STScI | SPACE TELESCOPE SCIENCE INSTITUTE

EXPANDING THE FRONTIERS OF SPACE ASTRONOMY

Python TAP Implementation at MAST: Lessons Learned

Joshua Fraustro

May 9th, 2023

The Development Team

MAST - Archive Science Applications Branch

“Evergreen Team”

Tom Donaldson

Theresea Dower

Chinwe Edeani

Ben Falk

Joshua Fraustro

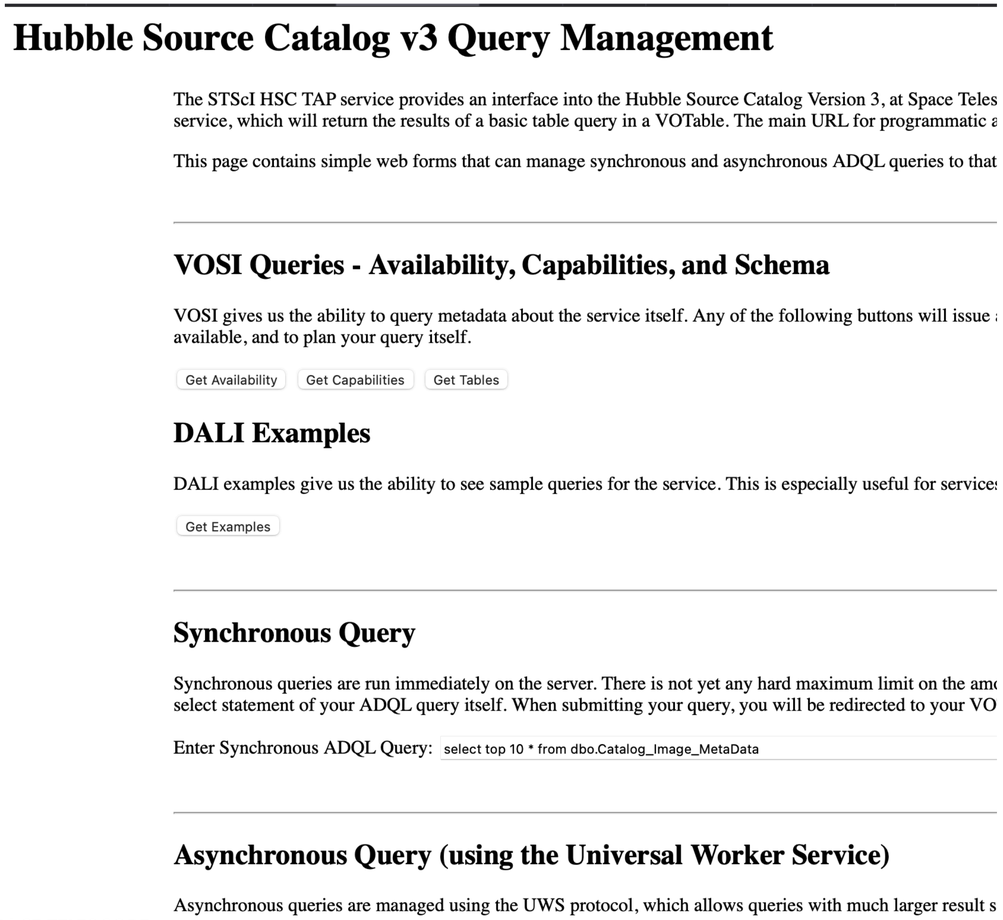
Sarah Weissman

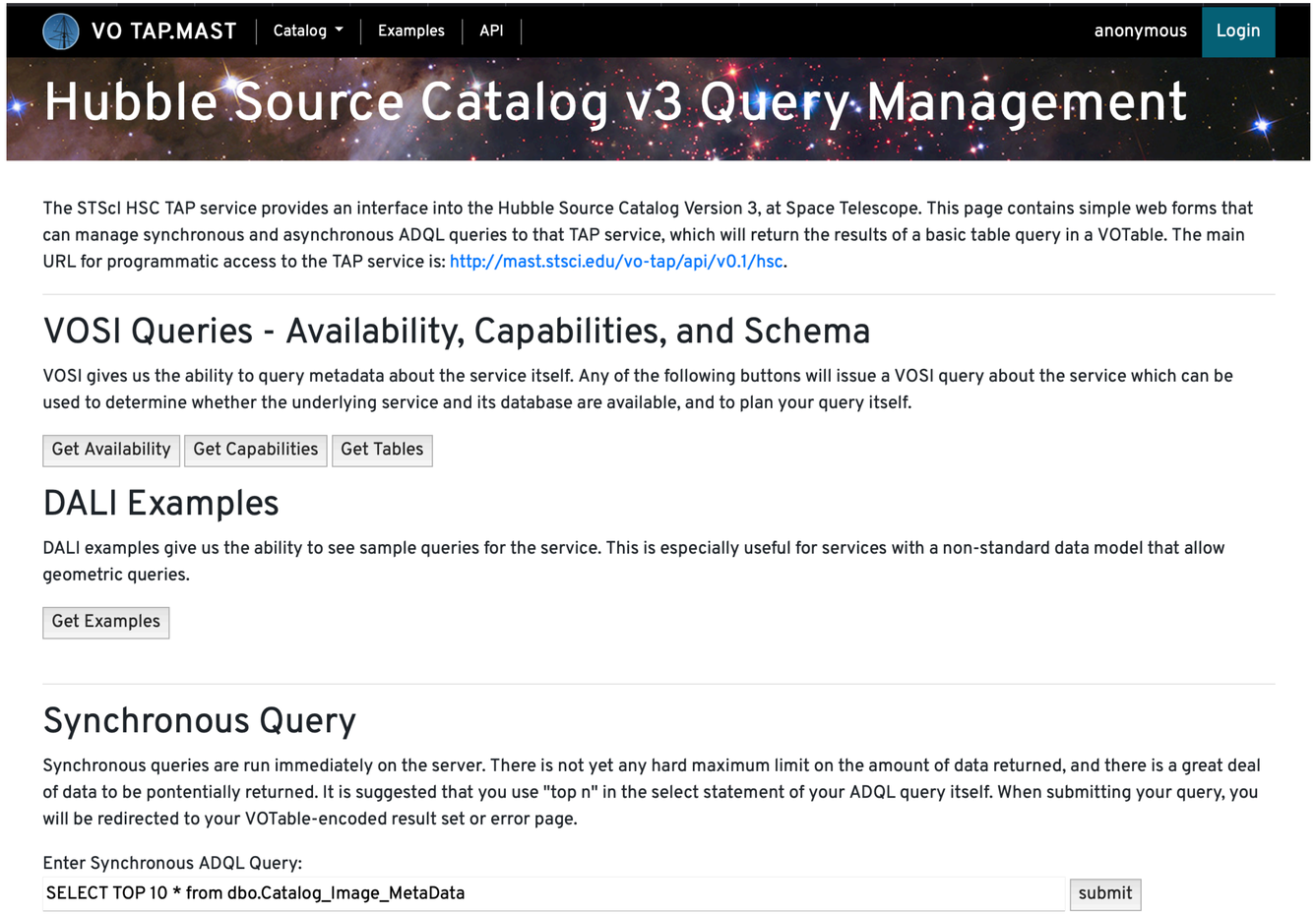
Implementation - VO-TAP

- TAP v1.1 Python Microservice
- Replaces MAST's current C# / Microsoft IIS service
- Uses FastAPI web framework
  - Simple, fast API building, used in our other web services
  - Pre-existing familiarity with the team
  - Takes advantage of Pydantic models for validating requests / responses
- Celery for asynchronous task management
  - Simple task queue / worker message system
  - Easily scalable
  - Uses shared Redis cache for backend / message brokering

Implementation - VO-TAP cont.

- Much needed visual facelift for service landing pages.

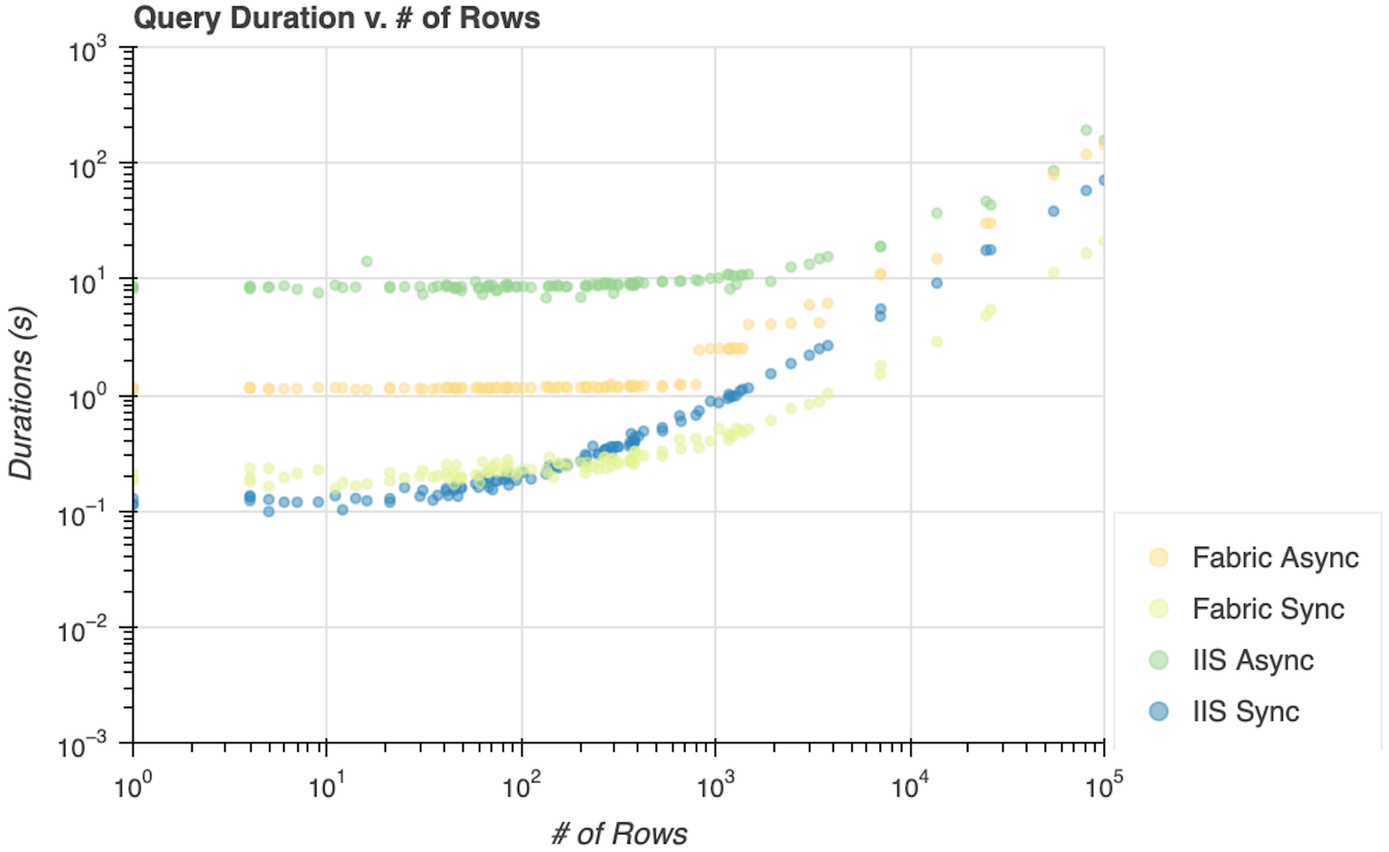




Implementation - VO-TAP cont.

Performance Improvements

- Approximately 6-7x faster asynchronous queries
- Outperforms current service in synchronous queries w/ 1000's-10,000's of rows



Facing the Standard(s)

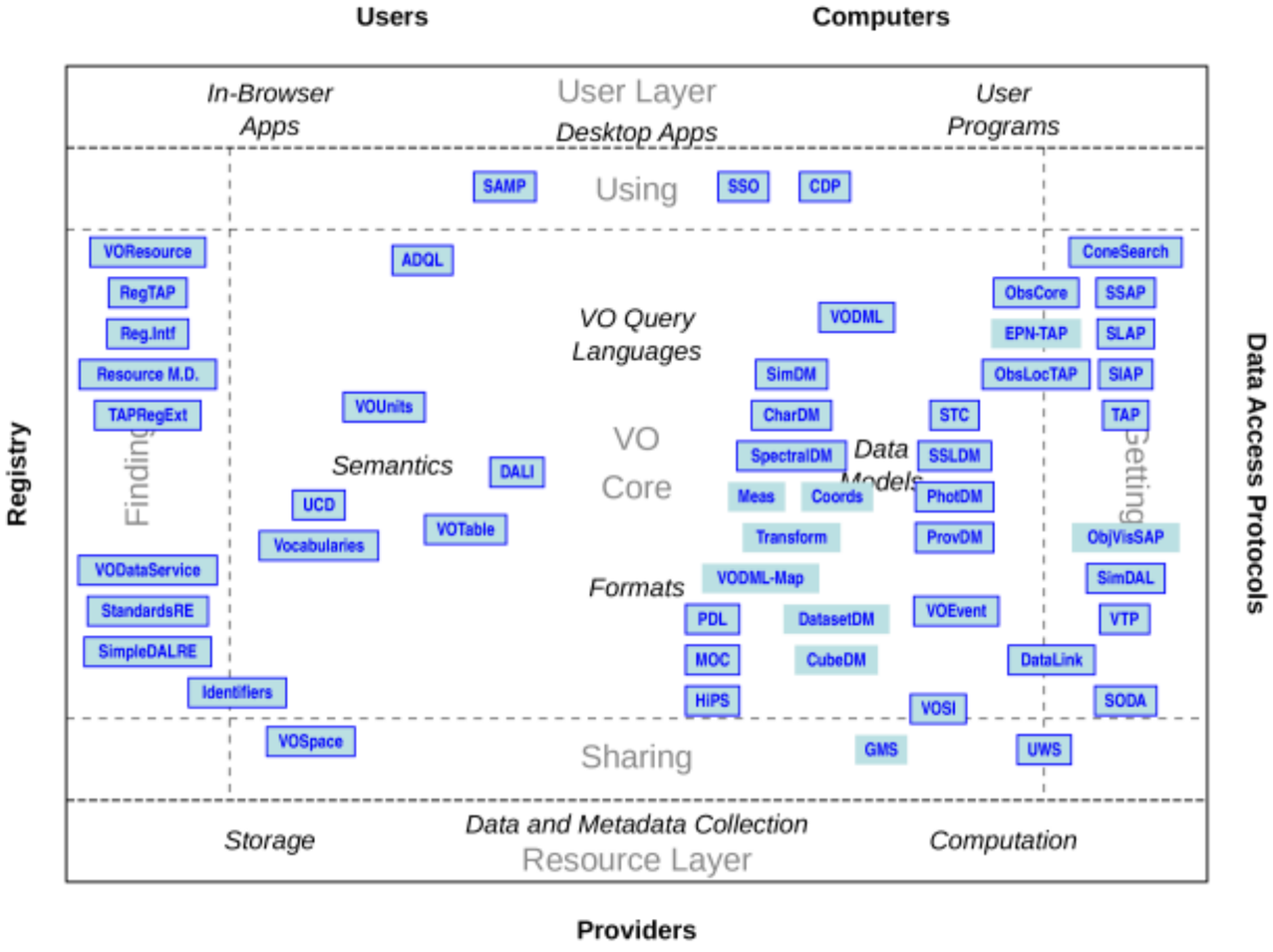
- Where to start?
  - Difficult to approach the standard(s) without a previous implementer / example service to reference.
  - Given the majority of the team's experience with VO services, many of us approached this service essentially from scratch.
  - There is a significant "bus-factor" in familiarity with decades of standards revisions.
  - Made all the more difficult by many overlapping standards.

Data Access

A suite of IVOA protocols specify basic access methods for different types of data resource - for example, Simple Image Access (SIA) for image archives, Simple Spectral Access (SSA) for spectra, Simple Cone Search (SCS) for source catalogues, and several others. These protocols are all "simple" because only a few parameters are involved in the data access requests - typically just RA, Dec, and search radius - and so implementing these services is relatively easy. For more flexible access, the IVOA provides Table Access Protocol (TAP). As well as giving a uniform way to issue SQL-like queries to large survey databases, it provides a richer way to find data from any data service. For example, rather than just looking for any images within a given radius of an RA and Dec, it could offer searching by observation date, camera filter, PI name, and so on. In principle, Structured Query Language (SQL) is already a standard for issuing queries to databases. In practice, there are vendor-dependant variants, and astronomy needs additional standardisation, such as how to specify searching within a circular region. The IVOA therefore produced a standard called "Astronomical Data Query Language (ADQL)", which can be expressed either in simple text strings, or in an XML format. Note that users don't necessarily need to learn this as a language - an application may allow a user to build a query with a graphical interface, or have its own scripting language, as long as the query is translated into standard ADQL. The receiving service likewise converts the standard ADQL into whatever its own database servers need. This is a good example of the VO philosophy : not to dictate what deployers do internally, but rather simply to make them commit to standard *interfaces*. VO data services normally return a standardised data format for tables, known as "VOTable". For a catalogue search, this might actually be the final desired data. For an image search, it is customarily a table including URLs of the images meeting the search criteria. The image data would not be transferred until specifically requested, with how this is done being dependent on the application. VO Tables are intended as an exchange medium, not as a primary storage format. Any FITS Table is expressible as a VO Table, but not necessarily vice versa, as the metadata in a VO Table can be structured hierarchically, as opposed to being a simple list of keyword-value pairs.

Facing the Standard(s)

- "Out of one (standard), many."
  - TAP requires comprehending:
    - UWS, VOSI, ADQL, DALI, VOTable, etc.
    - We were saved by our microservices approach and previous projects that touched these standards.
    - VOTable creation had already been implemented for SCS / others.
    - ADQL -> SQL translator could be reused.
  - These standards often do not agree or are ambiguously worded, despite efforts to clarify (RFC2119).
    - Some clients expect things that are fully optional (pyvo and WAIT).



Facing the Standard(s)

- … sometimes usefeul information is lost between versions.
  - “How do you handle overflows in other formats?”
- From TAP 1.1:

  Reporting of overflow depends on the output format and is described in DALI.

- DALI 1.1:
  - Only describes overflow handling for VOTables.
  - Is there a defined way to do it for other formats then?
  - Check the older standards…

4.4.1 Overflow

If an overflow occurs (result exceeds MAXREC), the service must include an *INFO* element in the *RESOURCE* with `name="QUERY\_STATUS"` and the `value="OVERFLOW"`. If the initial *INFO* element (above) specified the overflow, no further elements are needed, e.g.:

```
<RESOURCE type="results">
<INFO name="QUERY_STATUS" value="OVERFLOW"/>
...
<TABLE>...</TABLE>
</RESOURCE>
```

If the initial *INFO* element specified a status of OK then the service must append an *INFO* element for the overflo

- But in TAP 1.0:

  If the output format is VOTable, section 2.9.1 describes the method by which the overflow is reported. No method of reporting an overflow is defined for formats other than VOTable.

  - The detail of many standards varies between documents, versions.

Facing the Standard(s)

- Difficult to say which standard will address any particular parameter / specification.
  - “What is a valid RunID?”

- UWS:

2.1.9. RunId

The RunId object represents an identifier that the job creator uses to identify the job. Note that this is distinct from the Job Identifier that the UWS system itself assigns to each job. The UWS system should do no parsing or processing of the RunId, but merely pass back the value (if it exists) as it was passed to the UWS at job creation time. In particular it may be the case that multiple jobs have the same RunId, as this is a mechanism by which the calling process can identify jobs that belong to a particular group. The exact mechanism of setting the RunId is not specified here, but will be part of the specification of the protocol using the UWS pattern.

- TAP:

2.7.5 RUNID

The RUNID parameter is fully described in DALI.

- DALI:

3.4.6 RUNID

The service should implement the RUNID parameter, used to tag servic
RUNID value is a string with a maximum length of 64 characters.

For example, if a cross match portal issues multiple requests to remote
and the service logs could later be analysed to reconstruct the service
preserved in any service logs and should pass on the RUNID value in

The RUNID parameter is always single-valued.

Facing the Standard(s)

So, what to do?

- *If there was an easy answer, it probably would have been done!*
- For implementors, something between an “overview” and the technical standards.
  - “Service implementer’s guide”
    - James Tocknell, 2022
  - A “MUST/SHOULD/MAY” service cheatsheet.
- Standards documents:
  - Check “fully described by” references.
  - Dynamic linking to referenced standards would be nice.
  - Hard to discover issues until you try implementing the standard.

OPTIONAL	Support for VOTable output is mandatory, all other formats are optional.
Role within the VO Architecture	
MUST	1030O}] A standards-compliant TAP service must support queries written in the As
MUST	0920O}] All TAP services must be able to serve query results in the VOTable forma
REQUIRED	Note that while TAP 1.1 does not require the use of any particular minor version of unusable in practice.
MUST	For example, the overflow reporting and xtype attribute were introduced in VOTabl
SHOULD	0827D}] While there is no formal requirement to that effect, the response on a TAP capabilities in order to allow clients to discover several important aspects of a TAP
MAY	0218P}] TAP services that support authenticated requests may require delegation of credential service and could use delegated credentials for remote calls that require a
MAY	For example, a URL specified in the UPLOAD parameter may require authenticatio
Motivating Use Cases	
Discover Metadata	
MUST	Since content in relational databases is often custom and project-specific, users of a and other information necessary to construct meaningful correct queries.
Query Custom Tables	
	A TAP service should enable users to discover and query custom tables with a flexib

“How 1600 taplint errors brought me to Bologna…”

- Taplint was an absolutely invaluable resource to have during implementation!
- Routine testing during development caught minute “gotchas” in our standards implementation.
- Wished there was slightly more debugging / verbosity in outputs.
  - Had to dive into source code to figure out which query caused which failure.
  - Potentially add a flag to show traceback of failed validation step.
- Potential integration with our CI / CD pipeline to prevent gradual schema drift.

```
    E-EXA-EXPA-0001 Examples document not well-formed X
    E-EXA-EXDH-0001 Examples endpoint present but undec

    Totals: Errors: 1618; Warnings: 6
```

```
W-TMS-CLUN-3 Unused entry in TAP_SCHEMA.columns table: dbo.Sources
W-TMS-CLUN-4 Unused entry in TAP_SCHEMA.columns table: dbo.SourcePositionsView
W-TMS-CLUN-5 Unused entry in TAP_SCHEMA.columns table: dbo.SumMagAper2cat
E-TMS-TST0-1 Missing required table TAP_SCHEMA.schemas
E-TMS-TST0-2 Missing required table TAP_SCHEMA.tables
E-TMS-TST0-3 Missing required table TAP_SCHEMA.columns
E-TMS-TST0-4 Missing required table TAP_SCHEMA.keys
E-TMS-TST0-5 Missing required table TAP_SCHEMA.key_columns
E-TMS-CLOG-x (11 more)

Section TMC: Compare table metadata from /tables and TAP_SCHEMA
E-TMC-TM12-1 Table tap_schema.schemas from schema tap_schema exists in /tables b
E-TMC-TM12-2 Table tap_schema.tables from schema tap_schema exists in /tables bu
E-TMC-TM12-3 Table tap_schema.columns from schema tap_schema exists in /tables b
E-TMC-TM12-4 Table tap_schema.keys from schema tap_schema exists in /tables but
E-TMC-TM12-5 Table tap_schema.key_columns from schema tap_schema exists in /tabl
E-TMC-TM12-6 Table dbo.Catalog_ACS_SourceExtractor from schema dbo exists in /ta
E-TMC-TM12-7 Table dbo.Catalog_Image_MetaData from schema dbo exists in /tables
E-TMC-TM12-8 Table dbo.Catalog_WFC3_SourceExtractor from schema dbo exists in /t
E-TMC-TM12-9 Table dbo.Catalog_WFPC2_SourceExtractor from schema dbo exists in /
E-TMC-TM12-x (22 more)
```

“How 1600 taplint errors brought me to Bologna…”

- For messy or conflicting standards, conforming our service to what the validator expected was useful.
- Not necessarily best practice, but it meant our service conformed to ***something.***
- Sometimes that meant picking something that *made the most sense*, whether or not it was the “*most correct*”.
- Seen with the DALI /examples resource.
- We picked “TAPNOTE_VOCAB” to maintain compatibility with TOPCAT and our previous service.

```
// The 'correct' value for the RDFa @vocab attribute is a real mess.
// The values listed below have some claim to legitimacy.
// The worst problem is that TOPCAT versions 4.4 and earlier
// (from 4.3, when TAP examples support was introduced)
// required one of the forms TAPNOTE_VOCAB or PRAGMATIC_VOCAB,
// and failed to find the example elements otherwise
// (including in absence of any @vocab),
// so until TOPCAT versions 4.3-4.4 inclusive fall out of use,
// services probably need to use one of those forms,
// despite the fact that they are not permitted by any REC.
// If that's not a concern, the DALI 1.1 value is probably preferred.
// See dal list thread "DALI examples vocab" starting 19 May 2016,
// also private thread "examples @vocab" between MBT,
// Markus Demleitner and Pat Dowler in July 2017.
private static final String DALI10_VOCAB;    // DALI 1.0
private static final String TAPNOTE_VOCAB;   // TAP Implementation Note
private static final String PRAGMATIC_VOCAB; // Common practice mid-2017
private static final String DALI11_VOCAB;    // DALI 1.1
private static final String[] EXAMPLES_VOCABS = new String[] {
    DALI10_VOCAB =    "ivo://ivoa.net/std/DALI#examples",
    TAPNOTE_VOCAB =   "ivo://ivoa.net/std/DALI-examples",
```

Serialization Woes - “445 LoC to write a UWS response”

- Building XML documents for VO responses is more challenging than it seems at first glance.
- Previous approaches (for VOTable) involved string building / concatenation.
  - Issues:
    - Requires a lot of logic for element placement & validation.
    - Logic is specific to individual VO object types: JobSummary vs. ShortJobDescription
    - Tedious to write, easy to break in updates
      - Hard to cover version changes in a backwards-compatible way.
    - Separate functionality needed for reading / writing.
  - Benefits:
    - allows for streaming responses from the service for returning large tables
    - Enabled handling overflows easily, since we build the table one row at a time.

```
_VOT_INFO_NAME = ' name="%(name)s"'
_VOT_INFO_VALUE = ' value="%(value)s"'
_VOT_INFO_ATTRIBUTES = {"name": ("name", _VOT_INFO_NAME), "value": ("value
_VOT_INFO_BUILD_PLAIN = """
<INFO %(name)s%(value)s/>"""
_VOT_PARAM_ID = 'ID="%(id)s"'
_VOT_PARAM_NAME = ' name="%(name)s"'
_VOT_PARAM_DATATYPE = ' datatype="%(datatype)s"'
_VOT_PARAM_ARRAY = ' arraysize="%(arraysize)s"'
_VOT_PARAM_VALUE = ' value="%(value)s"'
_VOT_PARAM_ATTRIBUTES = {
    "id": ("id", _VOT_PARAM_ID),
    "name": ("name", _VOT_PARAM_NAME),
    "datatype": ("datatype", _VOT_PARAM_DATATYPE),
    "arraysize": ("arraysize", _VOT_PARAM_ARRAY),
    "value": ("value", _VOT_PARAM_VALUE)
}
```

```
def get_votable_info(info):
    """returns a votable INFO element"""
    string_builder = {}

    for attribute, builder in _VOT_INFO_ATTRIBUTES.items():
        info_attribute = info.get(attribute, None)
        if info_attribute is not None and str.upper(info_attribute) != "NULL":
            if isinstance(info_attribute, str):
                info_attribute = escape_xml(info_attribute).replace('"', "&quot;")
            info.update({attribute:info_attribute})
            string_builder[builder[0]] = builder[1] % info
        else:
            string_builder[builder[0]] = ""
    return _VOT_INFO_BUILD_PLAIN % string_builder
```

Serialization Woes - “445 LoC to write a UWS response”

- Set our sights on a better way to represent and handle UWS standard.
- Desired solution should have:
  - Automatic, internal VO schema validation.
    - Should validate when created in code, or read from a database.
  - Object-oriented creation / modification.
  - Hands-off XML output & serialization.
  - Developer-focused:
    - Don’t spend dev time on the minutiae
    - Ex: adding the RunId to a JobSummary should be as simple as job_summary.run_id = “xyz”
    - Shouldn’t have to know the schema or standard specifics to work with the code.
      - But also able to make changes when necessary!

Serialization Woes - “445 LoC to write a UWS response”

- “Pydantic-xml” package was our solution
  - https://github.com/dapper91/pydantic-xml

Pros:	Cons:
Any VO resource can be designed as a data model.	Can’t use objects with streaming responses
Object-oriented	A bit of upfront work to write the models
One-line XML serialization.	Package is early in development - documentation is lacking
Automatic schema validation *(if you wrote it right!)*	Optional “nillable” elements are a headache

Pydantic-xml JobSummary Example

Defining the JobSummary model:

```python
class JobSummary(BaseXmlModel, tag="job", ns="uws", nsmap=NSMAP):
    """JobSummary element - The complete representation of the state of a UWS Job…

    schema_version: UWSVersions = attr(name="version", default=UWSVersions.V1_1.value)

    job_id: str = element(tag="jobId", default_factory=lambda: str(uuid.uuid4()))
    run_id: Optional[str] = element(tag="runId")
    owner_id: Optional[NillableElement] = element(tag="ownerId")
    parameters: Parameters = element(tag="parameters", default_factory=lambda: [])
    results: Optional[Results] = element(tag="results")
    #...
```

Pydantic-xml JobSummary Example

Adding validators:

```
@validator("run_id")
def validate_runid_length(cls, value):
    """Validate the run_id is < 64 characters. Should also be handled by the FastAPI endpoint."""
    if value:
        if len(value) > 64:
            raise ValueError("runID value must be less than 64 characters")
    return value

@validator("owner_id", pre=True, always=True)
def validate_ownerid(cls, value):
    """Set the owner_id element if one was provided, otherwise set the element to nil"""
    return validate_nillable(value, str)
```

Pydantic-xml JobSummary Example

Instantiate and edit like any Python object:

```
job_summary = JobSummary()
uws_parameters = Parameters(
    parameters=[
        Parameter(id="query", value=query),
        Parameter(id="maxrec", value=maxrec),
        Parameter(id="catalog", value=catalog),
        Parameter(id="param_format", value=param_format),
    ]
)
job_summary.run_id = runid
job_summary.parameters = uws_parameters
```

Pydantic-xml JobSummary Example

Serializing the response:

```python
return XMLResponse(summary.to_xml(skip_empty=True))
```

```xml
<uws:job xmlns:uws="http://www.ivoa.net/xml/UWS/v1.0"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="1.1">
    <uws:jobId>hsc_70ef12ca-4029-44fc-984f-cc064c524822</uws:jobId>
    <uws:ownerId xsi:nil="true"/>
    <uws:phase>COMPLETED</uws:phase>
    <uws:quote xsi:nil="true"/>
    <uws:creationTime>2023-05-03T20:29:16.040Z</uws:creationTime>
    <uws:startTime>2023-05-03T20:29:16.046Z</uws:startTime>
    <uws:endTime>2023-05-03T20:29:16.767Z</uws:endTime>
    <uws:executionDuration>0</uws:executionDuration>
    <uws:destruction>2023-05-04T20:29:16.040Z</uws:destruction>
    <uws:parameters>
        <uws:parameter id="query">SELECT TOP 10 * FROM TAP_SCHEMA.columns</uws:parameter>
        <uws:parameter id="maxrec">100000</uws:parameter>
        <uws:parameter id="catalog">hsc</uws:parameter>
        <uws:parameter id="param_format">votable</uws:parameter>
    </uws:parameters>
    <uws:results>
        <uws:result id="result" xlink:type="simple" xlink:href="http://scarcury2022.local.
    </uws:results>
</uws:job>
```

Pydantic-xml JobSummary Example

Automatically handles child elements!

‘ResultReference’ and ‘Parameters’ are models too.

```python
class ResultReference(BaseXmlModel, tag="result", ns="uws",
    """ResultReference element - simple container for xlink

    id: str = attr(name="id", default="result")
    type: Optional[TypeValue] = attr(
        name="type",
        ns="xlink",
        default=TypeValue.SIMPLE,
    )
    href: Optional[str] = attr(ns="xlink")
    size: Optional[int] = attr()
    mime_type: Optional[str] = attr(name="mime-type")
```

```python
class Parameter(BaseXmlModel, tag="parameter", ns="uws
    """Parameter element - list of input parameters to

    value: str
    byReference: Optional[bool] = attr()
    id: str = attr(name="id")
    is_post: Optional[bool] = attr(name="isPost")
```

Pydantic-xml JobSummary Example

Automatically handles child elements!

For: /async/{job_id}/parameters:

```
return XMLResponse(summary.parameters.to_xml(skip_empty=True))
```

```
<uws:parameters xmlns:uws="http://www.ivoa.net/xml/UWS/v1.0"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <uws:parameter id="query">
        SELECT TOP 10 * FROM TAP_SCHEMA.columns
    </uws:parameter>
    <uws:parameter id="maxrec">100000</uws:parameter>
    <uws:parameter id="catalog">hsc</uws:parameter>
    <uws:parameter id="param_format">votable</uws:parameter>
</uws:parameters>
```

For: /async/{job_id}/results:

```
return XMLResponse(summary.results.to_xml())
```

```
<uws:results xmlns:uws="http://www.ivoa.net/xml/UWS/v1.0"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <uws:result id="result" xlink:type="simple" xlink:href="
</uws:results>
```

Metadata Madness

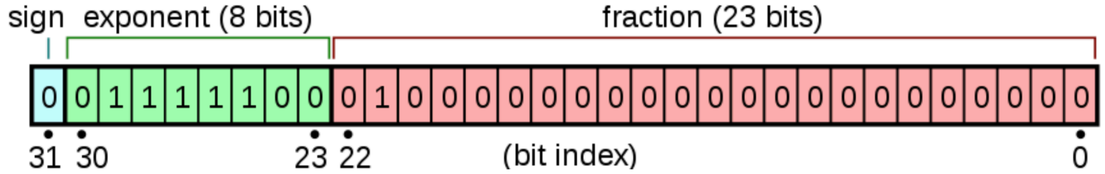
- How to ensure VOTable compliant datatypes without access to TAP_SCHEMA.columns?
  - We previously just assumed a “char(*)”!
- We use the pyodbc package for interacting with the database.
  - pyodbc will convert database results from their SQL_TYPE to the closest Python object type.
  - Because of the “flexible” nature of Python type objects, we lose the specificity of the datatype.
    - short, int, long -> Python int
- For other projects, we don’t want to rely on TAP_SCHEMA or make additional calls.

The following table describes how database results are converted to Python objects.

Description	ODBC Datatype	Python Datatype
NULL	any	None
bit	SQL_BIT	bool
integers	SQL_TINYINT, SQL_SMALLINT, SQL_INTEGER, SQL_BIGINT	int
floating point	SQL_REAL, SQL_FLOAT, SQL_DOUBLE	float
decimal, numeric	SQL_DECIMAL, SQL_NUMERIC	decimal.Decimal
1-byte text	SQL_CHAR	str via UTF-8 (1)
2-byte text	SQL_WCHAR	str via UTF-16LE (1)
binary	SQL_BINARY, SQL_VARBINARY	bytes
date	SQL_TYPE_DATE	datetime.date
time	SQL_TYPE_TIME	datetime.time
SQL Server time	SQL_SS_TIME2	datetime.time
timestamp	SQL_TIMESTAMP	datetime.datetime
UUID / GUID	SQL_GUID	str or uuid.UUID (2)
XML	SQL_XML	str via UTF-16LE (1)

Metadata Madness

- Pyodbc gives us the bit precision of the database column!
  - The precision is the fractional portion of the value + the sign.
    - A precision of 24 = float, 53 = double, etc.
  - We can then map them to their matching VOTable datatypes to fill out FIELD elements.
  - VOTable compliant datatypes can be provided without schema awareness and agnostic of database driver.




```
elif db_type_name in ["TINYINT", "SMALLINT"]:
    votable_datatype =  VOTableDataType.SHORT
elif db_type_name == "INTEGER":
    votable_datatype =  VOTableDataType.INT
elif db_type_name == "BIGINT":
    votable_datatype =  VOTableDataType.LONG

elif db_type_name in ["REAL", "FLOAT", "DECIMAL"]:
    votable_datatype =  VOTableDataType.FLOAT

elif db_type_name == "DOUBLE":
    votable_datatype =  VOTableDataType.DOUBLE
```