <phraseMaker>
    <code>
      ik = inputKeys[0]
      destRE = "^%s\\.[0-9]*$"%inPars[ik.name]
      yield "%s ~ (%%(%s)s)"%(ik.name,
        base.getSQLKey("destRE", destRE, outPars))
    </code>
  </phraseMaker>
</condDesc>
```

`getSQLKey` takes a suggested name, a value and a dictionary, which within phrase makers always is `outPars`. It will enter value with the suggested name as key into `outPars` or change the suggested name if there is a name clash. The generated name will be returned, and that is what is entered in the SQL statement.

The `outPars` dictionary is shared between all conddescs entering into a query. Hence, if you do anything with it except passing it to `base.getSQLKey`, you're voiding your entire warranty.

Here's how to define a condDesc doing a full text search in a column:

```
<condDesc>
  <inputKey original="source" description="Words from the catalog
    description, e.g., author names or title words.">
    <property name="adaptToRenderer">False</property>
  </inputKey>
  <phraseMaker>
    <code>
      yield ("to_tsvector('english', source)"
        " @@ plainto_tsquery('english', %%(%s)s)")%(
      base.getSQLKey("source", inPars["source"], outPars))
    </code>
  </phraseMaker>
</condDesc>
```

Incidentally, this would go with an index definition like:

```
<index columns="source" method="gin"
  >to_tsvector('english', source)</index>
```

Grouping Input Keys

For special effects, you can group inputKeys. This will make them show up under a common label and in a single line in HTML forms. Other renderers currently don't do anything with the groups.

Here's an example for a simple range selector:

```
<condDesc>
  <inputKey name="el" type="text" tablehead="Element"/>

  <inputKey name="mfmin" tablehead="Min. Mass Fraction \item">
    <property name="cssClass">a_min</property>
  </inputKey>

  <inputKey name="mfmax" tablehead="Max. Mass Fraction \item">
    <property name="cssClass">a_max</property>
  </inputKey>
```

```
  <group name="mf">
    <description>Mass fraction of an element. You may leave out
      either upper or lower bound.</description>
    <property name="label">Mass Fraction between...</property>
    <property name="style">compact</property>
  </group>
</condDesc>
```

You will probably want to style the result of this effort using the `service` element's `customCSS` property, maybe like this:

```
<service...>
  <property name="customCSS">
    input.a_min {width: 5em}
    input.a_max {width: 5em}
    input.formkey_min {width: 6em!important}
    input.formkey_max {width: 6em!important}
    span.a_min:before { content:" between "; }
    span.a_max:before { content:" and "; }
    tr.mflegend td {
      padding-top: 0.5ex;
      padding-bottom: 0.5ex;
      border-bottom: 1px solid black;
    }
  </property>
</service>
```

See also the entries on multi-line input, selecting input fields with a widget, and customizing generated SCS conditions in DaCHS' howto document.

Output tables

When determining what columns to include in a response from a table-based core, DaCHS follows relatively complicated rules because displays in the browser and almost anywhere else are subject to somewhat different constraints. In the following, when wie talk about "VOTable", we refer to all tabular formats produced by DaCHS (FITS binary, CSV, TSV...).

The column selection is influenced by:

- Verbosity. This is controlled by the `VERB` parameter (1..3) or preferentially `verbosity` (1..30). Only columns with `verbLevel` not exceeding verbosity (or, if not given, VERB*10) are included in the result set. This, in particular, means that columns with `verbLevel` larger than 30 are never automatically included in output tables (but they can be manually selected for HTML using `_ADDITEM`).

- Output Format. While VOTable takes the core's output table and apply the verbosity filter, HTML uses the service's output table as the basis from which to filter columns. On the other hand, in HTML output the core output table is used to create the list of potential additional columns.

- `votableRespectsOutputTable`. This is a property on services that makes DaCHS use the service's output table even when generating VOTable output if it is set to `True`. Write:

  ```
  <property name="votableRespectsOutputTable">True</property>
  ```

  in your service element to enable this behaviour.

- `_ADDITEM`. This parameter (used by DaCHS' web interface) lets users select columns not selected by the current settings or the service's output table. `_ADDITEM` is ignored in VOTable unless in HTML mode (which is used in transferring web results via SAMP).

- `noxml`. Columns can be furnished with a `displayHint="noxml=true"`, and they will never be included in VOTable output; use this when you use complex formatters to produce HTML displays.

- `_SET`. DaCHS supports "column sets", for instance, to let users select certain kinds of coordinates. See apfs/res/apfs_new.rd‘ for an example. Essentially, when defining an output table, each output field gets a `sets` attribute (default: no set; use ALL to have the column included in all outputs). Then, add a `_SET` service parameter (use `values` do declare the avaiable sets). Note that the `_SET` parameter changes VOTable column selection to `votableRespectsOutputTable` mode as discussed above. Services that use column sets should therefore set the property manually for consistency whether or not clients actually pass `_SET`.

Sorry for this mess; all this had, and by and large still has, good reasons.

Writing Custom Cores

While DaCHS provides cores for many common operations – in particular, database queries and wrapped external binaries –, there are of course services that need to do things not covered by what the shipped cores do. A common case is wrapping external binaries.

Many such cases still follow the basic premise of services: GET or POST parameters in, something table-like out. You should then use custom cores, which then still let you use normal DaCHS renderers (in particular `form` and `api/sync`). When that doesn't cut it, you'll need to use a custom renderer.

While a custom core is defined in a separate module – this also helps debugging since you can run it outside of DaCHS –, there's also the python core that keeps the custom code inside of the RD. This is very similar; Python Cores instead of Custom Cores explains the differences.

The following exposition is derived from the times service in the GAVO data center, a service wrapping some FORTRAN code wrapping SOFA (yes, we're aware that we would directly use SOFA through astropy; that's not the point here). Check out the sources at http://svn.ari.uni-heidelberg.de/svn/gavo/hdinputs/apfs; the RD is times.rd.

Defining a Custom Core

In an RD, a custom core is very typically just written with a reference to a defining module:

```
<customCore module="res/timescore"/>
```

The path is relative to the resdir, and you don't include the module's extension (DaCHS uses normal python module resolution, except for temporarily extending the search path with the enclosing directory). You can, in principle, declare the core's interface in that element, but that's typically not a good idea (see below).

The above declaration means you will find the core itself in `res/timescore.py`.

Ideally, you'll just use the DaCHS API in the core, since we try fairly hard to keep that api constant. The timescore doesn't quite follow that rule because it wants to expand VizieR expressions, which normal services probably won't do.

DaCHS expects the custom core under the name `Core`. Thus, the centerpiece of the module is:

```
from gavo import api

class Core(api.Core):
```

The core needs an InputTable and an OutputTable like all cores. You *could* define it in the resource descriptor like this:

```
<customCore id="createCore" module="bin/create">
  <inputTable>
    <inputKey .../>
  </inputTable>
  <outputTable>
    <column name="itemsAdded" type="integer" tablehead="Items added"/>
  </outputTable>
</customCore>
```

It's preferable to define at least the input in the code, though, since it's more likely to be kept in sync with the code in that case. Embedding the definitions is done using the class attribute `inputTableXML`:

```
class Core(core.Core):
  inputTableXML = """
    <inputTable>
      <inputKey name="ut1" type="vexpr-date" multiplicity="single"
        tablehead="UT1"
        description="Date and time (UT1)" ucd="time.epoch;meta.main"/>
      <inputKey name="interval" type="integer" multiplicity="single"
        tablehead="Interval"
        unit="s" ucd="time.interval"
        description="Interval between two sets of computed values"
      >3600</inputKey>
    </inputTable>
  """
```

There is also `outputTableXML`, which you should use if you were to compute stuff in some lines of Python, since then the fields are directly defined by the core core itself.

However, the case of timescore is fairly typical: There is some, essentially external, resource that produces something that needs to be parsed. In that case, it's a better idea to define the parsing logic in a normal RD `data` item. Its table then is the output table of the core. In the times example, the output of `timescompute` is described by the `build_result` data item in `times.rd`:

```
<table id="times">
  <column name="ut1" type="timestamp" tablehead="UT1"
    ucd="time.epoch;meta.main" verbLevel="1"
    description="Time and date (UT1)" displayHint="type=humanDate"/>
  <column name="gmst" type="time" tablehead="GMST"
    verbLevel="1" description="Greenwich mean sidereal time"
    xtype="adql:TIMESTAMP" displayHint="type=humanTime,sf=4"/>
  <column name="gast" type="time" tablehead="GAST"
    verbLevel="1" description="Greenwich apparent sidereal time"
    xtype="adql:TIMESTAMP" displayHint="type=humanTime,sf=4"/>
  <column name="era" type="double precision" tablehead="ERA"
    verbLevel="1" description="Earth rotation angle"
    displayHint="type=dms,sf=3" unit="deg"/>
</table>

<data id="build_result" auto="False">
  <reGrammar>
    <names>ut1,gmst,gast,era</names>
  </reGrammar>
  <make table="times">
    <rowmaker>
      <map dest="gmst">parseWithNull(@gmst, parseTime, "None")</map>
```

```
      ...
    </rowmaker>
  </make>
</data>
```

So, the core needs to say "my output table has the structure of #times". As usual with DaCHS structures, you should not override the constructor, as it is defined by a metaclass. Instead, Cores call, immediately after the XML parse (technically, as the first thing of their `completeElement` method), a method called `initialize`. This is where you should set the output table. For the times core, this looks like this:

```
def initialize(self):
  self.outputTable = api.OutputTableDef.fromTableDef(
    self.rd.getById("times"), None)
```

Of course, you are not limited to setting the output table there; as `initialize` is only called once while parsing, this is also a good place to perform expensive, one-time operations like reading and parsing larger external resources.

Giving the Core Functionality

To have the core do something, you have to override the run method, which has to have the following signature:

```
run(service, inputTable, queryMeta) -> stuff
```

The stuff returned will ususally be a Table or Data instance (that need not match the outputTable definition -- the latter is targetted at the registry and possibly applications like output field selection). The standard renderers also accept a pair of mime type and a string containing some data and will deliver this as-is. With custom renderers, you could return basically anything you want.

Services come up with some idea of the schema of the table they want to return and adapt tables coming out of the core to this. Sometimes, you want to suppress this behaviour, e.g., because the service's ideas are off. In that case, set a noPostprocess atttribute on the table to any value (the TAP core does this, for instance).

In `service` you get the service using the core; this may make a difference since different services can use the same core and could control details of its operations through properties, their output table, or anything else.

The `inputTable` argument is the `CoreArgs` instance discussed in Core Args. Essentially, you'll usually use its `args` attribute, a dictionary mapping the keys defined by your input table to values or lists of them.

The `queryMeta` argument is discussed in Database Options.

In the times example, the parameter interpretation is done in an extra function (which helps testability when there's a bit more complex things going on):

```
def computeDates(args):
  """yields datetimes at which to compute times from the ut1/interval
  inputs in coreArgs args.
  """
  interval = args["interval"] or 3600
  if args["ut1"] is None:
    yield datetime.datetime.utcnow()
    return

  try:
    expr = vizierexprs.parseDateExpr(args["ut1"])
    if expr.operator in set([',', '=']):
      for c in expr.children:
        yield c

    elif expr.operator=='..':
      for c in expandDates(expr.children[0],
          expr.children[1], interval):
        yield c

    elif expr.operator=="+/-":
      d0, wiggle = expr.children[0], datetime.timedelta(
        expr.expr.children[1])
      for c in expandDates(d0-wiggle, d0+wiggle):
        yield c

    else:
      raise api.ValidationError("This sort of date expression"
        " does not make sense for this service", colName="ut1")
  except base.ParseException, msg:
    raise api.ValidationError(
      "Invalid date expression (at %s)."%msg.loc,
      colName="ut1")
```

While the details of the parameter parsing and expansion don't really matter, note now exceptions are mapped to a ValidationError and give a `colName` – this lets the form renderer display error messages next to the inputs that caused the failure.

The next thing timescore does is build some input, which in this case is fairly trivial:

```
input = "\n".join(utils.formatISODT(date) for date in dates)+"\n"
```

If your input is more complex or you need input files or similar, you want to be a bit more careful. In particular, do *not* change directory (or, equivalently, use the utils.sandbox context manager); this may confuse the server, and in particular will break the first time two requests are served simultaneously: The core runs within the main process, and that can only have one current directory.

Instead, in such situations, make a temporary directory and manually place your inputs in there. The spacecore (http://svn.ari.uni-heidelberg.de/svn/gavo/hdinputs/sp_ace/res/spacecore.py) shows how this could look like, including tearing the stuff down safely when done (the runSpace function).

For the timescore, that is not necessary; you just run the wrapped program using standard subprocess functionality:

```
computer = service.rd.getAbsPath("bin/timescompute")

pipe = subprocess.Popen([computer],
    stdin=subprocess.PIPE, stdout=subprocess.PIPE, close_fds=True,
    cwd=os.path.dirname(computer))
data, errmsg = pipe.communicate(input)
if pipe.returncode:
  raise api.ValidationError("The backend computing program failed"
    " (exit code %s).  Messages may be available as"
    " hints."%pipe.returncode,
    "ut1",
    hint=errmsg)
```

Note that with today's computers, you shouldn't need to worry about streaming input or output until they are in the dozens of megabytes (in which case you should probably think hard about a custom UWS and keep the files in the job's working directories).

To turn the program's output into a table, you use the data item defined in the RD:

```
return api.makeData(
  self.rd.getById("build_result"),
  forceSource=StringIO(data))
```

When the core defines the data itself, you would skip `makeData`. Just directly produce the rowdicts and make the output table directly from the rows:

```
rows = [{"foo": 3*i, "bar": 8*i} for i in range(30)]
return rsc.TableForDef(self.outputTable, rows=rows)
```

Database Options

The standard DB cores receive a "table widget" on form generation, including sort and limit options. To make the Form renderer output this for your core as well, define a method `wantsTableWidget()` and return `True` from it.

The `queryMeta` that your `run` method receives has a dbLimit key. It contains the user selection or, as a fallback, the global db/defaultLimit value. These values are integers.

So, if you order a table widget, you should do something like:

```
cursor.execute("SELECT .... LIMIT %(queryLimit)s",
  {"queryLimit": queryMeta["dbLimit"],...})
```

In general, you should warn people if the query limit was reached; a simple way to do that is:

```
if len(res)==queryLimit:
  res.addMeta("_warning", "The query limit was reached.  Increase it"
    " to retrieve more matches.  Note that unsorted truncated queries"
    " are not reproducible (i.e., might return a different result set"
    " at a later time).")
```

where res would be your result table. _warning metadata is displayed in both HTML and VOTable output, though of course VOTable tools will not usually display it.

Python Cores instead of Custom Cores

If you only have a couple of lines of python, you don't have to have a separate module. Instead, use a python core. In it, you essentially have the `run` method as discussed in Giving the Core Functionality in a standard `procApp`. The advantage is that interface and implementation is nicely bundled together. The following example should illustrate the use of such python cores; note that `rsc` already is in the procApp's namespace:

```
<pythonCore>
  <inputTable>
    <inputKey name="opre" description="Operand, real part"
      required="True"/>
    <inputKey name="opim" description="Operand, imaginary part"
      required="True"/>
    <inputKey name="powers" description="Powers to compute"
      type="integer" multiplicity="multiple"/>
  </inputTable>
```

```
  <outputTable>
    <outputField name="re" description="Result, real part"/>
    <outputField name="im" description="Result, imaginary part"/>
    <outputField name="log"
      description="real part of logarithm of result"/>
  </outputTable>

  <coreProc>
    <setup>
      <code>
        import cmath
      </code>
    </setup>
    <code>
      powers = inputTable.args["powers"]
      if not powers:
        powers = [1,2]
      op = complex(inputTable.args["opre"],
        inputTable.args["opim"])

      rows = []
      for p in powers:
        val = op**p
        rows.append({
          "re": val.real,
          "im": val.imag,
          "log": cmath.log(val).real})

      return rsc.TableForDef(self.outputTable, rows=rows)
    </code>
  </coreProc>
</pythonCore>
```

As an additional service, DaCHS executes your python cores in a sandbox directory, so you can create temporary files to your heart's delight; they will be torn down once the core is finished.

Regression Testing

Introduction

Things break – perhaps because someone foolishly dropped a database table, because something happened in your upstream, because you changed something or even because we changed the API (if that's not mentioned in Changes, we owe you a beverage of your choice). Given that, having regression tests that you can easily run will really help your peace of mind.

Therefore, DaCHS contains a framework for embedding regression tests in resource descriptors. Before we tell you how these work, some words of advice, as writing useful regression tests is an art as much as engineering.

Don't overdo it. There's little point in checking all kinds of functionality that only uses DaCHS code – we're running our tests before committing into the repository, and of course before making a release. If the services just use condDescs with buildFrom and one of the standard renderers, there's little point in testing beyond a request that tells you the database table is still there and contains something resembling the data that should be there.

Don't be over-confident. Just because it seems trivial doesn't mean it cannot fail. Whatever code there is in the service processing of your RD, be it phrase makers, output field formatters, custom render or data functions, not to mention custom renderers and cores, deserves regression testing.

Be specific. In choosing the queries you test against, try to find something that won't change when data is added to your service, when you add input keys or when doing similar maintenance-like this. Change will happen, and it's annoying to have to fix the regression test every time the output might legitimately change. This helps with the next point.

Be pedantic. Do not accept failing regression tests, even if you think you know why they're failing. The real trick with useful testing is to keep "normal" output minimal. If you have to "manually" ignore diagnostics, you're doing it wrong. Also, sometimes tests may fail "just once". That's usually a sign of a race condition, and you should *really* try to figure out what's going on.

Make it fail first. It's surprisingly easy to write no-op tests that run but won't fail when the assertion you think you're making is no longer true. So, when developing a test, assert something wrong first, make sure there's some diagnostics, and only then assert what you really expect.

Be terse. While in unit tests it's good to test for maximally specific properties so failing unit tests lead you on the right track as fast as possible, in regression tests there's nothing wrong with plastering a number of assertions into one test. Regression tests actually make requests to a web server, and these are comparatively expensive. The important thing here is that regression testing is fast enough to let you run them every time you make a change.

Writing Regression Tests

DaCHS' regression testing framework is organized a bit along the lines of python's unittest and its predecessors, with some differences due to the different scope.

So, tests are grouped into suites, where each suite is contained in a regSuite element. These have a (currently unused) title and a boolean attribute `sequential` intended for when the tests contained must be executed in the sequence specified and not in parallel. It defaults to false, which means the requests are made

in random order and in parallel, which speeds up the test runs and, in particular, will help uncover race conditions.

On the other hand, if you're testing some sort of interaction across requests (e.g., make an upload, see if it's there, remove it again), this wouldn't work, and you must set *sequential="True"*. Keep these sequential suites as short as possible. In tests within such suites (and only there), you can pass information from one test to the following one by adding attributes to `self.followUp` (which are available as attributes of self in the next test). If you need to manipulate the next URL, it's at `self.followUp.url.content_`. For the common case of a redirect to the url in the location header (or a child thereof), there's the `pointNextToLocation(child="")` method of regression tests. In the tests that are manipulated like this, the URL given in the RD should conventionally be `overridden in the previous test`. Of course, additional parameters, httpMethods, etc, are still applied in the manipulated url element.

Regression suites contain tests, represented in regTest elements. These are procDefs (just like, e.g., rowmakery `apply`), so you can have setup code, and you could have a library of parametrizable regTests procDefs that you'd then turn into regTests by setting their parameters. We've not found that terribly useful so far, though.

You must given them a `title`, which is used when reporting problems with them. Otherwise, the crucial children of these are `url` and, as always with procDefs, `code`.

Here are some hints on development:

1) Give the test you're just developing an id; at the GAVO DC, we're usually using cur; that way, we run variations of `gavo test rdId#cur`, and only the test in question is run.

2) After defining the url, just put an `assert False` into the test code. Then run `gavo test -Devidence.xml rdId#cur` or similar. Then investigate `evidence.xml` (possibly after piping through `xmlstarlet fo`) for stable and strong indicators that things are working.

3) If you get a BadCode for a test you're just writing, the message may not always be terribly helpful. To see what's actually bugging python, run `gavo --debug test ...` and check dcInfos.

RegTest URLs

The url element encapsulates all aspects of building the request. In the simplest case, you just can have a simple URL, in which case it works as an attribute, like this:

```
<regTest title="example" url="svc/form">
  ...
```

URLs without a scheme and a leading slash are interpreted relative to the RD's root URL, so you'd usually just give the service id and the renderer to be applied. You can also specify root-relative and fully specified URLs as described in the documentation of the url element.

White space in URLs is removed, which lets you break long URLs as convenient.

You could have GET parameters in this URL, but that's inconvient due to both XML and HTTP escaping. So, if you want to pass parameters, just give them as attributes to the element:

```
<regTest title="example">
  <url RA="10" DEC="-42.3" SR="1" parSet="form">svc/form</url>
```

The `parSet=form` here sets up things such that processing for the form renderer is performed – our form library nevow formal has some hidden parameters that you don't want to repeat in every URL.

To easily translate URLs taken from a browser's address bar or the form renderer's result link, you can run `gavo totesturl` and paste the URLs there. Note that totesturl fails for values with embedded quotes, takes only the first value of repeated parameters and is a over-quick hack all around. Patches are gratefully accepted.

The `url` element hence accepts arbitary attributes, which can be a trap if you think you've given values to url's private attributes and mistyped their names. If uploads or authentication don't seem to happen, check if your attribute ended up the in the URL (which is displayed with the failure message) and fix the attribute name; most private url attributes start with `http`. If you really need to pass a parameter named like one of url's private attributes, pass it in the URL if you can. If you can't because you're posting, spank us. After that, we'll work out something not too abominable .

If you have services requiring authentication, use url's `httpAuthKey` attribute. We've introduced this to avoid having credentials in the RD, which, after all, should reside in a version control system which may be (and in the case of GAVO's data center is) public. The attribute's value is a key into the file `~/.gavo/test.creds`, which contains, line by line, this key, a username and a password, e.g.:

```
svc1 testuser notASecret
svc2 regtest NotASecretEither
```

A test using this would look like this:

```
<regTest title="Authenticated user can see the light">
  <url httpAuthKey="svc1">svc1/qp/light.txt</url>
  <code>
    self.assertHTTPStatus(200)
  </code>
</regTest>
```

By default, a test will perform a GET request. To change this, set the `httpMethod` attribute. That's particularly important with uploads (which must be POSTed).

For uploads, the url element offers two facilities. You can set a request payload from a file using the `postPayload` attribute (the path is interpreted relative to the resource directory), but it's much more common to do a file upload like browsers do them. Use the `httpUpload` element for this, as in:

```
<url> <httpUpload name="UPLOAD"
  fileName="remote.txt">a,b,c</httpUpload> svc1/async </url>
```

(which will work as if the user had selected a file remote.txt containing "a,b,c" in a browser with a file element named UPLOAD), or as in:

```
<url>
  <httpUpload name="UPLOAD" fileName="remote.vot"
    source="res/sample.regtest"/>
  svc1/async
</url>
```

(which will upload the file referenced in `source`, giving the remote server the filename `remote.vot`). The `fileName` attribute is optional.

Finally, you can pass arbitrary HTTP headers using the `httpHeader` element. This has an attribute `key`; the header's value is taken from the element content, like this:

```
<url postPayload="res/testData.regtest" httpMethod="POST">
  <httpHeader key="content-type">image/jpeg</httpHeader>
  >upload/custom</url>
```

RegTest Tests

Since regression tests are just procDefs, the actual assertions are contained in the `code` child of the `regTest`. The code in there sees the test itself in self, and it can access `self.data` (the response content), `self.headers` (a sequence of header name, value pairs; note that you should match the names case-insensitively here), and `self.status` (the HTTP response code), as well as the URL actually retrieved in `self.url.httpURL` (incidentally, that name is right; the regression framework only supports http, and it's not terribly likely that we'll change that).

You should probably only access those attributes in a pinch and instead use the pre-defined assertions, which are methods on the test objects as in pyunit – conventional assertions are clearer to read and less likely to break if fixes to the regression test API become necessary. If you still want to have custom tests, raise AssertionErrors to indicate a failure.

Here's a list of assertion methods defined right now:

assertHTTPStatus(self, expectedStatus) checks whether the request came back with expectedStatus.

assertHasStrings(self, *strings) checks that all its arguments are found within content.

assertHeader(self, key, value) checks that header key has value in the response headers.

keys are compared case-insensitively, values are compared literally.

assertLacksStrings(self, *strings) checks that all its arguments are *not* found within content.

assertValidatesXSD(self) checks whether the returned data are XSD valid.

This uses DaCHS built-in XSD validator with the built-in schema files; it hence will in general not retrieve schema files from external sources.

assertXpath(self, path, assertions) checks an xpath assertion.

path is an xpath (as understood by lxml), with namespace prefixes statically mapped; there's currently v2 (VOTable 1.2), v1 (VOTable 1.1), v (whatever VOTable version is the current DaCHS default), h (the namespace of the XHTML elements DaCHS generates), and o (OAI-PMH 2.0). If you need more prefixes, hack the source and feed back your changes (monkeypatching self.XPATH_NAMESPACE_MAP is another option).

path must match exactly one element.

assertions is a dictionary mapping attribute names to their expected value. Use the key None to check the element content, and match for None if you expect an empty element.

If you need an RE match rather than equality, there's EqualingRE in your code's namespace.

getFirstVOTableRow(self) interprets data as a VOTable and returns the first row as a dictionary

In test use, make sure the VOTable returned is sorted, or you will get randomly failing tests. Ideally, you'll constrain the results to just one match; database-querying cores (which is where order is an issue) also honor _DBOPTIONS_ORDER).

getVOTableRows(self) parses the first table in a result VOTable and returns the contents as a sequence of dictionaries.

getXpath(self, path, element=None) returns the equivalent of tree.xpath(path) for an lxml etree of the current document or in element, if passed in.

This uses the same namespace conventions as assertXpath.

All of these are methods, so you would actually write `self.assertHasStrings('a', 'b', 'c')` in your test code (rather than pass self explicitly).

When writing tests, you can, in addition, use assertions from python's unittest TestCases (e.g., assertEqual and friends). This is provided in particular for use to check values in VOTables coming back from services together with the `getFirstVOTableRow` method.

Also please note that, like all procDef's bodies, the test code is macro-expanded by DaCHS. This means that every backslash that should be seen by python needs to be escaped itself (i.e., doubled). An escaped backslash in python thus is four backslashes in the RD.

Finally, here's a piece of `.vimrc` that inserts a `regTest` skeleton if you type ge in command mode (preferably at the start of a line; you may need to fix the indentation if you're not indenting with tabs. We've thrown in a column skeleton on gn as well:

```
augroup rd
  au!
  autocmd BufRead,BufNewFile *.rd set ts=2 tw=79
  au BufNewFile,BufRead *.rd map gn i<tab><tab><lt>column name="" type=""<CR><tab>unit="" ucd=""
  au BufNewFile,BufRead *.rd map ge i<tab><tab><lt>regTest title=""><CR><tab><lt>url><lt>/url><C
augroup END
```

Running Tests

The first mode to run the regression tests is through `gavo val`. If you give it a `-t` flag, it will collect regression tests from all the RDs it touches and run them. It will then output a brief report listing the RDs that had failed tests for closer inspection.

It is recommended to run something like:

```
gavo val -tv ALL
```

before committing changes into your inputs repository. That way, regressions should be caught.

The tests are ran against the server described through the `[web]serverURL` config item. In the recommended setup, this would be a server started on your own development machine, which then would actually test the changes you made.

There is also a dedicated gavo sub-command `test` for executing the tests. This is what you should be using for developing tests or investigating failures flagged with `gavo val`. On its command line, you can give on of an RD id or a cross-rd reference to a test suite, or a cross-rd reference to an individual test. For example,

```
gavo test res1/q
gavo test res2/q#suite1
gavo test res2/q#test45
```

would run all the tests given in the RD `res1/q`, the tests in the regSuite with the `id` suite1 in `res2/q`, and a test with `id="test45` in `res2/q`, respectively.

To traverse inputs and run tests from all RDs found there, as well as tests from the built-in RDs, run:

```
gavo test ALL
```

`gavo test` by default has a very terse output. To see which tests are failing and what they gave as reasons, run it with the '-v' option.

To debug failing regression tests (or maybe to come up with good things to test for), use '-d', which dumps the server response of failing tests to stdout.

In the recommended setup with a production server and a development machine sharing a checkout of the same inputs, you can exercise production server from the development machine by giving the `-u` option with what your production server has in its `[web]serverURL` configuration item. So,

```
gavo test -u http://production.example.com ALL
```

is what might help your night's sleep.

Examples

Here are some examples how these constructs can be used. First, a simple test for string presence (which is often preferred even when checking XML, as it's less likely to break on schema changes; these usually count as noise in regression testing). Also note how we have escaped embedded XML fragments; an alternative to this shown below is making the code a CDATA section:

```
<regTest title="Info page looks ok"
  url="siap/info">
  <code>
    self.assertHasStrings("SIAP Query", "siap.xml", "form",
      "Other services", "SIZE&lt;/td>", "Verb. Level")
  </code>
</regTest>
```

The next is a test with a "rooted" URL that's spanning lines, has embedded parameters (not recommended), plus an assertion on binary data:

```
<regTest title="NV Maidanak product delivery"
  url="/getproduct/maidanak/data/Q2237p0305/Johnson_R/
    red_kk050001.fits.gz?siap=true">
  <code>
    self.assertHasStrings('\\x1f\\x8b\\x08\\x08')
  </code>
</regTest>
```

This is how parameters should be passed into the request:

```
<regTest title="NV Maidanak SIAP returns accref.">
  <url POS="340.12,3.3586" SIZE="0.1" INTERSECT="OVERLAPS"
    _TDENC="True" _DBOPTIONS_LIMIT="10">siap/siap.xml</url>
  <code>
    self.assertHasStrings('&lt;TD>AZT 22')
  </code>
</regTest>
```

Here's an example for a test with URL parameters and xpath assertions:

```
<regTest title="NV Maidanak SIAP metadata query"
    url="siap/siap.xml?FORMAT=METADATA">
  <code>
    self.assertXpath("//v1:FIELD[@name='wcs_cdmatrix']", {
      "datatype": "double",
      "ucd": "VOX:WCS_CDMatrix",
      "arraysize": "*",
      "unit": "deg/pix"})
    self.assertXpath("//v1:INFO[@name='QUERY_STATUS']", {
      "value": "OK",
      None: "OK",})
    self.assertXpath("//v1:PARAM[@name='INPUT:POS']", {
      "datatype": "char",
      "ucd": "pos.eq",
      "unit": "deg"})
  </code>
</regTest>
```

The following is a fairly complex example for a stateful suite doing inline uploads (and simple tests):

```
<regSuite title="GAVO roster publication cycle" sequential="True">
  <regTest title="Complete record yields some credible output">
    <url httpAuthKey="gvo" parSet="form" httpMethod="POST">
      <httpUpload name="inFile" fileName="testing_ignore.rd"
        ><![CDATA[
          <resource schema="gvo">
            <meta name="description">x</meta>
            <meta name="title">A test service</meta>
            <meta name="creationDate">2010-04-26T11:45:00</meta>
            <meta name="subject">Testing</meta>
            <meta name="referenceURL">http://foo.bar</meta>
            <nullCore id="null"/>
            <service id="run" core="null" allowed="external">
              <meta name="shortName">u</meta>
              <publish render="external" sets="gavo">
                <meta name="accessURL">http://foo/bar</meta>
              </publish></service></resource>
        ]]></httpUpload>upload/form</url>
    <code><![CDATA[
      self.assertHasStrings("#Published</th><td>1</td>")
    ]]></code>
  </regTest>

  <regTest title="Publication leaves traces on GAVO list" url="list/custom">
    <code>
      self.assertHasStrings(
        '"/gvo/data/testing_ignore/run/external">A test service')
    </code>
  </regTest>

  <regTest title="Unpublication yields some credible output">
```

```
    <url httpAuthKey="gvo" parSet="form" httpMethod="POST">
      <httpUpload name="inFile" fileName="testing_ignore.rd"
        ><![CDATA[
        <resource schema="gvo">
          <meta name="description">x</meta>
          <meta name="title">A test service</meta>
          <meta name="creationDate">2010-04-26T11:45:00</meta>
          <meta name="subject">Testing</meta>
          <meta name="referenceURL">http://foo.bar</meta>
          <service id="run" allowed="external">
            <nullCore/>
            <meta name="shortName">u</meta></service></resource>
        ]]></httpUpload>upload/form</url>
    <code><![CDATA[
      self.assertHasStrings("#Published</th><td>0</td>")
    ]]></code>
  </regTest>

  <regTest title="Unpublication leaves traces on GAVO list"
    url="list/custom">
    <code>
      self.assertLacksStrings(
        '"/gvo/data/testing_ignore/run/external">A test service')
    </code>
  </regTest>

</regSuite>
```

If you still run SOAP services, here's one way to test them:

```
<regTest id="soaptest" title="APFS SOAP returns something reasonable">
    <url postPayload="res/soapRequest.regtest" httpMethod="POST">
      <httpHeader key="SOAPAction">'"useService"'</httpHeader>
      <httpHeader key="content-type">text/xml</httpHeader
      >qall/soap/go</url>
    <code>
      self.assertHasStrings(
        '="xsd:date">2008-02-03Z&lt;/tns:isodate>',
      '&lt;tns:raCio xsi:type="xsd:double">25.35')
    </code>
  </regTest>
```

– here, `res/soapRequest.regtest` would contain the request body that you could, for example, extract from a tcpdump log.

Datalink and SODA

[Datalink] is an IVOA protocol that allows associating various products and artifacts with a data set id. Think the association of error or mask maps, progenitor datasets, or processed data products, with a data set.

It also lets you associate data processing services with datasets, which allows on-the-fly generation of cutouts, format conversions or recalibrations; a particular set of parameters for working with certain kinds of cubes is described in a standard called [SODA] (Serverside Operations for Data Access). Hence, we sometimes call the processing part of datalink SODA.

In DaCHS, Datalink is implemented by the `dlmeta` renderer, SODA by the `dlget` renderer. In all but fairly exotic cases, both renderers are used on the same service. While in DaCHS, you cannot use SODA without Datalink, there are perfectly sensible datalink services without SODA. In the following, we first treat the generation of "normal" datalinks and discuss processing services later.

A central term for datalink is the pubDID, or publisher DID. This is an identifier assigned (essentially) by you that points to a concrete dataset. In DaCHS, datalink services always use pubDIDs as the values of the datalink ID parameter.

Unless you arrange things differently (for which you should have good reasons), the pubDIDs used by DaCHS are formed as:

```
<authority>/~?<accref>
```

where the accref usually is the inputsDir-relative path to the file. If you use datalinks of that form, you should at some point run `gavo pub //products`; this will register the products deliverer as `<authority>/~`, which means that pubDIDs of this form are compliant with [IVOA Identifiers]_

When developing datalink services, it sometimes is useful to access datalink services directly, in particular because they don't usually have a useful web interface. Armed with the knowledge about the structure of DaCHS standard PubDIDs, you can easily build the URLs and parameters. For instance, to retrieve the datalink document for `mlqso/data/FBQ0951_data.fits` on the server dc.g-vo.org using the datalink renderer on the `mlqso/q/d` service, you'd write:

```
curl -FID=ivo://org.gavo.dc/~?mlqso/data/slits/FBQ0951_data.fits \
  http://dc.g-vo.org/mlqso/q/d/dlmeta | xmlstarlet fo
```

(of course, `xmlstarlet` isn't actually necessary, and you can use `wget` if you want, but you get the idea). Going on, you could pull out what parameters are mentioned somewhat like this:

```
curl -s -FID=ivo://org.gavo.dc/~?mlqso/data/slits/FBQ0951_data.fits \
  http://dc.g-vo.org/mlqso/q/d/dlmeta | \
  xmlstarlet sel -N v=http://www.ivoa.net/xml/VOTable/v1.3 -T \
  -t -m "//v:PARAM" -v "@name" -nl
```

In the remainder of this section, we first discuss the generation of datalinks and processing services "by example", which should do for a basic use of the facilities. We continue with a somewhat more in-depth look at the processing of a SODA request, after which we look more closely at the various elements that make up Datalink/SODA services.

Integrating Datalink Services

You generally declare datalink services on the table(s) that contain the identifiers the datalink service accepts. For that, you include two pieces of metadata: The identifier of the datalink service (which can be a cross-RD id with a hash; use the `_associatedDatalinkSvc.serviceId` meta key) and the column name within the table (use the `_associatedDatalinkSvc.idColumn` meta key). Both items will only be checked at run time, and broken links will be reported as warnings. If the following doesn't give you the datalink resources in results involving the tables, be sure to check the `dcInfos` log file.

The following example is a table that contains two sorts of identifiers that are understood by two different datalink services; one, `dlsvc` within the same RD, works on values in the `accref` column, the other, taken from a (hypothetical) `doires/q` RD, would work on the `doi` column:

```
<table id="datasets" onDisk="True">
  <meta name="_associatedDatalinkSvc">
    <meta name="serviceId">dlsvc</meta>
    <meta name="idColumn">accref</meta>
  </meta>
  <meta name="_associatedDatalinkSvc">
    <meta name="serviceId">doires/q#doidl</meta>
    <meta name="idColumn">doi</meta>
  </meta>

  <column name="accref" type="text".../>
  <column name="doi" type="text".../>
</table>

<service id="dlsvc" allowed="dlmeta,deget">
  <meta name="dlget.description">A service for
    slicing and dicing.</meta>
  ...
</service>
```

Note that forward references, which are generally not allowed in DaCHS, are possible in `serviceId` and `idColumn`.

An older way to associate datalink services with tables is to give certain services (most notably, SSA ones) a `datalink` property. This is deprecated now. If you see it in examples, please tell us so we can fix it.

Making Datalinks

A dataset frequently has associated data, like error or weight maps, derived data, or pieces of provenance. Datalink lets you tie these together algorithmically, using a specialised core (see element DatalinkCore) and the dlmeta renderer.

To produce datalinks, the datalink core must be furnished with

- exactly one descriptor generator (you can let DaCHS fall back to a default),
- one or more meta makers, generating related links.

Here is an example, adapted from boydende/q.rd:

```
<datalinkCore>
  <descriptorGenerator procDef="//soda#fits_genDesc"/>
  <metaMaker>
    <code>
      svc = rd.getById("dl")
      basename = descriptor.accref.split("/")[-1].split(".")[0]
      envPath = "data/static/envelopes/{0}.jpg".format(basename)

      yield descriptor.makeLinkFromFile(
        envPath,
        description="Scan of the plate envelope",
        semantics="#isMetadataFor")
    </code>
  </metaMaker>
</datalinkCore>
```

A descriptor generator – in the example, one that has additional functionality for FITS files, although the default (//soda#fromStandardPubDID) would work here, too – is passed the pubDID and returns an instance of `datalink.ProductDescriptor` (or a derived class). If a descriptor generator returns None, the datalink request will be rejected with a 404.

Whatever is returned by the descriptor generator is then available as `descriptor` to the remaining datalink procs (in this case, the meta makers). The columns of the product table (see dc.products) are available as attributes of this object. In addition, subclasses of `data.ProductDescriptor` may add more attributes; the `fits_genDesc` used in the example, for instance, provides a `hdr` attribute containing the primary header as given by pyfits.

The descriptor is then passed, in turn, to all meta makers given. These must `yield LinkDef` instances that describe additional data products; a single meta maker may yield zero or more of these. An example where multiple `LinkDef`

instances are yielded from a single `metaMaker` can be found in the `dl` service of cars/q.

When the links, as is quite common, correspond to simple files, the easiest way to generate them is through the descriptor's `makeLinkFromFile` method, that takes the source path, a description, and semantics (which should be taken from a controlled vocabulary at http://www.ivoa.net/rdf/datalink/core). File size and media type type, which otherwise should be given when constructing a `LinkDef`, then default to what's inferrable from the file (name).

`makeLinkFromFile` will create `NotFoundFault` error links if the file does not exist, thus alerting the user (and possibly you) that an expected file was not there. When missing files are expectable and should not cause diagnostics, pass a `suppressMissing=True` to `makeLinkFromFile`.

Another recommended pattern is used in the example: the datalink service itself is used to deliver the static, non-product files. This is effected by declaring the service embedding the core somewhat like this:

```
<service id="dl" allowed="dlget,dlmeta,static">
  <property name="staticData">data/static</property>
  <datalinkCore .../>
</service>
```

Note that, of course, exposing directory via the static renderer like this bypasses any access restrictions (e.g., embargos) on the respective data. So, do not expose you primary data in this way if you want to enforce access control.

A `LinkDef` for the product itself (semantics `#this`) and, if defined in the product table, a preview (semantics `#preview`) is automatically added by DaCHS unless a `suppressAutoLinks` attribute is set on the descriptor (you can set that in a meta maker or the descriptor generator).

For more information on the elements used here, see below.

Defining Processing Services

In DaCHS data processing services ("SODA services") use the same datalink cores as the datalink services, and they share the same descriptor. A datalink core does data processing when used by the dlget renderer.

To enable data processing, datalink cores additionally need data functions (see element dataFunction) and up to one data formatter (see element dataFormatter). The first data function must add a `data` attribute to the descriptor and thus plays a somewhat special role.

Processing services also use meta makers, but instead of links, these yield parameter definitions in the form of InputKeys (they are used by the datalink services, too, because the datalink documents contain the metadata of the processing services). So, typically, a given piece of SODA functionality comes as a pair of a meta maker and a data function, which then normally are combined in a STREAM (cf. Datalink-related Streams).

Processing services usually are a good deal more stereotypical than metadata generation; it is actually beneficial if different services have identical behaviour to facilitate the creation of interoperable clients. SODA itself essentially enumerates what in DaCHS are pre-defined meta makers and data functions. So, most of the time data processing will just re-use STREAMs and procDefs from the `//soda` RD.

The two most common cases are cutouts over FITS cubes and over spectra.

Processing services are referenced from the links table. In DaCHS, the description column for the services is empty by default, which you may want to change. Just set a `dlget.description` meta on the service. In the following example, there is a normal VOResource description that will end up in the Registry and the DaCHS web interface, and a description of the processing service:

```
<service id="dl" allowed="dlmeta,dlget">
  <meta name="description">A datalink service for the COOL data
    collection, giving provenance links, extracted sources, and
    cutouts</meta>
  <meta name="dlget.description">Use this interactive service
    to do cutouts, retrieved scaled images, and choose between
    FITS and JPEG results.</meta>
  ...
```

FITS/SODA processing

In the first case, the core would like this piece extracted from the `dl` service in califa/q3:

```
<datalinkCore>
  <descriptorGenerator procDef="//soda#fits_genDesc"
      name="genFITSDesc">
    <bind key="accrefPrefix">'califa/datadr3'</bind>
    <bind key="descClass">DLFITSProductDescriptor</bind>
  </descriptorGenerator>

  <FEED source="//soda#fits_standardDLFuncs" spectralAxis="3"/>
</datalinkCore>
```

Here, we use the //soda#fits_genDesc descriptor generator with a DLFITSProductDescriptor because CALIFA DR3 stores datalink URLs rather than actual file paths in the product table. You would leave the `descClass` parameter out when your products are the FITS files themselves.

Giving an `accrefPrefix` to anything using the product table to get accrefs (//soda#fromStandardPubDID is another example for these) usually is a good idea. If you don't give it, users can apply the datalink service to any dataset you publish, which might lead to information leaks and hard-to-understand error messages on the user side. `accrefPrefix` is simply a string that the accref of the product being processed must match. Since in the usual setup, the accref is the inputsDir-relative path of the file, you're usually fine if you just give the path to the directory containing the products in question.

The //soda#fits_standardDLFuncs STREAM arrange for all general FITS processing functions to be pulled in; these encompass the SODA parameters where applicable (at the time of this writing, there is no support for TIME and POL yet, but if you have such data, we'll be glad to add it), and some additional ones.

If you need extended functionality, it is a good idea to start from this STREAM. Copy it from `gavo adm dumpDF //soda` and hack from there.

SDM processing

The other very common sort of SODA-like processing is for spectra. A sketch for these from the `sdl` service in flashheros/q:

```
<datalinkCore>
  <descriptorGenerator procDef="//soda#sdm_genDesc">
    <bind name="ssaTD">"\rdId#data"</bind>
  </descriptorGenerator>
  <dataFunction procDef="//soda#sdm_genData">
    <bind name="builder">"\rdId#build_sdm_data"</bind>
  </dataFunction>
  <FEED source="//soda#sdm_plainfluxcalib"/>
  <FEED source="//soda#sdm_cutout"/>
  <FEED source="//soda#sdm_format"/>
</datalinkCore>
```

Here, the descriptor generator will in general be //soda#sdm_genDesc. It builds a special descriptor that contains the full metadata from an associated SSA row, which is why you need to give the id of the SSA table in the `ssaTD` parameter. Since pubDIDs will only be resolved within this table, no `accrefPrefix` is necessary or supported.

The first data function for spectra usually will be //soda#sdm_genData. This will read the entire spectrum into memory using a data item, the id of which is given in the `builder` parameter. This has to build an SDM-compliant spectrum. Some examples of how to do this can be found in cdfspect/q.rd (reading from half-broken FITS files), c8spect/q.rd (which shows how to create spectra that don't exist on disk as files), pcslg/q.rd (which nicely uses WCSAxis for parsing spectra that come as 1D-array, "IRAF-style"), or theossa/q.rd (which pulls the source files from a remote server and caches it). For more on generating SDM-compliant spectra, see SDM compliant tables.

For large spectra, reading the spectrum in its entirety may incur a significant CPU cost. When that becomes a problem for you, you'll need to write different data functions, perhaps only parsing a header, and implement, e.g., cutouts directly in a subsequent data function.

The two next STREAMs pulled in are just combinations of data functions and meta makers, one for optionally re-calibrating the spectrum (right now, only maximum normalisation is supported), the other for providing a SODA-like cutout.

Finally, //soda#sdm_format pulls in a meta maker defining a FORMAT parameter (letting people order several formats including VOTable, FITS binary table, and CSV) and a formatter that interprets it.

General Notes on Processing Services

This section contains an overview over how data processing services are built and executed. You should read it if you want to write data processing functions; for just using them, don't bother.

When a request for processed data comes in, the descriptor generator is used to make a product descriptor, and the input keys are adapted to the concrete dataset. This means that, contrary to normal DaCHS services, services with a Datalink core have a variable interface; in particular, the interface on the dlmeta renderer (essentially, just ID) is very different from the one on the dlget renderer (ID plus whatever the meta makers produce).

The input key so produced are used to build a context grammar that parses the request. If this succeeds, the data descriptor is passed to the initial data function together with the arguments parsed. This must set the `data` attribute of the descriptor or raise a ValidationError on the ID parameter; leaving `data` as `None` results in a 500 server error. Descriptor.data could an `rsc.InMemoryTable` (e.g., in SDM processing) or a `products.Products` instance, but as long as the other data functions and the formatter agree on what it is, anything goes.

The remaining data functions can change the data in place or potentially replace `descriptor.data`. When writing code, be aware, though, that a data function should only do something when the corresponding parameter has actually been used. When you change `descriptor.data` fundamentally, you'll probably make the lives of further data functions and the formatter a good deal harder.

Finally, the data enters the formatter, which actually generates the output, usually returning a pair of mime type and string to be delivered.

It is a design descision of the service creator which manipulations are done in the initial data function, which are in later filters, and which perhaps only in the formatter. The advantage of filters is that they are more flexible and can more easily be reused, while doing it things in the data generator itself will usually be more efficient, sometimes much so (e.g., sums being computed within a database rather than in a filter after all the data had to go through the interface of the database).

Descriptor Generators

Descriptor generators (see element descriptorGenerator) are procedure applications that, roughly, see a pubDID value and are expected to return a `datalink.ProductDescriptor` instance, or something derived from it.

Simple Product Descriptor Generator

In the end, this usually boils down to figuring out the value of accref in the product table and using what's there to construct the descriptor generator. In the simplest case, the pubDID will be in DaCHS' "standard" format (see the `getStandardPubDID` rowmaker function or the macro standardPubDID), in which case the default descriptor generator works and you don't have to specify anything. You could manually insert that default by saying:

```
<descriptorGenerator procDef="//soda#fromStandardPubDID"/>
```

This happens to be DaCHS' default if no descriptor generator is given, but as said above that is suboptimal as no accrefPrefix constrains what the service will run on.

The easiest way to furnish your descriptors with additional information is to grab that code (use `gavo adm dumpDF //soda`) and just add attributes to the `ProductDescriptor` generated in this way.

The default `ProductDescriptor` class exposes as attributes all the columns from the products table. See dc.products for their names and descriptions.

Spectrum Product Descriptor Generators

A slightly more interesting example is provided by datalink for SSA, where cutouts and similar is generated from spectra. The actual definition is in `//soda#sdm_genDesc`, but the gist of it is:

```
<procDef type="descriptorGenerator" id="sdm_genDesc">
  <setup>
    <par key="ssaTD" description="Full reference (like path/rdname#id)
      to the SSA table the spectrum's PubDID can be found in."/>
    <par key="descriptorClass" description="The SSA descriptor
      class to use.  You'll need to override this if the dc.products
      path doesn't actually lead to the file (see
      'custom generators &lt;#custom-product-descriptor-generators&gt;'_)."
      late="True">ssap.SSADescriptor</par>
    <code>
      from gavo import rscdef
      from gavo import rsc
      from gavo import svcs
      from gavo.protocols import ssap
      ssaTD = base.resolveCrossId(ssaTD, rscdef.TableDef)
    </code>
  </setup>

  <code>
    with base.getTableConn() as conn:
      ssaTable = rsc.TableForDef(ssaTD, connection=conn)
      matchingRows = list(ssaTable.iterQuery(ssaTable.tableDef,
        "ssa_pubdid=%(pubdid)s", {"pubdid": pubDID}))
      if not matchingRows:
        return DatalinkFault.NotFoundFault(pubDID,
          "No spectrum with this pubDID known here")

      # the relevant metadata for all rows with the same PubDID should
      # be identical, and hence we can blindly take the first result.
      return descriptorClass.fromSSARow(matchingRows[0],
        ssaTable.getParamDict())
  </code>
</procDef>
```

Here, we use `ssa.SSADescriptor`, derived from `ProductDescriptor`, rather than monkeypatching the extra `ssaRow` attribute the former provides; being explicit here may help when debugging. As usual, the descriptor generates encodes how to resolve a pubDID to an accref, in this case using an SSA table. If the product table just lists a datalink URL, you will want to override the accessPath this comes up with. See, for instance, pcslg/q for how to do this.

Incidentally, in this case you could stuff the entire code into the main `code` element, saving on the extra `setup` element. However, apart from a minor speed benefit, keeping things like function or class definitions in setup allows easier reuse of such definitions in procedure applications and is therefore recommended.

FITS Product Descriptor Generators

For FITS files, you will usually just use //soda#fits_genDesc, defining the `accrefStart` as discussed in FITS/SODA processing. This will produce *datalink.FITSProductDescriptor* instances. As in the SSA/SDM case, you may need different descriptor classes in special situations. Since for large FITS files, just delivering datalink files is a fairly compelling proposition, there is actually a predefined descriptor class to use with datalink access paths, *DLFITSProductDescriptor*; the `dl` service in califa/q3 shows how to use it.

Meta Makers

Meta makers (see element metaMaker) contain code that produces pieces of service metadata from a data descriptor. All meta makers belonging to a service are unconditionally executed, and all must be generator bodies (i.e., contain a yield statement).

Link Definitions

While meta makers see the `LinkDef` class itself, too, you should normally use the `makeLink` or `makeLinkFromFile` methods of the descriptor (they are available if the descriptor class was derived from `datalink.ProductDescriptor`, as it usually should).

These methods take a link or a path as the first argument, respectively. The rest are keyword arguments corresponding to the datalink columns, viz.,

description A human-readable short information on what's behind the link

semantics A term from a controlled-vocabulary describing what's behind the link (see below)

contentType An (advisory) media type of whatever this link points to. Please make sure it's consistent with what the server actually returns if the protocol used by accessURL supports that.

contentLength The (approximate) size of the resource at accessURL, in bytes (not for `makeLinkFromFile`, which takes it from the file system)

`makeLinkFromFile` additionally allows an argument service (see below).

With the exception of semantics, all auxillary data defaults to None if not given, and it's legal to leave it at that. Semantics must be non-NULL, even if an error message is generated. To make sure that's true, DaCHS inserts a non-informational URL, which preferentially shouldn't escape to the user. Hence,

please set semantics on LinkDefs, and if possible choose one of the terms given at http://www.ivoa.net/rdf/datalink/core

You can inspect the definition of the datalinks table active in your system by saying `gavo admin dumpDF //datalink | less` (the table definition is right at the top).

When returning link definitions, the tricky part mostly is to come up with the URLs. Use the `makeAbsoluteURL` rowmaker function to make them from relative URLs; the rest just depends on your URL scheme. An example could look like this:

```
<metaMaker>
  <code>
    yield descriptor.makeLink(
      makeAbsoluteURL("get/"+descriptor.accref[:-5]+".err.fits"),
      contentType="image/fits", semantics="#error",
      description="Errors for this dataset")
    yield descriptor.makeLink(
      "http://foo.bar/raw/"+descriptor.accref.split("/")[-1],
      contentType="image/fits", semantics="#progenitor",
      description="Un-flatfielded, uncalibrated source data")
  </code>
</metaMaker>
```

Parameter Definitions

To define a datalink service's processing capabilities, meta makers yield input keys (`InputKey` instances). The classes usually required to build input keys return (InputKey, Values, Option) are available to the code as local names. As usual, DaCHS structs should not be constructed directly but only using the `MS` helper (which is really an alias for base.makeStruct; it takes care that the special postprocessing of DaCHS structures takes place).

You should make sure that the input keys have proper annotation as regards minima, maxima, or enumerated values; clients, in general, have to way to guess what is sensible here.

The limits can usually be obtained from the descriptor (which, again, is available as `descriptor` in the meta maker. For instance, the FITS descriptor has a `header` attribute describing the instance that the core operates on, the SSA descriptor an attribute `ssaROW`.

A meta maker that generates an extra cutout parameter for radio astronomers (note that this is of course a bad idea -- unit adaption should be done on the client side) could be:

```
<metaMaker>
  <setup>
    <code>
      from gavo.utils import unitconv
    </code>
  </setup>
  <code>
    yield MS(InputKey, name="FREQ", unit="MHz", ucd="em.freq",
      description="Spectral cutout interval",
      type="double precision[2]" xtype="interval"
      multiplicity="forced-single"
      values=MS(Values,
        min=1e-6*unitconv.LIGHT_C/(descriptor.ssaRow["ssa_specstart"],
        max=1e-6*unitconv.LIGHT_C/descriptor.ssaRow["ssa_specend"]))
  </code>
</metaMaker>
```

The SODA-compliant version of this is in the `//soda#sdm_cutout` predefined stream.

The main point here is that you should follow section 4.3 for the [SODA] spec, i.e., use interval-xtyped parameters. Also, unless you're actually prepared to handle multiply-specified parameter values, you should use the `forced-single` mulitplicity, which makes DaCHS reject requests that contain a parameter more than once.

An extra complication occurs when SODA descriptors are generated for DAL responses. Currently, this is only envisaged for SSA. There, the descriptor has an extra limits attribute that gives, for each eligible column, minimum and maximum values or a set of values for enumerated columns.

Similar (if possibly less useful) mechanisms are conceivable for, say, partial obscore results or SIAv1. We suggest to keep the attribute name of this sort of collective characterisation as `limits`. DaCHS does not implement anything of this kind right now, though.

Metadata Error Messages

Both descriptor generators and meta makers can return (or yield, in the case of meta makers) error messages instead of either a descriptor or a link definition. This allows more fine-tuned control over the messages generated than raising an exception.

Error messages are constructed using class functions of `DatalinkFault`, which is visible to both procedure types. The class function names correspond to the message types defined in the datalink spec and match the semantics given there:

- AuthenticationFault

- AuthorizationFault
- NotFoundFault
- UsageFault
- TransientFault
- FatalFault
- Fault

Thus, a descriptor generator could look like this:

```
<descriptorGenerator>
  <code>
    with base.getTableConn() as conn:
      matchingRows = list(conn.queryToDicts(
        "select physPath from schema.myTable where pub_did=%(pubDID)s",
        locals()))
      if not matchingRows:
        return DatalinkFault.NotFoundFault(pubDID,
          "No dataset with this pubDID known here")
      return MyCustomDescriptor.fromFile(matchingRows[0]["physPath"])
  </code>
</descriptorGenerator>
```

Where sensible, you should pass (as a keyword argument) semantics (as for LinkDefs) to the `DatalinkFault`'s constructor; this would indicate what kind of link you wanted to create.

Data Functions

Data functions (see element dataFunction) generate or manipulate data. They see the descriptor and the arguments (as `args`), parsed according to the input keys produced by the meta makers, where the descriptor's `data` attribute is filled out by the first data function called (the "initial data function").

As described above, DaCHS does not enforce anything on the `data` attribute other than that it's not None after the first data function has run. It is the RD author's responsibility to make sure that all data functions in a given datalink core agree on what `data` is.

All code in a request for processed data is also passed the input parameters as processed by the context grammar. Hence, the code can rely on whatever contract is implicit in the context grammar, but not more. In particular, a datalink core has no way of knowing what data functions expects which parameters. If

no value for a parameter was provided on input, the corresponding value is None but a data function using it still is called.

An example for a generating data function is //soda#generateProduct, which may be convenient when the manipulations operate on plain local files; it basically looks like this:

```
<dataFunction>
  <code>
    descriptor.data = products.getProductForRAccref(descriptor.accref)
  </code>
</dataFunction>
```

(the actual implementation lets you require certain mime types and is therefore a bit more complicated).

You could do whatever you want, however. The following would work perfectly if you make your data functions handle lists of dicts:

```
<dataFunction>
  <setup>
    <code>
      import random
    </code>
  </setup>
  <code>
    descriptor.data = [{"pix": i, "val": random.random()}
      for i in range(20000)]
  </code>
</dataFunction>
```

It wouldn't be hard to come up with a formatter that turns this into a nice VOTable.

Filtering data functions should always come with a meta maker declaring their parameters. As an example, continuing the frequency cutout example above, consider this:

```
<dataFunction>
  <code>
    if not args.get("FREQ"):
      return

    lam_min, lam_max = (unitconv.LIGHT_C/(args[FREQ][0]*1e6)
      unitconv.LIGHT_C/(args[FREQ][1]*1e6))
    from gavo.protocols import sdm
    sdm.mangle_cutout(
      descriptor.data.getPrimaryTable(),
      lam_min, lam_max)
  </code>
</dataFunction>
```

(Ignoring for the moment troubles with half-open intervals).

There are situations in which a data function must shortcut, mostly because it is doing something other than just "pushing on" descriptor.data. Examples include preview producers or a data function that should produce the FITS header only. For cases like this, data functions can raise one of `DeliverNow` (which means `descriptor.data` must be something servable, see Data Formatters and causes that to be immediately served) or `FormatNow` (which immediately goes to the data formatter; this is less useful).

Here's an example for `DeliverNow`; a similar thing is contained in the STREAM `//soda#fits_genKindPar`:

```
<dataFunction>
  <setup>
    <code>
      from gavo.utils import fitstools
    </code>
  </setup>
  <code>
    if args["KIND"]=="HEADER":
      descriptor.data = ("application/fits-header",
        fitstools.serializeHeader(descriptor.data[0].header))
      raise DeliverNow()
  </code>
</dataFunction>
```

When writing data functions, you should raise `soda.EmptyData()` when a cutout results in empty data (e.g., because the cutout limits are out of range). If you don't, users of your service might become angry with you when they have to click away many empty windows (say).

For further examples of data functions, see the `//soda` RD coming with the distribution. If you write some, please consider whether they might be interesting for other DaCHS users, too, and submit them for inclusion into //soda.

Data Formatters

Data formatters (see element dataFormatter) take a descriptor's data attribute and build something servable out of it. Datalink cores do not absolutely need one; the default is to return `descriptor.data` (the `//soda#trivialFormatter`, which might be fine if that data is servable itself).

What is servable? The easiest thing to come up with is a pair of content type and data in byte strings; if `descriptor.data` is a Table or Data instance, the following could work:

```
<dataFormatter>
  <code>
    from gavo import formats

    return "text/plain", formats.getAsText(descriptor.data)
  </code>
</dataFormatter>
```

Another example is an excerpt from `//soda#sdm_cutout`:

```
<dataFormatter>
  <code>
    from gavo.protocols import sdm

    if len(descriptor.data.getPrimaryTable().rows)==0:
      raise base.ValidationError("Spectrum is empty.", "(various)")

    return sdm.formatSDMData(descriptor.data, args["FORMAT"])
  </code>
</dataFormatter>
```

(this goes together with a metaMaker for an input key describing FORMAT).

An alternative is to return something that has a `renderHTTP(ctx)` method that works in nevow. This is true for the Product instances that `//soda#generateProduct` generates, for example. You can also write something yourself by inheriting from protocols.products.ProductBase and overriding its iterData method.

If you don't inherit from ProductBase, be aware that this renderHTTP runs in the main server loop. If it blocks, the server blocks, so make sure that this doesn't happen. The conventional way would be to return, from the renderHTTP method, some twisted producer. Non-Product nevow resources will also not work with asynchronous datalink at this point.

Registry Matters

You can publish the metadata generating endpoint on your service by saying `<publish render="dlmeta" sets="ivo_managed"/>`. However, that is not recommended, as it clutters the registry with services that are not really usable after discovery.

Datalink services will, however, appear as capabilities of services that publish tables that have associated datalink services.

While it might be a good idea to provide some `_example` meta for all datalink services, when you register them, you really should provide one in any case so validators can pick up IDs and parameters to use when valdiating your service. Here is an example, taken from califa/q3:

```
CALIFA cubes can be cut out along RA, DEC, and spectral axes.
CIRCLE and POLYGON cutouts yield bounding boxes.  Also note that the
coverage of CALIFA cubes is hexagonal in space.  This explains
the empty area when cutting out :genparam:‘CIRCLE(225.5202 1.8486 0.001)‘
:genparam:‘BAND(366e-9 370e-9)‘ on
:dl-id:‘ivo://org.gavo.dc/~?califa/datadr3/V1200/UGC9661.V1200.rscube.fits‘.
```

Essentially, an identifier to use is given as the `dl-id` interpreted text role, whereas processing parameters are given as DALI genparams. In DaCHS, they are written as the parameter name and its value in parentheses.

Datalinks as Product URLs

In particular for larger datasets like cubes, it is rude to put the entire dataset into an obscore table. Although obscore gives expected download sizes, clients nevertheless do not usually expect to have to retrieve several gigabytes or even terabytes of data when dereferencing an obscore access URL.

While you could define additional datalink URLs and use these in Obscore – this is what lswscans/res/positions does, and there's a piece of text on this in the tutorial –, you should in general use datalinks as product URLs throughout with datasets larger than a couple of Megabytes. c8spect/q shows how to do that with completely virtual data, califa/q3 and pcslg/q are examples for what to do with FITS cubes or spectra.

This way, of course, without a datalink-enabled client people might be locked out from the dataset entirely. On the other hand, DaCHS comes with a stylesheet that enables datalink operation from a common web brower, so that's perhaps not too bad.

Aladin likes it when columns containing datalink URLs are marked up. DaCHS has two properties that let you add that markup, *targetType* and *targetTitle*. On a standalone datalink column that you just add to an output table, this could look like this (the datalink service would have an id of "dl" here; this also assumes you have a column named `pub_did`):

```
<outputField name="datalink" type="text" id="datalink_output"
    ucd="meta.ref.url"
    select="'\getConfig{web}{serverURL}/\rdId/dl/dlmeta?ID='
      || gavo_urlescape(pub_did)"
    tablehead="DL"
    description="URL of a datalink document for this dataset."
    displayHint="type=url" verbLevel="1">
  <property name="targetType"
    >application/x-votable+xml;content=datalink</property>
  <property name="targetTitle">Datalink</property>
</outputField>
```

When your product link is a datalink, you have to amend the `accref` column in your main table. This stereotypically looks like this:

```
<column original="accref">
  <property name="targetType"
    >application/x-votable+xml;content=datalink</property>
  <property name="targetTitle">Datalink</property>
</column>
```

To have datalinks rather than the plain dataset as what the accref points to, you need to change what DaCHS thinks of your dataset; this is what the //products#define rowfilter in your grammar is for:

```
<fitsProdGrammar qnd="True">
  <rowfilter procDef="//products#define">
    <bind key="path">\dlMetaURI{dl}</bind>
    <bind key="mime">'application/x-votable+xml;content=datalink'</bind>
    <bind key="fize">10000</bind>
    [...]
  </rowfilter>
  [...]
</fitsProdGrammar>
```

This includes the estimate that the datalink document will have about 10k octets; in that region, there is no need to be precise. Note that the argument to the macro dlMetaURI is the id of the datalink service; DaCHS has no way to work that out by itself.

When you do this, you must use a datalink-aware descriptor generator in SODA. When you use the recommended setup, where the accref is the inputsDir-relative path to the main file, and you're dealing with FITS, you can use the `DLFITSProductDescriptor` class. Thus, the base functionality of a FITS cutout service with datalink products would be:

```
 <service id="dl" allowed="dlget,dlmeta">
  <meta name="title">My Cutout Service</meta>
  <datalinkCore>
    <descriptorGenerator procDef="//soda#fits_genDesc"
      name="genFITSDesc">
      <bind key="accrefPrefix">'mysvcs/data'</bind>
      <bind key="descClass">DLFITSProductDescriptor</bind>
    </descriptorGenerator>
    <FEED source="//soda#fits_standardDLFuncs"/>
  </datalinkCore>
</service>
```

When not using FITS, you will need to change the descriptor generator's computation of the local file path yourself, as done, e.g., in pcslg/q.

SDM compliant tables

A common use for datalink cores in DaCHS is for server-side generation and processing of spectra as discussed in SDM processing . This almost invariably involves defining tables compliant with the spectral data model and filling them.

The `builder` parameter of //soda#sdm_genData expects a reference to an SDM compliant `data` element. To define it, you first need to define an instance table. The columns that are in there depend on your data. In the simplest case, the `//ssap#sdm-instance` mixin is sufficient and adds the columns `flux` and `spectral`. Here's how you'd add flux errors if you needed to:

```
<table id="instance" onDisk="False">
  <mixin ssaTable="slitspectra"
    spectralDescription="Wavelength"
    fluxDescription="Flux"
    >//ssap#sdm-instance</mixin>

  <column name="fluxerror"
    ucd="stat.error;phot.flux.density;em.wl"
    unit="m"
    description="Estimate for error in flux based on the procedure
      discussed at referenceURL"/>
</table>
```

What's referenced in //soda#sdm_genData is a `data` element that builds this table. Here's one that fills the table from the database:

```
<data id="get_slitcomponent">
  <!-- datamaker to pull spectra values out of the database -->
  <embeddedGrammar>
    <iterator>
      <code>
        obsId = self.sourceToken["accref"].split("/")[-1]
        with base.getTableConn() as conn:
          for row in conn.queryToDicts(
              "SELECT lambda as spectral, flux, error as fluxerror"
              " WHERE obsId=%(obsid)s ORDER BY lambda"):
            yield row
      </code>
    </iterator>
   </embeddedGrammar>
  <make table="instance">
     <parmaker>
       <apply procDef="//ssap#feedSSAToSDM"/>
     </parmaker>
  </make>
</data>
```

-- obviously, you can just as well fill it from a file (e.g., cdfspect/q, which also shows what to do when the metadata that comes with the files is boken).

The `parmaker` with the `//ssap#feedSSAToSDM` call is generic, i.e., you won't usually need any more tricks here.

Product Previews

DaCHS has built-in machinery to generate previews from normal, 2D FITS and JPEG files, where these are versions of the original dataset scaled to be about 200 pixels in width, delivered as JPEG files. These previews are shown on mousing over product links in the web interface, and they turn up as preview links in datalink interfaces. This also generates previews for cutouts.

For any other sort of data, DaCHS does not automatically generate previews. To still provide previews – which is highly recommended – there is a framework allowing you to compute and serve out custom previews. This is based on the `preview` and `preview_mime` columns which are usually set using parameters in `//products#define`.

You could use external previews by having http (or ftp) URLs, which could look like this:

```
<rowfilter procDef="//products#define">
  ...
  <bind key="preview">("http://example.org/previews/"
    +"/".join(\inputRelativePath.split("/")[2:]))</bind>
  <bind key="preview_mime">"image/jpeg"/bind>
</rowfilter>
```

(this assumes takes away to path elements from the relative paths, which typically reproduces an external hierachy). If you need to do more complex manipulations, you can have a custom rowfilter, maybe like this if you have both FITS files (for which you want DaCHS' default behaviour selected with `AUTO`) and `.complex` files with some external preview:

```
<rowfilter name="make_preview_paths">
  <code>
    srcName = os.path.basename(rowIter.sourceToken)
    if srcName.endswith(".fits"):
      row["preview"] = 'AUTO'
      row["preview_mime"] = None
    else:
      row["preview"] = ('http://example.com/previews'
        +os.path.splitext(srcName)[0]+"-preview.jpeg")
      row["preview_mime"] = 'image/jpeg'
    yield row
  </code>
```

```
</rowfilter>
<rowfilter procDef="//products#define">
  ...
  <bind key="preview">@preview</bind>
  <bind key="preview_mime">@preview_mime</bind>
</rowfilter>
```

More commonly, however, you'll have local previews. If they already exist, use a static renderer and enter full local URLs as above.

If you don't have pre-computed previews, let DaCHS handle them for you. You need to do three things:

a) define where the preview files are. This happens via a `previewDir` property on the importing data descriptor, like this:

```
<data id="import">
  <property key="previewDir">previews</property>
  ...
```

b) say that the previews are standard DaCHS generated in the `//products#define` rowfilter. The main thing you have to decide here is the MIME type of the previews you're generating. You will usually use either the macro standardPreviewPath (preferable when you have less than a couple of thousand products) or the macro splitPreviewPath to fill the preview path, but you can really enter whatever paths are convenient for you here:

```
<rowfilter procDef="//products#define">
  <bind name="table">"\schema.data"</bind>
  <bind name="mime">"image/fits"</bind>
  <bind name="preview_mime">"image/jpeg"</bind>
  <bind name="preview">\standardPreviewPath</bind>
</rowfilter>
```

c) actually compute the previews. This is usually not defined in the RD but rather using DaCHS' processing framework. Precomputing previews in the processor documentation covers this in more detail; the upshot is that this can be as simple as:

```
from gavo.helpers import processing

class PreviewMaker(processing.SpectralPreviewMaker):
  sdmId = "build_sdm_data"

if __name__=="__main__":
  processing.procmain(PreviewMaker, "flashheros/q", "import")
```

Custom UWSes

Universal Worker Systems (UWSes) allow the asynchronous operation of services, i.e., the server runs a job on behalf of the user without the need for a persistent connection.

DaCHS supports async operations of TAP and datalink out of the box. If you want to run async services defined by your own code, there are a few things to keep in mind.

(1) You'll need to prepare your database to keep track of your custom jobs (just once):

```
gavo imp //uws enable_useruws
```

(2) You'll have to allow the `uws.xml` renderer on the service in question.

(3) Things running within a UWS are fairly hard to debug in DaCHS right now. Until we have good ideas on how to make these things a bit more accessible, it's a good idea to at least for debugging also allow synchronous renderers, for instance, `form` or `api`. If something goes wrong, you can do a sync query that then drops you in a debugger in the usual manner (see the debugging chapter in the tutorial).

(4) For now, the usual queryMeta is not pushed into the uws handler (there's no good reason for that). We do, however, transport on DALI-type RESPONSEFORMAT. To enable that on automatic results (see below), say:

```
<inputKey name="responseformat" description="Preferred
  output format" type="text"/>
```

in your input table.

(5) All UWS parameters are lowercased and only available in lowercased form to server-side code. To allow cores to run in both sync and async without further worries, just have lowercase-only parameters.

(6) As usual, the core may return either a pair of (media type, content) or a data item, which then becomes a UWS result named `result` with the proper media type. You can also return None (which will make the core incompatible with most other renderers). That may be a smart thing to do if you're producing multiple files to be returned through UWS. To do that, there's a `job` attribute on the inputTable that has an `addResult(source, mediatype, name)` method. Source can be a string (in which case the string will be the result) or a file open for reading (in which case the result will be the file's content). Input tables of course don't have that attribute unless they come from the uws renderer. Hence, a typical pattern to use this would be:

```
if hasattr(inputTable, "job"):
  with inputTable.job.getWritable() as wjob:
    wjob.addResult("Hello World.\\n", "text/plain", "aux.txt")
```

or, to take the results from a file that's already on-disk:

```
if hasattr(inputTable, "job"):
  with inputTable.job.getWritable() as wjob:
    with open("other-result.txt") as src:
      wjob.addResult(src, "text/plain", "output.txt")
```

Right now, there's no facility for writing directly to UWS result files. Ask if you need that.

(7) UWS lets you add arbitrary files using standard DALI-style uploads. This is enabled if there are `file`-typed inputKeys in the service's input table. These inputKeys are otherwise ignored right now. See [DALI] for details on how these inputs work. To create an inline upload from a python client (e.g., to write a test), it's most convenient to use the requests package, like this:

```
import requests

requests.post("http://localhost:8080/data/cores/pc/uws.xml/D2hFEJ/parameters",
  {"UPLOAD": "stuff,param:upl"},
  files = {"upl": open("zw.py")})
```

From within your core, use the file name (the name of the input key) and pull the file from the UWS working directory:

```
with open(os.path.join(inputTable.job.getWD(), "mykey")) as f:
  ...
```

Hint on debugging: `gavo uwsrun` doesn't check the state the job is in, it will just try to execute it anyway. So, if your job went into error and you want to investicate why, just take its id and execute something like:

```
gavo --traceback uwsrun i1ypYX
```

Custom Pages

While DaCHS isn't actually intended to be an all-purpose server for web applications, sometimes you want to have some gadget for the browser that doesn't need VO protocols. For that, there is customPage, which is essentially a bare-bones nevow page. Hence, all (admittedly sparse) nevow documentation applies. Nevertheless, here are some hints on how to write a custom page.

First, in the RD, define a service allowing a custom page. These normally have no cores (the customPage renderer will ignore the core):

```
<service id="ui" core="null" allowed="custom"
  customPage="res/registration.py">
  <meta name="shortName">DOI registration</meta>
  <meta name="title">VOiDOI DOI registration web service</meta>
</service>
```

The python module referred to in customPage must define a `MainPage` nevow resource. The recommended pattern is like this:

```
from nevow import tags as T

from gavo import web
from gavo.imp import formal


class MainPage(
    formal.ResourceMixin,
    web.CustomTemplateMixin,
    web.ServiceBasedPage):

  name = "custom"
  customTemplate = "res/registration.html"

  workItems = None

  @classmethod
  def isBrowseable(self, service):
    return True

  def form_ivoid(self, ctx, data={}):
    form = formal.Form()
    form.addField("ivoid", formal.String(required=True), label="IVOID",
      description="An IVOID for a registred VO resource"),
    form.addAction(self.submitAction, label="Next")
    return form

  def render_workItems(self, ctx, data):
    if self.workItems:
      return ctx.tag[T.li[[m for m in self.workItems]]]
    return ""

  def submitAction(self, ctx, form, data):
    self.workItems = ["Working on %s"%data["ivoid"]]
    return self
```

The `formal.ResourceMixin` lets you define and interpret forms. The `web.ServiceBasedPage` does all the interfacing to the DaCHS (e.g., credential checking and the like). The `web.CustomTemplateMixin` lets you get your template from a DaCHS template (cf. templating guide) from a resdir-relative directory given in the `customTemplate` attribute. For widely distributed code, you should

additionaly provide some embedded stan fallback in the `defaultDocFactory` attribute -- of course, you can also give the template in stan in the first place.

On `form_invoid` and `submitAction` see below.

This template could, for this service, look like this:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:n="http://nevow.com/ns/nevow/0.1">
<head>
  <title>VOiDOI: Registration</title>
  <n:invisible n:render="commonhead"/>
</head>
<body n:render="withsidebar">
  <h1>VOiDOI: Register your VO resource</h1>
  <ul n:render="workItems"/>
  <p>VOiDOI lets you obtain DOIs for registered VO services.</p>

  <p>In the form below, enter the IVOID of the resource you want a DOI for.
  If the resource is known to our registry but has no DOI yet, the registred
  contact will be sent an e-mail to confirm DOI creation.</p>
  <n:invisible n:render="form ivoid"/>
</body>
</html>
```

Most of the details are explained in the templating guide. The exception is the `form ivoid`. This makes the `formal.ResourceMixin` call the `form_ivoid` in `MainPage` and put in whatever HTML/stan that returns. If nevow detects that the request already results from filling out the form, it will execute what your registred in `addAction` -- in this case, it's the `submitAction` method.

Important: anything you do within `addAction` runs within the (cooperative) server thread. If it blocks or performs a long computation, the server is blocked. You will therefore want to do non-trivial things either using asynchronous patterns or using `deferToThread`. The latter is less desirable but also easier, so here's how this looks like:

```
def submitAction(self, ctx, form, data):
  return threads.deferToThread(
    runRegistrationFor, data["ivoid"]
    ).addCallback(self._renderResponse
    ).addErrback(self._renderErrors)

def _renderResponse(self, result):
  # do something to render a success message (or return Redirect)
  return self
```

```
def _renderErrors(self, failure):
  # do something to render an error message, e.g., from
  # failure.getErrorMessage()
  return self
```

The embedding RD is available in the custom pages's global namespace as `RD`. Thus, the standard pattern for creating a read only table is:

```
with api.getTableConn() as conn: table =
  api.TableForDef(RD.getById("my_table"), connection=conn)
```

If you need write access, you would write:

```
with api.getWritableAdminConn() as conn:
  table = api.TableForDef(RD.getById("my_table"), connection=conn)
```

The `RD` attribute is *not* avalailable during module import. This is a bit annoying if you want to load resources from an RD-dependent place; this, in particular, applies to importing dependent modules. To provide a workaround, DaCHS calls a method `initModule(**kwargs)` after loading the module. You should accept arbitrary keyword arguments here so you code doesn't fail if we find we want to give `initModule` some further information.

The common case of importing a module from some RD-dependent place thus becomes:

```
from gavo import utils

def initModule(**kwargs):
  global oai2datacite
  modName = RD.getAbsPath("doitransfrom/oai2datacite")
  oai2datacite, _ = utils.loadPythonModule(modName)
```

Manufacturing Spectra

TODO: Update this for Datalink

Making SDM Tables

Compared to images, the formats situation with spectra is a mess. Therefore, in all likelihood, you will need some sort of conversion service to VOTables compliant to the spectral data model. DaCHS has a facility built in to support you with doing this on the fly, which means you only need to keep a single set of files around while letting users obtain the data in some format convenient to

them. The tutorial contains examples on how to generate metadata records for such additional formats.

First, you will have to define the "instance table", i.e., a table definition that will contain a DC-internal representation of the spectrum according to the data model. There's a mixin for that:

```
<table id="spectrum">
  <mixin ssaTable="hcdtest">//ssap#sdm-instance</mixin>
</table>
```

In addition to adding lots and lots of params, the mixin also defines two columns, `spectral` and `flux`; these have units and ucds as taken from the SSA metadata. You can add additional columns (e.g., a flux error depending on the spectral coordinate) as requried.

The actual spectral instances can be built by sdmCores and delivered through DaCHS' product interface. Note, however, that clients supporting getData wouldn't need to do this. You'll still have to define the data item defined below.

sdmCores, while potentially useful with common services, are intended to be used by the product renderer for dcc product table paths. They contain a data item that must yield a primary table that is basically sdm compliant. Most of this is done by the //ssap#feedSSAToSDM apply proc, but obviously you need to yield the spectral/flux pairs (plus potentially more stuff like errors, etc, if your spectrum table has more columns. This comes from the data item's grammar, which probably must always be an embedded grammar, since its sourceToken is an SSA row in a dictionary. Here's an example:

```
<sdmCore queriedTable="hcdtest" id="mksdm">
  <data id="getdata">
    <embeddedGrammar>
      <iterator>
        <code>
          labels = ("spectral", "flux")
          relPath = self.sourceToken["accref"].split("?")[-1]
          with self.grammar.rd.openRes(relPath) as inF:
            for ln in inF:
              yield dict(zip(labels,ln.split()))
        </code>
      </iterator>
    </embeddedGrammar>
    <make table="spectrum">
      <parmaker>
        <apply procDef="//ssap#feedSSAToSDM"/>
      </parmaker>
    </make>
  </data>
</sdmCore>
```

Note: spectral, flux, and possibly further items coming out of the iterator must be in the units units promised by the SSA metadata (fluxSI, spectralSI). Declarations to this effect are generated by the `//ssap#sdm-instance` mixin for the spectral and flux columns.

The sdmCores are always combined with the sdm renderer. It passes an accref into the core that gets turned into an row from queried table; this must be an "ssa" table (i.e., right now something that mixes in `//ssap#hcd`). This row is the input to the embedded data descriptor. Hence, this has no sources element, and you must have either a custom or embedded grammar to deal with this input.

Echelle Spectra

Echelle spectrographs "fold" a spectrum into several orders which may be delivered in several independent mappings from spectral to flux coordinate. In this split form, they pose some extra problems, dealt with in an extra system RD, `//echelle`. For merged Echelle spectra, just use the standard SSA framework.

Table

Echelle spectra have additional metadata that should end up in their SSA metadata table – these are things like the number of orders, the minimum and maximum (Echelle) order, and the like. To pull these columns into your metadata table, use the ssacols stream, for example like this:

```
<table id="ordersmeta" onDisk="True" adql="True">
  <meta name="description">SSA metadata for split-order
    Flash/Heros Echelle spectra</meta>
  <mixin
    [...]
    statSpectError="0.05"
    spectralResolution="2.5e-11"
  >//ssap#hcd</mixin>
  <mixin
    calibLevel="1">//obscore#publishSSAPHCD</mixin>
  <column name="localKey" type="text"
    ucd="meta.id"
    tablehead="Key"
    description="Local observation key."
    verbLevel="1"/>
  <STREAM source="//echelle#ssacols"/>
</table>
```

Supporting getData

DaCHS still has support the now-abandoned 2012 getData specification by Demleitner and Skoda. If you think you still want this, contact the authors; meanwhile, you really should be using datalink for whatever you think you need getData for.

Adapting Obscore

You may want extra, locally-defined columns in your obscore tables. To support this, there are three hooks in obscore that you can exploit. The hooks are in `userconfig.rd` (see Userconfig RD in the operator's guide to where it is and how to get started with it) It helps to have a brief look at the `//obscore` RD (e.g., using `gavo admin dumpDF //obscore`) to get an idea what these hooks do.

Within the template `userconfig.rd`, there are already three STREAMs with ids starting with obscore.; these are referenced from within the system `//obscore` RD. Here's an somewhat more elaborate example:

```
<STREAM id="obscore-extracolumns">
  <column name="fill_factor"
    description="Fill factor of the SED"
    verbLevel="20"/>
</STREAM>

<STREAM id="obscore-extrapars">
  <mixinPar name="fillFactor"
    description="The SED's fill factor">NULL</mixinPar>
</STREAM>

<STREAM id="obscore-extraevents">
  <property name="obscoreClause" cumulate="True">
    ,
    CAST(\\\\fillFactor AS real) AS fill_factor
  </property>
</STREAM>
```

(to be on the safe side: there need to be four backslashes in front of fillFactor; this is just a backslash doubly-escaped. Sorry about this).

The way this is used in an actual mixin would be like this:

```
<table id="specs" onDisk="True">
  <mixin ...>//ssap#hcd</mixin>
  <mixin
    ... (all the usual parameters)
    fillFactor="0.3">//obscore#publishSSAPMIXC</mixin>
</table>
```

What's going on here? Well, `obscore-extracolumns` is easy – this material is directly inserted into the definition of the obscore view (see the table with id `ObsCore` within the `//obscore` RD). You could abuse it to insert other stuff than columns but probably should not.

The tricky part is `obscore-extraevents`. This goes into the `//obscore#_publishCommon` STREAM and ends up in all the publish mixins

in obscore. Again, you could insert mixinPars and similar at this point, but the only thing you really must do is add lines to the big SQL fragment in the `obscoreClause` property that the mixin leaves in the table. This is what is made into the table's contribution to the big obscore union. Just follow the example above and, in particular, always CAST to the type you have in the metadata, since individual tables might have NULLs in the values, and you do not want misguided attempts by postgres to do type inference then.

If you actually must know why you need to double-escape fillFactor and what the magic with the `cumulate="True"` is, ask.

Finally, `obscore-extrapars` directly goes into a core component of obscore, one that all the various publish mixins there use. Hence, all of them grow your functionality. That is also why it is important to give defaults (i.e., element content) to all mixinPars you give in this way – without them, all those other publish mixins would fail unless their applications in the RDs were fixed.

If you change `%#obscore-extracolumns`, all the statement fragments contributed by the obscore-published tables need to be fixed. To spare you the effort of touching a potentially sizeable number of RDs, there's a data element in //obscore that does that for you; so, after every change just run:

```
gavo imp //obscore refreshAfterSchemaUpdate
```

This may fail if you didn't clean up properly after deleting a resource that once contributed to ivoa.obscore. In that case you'll see an error message like:

```
*** Error: table u'whatever.main' could not be located in dc_tables
```

In that case, just tell DaCHS to forget the offending table:

```
gavo purge whatever.main
```

Another problem can arise when a table once was published to obscore but now no longer is while still existing. DaCHS in that case will still have an entry for the table in ivoa._obscoresources, which results in an error like:

```
Table definition of whatever.main> has no property 'obscoreClause' set
```

The fastest way to fix this situation is to drop the offending line in the database manually:

```
psql gavo -c "delete from ivoa._obscoresources where tablename='whatever.main'"
```

Writing Custom Grammars

A custom grammar simply is a python module located within a resource directory defining a row iterator class derived from `gavo.grammars.customgrammar.CustomRowIterator`. This class must be called `RowIterator`. You want to override the `_iterRows` method. It will have to yield row dictionaries, i.e., dictionaries mapping string keys to something (preferably strings, but you will usually get away with returning complete values even without fancy rowmakers).

So, a custom grammar module could look like this:

```
from gavo.grammars.customgrammar import CustomRowIterator

class RowIterator(CustomRowIterator):
  def _iterRows(self):
    for i in xrange(int(self.sourceToken)):
      yield {'index': i, 'square': i**2}
```

This would be used with a `data` material like:

```
<sources><item>4</item><item>40</item></sources>
<customGrammar module="res/sillygrammar"/>
```

– `self.sourceToken` simply contains whatever the `sources` produce. One `RowIterator` will be constructed for each item.

It is highly recommended to keep track of the current position so DaCHS can give more useful error messages. When an error occurs, DaCHS will call the iterator's `getLocator` method. This returns an arbitrary string, where obviously it's a good idea if it leads users to somewhere close to where the problem will be. Here's a custom grammar reading space-separated key-value pairs from a file:

```
class RowIterator(CustomRowIterator):
  def _iterRows(self):
    self.lineNumber = 0
    with open(self.sourceToken) as f:
      for self.lineNumber, line in enumerate(f):
        yield dict(zip(["key", "value"], line.split(" ", 1)))

  def getLocator(self):
    return "line %s"%self.lineNumber
```

Note that `getLocator` doesn't include the source file name; it will be inserted into the error message by DaCHS.

Do not override magic methods, since you may lose row filters, sourceFields, and the like if you do. An exception is the constructor. If you must, you can override it, but you must call the parent constructor, like this:

```
class RowIterator(CustomRowIterator):
  def __init__(self, grammar, sourceToken, sourceRow=None):
    CustomRowIterator.__init__(self, grammar, sourceToken, sourceRow)
    <your code>
```

In practice (i.e., with `<sources pattern="*"/>`) ``self.sourceToken will be a file name. When you call `makeData` manually and pass a `forceSource` argument, its value will show up in `self.sourceToken` instead.

For development, it may be convenient to execute your custom grammar as a python module. To enable that, just append a:

```
if __name__=="__main__":
  import sys

  from gavo.grammars.customgrammar import CustomGrammar
  ri = RowIterator(CustomGrammar(None), sys.argv[1])
  for row in ri:
    print row
```

to your module. You can then run things like:

```
python res/mygrammar.py data/inhabitedplanet.fits
```

and see the rows as they're generated.

A row iterator will be instanciated for each source processed. Thus, you should usually not perform expensive operations in the constructor unless they depend on sourceToken. In general, you should rather define a function `makeDataPack` in the module. Whatever is returned by this function is available as `self.grammar.dataPack` in the row iterator.

The function receives an instance of the `customGrammar` as an argument. This means you can access the resource descriptor and properties of the grammar. As an example of how this could be used, consider this RD fragment:

```
<table id="defTable">
  ...
</table>

<customGrammar module="res/grammar">
  <property name="targetTable">defTable</property>
</customGrammar>
```

Then you could have the following in `res/grammar.py`:

```
def makeDataPack(grammar):
  return grammar.rd.getById(grammar.getProperty("targetTable"))
```

and access the table in the row iterator.

Also look into `EmbeddedGrammar`, which may be a more convenient way to achieve the same thing.

A fairly complex example for a custom grammar is a provisional Skyglow grammar .

Dispatching Grammars

With normal grammars, all rows are fed to all rowmakers of all makes within a data object. The rowmakers can then decide to not process a given row by raising `IgnoreThisRow` or using the trigger mechanism. However, when filling complex data models with potentially dozens of tables, this becomes highly inefficient.

When you write your own grammars, you can do better. Instead of just yielding a row from `_iterRows`, you yield a pair of a role (as specified in the `role` attribute of a `make` element) and the row. The machinery will then pass the row only to the feeder for the table in the corresponding make.

Currently, the only way to define such a dispatching grammar is to use a custom grammar or an embedded grammar. For these, just change your `_iterRows` and say `isDispatching="True"` in the `customGrammar` element. If you implement `getParameters`, you can return either pairs of role and row or just the row; in the latter case, the row will be broadcast to all parmakers.

Special care needs to be taken when a dispatching grammar parses products, because the product table is fed by a special make inserted from the products mixin. This make of course doesn't see the rows you are yielding from your dispatching grammar. This means that without further action, your files will not end up in the product table at all. In turn, getproducts will return 404s instead of your products.

To fix this, you need to explicitly yield the rows destined for the products table with a products role, from within your grammar. Where the grammar yield rows for the table with metadata (i.e., rows that actually contain the fields with prodtblAccref, prodtblPath, etc), yield to the products table, too, like this: `yield ("products", newRow)`.

Functions Available for Row Makers

In principle, you can use arbitrary python expressions in var, map and proc elements of row makers. In particular, the namespace in which these expressions are executed contains math, os, re, time, and datetime modules as well as gavo.base, gavo.utils, and gavo.coords.

However, much of the time you will get by using the following functions that are immediately accessible in the namespace:

TAItoTT(tai) returns TDT for a (datetime.datetime) TAI.

TTtoTAI(tdt) returns TAI for a (datetime.datetime) TDT.

bYearToDateTime(bYear) returns a datetime.datetime instance for a fractional Besselian year.

This uses the formula given by Lieske, J.H., A&A 73, 282 (1979).

computeMean(val1, val2) returns the mean value between two values.

Beware: Integer division done here for the benefit of datetime calculations.

```
>>> computeMean(1.,3)
2.0
>>> computeMean(datetime.datetime(2000, 10, 13),
...   datetime.datetime(2000, 10, 12))
datetime.datetime(2000, 10, 12, 12, 0)
```

dateTimeToJYear(dt) returns a fractional (julian) year for a datetime.datetime instance.

dateTimeToJdn(dt) returns a julian day number (including fractionals) from a datetime instance.

dateTimeToMJD(dt) returns a modified julian date for a datetime instance.

dmsToDeg(dmsAngle, sepChar=None) returns the degree minutes seconds-specified dmsAngle as a float in degrees.

```
>>> "%3.8f"%dmsToDeg("45 30.6")
'45.51000000'
>>> "%3.8f"%dmsToDeg("45:30.6", ":")
'45.51000000'
>>> "%3.8f"%dmsToDeg("-45 30 7.6")
'-45.50211111'
>>> dmsToDeg("junk")
Traceback (most recent call last):
ValueError: Invalid dms value with sepChar None: 'junk'
```

getAccrefFromStandardPubDID(pubdid, authBase=u'ivo://org.gavo.dc/~?') returns an accref from a standard DaCHS PubDID.

This is basically the inverse of getStandardPubDID. It will raise ValueErrors if pubdid doesn't start with ivo://<authority>/~?.

The function does not check if the remaining characters are a valid accref, much less whether it can be resolved.

authBase's default will reflect you system's settings on your installation, which probably is not what's given in this documentation.

getDatalinkMetaLink(dlSvc, accref) returns a datalink URL for the product referenced through accref with the datalink service dlSvc.

This assumes that dlSvc uses the standard DaCHS pubDIDs. dlSvc needs to be the service element.

A typical use is in a metaMaker and would look like this:

```
getDatalinkMetaLink(rd.getById("dl"), descriptor.accref)
```

getFileStem(fPath) returns the file stem of a file path.

The base name is what remains if you take the base name and split off extensions. The extension here starts with the last dot in the file name, except up to one of some common compression extensions (.gz, .xz, .bz2, .Z, .z) is stripped off the end if present before determining the extension.

```
>>> getFileStem("/foo/bar/baz.x.y")
'baz.x'
>>> getFileStem("/foo/bar/baz.x.gz")
'baz'
>>> getFileStem("/foo/bar/baz")
'baz'
```

getFlatName(accref) returns a unix-compatible file name for an access reference.

The file name will not contain terrible characters, let alone slashes. This is used to, e.g., keep all previews in one directory.

getInputsRelativePath(absPath, liberalChars=True) returns absath relative to the DaCHS inputsDir.

If `absPath` is not below `inputsDir`, a `ValueError` results. On `liberalChars`, wee see the function getRelativePath.

In rowmakers and rowfilters, you'll usually use the macro `\inputRelativePath` that inserts the appropriate code.

getQueryMeta() returns a query meta object from somewhere up the stack.

This is for row makers running within a service. This can be used to, e.g., enforce match limits by writing getQueryMeta()["dbLimit"].

getRelativePath(fullPath, rootPath, liberalChars=True) returns rest if fullPath has the form rootPath/rest and raises an exception otherwise.

Pass `liberalChars=False` to make this raise a ValueError when URL-dangerous characters (blanks, amperands, pluses, non-ASCII, and similar) are present in the result. This is mainly for products.

getStandardPubDID(path) returns the standard DaCHS PubDID for `path`.

The publisher dataset identifier (PubDID) is important in protocols like SSAP and obscore. If you use this function, the PubDID will be your authority, the path compontent ~, and the inputs-relative path of the input file as the parameter.

`path` can be relative, in which case it is interpreted relative to the DaCHS `inputsDir`.

You *can* define your PubDIDs in a different way, but you'd then need to provide a custom descriptorGenerator to datalink services (and might need other tricks). If your data comes from plain files, use this function.

In a rowmaker, you'll usually use the standardPubDID macro.

getWCSAxis(header, axisIndex, forceSeparable=False) returns a WCSAxis instance from an axis index and a FITS header.

If the axis is mentioned in a transformation matrix (CD or PC), a `ValueError` is raised (use `forceSeparable` to override).

The `axisIndex` is 1-based; to get a transform for the axis described by CTYPE1, pass 1 here.

The object returned has methods like `pixToPhys`, `physToPix` (and their `pix0` brethren), and `getLimits`.

Note that at this point WCSAxis only supports linear transforms (it's a DaCHS-specific implementation). We'll extend it on request.

hmsToDeg(hms, sepChar=None) returns the time angle (h m s.decimals) as a float in degrees.

```
>>> "%3.8f"%hmsToDeg("22 23 23.3")
'335.84708333'
>>> "%3.8f"%hmsToDeg("22:23:23.3", ":")
'335.84708333'
>>> "%3.8f"%hmsToDeg("222323.3", "")
'335.84708333'
>>> hmsToDeg("junk")
Traceback (most recent call last):
ValueError: Invalid time with sepChar None: 'junk'
```

iterSimpleText(f) iterates over `(physLineNumber, line)` in f with some usual conventions for simple data files.

You should use this function to read from simple configuration and/or table files that don't warrant a full-blown grammar/rowmaker combo. The intended use is somewhat like this:

```
with open(rd.getAbsPath("res/mymeta")) as f:
  for lineNumber, content in iterSimpleText(f):
    try:
      ...
    except Exception, exc:
      sys.stderr.write("Bad input line %s: %s"%(lineNumber, exc))
```

The grammar rules are, specifically:

- leading and trailing whitespace is stripped
- empty lines are ignored
- lines beginning with a hash are ignored
- lines ending with a backslash are joined with the following line; to have intervening whitespace, have a blank in front of the backslash.

jdnToDateTime(jd) returns a `datetime.datetime` instance for a julian day number.

killBlanks(literal) returns the string literal with all blanks removed.

This is useful when numbers are formatted with blanks thrown in.

Nones are passed through.

lastSourceElements(path, numElements) returns a path made up from the last `numElements` items in `path`.

loadPythonModule(fqName, relativeTo=None) imports fqName and returns the module with a module description.

The module description is what what find_module returns; you may need this for reloading and similar.

Do not use this function to import DC-internal modules; this may mess up singletons since you could bypass python's mechanisms to prevent multiple imports of the same module.

fqName is a fully qualified path to the module without the .py, unless relativeTo is given, in which case it is interpreted as a relative path. This for letting modules in resdir/res import each other by saying:

```
mod, _ = api.loadPythonModule("foo", relativeTo=__file__)
```

The python path is temporarily amended with the path part of the source module.

If the module is in /var/gavo/inputs/foo/bar/mod.py, Python will know the module as foo_bar_mod (the last two path components are always

added). This is to keep Python from using the module when someone writes import mod.

makeAbsoluteURL(path, canonical=False) returns a fully qualified URL for a rooted local part.

This will reflect the http/https access mode unless you pass canonical=True, in which case [web]serverURL will be used unconditionally.

makeProductLink(key, withHost=True, useHost=None) returns the URL at which a product can be retrieved.

key can be an accref string or an RAccref

makeSitePath(path) returns a rooted local part for a server-internal URL.

uri itself needs to be server-absolute; a leading slash is recommended for clarity but not mandatory.

makeTimestamp(date, time) makes a datetime instance from a date and a time.

mjdToDateTime(mjd) returns a `datetime.datetime` instance for a modified julian day number.

Beware: This loses a couple of significant digits due to transformation to jd.

parseAngle(literal, format, sepChar=None) converts the various forms angles might be encountered to degrees.

format is one of hms, dms, fracHour. For sexagesimal/time angles, you can pass a sepChar (default: split at blanks) that lets you specify what separates hours/degrees, minutes, and seconds.

```
>>> str(parseAngle("23 59 59.95", "hms"))
'359.999791667'
>>> "%10.5f"%parseAngle("-20:31:05.12", "dms", sepChar=":")
' -20.51809'
>>> "%010.6f"%parseAngle("21.0209556", "fracHour")
'315.314334'
```

parseBooleanLiteral(literal) returns a python boolean from some string.

Boolean literals are strings like True, false, on, Off, yes, No in some capitalization.

parseDate(literal, format='%Y-%m-%d') returns a `datetime.date` object of literal parsed according to the strptime-similar format.

The function understands the special `dateFormat !!jYear` (stuff like 1980.89).

parseFloat(literal) returns a float from a literal, or None if literal is None or an empty string.

Temporarily, this includes a hack to work around a bug in psycopg2.

```
>>> parseFloat("   5e9 ")
5000000000.0
>>> parseFloat(None)
>>> parseFloat("  ")
>>> parseFloat("wobbadobba")
Traceback (most recent call last):
ValueError: could not convert string to float: wobbadobba
```

parseISODT(literal) returns a datetime object for a ISO time literal.

There's no real timezone support yet, but we accept and ignore various ways of specifying UTC.

```
>>> parseISODT("1998-12-14")
datetime.datetime(1998, 12, 14, 0, 0)
>>> parseISODT("1998-12-14T13:30:12")
datetime.datetime(1998, 12, 14, 13, 30, 12)
>>> parseISODT("1998-12-14T13:30:12Z")
datetime.datetime(1998, 12, 14, 13, 30, 12)
>>> parseISODT("1998-12-14T13:30:12.224Z")
datetime.datetime(1998, 12, 14, 13, 30, 12, 224000)
>>> parseISODT("19981214T133012Z")
datetime.datetime(1998, 12, 14, 13, 30, 12)
>>> parseISODT("19981214T133012+00:00")
datetime.datetime(1998, 12, 14, 13, 30, 12)
>>> parseISODT("junk")
Traceback (most recent call last):
ValueError: Bad ISO datetime literal: junk (required format: yyyy-mm-ddThh:mm:ssZ)
```

parseInt(literal) returns an int from a literal, or None if literal is None or an empty string.

```
>>> parseInt("32")
32
>>> parseInt("")
>>> parseInt(None)
```

parseTime(literal, format='%H:%M:%S') returns a `datetime.timedelta` object for literal parsed according to format.

For format, you can the magic values `!!secondsSinceMidnight`, `!!decimalHours` or a strptime-like spec using the H, M, and S codes.

```
>>> parseTime("89930", "!!secondsSinceMidnight")
datetime.timedelta(1, 3530)
>>> parseTime("23.4", "!!decimalHours")
datetime.timedelta(0, 84240)
>>> parseTime("3.4:5", "%H.%M:%S")
datetime.timedelta(0, 11045)
```

```
>>> parseTime("20:04", "%H:%M")
datetime.timedelta(0, 72240)
```

parseTimestamp(literal, format='%Y-%m-%dT%H:%M:%S') returns a `datetime.datetime` object from a literal parsed according to the strptime-similar format.

A `ValueError` is raised if literal doesn't match format (actually, a parse with essentially DALI-standard ISO representation is always tried)

parseWithNull(literal, baseParser, nullLiteral=<Undefined>, default=None, checker=None) returns default if literal is `nullLiteral`, else `baseParser(literal)`.

If `checker` is non-None, it must be a callable returning `True` if its argument is a null value.

`nullLiteral` is compared against the unprocessed literal (usually, a string). The intended use is like this (but note that often, a `nullExcs` attribute on a rowmaker `map` element is the more elegant way:

```
>>> parseWithNull("8888.0", float, "8888")
8888.0
>>> print(parseWithNull("8888", float, "8888"))
None
>>> print(parseWithNull("N/A", int, "N/A"))
None
```

quoteProductKey(key) returns key as getproduct URL-part.

If `key` is a string, it is quoted as a naked accref so it's usable as the path part of an URL. If it's an `RAccref`, it is just stringified. The result is something that can be used after getproduct in URLs in any case.

requireValue(val, fieldName) returns `val` unless it is `None`, in which case a `ValidationError` for `fieldName` will be raised.

scale(val, factor, offset=0) returns val*factor+offset if val is not None, None otherwise.

This is when you want to manipulate a numeric value that may be NULL. It is a somewhat safer alternative to using nullExcs with scaled values.

toMJD(literal) returns a modified julian date made from some datetime representation.

Valid representations include:

- MJD (a float smaller than 1e6)
- JD (a float larger than 1e6)
- datetime.datetime instances
- ISO time strings.

Scripting

As much as it is desirable to describe tables in a declarative manner, there are quite a few cases in which some imperative code helps a lot during table building or teardown. Resource descriptors let you embed such imperative code using script elements. These are children of the make elements since they are exclusively executed when actually importing into a table.

Currently, you can enter scripts in SQL and python, which may be called at various phases during the import.

SQL scripts

In SQL scripts, you separate statements with semicolons. Note that no statements in an SQL script may fail since that will invalidate the transaction. This is a serious limitation since you must not commit or begin transactions in SQL scripts as long as Postgres does not support nested transactions.

You can use table macros in the SQL scripts to parametrize them; the most useful among those probably is `\curtable` containing the fully qualified name of the table being processed.

Python scripts

Python scripts can be indented by a constant amount.

The table object currently processed is accessible as table. In particular, you can use this to issue queries using `table.query(query, arguments)` (parallel to dbapi.execute) and to delete rows using `table.deleteMatching(condition, pars)`. The current RD is accessible as `table.rd`, so you can access items from the RD as `table.rd.getById("some_id")`, and the recommended way to read stuff from the resource directory is `table.rd.openRes("res/some_file)`.

Some types of scripts may have additional names available. Currently:

- `newSource` and `sourceDone` have the name `sourceToken`; this is the sourceToken as passed to the grammar; usually, that's the file name that's parsed from, but other constellations are possible.
- `sourceDone` has `feeder` -- that is the DaCHS-internal glue to filling tables. The main use of this is that you can call its `flush()` method, followed by a `table.commit()`. This may be interesting in updating grammars where you preserve what's already imported. Note, however, that this may come with a noticeable performance penalty.

Script types

The type of a script corresponds to the event triggering its execution. The following types are defined right now:

- preImport -- before anything is written to the table
- preIndex -- before the indices on the table are built
- preCreation -- immediately before the table DDL is executed
- postCreation -- after the table (incl. indices) is finished
- beforeDrop -- when the table is about to be dropped
- newSource -- every time a new source is started
- sourceDone -- every time a source has been processed

Note that preImport, preIndex, and postCreation scripts are not executed when the make's table is being updated, in particular, in data items with `updating="True"`. The only way to run scripts in such circumstances is to use newSource and sourceDone scripts.

Examples

This snippet sets a flag when importing some source (in this case, that's an RD, so we can access sourceToken.sourceId:

```
<script type="newSource" lang="python" id="markDeleted">
  table.query("UPDATE %s SET deleted=True"
    " WHERE sourceRD=%%(sourceRD)s"%id,
    {"sourceRD": sourceToken.sourceId})
</script>
```

This is a hacked way of ensuring some sort of referential integrity: When a table containing "products" is dropped, the corresponding entries in the products table are deleted:

```
<script type="beforeDrop" lang="SQL" name="clean product table">
  DELETE FROM products WHERE sourceTable='\curtable'
</script>
```

Note that this is actually quite hazardous because if the table is dropped in any way not using the make element in the RD, this will not be executed. It's usually much smarter to tell the database to do the housekeeping. Rules are typically set in postCreation scripts:

```
<script type="postCreation" lang="SQL">
  CREATE OR REPLACE RULE cleanupProducts AS
    ON DELETE TO \curtable DO ALSO
    DELETE FROM products WHERE key=OLD.accref
</script>
```

The decision if such arrangements are make before the import, before the indexing or after the table is finished needs to be made based on the script's purpose.

Another use for scripts is SQL function definition:

```
<script type="postCreation" lang="SQL" name="Define USNOB matcher">
  CREATE OR REPLACE FUNCTION usnob_getmatch(alpha double precision,
    delta double precision, windowSecs float
  ) RETURNS SETOF usnob.data AS $$
  DECLARE
    rec RECORD;
  BEGIN
    FOR rec IN (SELECT * FROM usnob.data WHERE
      q3c_join(alpha, delta, raj2000, dej2000, windowSecs/3600.))
    LOOP
      RETURN NEXT rec;
    END LOOP;
  END;
  $$ LANGUAGE plpgsql;
</script>
```

You can also load data, most usefully in preIndex scripts (although beforeImport would work as well here):

```
<script type="preIndex" lang="SQL" name="create USNOB-PPMX crossmatch">
    SET work_mem=1000000;
    INSERT INTO usnob.ppmxcross (
      SELECT q3c_ang2ipix(raj2000, dej2000) AS ipix, p.localid
      FROM
        ppmx.data AS p,
        usnob.data AS u
      WHERE q3c_join(p.alphaFloat, p.deltaFloat,
        u.raj2000, u.dej2000, 1.5/3600.))
</script>
```

ReStructuredText

Text needing some amount of markup within DaCHS is almost always input as ReStructuredText (RST). The source versions of the DaCHS documentation give examples for such markup, and DaCHS users should at least briefly skim the ReStructuredText primer.

DaCHS contains some RST extensions. Those specifically targeted at writing DALI-compliant examples of them are discussed with the examples renderer

Generally useful extensions include:

bibcode This text role formats the argument as a link into ADS when rendered as HTML. For technical reasons, this currently ignores the configured ADS mirror and always uses the Heidelberg one. Complain if this bugs you. To use it, you'd write:

```
See also :bibcode:‘2011AJ....142....3H‘.
```

Extensions for writing DaCHS-related documentation include:

dachsdoc A text role generating a link into the current DaCHS documentation. The argument is the relative path, e.g., `:dachsdoc:‘opguide.html#userconfig-rd‘`.

dachsref A text role generating a link into the reference documentation. The argument is a section header within the reference documentation, e.g., `:dachsref:‘//epntap2#populate-2_0‘` or `:dachsref:‘the form renderer‘`.

samplerd A text role generating a link to an RD used by the GAVO data center (exhibiting some feature). The argument is the relative path to the RD (or, really, anything else in the VCS), e.g., `:samplerd:‘ppmxl/q.rd‘`.

(if you add anything here, please also amend the document source's README).

Code in DaCHS

This section contains a few general points for python code embedded in DaCHS, be it custom pages, cores, or grammars, or even procs.

Importing modules

To keep the various resources as separate from each other as possible, DaCHS does not manipulate Python's import path. However, one frequently wants to have library-like modules providing common functionality or configuration in a resdir (the conventional place for these would be in `res/`).

To import these, use `api.loadPythonModule(path)`. Path, here, is the full path to the file containing the python code, but without the `.py`. When you have the RD, the conventional pattern is:

```
mymod, _ = api.loadPythonModule(rd.getAbsPath("res/mymod"))
```

instead of `import mymod`.

As you can see `loadPythonModule` returns a tuple; you're very typically only interested in the first element.

Note in particular that for modules loaded in this way, the usual rule that you can just import modules next to you does not apply. To import a modules "next to" you without having to go through the RD, use the special form:

```
siblingmod, _ = api.loadPythonModule("siblingmod", relativeTo=__file__)
```

instead of `import siblingmod`. This will take the directory part for what's in `relativeTo` (here, the module's own path) and make a full path out of the first argument to pull the modules from there.

The DaCHS API

User extension code (e.g., custom cores, custom grammars, processors) for DaCHS should only use DaCHS functions from its api as described below. We will try to keep it stable and at any rate warn in the release notes if we change it. For various reasons, the module also contains a few modules. These, and in particular their content, are *not* part of the API.

Note that at this point this is *not* what is in the namespace of rowmakers, rowfilters, and similar in-RD procedures. We do not, at this point, recommend importing the api. If you do it anyway, we'd appreciate if you told us.

Before using non-API DaCHS functions, please inquire on the dachs-support mailing list (cf. http://docs.g-vo.org/DaCHS).

To access DaCHS API functions, say:

```
from gavo import api
```

(perhaps adding an `as dachsapi` if there is a risk of confusion) and reference symbols with the explicit module name (i.e., `api.makeData` rather than picking individual names) in order to help others understand what you've written.

Here is an alphabetical list of the documented API functions:

Class ADQLTAPJob

A facade for an ADQL-based async TAP job.

Construct it with the URL of the async endpoint and a query.

Alternatively, you can give the endpoint URL and a jobId as a keyword parameter. This only makes sense if the service has handed out the jobId before (e.g., when a different program takes up handling of a job started before).

See adql.html for details.

Class AnetHeaderProcessor

A file processor for calibrating FITS frames using astrometry.net.

It might provide calibration for "simple" cases out of the box. You will usually want to override some solver parameters. To do that, define class attributes sp_<parameter name>, where the parameters available are discussed in helpers.anet's docstring. sp_indices is one thing you will typically need to override.

To use SExtractor rather than anet's source extractor, override sexControl, to use an object filter (see anet.getWCSFieldsFor), override the objectFilter attribute.

To add additional fields, override _getHeader and call the parent class' _getHeader method. To change the way astrometry.net is called, override the _solveAnet method (it needs to return some result anet.of getWCSFieldsFor) and call _runAnet with your custom arguments for getWCSFieldsFor.

See processors#astrometry-net for details.

Class Authenticate

raised to initiate an authentication request.

Authenticates are optionally constructed with the realm the user shall authenticate in. If you leave the realm out, the DC-wide default will be used.

Class BadCode

is raised when some code could not be compiled.

BadCodes are constructed with the offending code, a code type, the original exception, and optionally a hint and a position.

Class Binding

OpenSSL API wrapper.

Class CannotComputeHeader

is raised when no FITS header was generated by a HeaderProcessor.

Specifically, this is what gets raised when `_getHeader` returns `None`.

Core

A definition of the "active" part of a service.

A core will receive input from a renderer in the form of a `svcs.CoreArgs` (see Core Args). A core will return a table or perhaps directly data as discussed in DaCHS' Service Interface .

The abstract core element will never occur in resource descriptors. See Cores Available for concrete cores. Use the names of the concrete cores in RDs.

Class DBError

Base class for error exceptions.

Class DBTable

An interface to a table in the database.

These are usually created using `api.TableForDef(tableDef)` with a table definition obtained, e.g., from an RD, saying `onDisk=True`.

When constructing a DBTable, it will be created if necessary (unless `create=False` is passed), but indices or primary keys keys will only be created on a call to `importFinished`.

The constructor does not check if the schema of the table on disk matches the tableDef. If the two diverge, all kinds of failures are conceivable; use `dachs val -c` to make sure on-disk structure match the RDs.

You can pass a `nometa` boolean kw argument to suppress entering the table into the `dc_tables` table.

You can pass an exclusive boolean kw argument; if you do, the `iterQuery` (and possibly similar methods in the future) method will block concurrent writes to the selected rows ("FOR UPDATE") as long as the transaction is active.

The main attributes (with API guarantees) include:

- tableDef -- the defining tableDef
- getFeeder() -- returns a function you can call with rowdicts to insert them into the table.
- importFinished() -- must be called after you've fed all rows when importing data.
- drop() -- drops the table in the database
- recreate() -- drops the table and generates a new empty one.
- getTableForQuery(...) -- returns a Table instance built from a query over this table (you probably to use `conn.query*` and `td.getSimpleQuery` instead).

Constant DEG

A constant, valued 0.0174532925199

Constant DEG_ARCSEC

A constant, valued 0.000277777777778

Constant DEG_MAS

A constant, valued 2.77777777778e-07

Class Data

A collection of tables.

`Data`, in essence, is the instanciation of a `DataDescriptor`.

It is what `makeData` returns. In typical one-table situations, you just want to call the `getPrimaryTable()` method to obtain the table built.

Class DataError

is raised when something is wrong with a data set.

When facing the web, these yield HTTP status 406.

Class Error

The base class for all exceptions that can be expected to escape a module.

Apart from the normal message, you can give a `hint` constructor argument.

Class FileProcessor

An abstract base for a source file processor.

In concrete classes, you need to define a `process(fName)` method receiving a source as returned by the dd (i.e., usually a file name).

You can override the method `_createAuxiliaries(dataDesc)` to compute things like source catalogues, etc. Thus, you should not need to override the constructor.

These objects are usually constructed thorough `api.procmain` as discussed in processing.html.

Class ForbiddenURI

raised to generate an HTTP 403 response.

Class HeaderProcessor

A base for processors doing FITS header manipulations.

The processor builds naked FITS headers alongside the actual files, with an added extension .hdr (or whatever is in the headerExt attribute). The presence of a FITS header indicates that a file has been processed. The headers on the actual FITS files are only replaced if necessary.

The basic flow is: Check if there is a header. If not, call _getNewHeader(srcFile) -> hdr. Store hdr to cache. Insert cached header in the new FITS if it's not there yet.

You have to implement the _getHeader(srcName) -> pyfits header object function. It must raise an exception if it cannot come up with a header. You also have to implement _isProcessed(srcName) -> boolean returning True if you think srcName already has a processed header.

This basic flow is influenced by the following opts attributes:

- reProcess -- even if a cache is present, recompute header values
- applyHeaders -- actually replace old headers with new headers
- reHeader -- even if _isProcessed returns True, write a new header
- compute -- perform computations

The idea is that you can:

- generate headers without touching the original files: proc
- write all cached headers to files that don't have them proc --apply --nocompute
- after a bugfix force all headers to be regenerated: proc --reprocess --apply --reheader

All this leads to the messy logic. Sorry 'bout this.

Class IgnoreThisRow

can be raised by user code to indicate that a row should be skipped when building a table.

Class ImmediateHeaderProcessor

An base for processors doing simple FITS manipulations to the primary FITS header.

To define these, override `_isProcessed(self, srcName, hdr)` and `_changeHeader(self, hdr)`.

`_changeHeader` can change the pyfits header `hdr` in place. It will then be replaced on the actual file.

For complex operations, it is probably advisable to use `HeaderProcessor` which gives you a two-step process of first having the detached headers that you can check before applying them.

Class IntegrityError

Error related to database integrity.

Constant LIGHT_C

A constant, valued 299792458.0

Class LiteralParseError

is raised if an attribute literal is somehow bad.

LiteralParseErrors are constructed with the name of the attribute that was being parsed, the offending literal, and optionally a parse position and a hint.

Function MS

Signature: `MS(structClass, **kwargs)`

creates a parentless instance of structClass with `**kwargs`.

You can pass in a `parent_` kwarg to force a parent.

This is the preferred way to create struct instances in DaCHS, as it will cause the sequence of completers and validators run. Use it like this:

```
MS(rscdef.Column, name="ra", type="double precision)
```

Class NoMetaKey

is raised when a meta key does not exist (and raiseOnFail is True).

Class NotFoundError

is raised when something is asked for something that does not exist.

lookedFor can be an arbitrary object, so be careful when your repr it -- that may be long.

OutputTableDef

A table that has outputFields for columns.

Cores always have one of these, but they are implicitly defined by the underlying database tables in case of dbCores and such.

Services may define output tables to modify what is coming back fromt the core. Note that this usually only affects the output to web browsers. To use the output table also through VO protocols (and when producing VOTables, FITS files, and the like), you need to set the service's votableRespectsOutputTable property to True.

Constant PLANCK_H

A constant, valued 6.62607004e-34

Class PlainUI

An Observer spitting out most info to the screen.

This is to configure the UI. Enable it by calling `api.PlainUI(api.ui)`.

Class PreviewMaker

A file processor for generating previews.

For these, define a method getPreviewData(accref) -> string returning the raw preview data.

RD

A resource descriptor.

RDs collect all information about how to parse a particular source (like a collection of FITS images, a catalogue, or whatever), about the database tables the data ends up in, and the services used to access them.

In DaCHS' RD XML serialisation, they correspond to the root element.

Class RDNotFound

is raised when an RD cannot be located.

Class ReportableError

is raised when something decides it can come up with an error message that should be presented to the user as-is.

UIs should, consequently, just dump the payload and not try adornments. The content should be treated as a unicode string.

Class SeeOther

raised to redirect a user agent to a different resource (HTTP 303).

SeeOthers are constructed with the destination URL that can be relative (to webRoot) or absolute (starting with http).

They are essentially like WebRedirect, except they put out a 303 instead of a 301.

Class SourceParseError

is raised when some syntax error occurs during a source parse.

They are constructed with the offending input construct (a source line or similar, None in a pinch) and the result of the row iterator's getLocator call.

Class StingyPlainUI

An Observer swallowing infos, warnings, and the like.

This is to configure the UI. Enable it by calling `api.StingyPlainUI(api.ui)`.

Class StructureError

is raised if an error occurs during the construction of structures.

You can construct these with pos; this is an opaque object that, when stringified, should expand to something that gives the user a rough idea of where something went wrong.

Since you will usually not know where you are in the source document when you want to raise a StructureError, xmlstruct will try to fill pos in when it's still None when it sees a StructureError. Thus, you're probably well advised to leave it blank.

Function TAItoTT

Signature: `TAItoTT(tai)`

returns TDT for a (datetime.datetime) TAI.

Function TTtoTAI

Signature: `TTtoTAI(tdt)`

returns TAI for a (datetime.datetime) TDT.

TableDef

A definition of a table, both on-disk and internal.

Some attributes are ignored for in-memory tables, e.g., roles or adql.

Properties for tables:

- supportsModel -- a short name of a data model supported through this table (for TAPRegExt dataModel); you can give multiple names separated by commas.
- supportsModelURI -- a URI of a data model supported through this table. You can give multiple URIs separated by blanks.

If you give multiple data model names or URIs, the sequences of names and URIs must be identical (in particular, each name needs a URI).

Function TableForDef

Signature: `TableForDef(tableDef, suppressIndex=False, parseOptions=<ParseOptions validateRows=False maxRows=None keepGoing=False>, **kwargs)`

returns a table instance suitable for holding data described by tableDef.

This is the main interface to table instancation.

suppressIndex=True can be used to suppress index generation on in-memory tables with primary keys. Use it when you are sure you will not need the index (e.g., if staging an on-disk table).

See the function getParseOptions for what you can pass in as `parseOptions`; arguments there can also be used here.

Class UnknownURI

raised to generate an HTTP 404 response.

Class UnmanagedQuerier

A simple interface to querying the database through a connection managed by someone else.

This is typically used as in:

```
with base.getTableConn() as conn:
  q = UnmanagedQuerier(conn)
  ...
```

This contains numerous methods abstracting DB functionality a bit. Documented ones include:

- schemaExissts(schema)
- getColumnsFromDB(tableName)
- getTableType(tableName) -- this will return None for non-existing tables, which is DaCHS' official way to determine table existence.
- getTimeout() -- returns the current query timeout in seconds
- setTimeout(timeout) -- sets a timeout in seconds.

Class VOTableContext

A context object for writing VOTables.

The constructor arguments work as keyword arguments to `getAsVOTable`. Some other high-level functions accept finished contexts.

This class provides management for unique ID attributes, the value mapper registry, and possibly additional services for writing VOTables.

VOTableContexts optionally take

- a value mapper registry (by default, valuemappers.defaultMFRegistry)
- the tablecoding (currently, td, binary, or binary2)
- version=(1,1) to order a 1.1-version VOTable, (1,2) for 1.2. (default is now 1.3).
- acquireSamples=False to suppress reading some rows to get samples for each column
- suppressNamespace=False to leave out a namespace declaration (mostly convenient for debugging)
- overflowElement (see votable.tablewriter.OverflowElement)

There's also an attribute produceVODML that will automatically be set for VOTable 1.4; you can set it to true manually, but the resulting VOTables will probably be invalid.

If VO-DML processing is enabled, the context also manages models declared; that's the modelsUsed dictionary, mapping prefix -> dm.Model instances

Class VOTableError

The base class of VOTable-related errors.

Class ValidationError

is raised when the validation of a field fails.

ValidationErrors are constructed with a message, a column name, and optionally a row (i.e., a dict) and a hint.

Class WebRedirect

raised to redirect a user agent to a different resource (HTTP 301).

WebRedirectes are constructed with the destination URL that can be relative (to webRoot) or absolute (starting with http).

Function bYearToDateTime

Signature: `bYearToDateTime(bYear)`

returns a datetime.datetime instance for a fractional Besselian year.

This uses the formula given by Lieske, J.H., A&A 73, 282 (1979).

Function computeMean

Signature: `computeMean(val1, val2)`

returns the mean value between two values.

Beware: Integer division done here for the benefit of datetime calculations.

```
>>> computeMean(1.,3)
2.0
>>> computeMean(datetime.datetime(2000, 10, 13),
...   datetime.datetime(2000, 10, 12))
datetime.datetime(2000, 10, 12, 12, 0)
```

Function createDump

Signature: `createDump(tableIds, destFile)`

writes a DaCHS dump of tableIds to destFile.

tableIds is a list of rd-id#table-id identifiers (all must resolve), destFile is a file object opened for writing.

Function dateTimeToJYear

Signature: `dateTimeToJYear(dt)`

returns a fractional (julian) year for a datetime.datetime instance.

Function dateTimeToJdn

Signature: `dateTimeToJdn(dt)`

returns a julian day number (including fractionals) from a datetime instance.

Function dateTimeToMJD

Signature: `dateTimeToMJD(dt)`

returns a modified julian date for a datetime instance.

Function dmsToDeg

Signature: `dmsToDeg(dmsAngle, sepChar=None)`

returns the degree minutes seconds-specified dmsAngle as a float in degrees.

```
>>> "%3.8f"%dmsToDeg("45 30.6")
'45.51000000'
>>> "%3.8f"%dmsToDeg("45:30.6", ":")
'45.51000000'
>>> "%3.8f"%dmsToDeg("-45 30 7.6")
'-45.50211111'
>>> dmsToDeg("junk")
Traceback (most recent call last):
ValueError: Invalid dms value with sepChar None: 'junk'
```

Function formatData

Signature: `formatData(formatName, table, outputFile, acquireSamples=True, **moreFormatterArgs)`

writes a table to outputFile in the format given by key.

Table may be a table or a `Data` instance. `formatName` is a format shortcut (`formats.iterFormats()` gives keys available) or a media type. If you pass None, the default VOTable format will be selected.

This raises a `CannotSerializeIn` exception if `formatName` is not recognized. Note that you have to import the serialising modules from the format package to make the formats available (fitstable, csvtable, geojson, jsontable, texttable, votable; api itself already imports the more popular of these).

If a client knows a certain formatter understands additional arguments, it can hand them in as keywords arguments. This will raise an error if another formatter that doesn't understand the argument is being used.

Function formatISODT

Signature: `formatISODT(dt)`

returns some ISO8601 representation of a datetime instance.

The reason for preferring this function over a simple str is that datetime's default representation is too difficult for some other code (e.g., itself); hence, this code suppresses any microsecond part and always adds a Z (where strftime works, utils.isoTimestampFmt produces an identical string).

The behaviour of this function for timezone-aware datetimes is undefined.

For convenience, None is returned as None

```
>>> formatISODT(datetime.datetime(2015, 10, 20, 12, 34, 22, 250))
'2015-10-20T12:34:22Z'
>>> formatISODT(datetime.datetime(1815, 10, 20, 12, 34, 22, 250))
'1815-10-20T12:34:22Z'
```

Function genLimitKeys

Signature: `genLimitKeys(inputKey)`

yields _MAX and _MIN inputKeys from a single input key.

This also tries to sensibly fix descriptions and ucds. This is mainly for datalink metaMakers; condDescs may use a similar thing, but that's not exposed to RDs.

Don't use this function any more. It will go away soon.

Function getAccrefFromStandardPubDID

Signature: `getAccrefFromStandardPubDID(pubdid, authBase=u'ivo://org.gavo.dc/~?')`

returns an accref from a standard DaCHS PubDID.

This is basically the inverse of getStandardPubDID. It will raise ValueErrors if pubdid doesn't start with ivo://<authority>/~?.

The function does not check if the remaining characters are a valid accref, much less whether it can be resolved.

authBase's default will reflect you system's settings on your installation, which probably is not what's given in this documentation.

Function getAsVOTable

Signature: `getAsVOTable(data, ctx=None, **kwargs)`

returns a string containing a VOTable representation of data.

`kwargs` can be constructor arguments for VOTableContext.

Function getDBConnection

Signature: `getDBConnection(profile, debug=False, autocommitted=False)`

returns an enhanced database connection through profile.

You will typically rather use the context managers for the standard profiles (`getTableConnection` and friends). Use this function if you want to keep your connection out of connection pools or if you want to use non-standard profiles.

profile will usually be a string naming a profile defined in `GAVO_ROOT/etc`.

Function getDatalinkMetaLink

Signature: `getDatalinkMetaLink(dlSvc, accref)`

returns a datalink URL for the product referenced through accref with the datalink service dlSvc.

This assumes that dlSvc uses the standard DaCHS pubDIDs. dlSvc needs to be the service element.

A typical use is in a metaMaker and would look like this:

```
getDatalinkMetaLink(rd.getById("dl"), descriptor.accref)
```

Function getFileStem

Signature: `getFileStem(fPath)`

returns the file stem of a file path.

The base name is what remains if you take the base name and split off extensions. The extension here starts with the last dot in the file name, except up to one of some common compression extensions (.gz, .xz, .bz2, .Z, .z) is stripped off the end if present before determining the extension.

```
>>> getFileStem("/foo/bar/baz.x.y")
'baz.x'
>>> getFileStem("/foo/bar/baz.x.gz")
'baz'
>>> getFileStem("/foo/bar/baz")
'baz'
```

Function getFlatName

Signature: `getFlatName(accref)`

returns a unix-compatible file name for an access reference.

The file name will not contain terrible characters, let alone slashes. This is used to, e.g., keep all previews in one directory.

Function getFormatted

Signature: `getFormatted(formatName, table, acquireSamples=False)`

returns a string containing a representation of table in the format given by formatName.

This is just wrapping the function formatData; se there for formatName. This function will use large amounts of memory for large data.

Function getInputsRelativePath

Signature: `getInputsRelativePath(absPath, liberalChars=True)`

returns absath relative to the DaCHS inputsDir.

If `absPath` is not below `inputsDir`, a `ValueError` results. On `liberalChars`, wee see the function getRelativePath.

In rowmakers and rowfilters, you'll usually use the macro `\inputRelativePath` that inserts the appropriate code.

Function getMetaText

Signature: `getMetaText(ob, key, default=None, **kwargs)`

returns the meta item key form ob in text form if present, default otherwise.

You can pass getMeta keyword arguments (except default).

Additionally, there's acceptSequence; if set to true, this will return the first item of a sequence-valued meta item rather than raising an error.

ob will be used as a macro package if it has an expand method; to use something else as the macro package, pass a macroPackage keyword argument.

Function getParseOptions

Signature: `getParseOptions(validateRows=True, doTableUpdates=False, batchSize=1024, maxRows=None, keepGoing=False, dropIndices=False, dumpRows=False, metaOnly=False, buildDependencies=True, systemImport=False, commitAfterMeta=False, dumpIngestees=False)`

returns an object with some attributes set.

This object is used in the parsing code in dddef. It's a standin for the the command line options for tables created internally and should have all attributes that the parsing infrastructure might want from the optparse object.

So, just configure what you want via keyword arguments or use the prebuilt objects parseValidating and and parseNonValidating below.

See commandline.py for the meaning of the attributes.

The exception is buildDependencies. This is true for most internal builds of data (and thus here), but false when we need to manually control when dependencies are built, as in user.importing and while building the dependencies themselves.

Function getQueryMeta

Signature: `getQueryMeta()`

returns a query meta object from somewhere up the stack.

This is for row makers running within a service. This can be used to, e.g., enforce match limits by writing getQueryMeta()["dbLimit"].

Function getReferencedElement

Signature: `getReferencedElement(refString, forceType=None, **kwargs)`

returns the element for the DaCHS reference `refString`.

`refString` has the form `rdId[#subRef]`; `rdId` can be filesystem-relative, but the RD referenced must be below `inputsDir` anyway.

You can pass a structure class into `forceType`, and a `StructureError` will be raised if what's pointed to by the id isn't of that type.

You should usually use `base.resolveCrossId` instead of this from *within* DaCHS. This is intended for code handling RD ids from users.

This supports further keyword arguments to getRD.

Function getRelativePath

Signature: `getRelativePath(fullPath, rootPath, liberalChars=True)`

returns rest if fullPath has the form rootPath/rest and raises an exception otherwise.

Pass `liberalChars=False` to make this raise a ValueError when URL-dangerous characters (blanks, amperands, pluses, non-ASCII, and similar) are present in the result. This is mainly for products.

Function getStandardPubDID

Signature: `getStandardPubDID(path)`

returns the standard DaCHS PubDID for `path`.

The publisher dataset identifier (PubDID) is important in protocols like SSAP and obscore. If you use this function, the PubDID will be your authority, the path compontent ~, and the inputs-relative path of the input file as the parameter.

`path` can be relative, in which case it is interpreted relative to the DaCHS `inputsDir`.

You *can* define your PubDIDs in a different way, but you'd then need to provide a custom descriptorGenerator to datalink services (and might need other tricks). If your data comes from plain files, use this function.

In a rowmaker, you'll usually use the standardPubDID macro.

Function getTableDefForTable

Signature: `getTableDefForTable(connection, tableName)`

returns a TableDef object for a SQL table name.

connection needs to be TableConnection of higher.

This really has little to do with resolving identifiers, but this module already has getRDs and similar, so it seemed the least unnatural place.

Function getWCSAxis

Signature: `getWCSAxis(header, axisIndex, forceSeparable=False)`

returns a WCSAxis instance from an axis index and a FITS header.

If the axis is mentioned in a transformation matrix (CD or PC), a `ValueError` is raised (use `forceSeparable` to override).

The `axisIndex` is 1-based; to get a transform for the axis described by CTYPE1, pass 1 here.

The object returned has methods like `pixToPhys`, `physToPix` (and their `pix0` brethren), and `getLimits`.

Note that at this point WCSAxis only supports linear transforms (it's a DaCHS-specific implementation). We'll extend it on request.

Function getXMLTree

Signature: `getXMLTree(xmlString, debug=False)`

returns an `libxml2` etree for `xmlString`, where, for convenience, all namespaces on elements are nuked.

The libxml2 etree lets you do xpath searching using the `xpath` method.

Nuking namespaces is of course not a good idea in general, so you might want to think again before you use this in production code.

Function hmsToDeg

Signature: `hmsToDeg(hms, sepChar=None)`

returns the time angle (h m s.decimals) as a float in degrees.

```
>>> "%3.8f"%hmsToDeg("22 23 23.3")
'335.84708333'
>>> "%3.8f"%hmsToDeg("22:23:23.3", ":")
'335.84708333'
>>> "%3.8f"%hmsToDeg("222323.3", "")
'335.84708333'
>>> hmsToDeg("junk")
Traceback (most recent call last):
ValueError: Invalid time with sepChar None: 'junk'
```

Function iterSimpleText

Signature: `iterSimpleText(f)`

iterates over `(physLineNumber, line)` in f with some usual conventions for simple data files.

You should use this function to read from simple configuration and/or table files that don't warrant a full-blown grammar/rowmaker combo. The intended use is somewhat like this:

```
with open(rd.getAbsPath("res/mymeta")) as f:
  for lineNumber, content in iterSimpleText(f):
    try:
      ...
    except Exception, exc:
      sys.stderr.write("Bad input line %s: %s"%(lineNumber, exc))
```

The grammar rules are, specifically:

- leading and trailing whitespace is stripped
- empty lines are ignored
- lines beginning with a hash are ignored
- lines ending with a backslash are joined with the following line; to have intervening whitespace, have a blank in front of the backslash.

Function jYearToDateTime

Signature: `jYearToDateTime(jYear)`

returns a datetime.datetime instance for a fractional (julian) year.

This refers to time specifications like J2001.32.

Function jdnToDateTime

Signature: `jdnToDateTime(jd)`

returns a `datetime.datetime` instance for a julian day number.

Function killBlanks

Signature: `killBlanks(literal)`

returns the string literal with all blanks removed.

This is useful when numbers are formatted with blanks thrown in.

Nones are passed through.

Function lastSourceElements

Signature: `lastSourceElements(path, numElements)`

returns a path made up from the last `numElements` items in `path`.

Function loadPythonModule

Signature: `loadPythonModule(fqName, relativeTo=None)`

imports fqName and returns the module with a module description.

The module description is what what find_module returns; you may need this for reloading and similar.

Do not use this function to import DC-internal modules; this may mess up singletons since you could bypass python's mechanisms to prevent multiple imports of the same module.

fqName is a fully qualified path to the module without the .py, unless relativeTo is given, in which case it is interpreted as a relative path. This for letting modules in resdir/res import each other by saying:

```
mod, _ = api.loadPythonModule("foo", relativeTo=__file__)
```

The python path is temporarily amended with the path part of the source module.

If the module is in /var/gavo/inputs/foo/bar/mod.py, Python will know the module as foo_bar_mod (the last two path components are always added). This is to keep Python from using the module when someone writes import mod.

Function makeAbsoluteURL

Signature: `makeAbsoluteURL(path, canonical=False)`

returns a fully qualified URL for a rooted local part.

This will reflect the http/https access mode unless you pass canonical=True, in which case [web]serverURL will be used unconditionally.

Function makeData

Signature: `makeData(dd, parseOptions=<ParseOptions validateRows=False maxRows=None keepGoing=False>, forceSource=None, connection=None, data=None, runCommit=True)`

returns a data instance built from `dd`.

It will arrange for the parsing of all tables generated from dd's grammar.

If database tables are being made, you *must* pass in a connection. The entire operation will then run within a single transaction within this connection (except for building dependents; they will be built in separate transactions).

The connection will be rolled back or committed depending on the success of the operation (unless you pass `runCommit=False`, in which case even a successful import will not be committed)..

You can pass in a data instance created by yourself in data. This makes sense if you want to, e.g., add some meta information up front.

Function makeDependentsFor

Signature: `makeDependentsFor(dds, parseOptions, connection)`

rebuilds all data dependent on one of the DDs in the dds sequence.

Function makeProductLink

Signature: `makeProductLink(key, withHost=True, useHost=None)`

returns the URL at which a product can be retrieved.

key can be an accref string or an RAccref

Function makeSitePath

Signature: `makeSitePath(path)`

returns a rooted local part for a server-internal URL.

uri itself needs to be server-absolute; a leading slash is recommended for clarity but not mandatory.

Function makeStruct

Signature: `makeStruct(structClass, **kwargs)`

creates a parentless instance of structClass with `**kwargs`.

You can pass in a `parent_` kwarg to force a parent.

This is the preferred way to create struct instances in DaCHS, as it will cause the sequence of completers and validators run. Use it like this:

```
MS(rscdef.Column, name="ra", type="double precision)
```

Function makeTimestamp

Signature: `makeTimestamp(date, time)`

makes a datetime instance from a date and a time.

Function mjdToDateTime

Signature: `mjdToDateTime(mjd)`

returns a `datetime.datetime` instance for a modified julian day number.

Beware: This loses a couple of significant digits due to transformation to jd.

Function parseAngle

Signature: `parseAngle(literal, format, sepChar=None)`

converts the various forms angles might be encountered to degrees.

format is one of hms, dms, fracHour. For sexagesimal/time angles, you can pass a sepChar (default: split at blanks) that lets you specify what separates hours/degrees, minutes, and seconds.

```
>>> str(parseAngle("23 59 59.95", "hms"))
'359.999791667'
>>> "%10.5f"%parseAngle("-20:31:05.12", "dms", sepChar=":")
' -20.51809'
>>> "%010.6f"%parseAngle("21.0209556", "fracHour")
'315.314334'
```

Function parseBooleanLiteral

Signature: `parseBooleanLiteral(literal)`

returns a python boolean from some string.

Boolean literals are strings like True, false, on, Off, yes, No in some capitalization.

Function parseCooPair

Signature: `parseCooPair(soup)`

returns a pair of RA, DEC floats if they can be made out in soup or raises a value error.

No range checking is done (yet), i.e., as long as two numbers can be made out, the function is happy.

```
>>> parseCooPair("23 12")
(23.0, 12.0)
>>> parseCooPair("23.5,-12.25")
(23.5, -12.25)
>>> parseCooPair("3.75 -12.125")
(3.75, -12.125)
>>> parseCooPair("3 25,-12 30")
(51.25, -12.5)
>>> map(str, parseCooPair("12 15 30.5 +52 18 27.5"))
['183.877083333', '52.3076388889']
>>> parseCooPair("3.39 -12 39")
Traceback (most recent call last):
ValueError: Invalid time with sepChar None: '3.39'
>>> parseCooPair("12 15 30.5 +52 18 27.5e")
Traceback (most recent call last):
ValueError: 12 15 30.5 +52 18 27.5e has no discernible position in it
>>> parseCooPair("QSO2230+44.3")
Traceback (most recent call last):
ValueError: QSO2230+44.3 has no discernible position in it
```

Function parseDate

Signature: `parseDate(literal, format='%Y-%m-%d')`

returns a `datetime.date` object of literal parsed according to the strptime-similar format.

The function understands the special `dateFormat !!jYear` (stuff like 1980.89).

Function parseFloat

Signature: `parseFloat(literal)`

returns a float from a literal, or None if literal is None or an empty string.

Temporarily, this includes a hack to work around a bug in psycopg2.

```
>>> parseFloat("   5e9 ")
5000000000.0
>>> parseFloat(None)
>>> parseFloat("  ")
>>> parseFloat("wobbadobba")
Traceback (most recent call last):
ValueError: could not convert string to float: wobbadobba
```

Function parseFromString

Signature: `parseFromString(rootStruct, inputString, context=None)`

parses a DaCHS RD tree rooted in `rootStruct` from a string.

It returns the root element of the resulting tree. You would use this like this:

```
parseFromString(rscdef.Column, "<column name='foo'/>")
```

Function parseISODT

Signature: `parseISODT(literal)`

returns a datetime object for a ISO time literal.

There's no real timezone support yet, but we accept and ignore various ways of specifying UTC.

```
>>> parseISODT("1998-12-14")
datetime.datetime(1998, 12, 14, 0, 0)
>>> parseISODT("1998-12-14T13:30:12")
datetime.datetime(1998, 12, 14, 13, 30, 12)
>>> parseISODT("1998-12-14T13:30:12Z")
datetime.datetime(1998, 12, 14, 13, 30, 12)
>>> parseISODT("1998-12-14T13:30:12.224Z")
datetime.datetime(1998, 12, 14, 13, 30, 12, 224000)
>>> parseISODT("19981214T133012Z")
datetime.datetime(1998, 12, 14, 13, 30, 12)
>>> parseISODT("19981214T133012+00:00")
datetime.datetime(1998, 12, 14, 13, 30, 12)
>>> parseISODT("junk")
Traceback (most recent call last):
ValueError: Bad ISO datetime literal: junk (required format: yyyy-mm-ddThh:mm:ssZ)
```

Function parseInt

Signature: `parseInt(literal)`

returns an int from a literal, or None if literal is None or an empty string.

```
>>> parseInt("32")
32
>>> parseInt("")
>>> parseInt(None)
```

parseNonValidating

see function getParseOptions .

Function parseSPoint

Signature: `parseSPoint(soup)`

returns an `SPoint` for a coordinate pair.

The coordinate pair can be formatted in a variety of ways; see the function parseCooPair. Input is always in degrees.

Function parseTime

Signature: `parseTime(literal, format='%H:%M:%S')`

returns a `datetime.timedelta` object for literal parsed according to format.

For format, you can the magic values `!!secondsSinceMidnight`, `!!decimalHours` or a strptime-like spec using the H, M, and S codes.

```
>>> parseTime("89930", "!!secondsSinceMidnight")
datetime.timedelta(1, 3530)
>>> parseTime("23.4", "!!decimalHours")
datetime.timedelta(0, 84240)
>>> parseTime("3.4:5", "%H.%M:%S")
datetime.timedelta(0, 11045)
>>> parseTime("20:04", "%H:%M")
datetime.timedelta(0, 72240)
```

Function parseTimestamp

Signature: `parseTimestamp(literal, format='%Y-%m-%dT%H:%M:%S')`

returns a `datetime.datetime` object from a literal parsed according to the strptime-similar format.

A `ValueError` is raised if literal doesn't match format (actually, a parse with essentially DALI-standard ISO representation is always tried)

parseValidating

see function getParseOptions .

Function parseWithNull

Signature: `parseWithNull(literal, baseParser, nullLiteral=<Undefined>, default=None, checker=None)`

returns default if literal is `nullLiteral`, else `baseParser(literal)`.

If `checker` is non-None, it must be a callable returning `True` if its argument is a null value.

`nullLiteral` is compared against the unprocessed literal (usually, a string). The intended use is like this (but note that often, a `nullExcs` attribute on a rowmaker `map` element is the more elegant way:

```
>>> parseWithNull("8888.0", float, "8888")
8888.0
>>> print(parseWithNull("8888", float, "8888"))
None
>>> print(parseWithNull("N/A", int, "N/A"))
None
```

Function procmain

Signature: `procmain(processorClass, rdId, ddId)`

The "standard" main function for processor scripts.

The function returns the instanciated processor so you can communicate from your processor back to your own main.

See processors.html for details.

Function quoteProductKey

Signature: `quoteProductKey(key)`

returns key as getproduct URL-part.

If `key` is a string, it is quoted as a naked accref so it's usable as the path part of an URL. If it's an `RAccref`, it is just stringified. The result is something that can be used after getproduct in URLs in any case.

Function requireValue

Signature: `requireValue(val, fieldName)`

returns `val` unless it is `None`, in which case a `ValidationError` for `fieldName` will be raised.

Function resolveCrossId

Signature: `resolveCrossId(id, forceType=None, **kwargs)`

resolves `id`, where id is of the form `rdId#id`.

`forceType`, if non-None must be a DaCHS struct type (e.g., rscdef.Table); a `StructureError` will be raised if the reference resolves to something else than an instance of that type.

`id` can also be a simple rd id.

`kwargs` lets you pass additional keyword arguments to the `getRD` calls that may be triggered by this.

Function restoreDump

Signature: `restoreDump(dumpFile)`

restores a dump.

dumpFile is an open file object containing a file crated by createDump.

This comprises recrating all mentioned tables, copying in the associated data, and re-creating all indices.

Each table is handled in a separate transaction, we do not stop if a single restore has failed.

Function scale

Signature: `scale(val, factor, offset=0)`

returns val*factor+offset if val is not None, None otherwise.

This is when you want to manipulate a numeric value that may be NULL. It is a somewhat safer alternative to using nullExcs with scaled values.

setConfig

sets a configuration item to a value.

arg1 can be a section, in which case arg2 is a key and arg3 is a value; alternatively, if arg3 is not given, arg1 is a key in the defaultSection, and arg2 is the value.

All arguments are strings that must be parseable by the referenced item's _parse method.

Origin is a tag you can use to, e.g., determine what to save.

Function toMJD

Signature: `toMJD(literal)`

returns a modified julian date made from some datetime representation.

Valid representations include:

- MJD (a float smaller than 1e6)
- JD (a float larger than 1e6)
- datetime.datetime instances
- ISO time strings.

ui

is the central event dispatcher.

Events are posted by using notify* methods. Various handlers can then attach to them.

Function writeAsVOTable

Signature: `writeAsVOTable(data, outputFile, ctx=None, **kwargs)`

writes `data` to the `outputFile`.

data can be a table or `Data` item.

`ctx` can be a `VOTableContext` instance; alternatively, `VOTableContext` constructor arguments can be passed in as `kwargs`.

System Tables

DaCHS uses a number of tables to manage services and implement protocols. Operators should not normally be concerned with them, but sometimes having a glimpse into them helps with debugging.

If you find yourself wanting to change these tables' content, please post to dachs-support first describing what you're trying to do. There should really be commands that do what you want, and it's relatively easy to introduce subtle problems by manipulating system tables without going through those.

Having said that, here's a list of the system tables together with brief descriptions of their role and the columns contained. Note that your installation might not have all of those; some only appear after a `gavo imp` of the RD they are defined in -- which you of course only should do if you know you want to enable the functionality provided.

The documentation given here is extracted from the resource descriptors, which, again, you can read in source using `gavo admin dumpDF //<rd-name>`.

dc.authors

Defined in //services

A table that contains the (slightly processed) creator.name metadata from published services. It is used by the shipped templates of the root pages.

Manipulate through gavo pub; to remove entries from this table, remove the publication element of the service or table in question and re-run gavo pub on the resource descriptor.

sourceRD (text) -- Id of the RD (essentially, the inputsDir-relative path, with the .rd cut off).

resId (text) -- Id of the service, data or table within the RD. Together with the RD id, this uniquely identifies the resource to DaCHS.

author (unicode) -- An author name taken from creator.name; DaCHS assumes this to be in the form Last, I.

dc.datalinkjobs

Defined in //datalink

A table managing datalink jobs submitted asynchronously (the dlasync renderer)

jobId (text) -- Internal id of the job. At the same time, uwsDir-relative name of the job directory.

phase (text) -- The state of the job.

executionDuration (integer) -- Job time limit

destructionTime (timestamp) -- Time at which the job, including ancillary data, will be deleted

owner (text) -- Submitter of the job, if verified

parameters (text) -- Pickled representation of the parameters (except uploads)

runId (text) -- User-chosen run Id

startTime (timestamp) -- UTC job execution started

endTime (timestamp) -- UTC job execution finished

error (text) -- some suitable representation an error that has occurred while executing the job (null means no error information has been logged)

creationTime (timestamp) -- UTC job was created

pid (integer) -- A unix pid to kill to make the job stop

dc.groups

Defined in //users

Assignment of users to groups.

Conceptually, each user has an associated group of the same name. A user always is a member of her group. Other users can be added to that group, essentially as in the classic Unix model.

Manipulate this table through gavo admin addtogroup and gavo admin delfromgroup.

username (text) -- Name of the user belonging to the group

groupname (text) -- Name of the group

dc.interfaces

Defined in //services

A table that has "interfaces", i.e., actual URLs under which services are accessible. This is in a separate table, as services can have multiple interfaces (e.g., SCS and form).

Manipulate through gavo pub; to remove entries from this table, remove the publication element of the service or table in question and re-run gavo pub on the resource descriptor.

sourceRD (text) -- Id of the RD (essentially, the inputsDir-relative path, with the .rd cut off).

resId (text) -- Id of the service, data or table within the RD. Together with the RD id, this uniquely identifies the resource to DaCHS.

accessURL (text) -- The URL this service with the given renderer can be accessed under.

referenceURL (text) -- The URL this interface is explained at. In DaCHS, as in VOResource, this column should actually be in dc.resources, but we don't consider that wart bad enough to risk any breakage.

browseable (boolean) -- True if this interface can sensibly be operated with a web browser (e.g., form, but not scs.xml; browseable service interfaces are eligible for being put below the 'Use this service with your browser' button on the service info page.

renderer (text) -- The renderer used for this interface.

dc.metastore

Defined in //dc_tables

A table for storing all kinds of key-value pairs. Key starting with an underscore are for use by user RDs.

Only one pair per key is supported, newer keys overwrite older ones.

Currently, this is only used for schemaversion, the version of the DaCHS system tables as used by gavo upgrade to figure out what to change. gavo upgrade manages this.

From your code, you can use base.getDBMeta(key) and base.setDBMeta(connection, key, value) to put persistent, string-valued metadata in here; if you use this, would you tell us your use case?

"key" (text) -- A key; everything that starts with an underscore is user defined.

"value" (text) -- A value; no serialization format is defined here, but you are encouraged to use python literals for non-strings.

dc.products

Defined in //products

The products table keeps information on "products", i.e. datasets delivered to the users.

It is normally fed through the products#define rowfilter and a mixin like products#table (or other mixins using it like siap#pgs or ssap#mixc).

/getproducts inspects this table before handing out data to enforce embargoes and similar restrictions, and this is also where it figures out where to go for previews.

accref (text) -- Access key for the data

owner (text) -- Owner of the data

embargo (date) -- Date the data will become/became public

mime (text) -- MIME type of the file served

accessPath (text) -- Inputs-relative filesystem path to the file

sourceTable (text) -- Name of table containing metadata

preview (text) -- Location of a preview; this can be NULL if no preview is available, 'AUTO' if DaCHS is supposed to try and make its own previews based on MIME guessing, or a file name, or an URL.

datalink (text) -- A fully qualified URL of a datalink document for this dataset. This is to allow the global datalink service (sitting on the ~ resource and used by obscore) to forward datalink requests globally.

preview_mime (text) -- MIME type of a preview (if any)

dc.res_dependencies

Defined in //services

An RD-level map of dependencies, meaning that before generating resource records from rd, prereq should be imported (think: TAP needs the metadata of all dependent tables).

This is managed by gavo pub and used in the OAI-PMH interface.

rd (text) -- id of an RD

prereq (text) -- id of an RD that should be imported before records from rd are generated.

sourceRD (text) -- id of the RD that introduced this dependency

dc.resources

Defined in //services

The table of published "resources" (i.e., services, tables, data collections) within this data center. There are separate tables of the interfaces these resources have, their authors, subjects, and the sets they belong to.

Manipulate through gavo pub; to remove entries from this table, remove the publication element of the service or table in question and re-run gavo pub on the resource descriptor.

sourceRD (text) -- Id of the RD (essentially, the inputsDir-relative path, with the .rd cut off).

resId (text) -- Id of the service, data or table within the RD. Together with the RD id, this uniquely identifies the resource to DaCHS.

shortName (text) -- The content of the service's shortName metadata. This is not currently used by the root pages delivered with DaCHS, so this column essentially is ignored.

title (text) -- The content of the service's title metadata (gavo pub will fall back to the resource's title if the service doesn't have a description of its own).

description (text) -- The content of the service's description metadata (gavo pub will fall back to the resource's description if the service doesn't have a description of its own).

owner (text) -- NULL for public services, otherwise whatever is in limitTo. The root pages delivered with DaCHS put a [P] in front of services with a non-NULL owner.

dateUpdated (timestamp) -- Date of last update on the resource itself (i.e., run of gavo imp).

recTimestamp (timestamp) -- UTC of gavo publish run on the source RD

deleted (boolean) -- True if the service is deleted. On deletion, services are not removed from the resources and sets tables so the OAI-PMH service can notify incremental harvesters that a resource is gone.

ivoid (text) -- The full ivo-id of the resource. This is usually ivo://auth/rdid/frag but may be overridden (you should probably not create records for which you are not authority, but we do not enforce that any more).

authors (text) -- Resource authors in source sequence

dc.resources_join

Defined in //services

A join of resources, interfaces, and sets used internally.

sourceRD (text) -- Id of the RD (essentially, the inputsDir-relative path, with the .rd cut off).

resId (text) -- Id of the service, data or table within the RD. Together with the RD id, this uniquely identifies the resource to DaCHS.

title (text) -- The content of the service's title metadata (gavo pub will fall back to the resource's title if the service doesn't have a description of its own).

description (text) -- The content of the service's description metadata (gavo pub will fall back to the resource's description if the service doesn't have a description of its own).

owner (text) -- NULL for public services, otherwise whatever is in limitTo. The root pages delivered with DaCHS put a [P] in front of services with a non-NULL owner.

dateUpdated (timestamp) -- Date of last update on the resource itself (i.e., run of gavo imp).

recTimestamp (timestamp) -- UTC of gavo publish run on the source RD

deleted (boolean) -- True if the service is deleted. On deletion, services are not removed from the resources and sets tables so the OAI-PMH service can notify incremental harvesters that a resource is gone.

accessURL (text) -- The URL this service with the given renderer can be accessed under.

referenceURL (text) -- The URL this interface is explained at. In DaCHS, as in VOResource, this column should actually be in dc.resources, but we don't consider that wart bad enough to risk any breakage.

browseable (boolean) -- True if this interface can sensibly be operated with a web browser (e.g., form, but not scs.xml; browseable service interfaces are eligible for being put below the 'Use this service with your browser' button on the service info page.

renderer (text) -- The renderer used for this interface.

setName (text) -- Name of an OAI set.

ivoid (text) -- The full ivo-id of the resource. This is usually ivo://auth/rdid/frag but may be overridden (you should probably not create records for which you are not authority, but we do not enforce that any more).

dc.sets

Defined in //services

A table that contains set membership of published resources. For DaCHS, the sets ivo_managed ("publish to the VO") and local ("show on a generated root page" if using one of the shipped root pages) have a special role.

Manipulate through gavo pub; to remove entries from this table, remove the publication element of the service or table in question and re-run gavo pub on the resource descriptor.

sourceRD (text) -- Id of the RD (essentially, the inputsDir-relative path, with the .rd cut off).

resId (text) -- Id of the service, data or table within the RD. Together with the RD id, this uniquely identifies the resource to DaCHS.

setName (text) -- Name of an OAI set.

renderer (text) -- The renderer used for the publication belonging to this set. Typically, protocol renderers (e.g., scs.xml) will be used in VO publications, whereas form and friends might be both in local and ivo_managed

deleted (boolean) -- True if the service is deleted. On deletion, services are not removed from the resources and sets tables so the OAI-PMH service can notify incremental harvesters that a resource is gone.

dc.subjects

Defined in //services

A table that contains the subject metadata for published services. It is used by the shipped templates of the root pages ("...by subject").

Manipulate through gavo pub; to remove entries from this table, remove the publication element of the service or table in question and re-run gavo pub on the resource descriptor.

sourceRD (text) -- Id of the RD (essentially, the inputsDir-relative path, with the .rd cut off).

resId (text) -- Id of the service, data or table within the RD. Together with the RD id, this uniquely identifies the resource to DaCHS.

subject (text) -- A subject heading. Terms should ideally come from the IVOA thesaurus.

dc.subjects_join

Defined in //services

A join of resources, subjects, and sets used internally.

subject (text) -- A subject heading. Terms should ideally come from the IVOA thesaurus.

sourceRD (text) -- Id of the RD (essentially, the inputsDir-relative path, with the .rd cut off).

resId (text) -- Id of the service, data or table within the RD. Together with the RD id, this uniquely identifies the resource to DaCHS.

title (text) -- The content of the service's title metadata (gavo pub will fall back to the resource's title if the service doesn't have a description of its own).

owner (text) -- NULL for public services, otherwise whatever is in limitTo. The root pages delivered with DaCHS put a [P] in front of services with a non-NULL owner.

accessURL (text) -- The URL this service with the given renderer can be accessed under.

referenceURL (text) -- The URL this interface is explained at. In DaCHS, as in VOResource, this column should actually be in dc.resources, but we don't consider that wart bad enough to risk any breakage.

browseable (boolean) -- True if this interface can sensibly be operated with a web browser (e.g., form, but not scs.xml; browseable service interfaces are eligible for being put below the 'Use this service with your browser' button on the service info page.

setName (text) -- Name of an OAI set.

ivoid (text) -- The full ivo-id of the resource. This is usually ivo://auth/rdid/frag but may be overridden (you should probably not create records for which you are not authority, but we do not enforce that any more).

dc.tablemeta

Defined in //dc_tables

A table mapping table names and schemas to the resource descriptors they come from and whether they are open to ADQL queries.

This is used wherever DaCHS needs to go from a database name to the resource description, e.g., when generating tableinfo.

The table is maintained through gavo imp; to force things out of here, there's gavo drop (for RDs; use -f if the RD is gone or meoved away) or gavo purge (for single tables).

tableName (text) -- Fully qualified table name

sourceRD (text) -- Id of the resource descriptor containing the table's definition

tableDesc (text) -- Description of the table content

resDesc (text) -- Description of the resource this table is part of

adql (boolean) -- True if this table may be accessed using ADQL

dc.users

Defined in //users

Users known to the data center, together with their credentials.

Right now, DaCHS only supports user/password. Note that passwords are currently stored in cleartext, so do discourage your users from using valuable passwords here (whether you explain to them that DaCHS so far only provides "mild security" is up to you).

Manipulate this table through gavo admin adduser, gavo admin deluser, and gavo admin listusers.

username (text) -- Name of the user.

password (text) -- Password in clear text.

remarks (text) -- Free text mainly intended to explain what the user is supposed to be/do

ivoa.ObsCore

Defined in //obscore

The IVOA-defined obscore table, containing generic metadata for datasets within this datacenter.

dataproduct_type (text) -- High level scientific classification of the data product, taken from an enumeration

dataproduct_subtype (text) -- Data product specific type

calib_level (smallint) -- Amount of data processing that has been applied to the data

obs_collection (text) -- Name of a data collection (e.g., project name) this data belongs to

obs_id (text) -- Unique identifier for an observation

obs_title (text) -- Free-from title of the data set

obs_publisher_did (text) -- Dataset identifier assigned by the publisher.

obs_creator_did (text) -- Dataset identifier assigned by the creator.

access_url (text) -- The URL at which to obtain the data set.

access_format (text) -- MIME type of the resource at access_url

access_estsize (bigint) -- Estimated size of data product

target_name (text) -- Object a targeted observation targeted

target_class (text) -- Class of the target object (star, QSO, ...)

s_ra (double precision) -- RA of (center of) observation, ICRS

s_dec (double precision) -- Dec of (center of) observation, ICRS

s_fov (double precision) -- Approximate spatial extent for the region covered by the observation

s_region (spoly) -- Region covered by the observation, as a polygon

s_resolution (double precision) -- Best spatial resolution within the data set

t_min (double precision) -- Lower bound of times represented in the data set, as MJD

t_max (double precision) -- Upper bound of times represented in the data set, as MJD

t_exptime (real) -- Total exporure time

t_resolution (real) -- Minimal significant time interval along the time axis

em_min (double precision) -- Minimal wavelength represented within the data set

em_max (double precision) -- Maximal wavelength represented within the data set

em_res_power (double precision) -- Spectral resolving power delta lambda/lamda

o_ucd (text) -- UCD for the product's observable

pol_states (text) -- List of polarization states in the data set

facility_name (text) -- Name of the facility at which data was taken

instrument_name (text) -- Name of the instrument that produced the data

s_xel1 (bigint) -- Number of elements (typically pixels) along the first spatial axis.

s_xel2 (bigint) -- Number of elements (typically pixels) along the second spatial axis.

t_xel (bigint) -- Number of elements (typically pixels) along the time axis.

em_xel (bigint) -- Number of elements (typically pixels) along the spectral axis.

pol_xel (bigint) -- Number of elements (typically pixels) along the polarization axis.

s_pixel_scale (double precision) -- Sampling period in world coordinate units along the spatial axis

em_ucd (text) -- Nature of the product's spectral axis

ivoa._obscoresources

Defined in //obscore

This table contains the SQL fragments that make up this installation's ivoa.obscore view. Whenever a participating table is re-made, the view definition is renewed with a statement made up of a union of all sqlFragments present in this table.

Manipulate this table through gavo imp on tables that have an obscore mixin, or by dropping RDs or purging tables that are part of obscore.

tableName (text) --

sqlFragment (text) --

ivoa.emptyobscore

Defined in //obscore

An empty table having all columns of the obscore table. Useful internally, and sometimes for tricky queries.

dataproduct_type (text) -- High level scientific classification of the data product, taken from an enumeration

dataproduct_subtype (text) -- Data product specific type

calib_level (smallint) -- Amount of data processing that has been applied to the data

obs_collection (text) -- Name of a data collection (e.g., project name) this data belongs to

obs_id (text) -- Unique identifier for an observation

obs_title (text) -- Free-from title of the data set

obs_publisher_did (text) -- Dataset identifier assigned by the publisher.

obs_creator_did (text) -- Dataset identifier assigned by the creator.

access_url (text) -- The URL at which to obtain the data set.

access_format (text) -- MIME type of the resource at access_url

access_estsize (bigint) -- Estimated size of data product

target_name (text) -- Object a targeted observation targeted

target_class (text) -- Class of the target object (star, QSO, ...)

s_ra (double precision) -- RA of (center of) observation, ICRS

s_dec (double precision) -- Dec of (center of) observation, ICRS

s_fov (double precision) -- Approximate spatial extent for the region covered by the observation

s_region (spoly) -- Region covered by the observation, as a polygon

s_resolution (double precision) -- Best spatial resolution within the data set

t_min (double precision) -- Lower bound of times represented in the data set, as MJD

t_max (double precision) -- Upper bound of times represented in the data set, as MJD

t_exptime (real) -- Total exporure time

t_resolution (real) -- Minimal significant time interval along the time axis

em_min (double precision) -- Minimal wavelength represented within the data set

em_max (double precision) -- Maximal wavelength represented within the data set

em_res_power (double precision) -- Spectral resolving power delta lambda/lamda

o_ucd (text) -- UCD for the product's observable

pol_states (text) -- List of polarization states in the data set

facility_name (text) -- Name of the facility at which data was taken

instrument_name (text) -- Name of the instrument that produced the data

s_xel1 (bigint) -- Number of elements (typically pixels) along the first spatial axis.

s_xel2 (bigint) -- Number of elements (typically pixels) along the second spatial axis.

t_xel (bigint) -- Number of elements (typically pixels) along the time axis.

em_xel (bigint) -- Number of elements (typically pixels) along the spectral axis.

pol_xel (bigint) -- Number of elements (typically pixels) along the polarization axis.

s_pixel_scale (double precision) -- Sampling period in world coordinate units along the spatial axis

em_ucd (text) -- Nature of the product's spectral axis

tap_schema.columns

Defined in //tap

Columns in tables available for ADQL querying.

table_name (text) -- Fully qualified table name

column_name (text) -- Column name

description (unicode) -- Brief description of column

unit (text) -- Unit in VO standard format

ucd (text) -- UCD of column if any

utype (text) -- Utype of column if any

datatype (text) -- ADQL datatype

arraysize (text) -- Arraysize in VOTable notation

xtype (text) -- VOTable extended type information (for special interpretation of data content, e.g., timestamps or points)

"size" (integer) -- Legacy length (ignore if you can).

principal (integer) -- Is column principal?

indexed (integer) -- Is there an index on this column?

std (integer) -- Is this a standard column?

sourceRD (text) -- Id of the originating rd (local information)

column_index (smallint) -- 1-based index of the column in database order.

tap_schema.groups

Defined in //tap

Columns that are part of groups within tables available for ADQL querying.

table_name (text) -- Fully qualified table name

column_name (text) -- Name of a column belonging to the group

column_utype (text) -- utype the column withing the group

group_name (text) -- Name of the group

group_utype (text) -- utype of the group

sourceRD (text) -- Id of the originating rd (local information)

tap_schema.key_columns

Defined in //tap

Columns participating in foreign key relationships between tables available for ADQL querying.

key_id (text) -- Key identifier from TAP_SCHEMA.keys

from_column (text) -- Key column name in the from table

target_column (text) -- Key column in the target table

sourceRD (text) -- Id of the originating rd (local information)

tap_schema.keys

Defined in //tap

Foreign key relationships between tables available for ADQL querying.

key_id (text) -- Unique key identifier

from_table (text) -- Fully qualified table name

target_table (text) -- Fully qualified table name

description (text) -- Description of this key

utype (text) -- Utype of this key

sourceRD (text) -- Id of the originating rd (local information)

tap_schema.schemas

Defined in //tap

Schemas containing tables available for ADQL querying.

schema_name (text) -- Fully qualified schema name

description (text) -- Brief description of the schema

utype (text) -- utype if schema corresponds to a data model

tap_schema.supportedmodels

Defined in //tap

Standard data models supported by this service.

This is a non-standard tap_schema table used by DaCHS in the creation of registry records. It is manipulated through gavo imp on tables with supportsModel and supportsModelURI properties.

sourceRD (text) -- Id of the originating rd (local information)

dmname (text) -- Human-readable name of the data model

dmivorn (text) -- IVOID of the data model (sorry for the legacy name).

tap_schema.tables

Defined in //tap

Tables available for ADQL querying.

schema_name (text) -- Fully qualified schema name

table_name (text) -- Fully qualified table name

table_type (text) -- One of: table, view

description (text) -- Brief description of the table

utype (text) -- utype if the table corresponds to a data model

table_index (integer) -- Suggested position this table should take in a sorted list of tables from this data center

sourceRD (text) -- Id of the originating rd (local information)

tap_schema.tapjobs

Defined in //tap

A non-standard (and not tap-accessible) table used for managing asynchronous TAP jobs. It is manipulated through TAP job creation and destruction internally. Under very special circumstances, operators can use the gavo admin cleantap command to purge jobs from this table.

Note that such jobs have corresponding directories in $STATEDIR/uwsjobs, which will be orphaned if this table is manipulated through SQL.

jobId (text) -- Internal id of the job. At the same time, uwsDir-relative name of the job directory.

phase (text) -- The state of the job.

executionDuration (integer) -- Job time limit

destructionTime (timestamp) -- Time at which the job, including ancillary data, will be deleted

owner (text) -- Submitter of the job, if verified

parameters (text) -- Pickled representation of the parameters (except uploads)

runId (text) -- User-chosen run Id

startTime (timestamp) -- UTC job execution started

endTime (timestamp) -- UTC job execution finished

error (text) -- some suitable representation an error that has occurred while executing the job (null means no error information has been logged)

creationTime (timestamp) -- UTC job was created

pid (integer) -- A unix pid to kill to make the job stop

uws.userjobs

Defined in //uws

The jobs table for user-defined UWS jobs. As the jobs can come from all kinds of services, this must encode the jobClass (as the id of the originating service).

jobId (text) -- Internal id of the job. At the same time, uwsDir-relative name of the job directory.

phase (text) -- The state of the job.

executionDuration (integer) -- Job time limit

destructionTime (timestamp) -- Time at which the job, including ancillary data, will be deleted

owner (text) -- Submitter of the job, if verified

parameters (text) -- Pickled representation of the parameters (except uploads)

runId (text) -- User-chosen run Id

startTime (timestamp) -- UTC job execution started

endTime (timestamp) -- UTC job execution finished

error (text) -- some suitable representation an error that has occurred while executing the job (null means no error information has been logged)

creationTime (timestamp) -- UTC job was created

pid (integer) -- A unix pid to kill to make the job stop

jobClass (text) -- Key for the job class to use here. This is, as an implementation detail, simply the cross-id of the service processing this.

References

[RMI] Hanisch, R., et al, "Resource Metadata for the Virtual Observatory", http://www.ivoa.net/Documents/latest/RM.html

[VOTSTC] Demleitner, M., Ochsenbein, F., McDowell, J., Rots, A.: "Referencing STC in VOTable", Version 2.0, http://www.ivoa.net/Documents/Notes/VOTableSTC/20100618/NOTE-VOTableSTC-2.0-20100618.pdf

[DALI] Dowler, P, et al, "Data Access Layer Interface Version 1.0", http://ivoa.net/documents/DALI/20131129/

[SODA] Bonnarel, F., et al, "IVOA Server-side Operations for Data Access", http://ivoa.net/documents/SODA/

[Datalink] Dowler, P., et al, "IVOA DataLink", http://ivoa.net/documents/DataLink/