NodeSoftware Documentation

Document Information

Editors:	T. Marquart
Authors:	T. Marquart, S. Regandell, G. Rixon
Contributors:	VAMDC WP7 working group
Type of document:	software documentation
Status:	release
Distribution:	public
Work package:	WP7
Version:	12.07
Document code:	
Directory and file name:	

Abstract: This document describes the functionality and the setup of the VAMDC NodeSoftware which implements the standards for a VAMDC service.

Version History

Version	Date	Modified By	Description of Change
0.1	16/12/2010	T. Marquart	first draft
0.2	27/01/2011	T. Marquart	version for implementation workshop
11.5	27/05/2011	T. Marquart	release together with standards
11.5r1	15/06/2011	T. Marquart	update to match software release 11.5r1
11.12	25/01/2012	T. Marquart	updates to match software release 11.12
11.12r1	13/01/2012	T. Marquart	minor updates for bugfix release
11.12r2	29/05/2012	T. Marquart	updates for bugfix release
11.12r3	05/06/2012	T. Marquart	minor updates for bugfix release
12.07	05/12/2012	T. Marquart	minor updates for this release

Acknowledgements

VAMDC is funded under the "Combination of Collaborative Projects and Coordination and Support Actions" Funding Scheme of The Seventh Framework Program. Call topic: INFRA-2008-1.2.2 Scientific Data Infrastructure. Grant Agreement number: 239108.

Contents

This document covers the **release 12.07** of the NodeSoftware.

Links to HTML-versions:

- Last release: http://readthedocs.org/docs/vamdc-nodesoftware/en/release/
- Latest development: http://readthedocs.org/docs/vamdc-nodesoftware/en/latest/

Links to PDF-versions:

- Last release: http://media.readthedocs.org/pdf/vamdc-nodesoftware/release/vamdc-nodesoftware.pdf
- Latest development: http://media.readthedocs.org/pdf/vamdc-nodesoftware/latest/vamdc-nodesoftware.pdf

CHAPTER 1

Introduction

1.1 About VAMDC

The Virtual Atomic and Molecular Data Center is a EU FP7 research infrastructure project and you can read all about it on http://vamdc.eu/

1.2 VAMDC nodes

A "node" within VAMDC is a data service that offers its data using the standards and protocols defined by the VAMDC. They are web services with a simple API, the specification of which can be found in the documentation for the VAMDC standards: http://vamdc.org/documents/standards/

The scope of this document is to serve as documentation for the reference implementation of such a service. The goal of this implementation is to serve as publishing tools for new data services, i.e. it is meant to be easily deployed at multiple nodes.

1.3 A versatile implementation of VAMDC standards

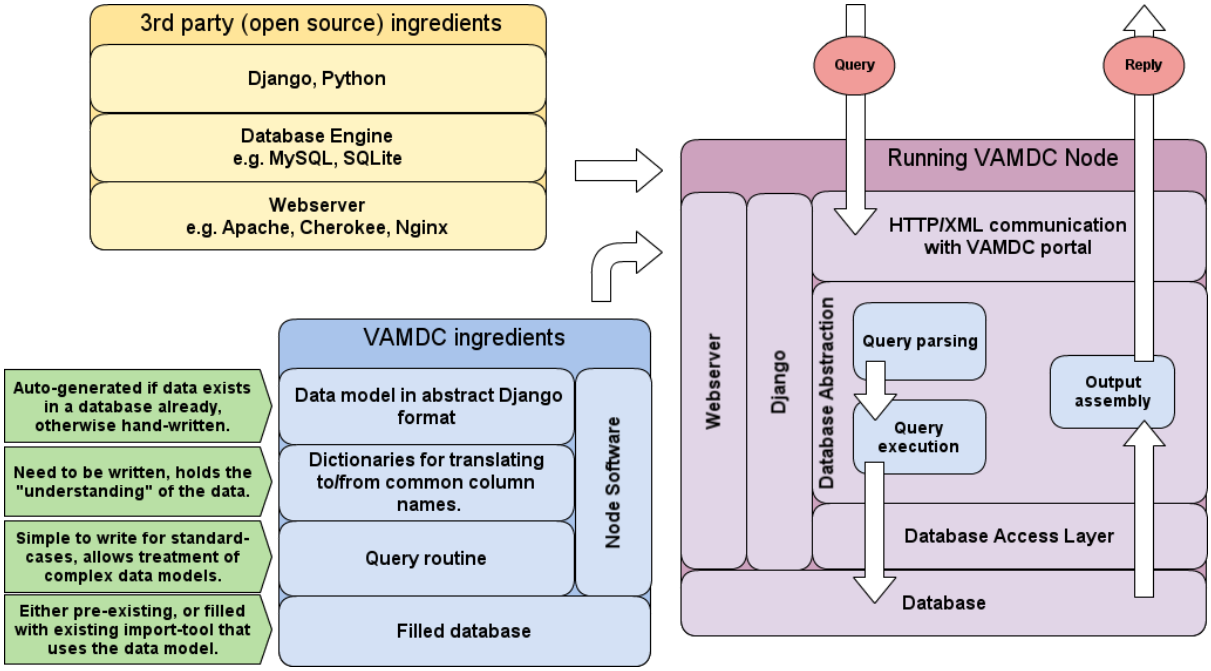
Principle design decisions that were made to arrive at this software package include

- *Open source*. No software licences need to be bought and the used software can be adapted if needed.
- The data must exist in a *relational database*. If this is not the case yet, a tool for creating it is provided.
- *Flexibility in the data structure*. The service should be able to be plugged on top of existing databases and therefore needs to cope with almost arbitrary internal data formats.
- *Re-usable code*. The implementation of the VAMDC standards and protocols themselves should not depend on the requirements of a specific node.

Since the last two points contradict each other in practice, there needs to be an intermediate layer of abstraction that hides the node-specific details like the database layout from the parts of code that are shared between nodes.

Our implementation of the VAMDC node software is therefore based on a framework called Django (which in turn is based on the programming language Python) that provides both the database abstraction layer and high level tools for implementing web services.

The ingredients for a VAMDC node based on this software package and its operation look schematically like this:


3rd party (open source) ingredients
Django, Python
Database Engine
e.g. MySQL, SQLite
Webserver
e.g. Apache, Cherokee, Nginx
VAMDC ingredients
Auto-generated if data exists in a database already, otherwise hand-written.
Data model in abstract Django format
Need to be written, holds the "understanding" of the data.
Dictionaries for translating to/from common column names.
Simple to write for standard-cases, allows treatment of complex data models.
Query routine
Either pre-existing, or filled with existing import-tool that uses the data model.
Filled database
Node Software
Query
Reply
Running VAMDC Node
Webserver
Django
Database Abstraction
HTTP/XML communication with VAMDC portal
Query parsing
Query execution
Output assembly
Database Access Layer
Database

CHAPTER 2

Changelog

Note: This chapter will be difficult to understand if you have not read the whole document before, since terms are used that are introduced later. It is meant for returning readers, especially the maintainers of VAMDC nodes.

2.1 Nov 15, 2012

Version 12.07. This is version 12.07 of the NodeSoftware that implements the VAMDC standards with equal version number. There have been no conceptual changes in the inner workings and upgrading should be painless. A few points are however worth mentioning.

Mirrors. VAMDC nodes can have mirrors and each deployment should know about the others in order to convey the URLs through the `/tap/capabilities` endpoint. The way to do this is through a variable in `settings.py` like this:

```
MIRRORS = ['http://mirror1.domain/tap/', 'http://mirror2.domain/tap/']
```

Note that the URL of the current node should not be repeated here and that they should end including the trailing slash.

Last modified. The information on when the data that are returned in a query were last modified, e.g. to use a HEAD (preview) request to check for new data, nodes can fill the HTTP header `Last-Modified` to convey this information. It is up to each node whether this will be done on a fine-grained level of individual data or globally for the whole database. The values can be set from `settings.py` (variable LAST_MODIFIED) or from the node's `setupresults()` function as LAST-MODIFIED in the header dictionary. As values, either a string (like: *Sat, 10 Nov 2012 23:00:00 GMT*) or a Python `datetime.date` instance that will be formatted automatically.

Basis States. The dictionary keywords for the molecular basis states were named inconsistently. In order to keep the naming scheme somewhat sane, words like *BasisState** have been renamed to *MoleculeBasisState**. The words for the basis state quantum numbers like *BasisStateQN** have been renamed to *MoleculeBQN**.

License. A file LICENSE has been added to the package and contains the agreed-upon software license, as taken from http://opensource.org/licenses/BSD-3-Clause

Django upgrade. Lastly, please do not forget to keep the underlying software packages up to date. Currently Django is at version 1.4.2 and upgrading should not make much problems. Also upgrade the rest of your nodes' stack, like Nginx, Gunicorn or Apache.

2.2 June 5, 2012

Version 11.12r3. This is a bugfix-release over 11.12r2 with few changes. Among them are two bug-fixes however that make this quick release necessary. One was breaking XSAMS validation with molecule chemical names and another made the nodes reject queries from the portal.

2.3 May 23, 2012

Version 11.12r2. This is a bugfix-release of the NodeSoftware, implementing the VAMDC standards 11.12. There are no major internal changes that should require updating the code that is specific for each data node.

Django 1.4. A new version of Django, the framework that we build upon, has been released. With the changes contained in this release, deployment should not break when upgrading to Django 1.4 and we encourage all nodes to do so. (However staying at 1.3 for a while is no security risk.) If you installed Django via *pip* as is recommended later in this document, upgrading is as simple as:

```
pip install --upgrade django
```

Bugfixes. Thanks everyone who reported issues; many small improvements are included in this release. A notable change is that partly invalid queries are now rejected whereas the valid part was executed before.

Pybtex is no longer a dependency unless a nodes uses the automatic BibTex to XSAMS conversion.

Future. A small number of "post 11.12 standards" things are included already (like molecular basis states) but nothing that changes previously existing parts of the XML generator.

Documentation. Apart from minor updates, information has been added on *How to update an existing database*.

2.4 February 13, 2012

Version. This is for *NodeSoftware 11.12r1* which is the first bugfix-release for version 11.12 released before.

No major internal changes that require updating the code that is specific for each node, except:

NormalModes. Previously, the NormalModes in the atomic state composition of XSAMS were wrongly attached to each Atom object, now they need to be handed to the generator as `AtomState.NormalModes`. This means that nodes which use this part of the schema need to update their query-function.

2.5 January 22, 2012

Version. This is for *NodeSoftware 11.12* which implements the VAMDC standards 11.12. (Please make sure to also read the changes for the beta release below.)

Since the beta-release (11.10beta), there are no major changes of the internal workings, which means that you most likely do not need to change the query-function if it worked with that. However, please test your node after an upgrade anyway.

Dictionary. Some keywords have changed, both Restrictables and Returnables (due to the changes in the schema), so please double-check the node's `dictionaries.py` against http://dictionary.vamdc.org/.

DEPLOY_URL. You can now override the automatic determination of the URL at which a node is deployed, see *Setting the deployment URL*.

New IDs. The XSAMS standard now makes mandatory several IDs in an XSAMS document, for example each process must have an ID now. Please read *Filling the IDs* on how to do this.

Advanced treatment of Restrictables. If a node wants to support a Restrictable that does not match a field in the database, this can now be handled with some custom code. See *Treating a Restrictable as a special case*.

Finding the bug. For debugging purposes, it may help to manually go through the steps that happen when a query comes to a node. See *Quick debugging and testing* for information on how to do this.

Self-referencing <Source>. In the bibliographical part of the XSAMS schema, i.e. the <Source> elements, the xml-generator now automatically adds such an element in order to describe the document itself. It contains a timestamp and the full query URL, among other things. Please check the output if this works correctly for your node.

Last, but not least, since we often are asked how to test a node, we'd like to mention that there is a very convenient software called **TAPvalidator** (see http://www.vamdc.org/software) which can be used to query a node, browse the output and check that it is valid with respect to the xsams standard.

2.6 September 30, 2011

Version. This is for *NodeSoftware 11.10beta*, which has most of the changes for the upcoming 11.10 standards release and is aleady more robust than previous releases. All nodes are encouraged to upgrade.

Query functions. The standard way of starting a node's query function has changed: the function *where2q()* is superseded by *sql2Q()*. **This means you should change this in your code!** See the updated example in *The query routine*.

Requestables. Queries to the nodes can now ask to return only a certain part of the XML document, for example "SELECT Spiecies WHERE ..." instead of "SELECT ALL WHERE ...". This works behind the scenes, but a node's query function might want to skip some of the work, see *Handling the Requestables better*

Returnables. Many Returnables (e.g. all that correspond to a DataType in the XML schema) now can receive vectors which allows to give several values of the same quantity. See *Using a custom model method for filling a Returnable* on how to do this.

Unit conversions. Each Restrictable has a default unit in which the queries are formulated. If a node's database has the quantity in a different unit, the value in the query needs to be converted to the internal unit. There is now a comfortable mechanism to do this, see *Unit conversions for Restrictables*

Dictionaries. While we're at Restrictables, it is good to keep in mind that a node is the more useful the more Restrictables it supports, simply because it will be able to answer a higer fraction of queries. All nodes that have data about radiative transitions are **highly encouraged** to support RadTransWavelength, even if they internally keep frequency or wavenumber. Some clients, like the current portal, made the choice to always use wavelength.

Restrictable prefixes. Apart from the Requestables (see above) the second major addition in the query language VSS2 is that Restrictables can have prefixes, separated by a dot from the usual keyword. For example *SELECT * WHERE Upper.AtomStateEnergy > 13*. See the standard documentation for all available prefixes. Currently the easiest way for a node to support these is to treat them as separate Restrictables in `dictionaries.py`. This becomes tricky for collisions where the prefixes allow to group Restrictables to belong to reactants and/or products. Since this very much depends on the individual node, there are currently no specific tools for this, but we are certainly open for ideas on how to solve this.

Special Restrictables. If a node needs to handle one or more Restrictables as special cases, for example because the corresponding value is not in the database, this is certainly possible. See *Treating a Restrictable as a special case*

Custom return formats. This goes beyong the VAMDC standard but if you are interested to return other formats from your node, you can have a look at *How to skip the XSAMS generator and return a custom format*.

The section on *Logging* has been extended as well and a few notes about *Making more use of Django* were added.

2.7 June 15, 2011

Version. This documentation has been updated to match the release of the NodeSoftware 11.5r1 which implements the VAMDC Standards release 11.5. NodeSoftware 11.5r1 supersedes and obsoletes version 11.5 (released May 26) and all nodes are encouraged to upgrade. This is mainly a bug-fix release and upgraded nodes will only have to do the two small changes mentioned below.

Example Queries. The way to define example queries in each node's `settings.py` has changed in order to allow several of them. They will be used for automated testing and are as of this version returned to the VAMDC registry. New example:

```
EXAMPLE_QUERIES = [\
    'SELECT ALL WHERE RadTransWavelength > 4000 AND RadTransWavelength < 4005',
    'SELECT ALL WHERE AtomSymbol = "Fe"',
    ]
```

CaselessDict. The import and use of *CaselessDict* in the nodes' `dictionaries.py` or `queryfunc.py` is not longer necessary and should be removed.

Limitations. A chapter on the limitations of the NodeSoftware has been addedd to the documentation: *Known limitations*

Dictionary. The NodeSoftware makes use of dictionary keywords that are not in the VAMDC Standards 11.5 but will be in the next Standards release (11.7). If you want to use the NodeSoftware's XML-generator for solids, particles or molecular quantum numbers, please see http://dictionary.vamdc.org/dict/ for the new keywords.

Registration. The NodeSoftware now automatically reports its own version and the standards version it implements at *tap/capabilities*. You might want to make the VAMDC Registry re-read this information (click "Edit metadata" and "Update the registry entry").

Virtual Machine. The virutal machine has been updated to include Django 1.3 and NodeSoftware 11.5r1.

2.8 May 26, 2011

Version numbers. As of now, we introduce version numbers for both the standards (XSAMS, VAMDC-TAP, see separate documentation) and for their implementation in the NodeSoftware which is the concern of this document. Version numbers follow the format YY.MMrX where YY is for the year, MM the month, and X an increasing number for bugfix revisions that do not affect the usage of the NodeSoftware.

The most important changes from the perspective of a node-operator who wants to upgrade to this *11.5* release are:

Update to Django 1.3. The NodeSoftware now requires Django version 1.3 and node operators probably need to upgrade their installation of Django. See *Upgrading*.

Email. Make sure you have set a correct email address in `settings.py`. It will be used to report critical errors to, including reports on what went wrong.

Logging. The capabilities to log debug and error-messages have been extended. See *Logging*.

Example query. As soon as a node becomes operational, please add an example query to its `settings.py`. It will be used for automated testing. Example:

```
EXAMPLE_QUERY = 'SELECT ALL WHERE RadTransWavelength > 4000 AND RadTransWavelength < 4005'
```

Volume estimate. In order to allow the portal (and other queries to your node) to find out how big the resulting XML-output for a particular query will be, nodes should estimate this and relay it via the new HTTP-header *VAMDC-APPROX-SIZE*. The easiest way to do this is to run a test query, determine the outputs size (in MB) and divide it by the number of items (e.g. transitions, if these dominate your results). This number can then be used to estimate the size of any query, see the updated example at *The query routine*.

Other Header changes. The header *VAMDC-COUNT-SPECIES* has been replaced by *VAMDC-COUNT-ATOMS* and *VAMDC-COUNT-MOLECULES*. See the standards documentation for the full definition.

Error handling in urls.py. The NodeSoftware has become more error-safe and tries to handle unexected input and "crashes" more gracefully. You need not care about this, excpet making sure that the following two lines are present at the end of the file `urls.py` in your node's main directory:

```
handler500 = 'vamdctap.views.tapServerError'
handler404 = 'vamdctap.views.tapNotFoundError'
```

Dictionary changes. Since the XSAMS-schema has changed, so have the dictionary keywords, especially in the Broadening-part of radiative transitions and the atomic quantum numbers. Also new keywords have been added for the bits that are newly implemented in the XML-generator.

Stricter format for accuracies. In compliance with XSAMS' new way of defining a value's accuracy, the keywords that are not explicity given for *DataTypes* have become more. Any word *SomeKeyword* that is marked as a *DataType* in the dictionary allows for use of the following words as well: SomeKeywordUnit, SomeKeywordRef, SomeKeywordComment, SomeKeywordMethod, SomeKeywordAccuracyCalibration, SomeKeywordAccuracyQuality, SomeKeywordAccuracySystematic, SomeKeywordAccuracySystematicConfidence, SomeKeywordAccuracySystematicRelative, SomeKeywordAccuracyStatistical, SomeKeywordAccuracyStatisticalConfidence, SomeKeywordAccuracyStatisticalRelative, SomeKeywordAccuracyStatLow, SomeKeywordAccuracyStatLowConfidence, SomeKeywordAccuracyStatLowRelative, SomeKeywordAccuracyStatHigh, SomeKeywordAccuracyStatHighConfidence, SomeKeywordAccuracyStatHighRelative. See also the standards documentation.

Note: The last two points mean that you probably have to update your `dictionaries.py`.

2.9 March 10, 2011

The chapter *The main concepts behind the implementation* now has more detail on the XSAMS schema.

A large part of the XML/XSAMS generator has been rewritten, both to comply with the new version of the schema and in terms of its structure. In addition the keywords in the VAMDC dictionary have changed somewhat. This means that **you will probably need to update your query function and dictionaries when you update the NodeSoftware.**

Step by step guide to a new VAMDC node has been updated and extended accordingly.

A new version of the virtmach has also been uploaded, containing the latest NodeSoftware and operating system.

2.10 February 2011

The deployment of nodes is now covered in more detail at *Deployment of your node*.

CHAPTER 3

The main concepts behind the implementation

The following is a glossary-like list that shortly touches upon various subjects that one should be aware of before setting up a new VAMDC node.

3.1 The database

As already mentioned in the *Introduction*, data needs to reside in a relational database in order to use the node software for a VAMDC node. This is what we mean by *database* in the following, in contrast to *data set* which means the data in any format or *data model*:

3.2 The data model(s)

The *data model* is a definition of the database layout in form of Python code where a *class* is defined for each table in the database and the members of the class are *fields* that correspond to the tables' columns. The data model also defines the connections between tables. For an existing database the data model can be automatically generated, otherwise it needs to be written for a new node (see *Step by step guide to a new VAMDC node* later) and will then be used to create the database.

Having this code representation of the database layout has many advantages, among these are:

- automatic (re-)creation of the database, independent of the engine
- no need to learn SQL
- easy queries
- additional features like easily traversing linked tables in both directions.

Note: Sometimes the singular *data model* refers to a single model (i.e. a table in the database) and sometimes the full set of models, describing the whole database layout.

3.3 The VAMDC dictionary

In order to facilitate automated communication, there is a need for a set of names that identify a certain type of data. Each name is unique and is uniquely associated with a description, a data type, a unit where applicable and a (non-mandatory) restriction.

For illustration, let's have a look at one entry of the dictionary:

Keyword	short descr	long description	data type	re-stric-tion	unit
Atom-Mass-Number	Atomic mass	Atomic mass in Daltons, which is the same as the unified mass units (1Da = 1u = 1.660 538 86 (28) e-27)	(Float|Double)	>1	amu

It is the first column that contains the *name* that we use globally within VAMDC for a certain bit of information. This is what we mean in the following when we talk about "global names" or "keywords".

The full VAMDC dictionary is still being worked on and it currently resides at http://dictionary.vamdc.eu/ where also some helper tools are provided.

At the nodes, the dictionary is used in the following different ways. Note that some keywords do not make sense being used in all three cases. Common sense applies.

Note: The Returnables and Restrictables, as described in the following, are different for each node (depending on the data it offers and its structure) and need to be written when setting up a new node.

3.3.1 Returnables

Each node keeps a list of global names that we call the *Returnables*. This list contains the names associated with the kinds of information that a node has to offer. This is list is offered as XML at the *tap/capabilities/* URL end point which allows user applications to decide whether it is worth to query a certain node for a certain bit of data, or not.

The node software stores the Returnables not only as a list of global names, but as a list of key-value pairs where the names are the keys and the values are the corresponding places of the data in the *data model* (see above). This way, the Returnables become a simple one-to-one map between the global names, used by all VAMDC nodes, and the node-specific layout of the database.

This "translation" is then used, among other things, by the code that fills the data into a certain output format which in turn can become node-independant. Thereby each Returnable corresponds to a certain place (a column in table format, or a certain XML tag) in the output format.

3.3.2 Restrictables

It is the list of global names that make sense to use as constraints for a certain node and therefore tells which names from the dictionary can be used in the WHERE-clause of a query to the node (see query language below).

Again, the node software uses the Restrictables as a list of key-value pairs where the keys are the global names and the values are the corresponding place in the data model. As for the Returnables, this one-to-one map of global names to custom data model allows to translate between the two - this time when the query is parsed at arrival. The code for parsing the query uses this and can thus be re-used by all nodes without altering the code.

3.3.3 Requestables

Requestables are a third way of using the dictionary. They are used in the SELECT-clause of the SQL expression when one wants to recieve only a subset of the data that matches the restrictions. For example, *SELECT Species, RadiativeTransitions* would return only the fields in this group and skip any information about the states, if it were available.

3.4 The registry

The registry is a central web service where all VAMDC nodes are registered with their access URL and some additional information. This allows finding nodes before sending queries to them. You will need to register your

node there once the setup is complete.

Note: What follows below is not necessary to know for setting up a new VAMDC node but it helps to get the broader picture.

3.5 TAP services

TAP stands for *Table Access Protocol* and is a Virtual Observatory standard definition of a web service. The detailed specs can be found here. All VAMDC nodes offer their data though a TAP-like interface which means that the URL end-points are named like in TAP, the most important being */tap/sync* for a data query which returns the data synchronously (in the immediate reply). Also the attribute names for submitting a query are strongly inspired by TAP so that a query to a single VAMDC node looks something like this:

```
http://domain.of.your.node/tap/sync/?LANG=VSS1&FORMAT=XSAMS&QUERY=query string
```

VAMDC nodes currently only use and support a subset of the TAP standard, i.e. that parts that are needed within the VAMDC. Keep in mind that users will not primarily query an individual node but use a higher level tool like the VAMDC portal for querying many nodes at once. Data providers that want to set up their own VAMDC node do not really need to care about TAP either.

The more detailed specification of the VAMDC variant of a TAP service can be found in the standards documentation at http://vamdc.org/documents/standards/.

3.6 The query language

The node software uses the *VAMDC SQL-subset 2* (VSS2) which is a superset of VSS1 for query language as defined in the VAMDC standards. This is basically a SQL-like string where the database layout of the node does not need to be known - instead one uses the keywords from the dictionary in the WHERE part to restrict the selection of data. This means that all nodes understand identical queries and there is no need to adapt the query to a certain node.

Details can be found in the VAMDC-TAP specification (see link above) and should not be necessary to know for setting up a new VAMDC node. Defining the Restrictables and Returnables is enough for allowing the node software to take care of the rest.

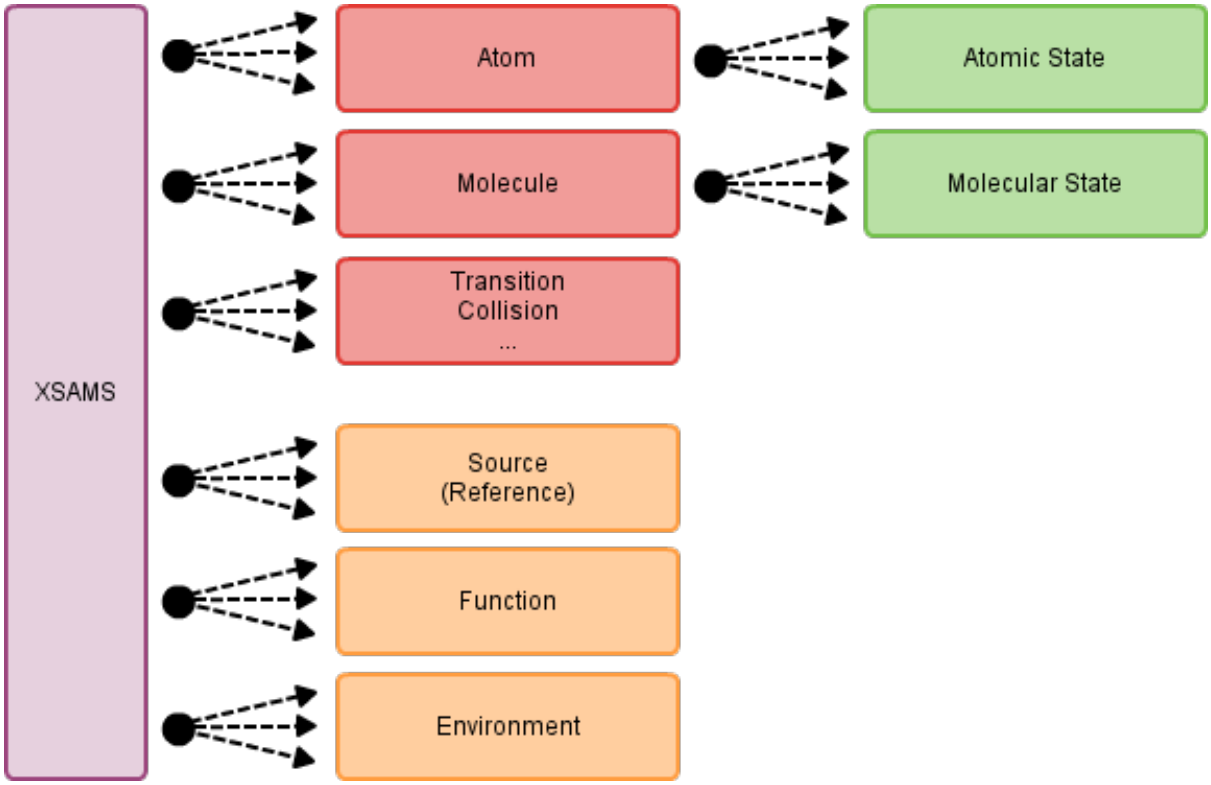
3.7 The XSAMS schema

XSAMS stands for XML Schema for Atoms, Molecules and Solids. It defines a strict way to represent data in XML. XSAMS is the format in which VAMDC nodes send their data replies.

Link to the VAMDC-XSAMS project on Sourceforge.

The NodeSoftware provides an implementation of the XSAMS schema and data providers need not know it in detail to set up a VAMDC node. However, basic knowledge of its structure is needed to be able to write the few bits of code as explained in the next chapter.

XSAMS is a hierarchical structure which simplified looks like this:



Inside each box resides all the data that corresponds to it. The *atom* box holds the name, atomic number, masses, iosotopes, ionization and so on. The *atomic state* box holds the state energy, quantum numbers and so on.

The different parts are interlinked, for example each atomic state has an ID and the transitions can refer to them as their initial and final state. Sources (i.e. publications) can be referenced, again by their ID, for each bit of information provided.

3.8 The generic XSAMS generator

The node software comes with an implementation of the XSAMS that can (but need not necessesarily) be used by all nodes, aka the XSAMS *generator*. This frees data providers from the need to know about XML and the details of the schema. In order for this to work, data providers need fill the *Returnables* as described above and in the next chaper. The generator then knows how to put the data into the schema.

In principle, XSAMS allows many more nested loops than are shown in the diagram above. But since each node needs to build from its database the structure that matches the hierarchy, we have made some deliberate simplifications. For example we treat each ion and isotope or an atom as a different atom/species. This means that skip the complexity of having five or more nested loops at the expense of replicating some information.

3.9 The portal

The portal is the obvious example of a *user application* that makes use of VAMDC nodes. It is a web site that facilitates the submission of a query to many nodes at once by providing a web form out of which it assembles the query string which it then sends to one or many nodes, gathers the results from each of them and presents them to the user.

CHAPTER 4

Software prerequisits and installation

4.1 Quick start

If you use a Linux-distribution like Debian (squeeze) or Ubuntu (some not too old version), you can simply run the following command (with root-rights) to install all software that you need:

```
$ apt-get update && apt-get install python python-pip python-pyparsing python-mysqldb gunicorn ng
$ pip install django
```

This will automagically install some more packages that the above ones depend upon. There are most probably similar packages for other linux distributions than Debian. All software should be able to be installed on Windows and OSX as well but it probably involves some more effort and we unfortunately cannot give support for this.

We also provide a virtual machine appliance with Debian/Linux and all required software installed into it. You can then run this virtual machine on a host computer, using VirtualBox which is available for free on most operating systems. See virtmach for more detail on this.

If the commands above worked or you run the virtual machine, you might want to skip to *Test your installation*. Otherwise continue reading for a list of the individual software dependencies..

4.2 Python plus some modules

Python is a wide-spread, open-source, object-oriented, dynamically-typed, interpreted programming language. You can read all about it at http://python.org and there exist installation packages for all operating systems and architectures.

We require Python between (and including) versions 2.5 and 2.7.

We recommend to also install IPython (http://ipython.scipy.org/), an improved interactive shell for Python.

4.2.1 Database access library

This depends on your choice of database engine (see below). The two choices we support primarily are SQLite (access library comes with Python itself) and MySQL (access library at http://pypi.python.org/pypi/MySQL-python/ but preferrably installed by your OS's package manager).

4.2.2 PyParsing

This is needed for our SQL-parser and you can read about it at http://pypi.python.org/pypi/pyparsing

Again, it is best installed via your distribution's package manager.

4.3 Django

Django is the Python-based web-framework that we use to run the services (see *Introduction* and http://djangoproject.com). We currently use Django 1.3.X (where X is the latest bug-fix version number) but newer versions will be supported as they are released.

The packaged version of your OS is probably outdated. This is why we recommend to install Django using `pip` (see command above). Alternatively follow the installation instructions on the Django website.

4.4 Database engine

If the data that your node should serve reside already in a relational database, there is most probably no need to set up a new one but you instead deploy the node software directly on top of the existing database. The list of databases that Django can handle can be found at http://docs.djangoproject.com/en/1.3/ref/databases/

When setting up a new database, we recommend one of the following two

- SQLite http://www.sqlite.org/
- MySQL http://mysql.com/ (or, if ORACLE succeeds in messing MySQL up, the MySQL fork called MariaDB http://mariadb.org/)

Unless the data set is extremely large and/or complex, the choice between the two is of minor importance. SQLite has the advantage of not relying on a separate server software and is often on par with MySQL in terms of speed. Its limitation in terms of concurrent write access is not relevant in our typical use case where the database is only read, not written to, during standard operation.

4.5 Webserver

The node software needs to run within a webserver. The two setups that we successfully tested are *Gunicorn* (together with *nginx*) and the Apache webserver (with its WSGI module).

This is covered in more detail in *Deployment of your node*.

4.6 Git version control

This is not a real requirement since you can download the node software (see *The Code*) directly. However, using the version control system *git* (http://git-scm.com/), it becomes easier to update your installation and to re-submit your changes.

4.7 The node software itself

See *The Code* on how to obtain the source code.

4.8 Test your installation

None of the following commands should give you an error:

```
$ python -c "import django"
$ python -c "import pyparsing"

$ cd /path/to/where/you/downloaded/NodeSoftware
$ cd nodes/ExampleNode
```

```
$ ./manage.py
$ ./manage.py test
$ ./manage.py shell
```

The last command will open an interactive Python shell for you (IPython, if you have it installed, otherwise standard Python) and in there you should be able to run:

```
>>> from node.models import *
>>> import vamdctap
>>> exit()
```

If any of this fails, please make sure you have installed all of the above correctly and ask your system administrator for help. For contacting us, see *Bugs and Contact*.

Note: The above only tests that you have installed the software correctly, not the setup and configuration of the node in question.

CHAPTER 5

Upgrading

5.1 NodeSoftware

The simplest way is to simply download the latest tar.gz-archive and extract it on top of you previous installation. We however strongly recommend to backup the files in your node-directory before doing this; alternatively moving the old NodeSoftware to a different location and then copy the files you need from there into the new version.

If you instead use our version control system, please see *Collaborating with git and GitHub* on how to get the latest.

Note: After upgrading the NodeSoftware, you should check that your node is still running properly. We cannot (yet) guarantee that you need not update your node-specific code to fit the latest version. Larger changes will be mentioned in the *Changelog*.

5.2 Django

This depends on how you installed Django. With `pip` it is enough to run:

```
$ pip install --upgrade django
```

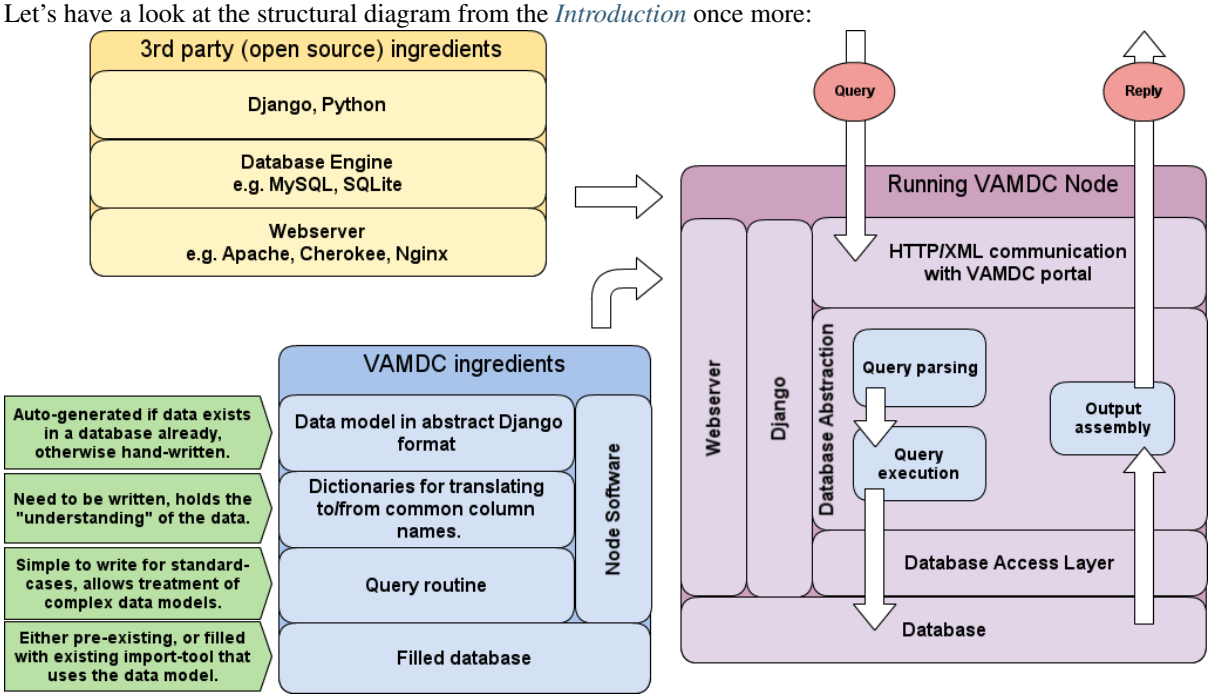
5.3 Everything else

If you have installed all the prerequisites from Debian or Ubuntu packages as recommended, you can simply run the following regularly to keep your system up to date:

```
$ apt-get update
$ apt-get upgrade
```

Step by step guide to a new VAMDC node

Let's have a look at the structural diagram from the *Introduction* once more:



If you have followed the instructions of the page on *Software prerequisits and installation*, you are done with the yellow box in the figure. This page will tell you first how to configure and write the few code bits that your node needs before running (blue box), and then how to deploy the node and make it run as shown in the violet box.

It goes like this:

- Get the Nodesoftware and make a copy of the example node.
- Auto-create a new settings file and put your database connection there.
- **Either**
  - Write your data model and let Django create the database from it. Then use the import tool to put your data there.
  - Let Django write the model from an existing database that you already have.
- Assign names from the VAMDC dictionary to your data to make them globally understandable.
- Start your node and test it.

But let's take it step by step:

6.1 The main directory of your node

Let's give the directory which holds your copy of the NodeSoftware a name and call it *$VAMDCROOT*. (It is called *NodeSoftware* by default and exists whereever you downloaded and extracted it, unless you moved it elsewhere and/or renamed it, which is no problem to do) a name and call it *$VAMDCROOT*. Let's also assume the name of the dataset is *YourDBname*.

Inside $VAMDCROOT you find several subdirectories. For setting up a new node, you only need to care about the one called *nodes/* which contains the files for several nodes already, plus the example node. The first thing to do, is to make a copy of the ExampleNode:

```
$ export VAMDCROOT=/your/path/to/NodeSoftware/
$ # (the last line is for Bash-like shells, for C-Shell use `setenv` instead of `export`
$ cd $VAMDCROOT/nodes/
$ cp -a ExampleNode YourDBname
$ cd YourDBname/
```

Note: In the following you always work within this newly created directory for your node. You should not need to touch any files or run commands outside it.

6.2 Inside your node directory

The first thing to do inside your node directory is to run:

```
$ ./manage.py
```

This will generate a new file `settings.py` for you. This file is where you override the default settings which reside in `settings_default.py` (which you should not edit!). There are only a few configuration items that you need to fill

- The information on how to connect to your database.
- A name and email address for the node administrator(s).
- Example queries that makes sense with your data.
- Optionally you can set the location of the log-file and override other options by copying from `settings_default.py`.

The structure for filling in this information is already inside the newly created file. You can leave the default values for now, if you do not yet know what to fill in.

There are only three more files that you will need to care about in the following:

- `node/models.py` is where you put the data model,
- `node/dictionaries.py` is where you put the dictionaries and
- `node/queryfunc.py` is where you write the query function,

all of which will be explained in detail in the following.

6.3 The data model and the database

By *data model* we mean the piece of Python code that tells Django the layout of the database, including the relations between the tables. By *database* we mean the actual relational database that is to hold the data. (See also *The main concepts behind the implementation*).

There are two basic scenarios to come up with these two ingredients. Either the data are already in a relational database, or you want to create one.

6.3.1 Case 1: Existing database

If you want to deploy the VAMDC node software on top of an existing relational database, the *data model* for Django can be automatically generated by running:

```
$ ./manage.py inspectdb > node/models.py
```

This will look into the database that you told Django about in `settings.py` above and create a Python class for each table in the database and attributes for these that correspond to the table columns. An example may look like this:

```
from django.db.models import *

class Species(Model):
    id = IntegerField(primary_key=True)
    name = CharField(max_length=30)
    ion = IntegerField()
    mass = DecimalField(max_digits=7, decimal_places=2)
    class Meta:
        db_table = u'species'
```

There is one important thing to do with these model definitions, apart from checking that the columns were detected correctly: The columns that act as a pointer to another table need to be replaced by *ForeignKeys*, thereby telling the framework how the tables relate to each other. This is best illustrated in an example. Suppose you have a second model, in addition to the one above, that was auto-detected as follows:

```
class State(Model):
    id = IntegerField(primary_key=True)
    species = IntegerField()
    energy = DecimalField(max_digits=17, decimal_places=4)
    ...
```

Now suppose you know that the field called *species* is acutally a reference to the species-table. You would then change the class *State* as such:

```
class State(Model):
    id = IntegerField(primary_key=True)
    species = ForeignKey(Species)
    energy = DecimalField(max_digits=17, decimal_places=4)
    ...
```

Note: You will probably have to re-order the classes inside the file `models.py`. The class that is referred to needs to be defined before the one that refers to it. In the example, *Species* must be above *State*.

Let's add a third model:

```
class Transition(Model):
    id = IntegerField(primary_key=True)
    species = ForeignKey(Species)
    upper_state = ForeignKey(State, related_name='transup')
    lower_state = ForeignKey(State, related_name='translo')
    wavelength = FloatField()
```

The important thing here is the *related_name*. Whenever you want to define more than one *ForeignKey* to the *same* model, you need to set this to an arbitrary name. This is because Django will automatically set up the reverse key for you and needs to give it a unique name. The reverse key in this example could be used to get all the Transitions that have a given State as upper or lower state. More on this at *Setting the related name of a field*.

Once you have finished your model, you should test it. Continuing the example above you could do something like:

```
$ ./manage.py shell
>>> from node.models import *
```

```
>>> allspecies = Species.objects.all()
>>> allspecies.count() # the number of species is returned
>>> somestates = State.objects.filter(species__name='He')
>>> for state in somestates: print state.energy
>>> sometransitions = Transition.objects.filter(wavelength__lt=500)
>>> atransition = sometransitions[5]
>>> othertransitions = atransition.upper_state.transup.objects.all()
>>> othertransitions.count() # gives the number of transitions with the
                            # same upper state.
```

Detailed information on how to use your models to run queries can be found in Django's own excellent documentation: http://docs.djangoproject.com/en/1.3/topics/db/queries/

6.3.2 Case 2: Create a new database

In this case we assume that the data are in ascii tables of arbitrary layout. The steps now are as follows:

1. Write the data model in your `node/models.py`.
2. Create an empty database with corresponding user and password
3. Tell the node software where to find this database.
4. Let the node software create the tables
5. Use the import tool to fill the database with the data.

First of all, you need to think about how the data should be structured. Data conversion (units, structure etc) can and should be done while importing the data since this saves work and execution time later. Since the data will need to be represented in the common XSAMS format, it is recommended to adopt a layout with separate tables for species, states, processes (radiative, collisions etc) and references.

Deviating data models are certainly possible, but will involve some more work on the query function (see below). In any case, do not so much think about how your data is structured now, but how you want it to be structured in the database, when writing the models.

Writing your data models is best learned from example. Have a look at the example from Case 1 above and at file `$VAMDCROOT/nodes/vald/node/models.py` inside the NodeSoftware to see how the model for VALD looks like. Keep in mind the following points:

- As mentioned, a *class* in the model becomes a *table* in the database and the fields/members of the class correspond to the table columns.
- Each class should have one member with *primary_key=True*. If not, one called *id* will be implicitly created for you.
- How you name your classes and fields is up to you. Sensible names will make it easier to write the dictionaries below.
- Use the appropriate field type for each bit of data, e.g. BooleanField, CharField, PositiveSmallIntegerField, FloatField. There is also a DecimalField that allows you to specify arbitrary precision which will also be used in later ascii-representations of data.
- Use *ForeignKey()* to another class's primary key to connect your tables.
- The full list of possible fields can be found at http://docs.djangoproject.com/en/1.3/ref/models/fields/.
- If you know that a field will be empty sometimes, add *null=True* to the field definition inside the brackets ().
- For fields that are frequent selection criteria (like wavelength for a transition database), you can add *db_index=True* to the field to speed up searches along this column (at the expense of some disk space and computation time at database creation).
- If you do not define a table name for your model with the Meta class, as in the first example above, the table in the database will be named as the model, but lowercase and with a prefix *node_*.

Once you have a first draft of your data model, you test it by running (inside your node directory):

```
$ ./manage.py sqlall node
```

This will (if you have no error in the models) print the SQL statements that Django will use to create the database, using the connection information in `settings.py`. If you do not know SQL, you can ignore the output and move straight on to creating the database:

```
$ ./manage.py syncdb
```

Now you have a fresh empty database. You can test it with the same commands as mentioned at the end of Case 1 above, replacing "Species" and "State" by your own model names.

Note: There is no harm in deleting the database and re-creating it after improving your models. After all, the database is still empty at this stage and *syncdb* will always create it for you from the models, even if you change your database engine in `settings.py`. The command for re-creating the tables in the database (deleting all data!) is `./manage.py reset node`.

Note: If you use MySQL as your database engine, we recommend its internal storage engine InnoDB over the standard MyISAM. You can set this in your settings.py by adding *'OPTIONS': {"init_command": "SET storage_engine=INNODB"}* to your database setup. We also recommend to use UTF8 as default in your MySQL configuration or create your database with *CREATE DATABASE <dbname> CHARACTER SET utf8;*

How you fill your database with information from ascii-files is explained in the next chapter: *How to get your data into the database*. You can do this now and return here later, or continue with the steps below first.

6.4 Using the XML generator

Before we go on to the remaining two ingredients, the *query function* and the *dictionaries*, we need to have an understanding on how they play together in the XML generator. As you remember from *The XSAMS schema*, the goal is to run queries on your models and pass on the output to the generator so that it can looped over them to fill the hierarchical XSAMS structure.

In order to make this work, we need to name the variables that you pass into the generator (as explained below) and the loop variables that you use in the Returnables. For example, continuing on the model above: Assume you have made a selection of your Transition model; you pass this on under the name *RadTrans*; the generator loops over it, calling each Transition insite its loop *RadTran* (note the singular!). *RadTran* is now a single instance of your Transition model and has the wavelength as *RadTran.wavelength* since we called the field this way above. The entry in the RETURNABLES would therefore look like *'RadTranWavelenth':'RadTran.wavelength'* - where the first part is the keyword from the VAMDC dictionary (which the generator knows where in the schema it should end up) and the second part tells it how to get the value from the query results that it got from your query function.

Do not fret if this sounded complicated, it will become clear in the examples below. Just read the previous paragraph again after that.

Here is a table that lists the variables names that you can pass into the generator and the loop variables that you use in the Returnables. The one is simply the plural of the other.

Passed into generator	Loop variable	Object looped over	Loop variable
Atoms	Atom		
		Atom.States	AtomState
		Atom.Components	Component
		Atom.Component.SuperShells	AtomSuperShell
		Atom.Component.Shells	AtomShell
		Atom.Component.ShellPairs	AtomShellPair
Molecules	Molecule		

Continued on next page

Table 6.1 – continued from previous page

Passed into generator	Loop variable	Object looped over	Loop variable
		Molecule.States	MoleculeState
		Molecule.State.Parameters	Parameter
		Molecule.State.Parameter.Vector	VectorValueOA
		Molecule.NormalModes	NormalMode
		Molecule.State.Expansions	Expansion
		Molecule.State.Expansion.Coefficients	Coefficient
Solids	Solid		
		Solid.Layers	Layer
		Solid.Layer.Components	Component
Particles	Particle		
RadTrans	RadTran		
		RadTran.Shiftings	Shifting
		RadTran.Shifting.ShiftingParams	ShiftingParam
		RadTran.Shifting.ShiftingParam.Fits	Fit
		RadTran.Shifting.ShiftingParam.Fit.Parameters	Parameter
RadCross	RadCros		
		RadCros.BandModes	BandMode
CollTrans	CollTran		
		CollTran.Reactants	Reactant
		CollTran.IntermediateStates	IntermediateState
		CollTran.Products	Product
		CollTran.DataSets	DataSet
		CollTran.DataSet.FitData	FitData
		CollTran.DataSet.FitData.Arguments	Argument
		CollTran.DataSet.FitData.Parameters	Parameter
		CollTran.DataSet.TabData	TabData
NonRadTrans	NonRadTran		
Environments	Environment		
		Environment.Species	EnvSpecies
Particles	Particle		
Sources	Source		
Methods	Method		
Functions	Function		
		Function.Parameters	Parameter

The third and fourth columns are for an inner loop. So for example the generator loops over all *Atoms*, calling each atom insteance *Atom*. To extract all states being a part of this particualar Atom, the generator will assume that there is an iterable *States* defined on each *Atom* over which it will iterate. So it will loop over *Atom.States*, calling each of state *AtomState* in the inner loop, like this:

```
for Atom in Atoms:

    [...]

    for AtomState in Atom.States:

        [...]
```

It is up to you to make sure the *Atom.States* is defined if you want to output state information. This is covered in the next section.

6.5 The query routine

Now that we have a working database and data model and know in principle how the generator works, we simply need to tell the framework how to run a query and pass the output to the generator. This is done in a single function called *setupResults()* which must be written in the file `node/queryfunc.py` in your node directory. It works like this:

- *setupResults()* is called from elsewhere and you need not run it yourself.
- *setupResults()* gets an object as input, called *sql*. This is a parsed version of the query that comes in. It holds the WHERE-part as *sql.where* and so on.
- We now need to run this query on the data model in order to get so called *QuerySets* which are basically unevaluated queries that are then passed on to the XML generator which takes care of the rest.
- If you want to enforce limits on how much data can be returned in one query, this can be done here as well.
- You should also calculate some statistics on how much information a query returns and return it as header information.

In a concrete example of an atomic transition database, it looks like this:

```
from django.db.models import Q
from vamdctap.sqlparse import *
from dictionaries import *
from models import *

LIMIT = 10000

def setupResults(sql):
    q = sql2Q(sql)
    transs = Transition.objects.filter(q).order_by('wavelength')
    ntranss = transs.count()

    if ntranss > LIMIT:
        percentage = '%.1f'%(float(LIMIT)/ntranss *100)
        limitwave = transs[LIMIT].wavelength
        transs = Transition.objects.filter(q,Q(vacwave__lt=limitwave))
    else: percentage=None

    spids = set( transs.values_list('species_id',flat=True) )
    species = Species.objects.filter(id__in=spids)
    nspecies = species.count()
    nstates = 0
    for specie in species:
        subtranss = transs.filter(species=specie)
        up=subtranss.values_list('upper_state_id',flat=True)
        lo=subtranss.values_list('lower_state_id',flat=True)
        sids = set(up+lo)
        specie.States = State.objects.filter(id__in = sids)
        nstates += len(sids)

    headerinfo={'TRUNCATED':percentage,
                'COUNT-ATOMS':nspecies,
                'COUNT-STATES':nstates,
                'COUNT-RADIATIVE':ntranss
                'APPROX-SIZE':ntranss*0.001
               }

    return {'RadTrans':transs,
            'Atoms':species,
            'HeaderInfo':headerinfo
           }
```

Explanations on what happens here:

- Lines 1-4: We import some helper functions from the sqlparser and the dictionaries and models that reside in the same directory as `queryfunc.py`
- Line 6: Set the limit of transitions for use below.
- Line 7: Begin the function *setupResults*. Do not change this line.
- Line 9: This uses the helper function where2q() to convert the information in *sql.where* to QueryObjects that match your model, using the RESTRICTABLES (see below). The result from where2q() is a string that needs to be executed with eval().
- In line 10 we simply pass these QueryObjects to the Transition model's filter function. This returns a QuerySet, an unevaluated version of the query, which we assign to the variable *transs*. We also ordered it by wavelength.
- Line 11: We use the count() method on the QuerySet to get the number of transitions which we later pass into the header.
- Line 13-17: We check if the number is larger than our limit and shorten the QuerySet if necessary. This is done by getting the wavelength at the limit and making a new QuerySet that has as an additional restriction the new upper wavelength limit. We also prepare a string with the percentage for the headers.
- Lines 19-29: Here comes the tricky part. For the selected transitions, we now need to create the corresponding atoms/species, since they go into different parts of the generator, see the table above. Not only that, each atom should have attached its list of states that are upper or lower states for the selected transitions - there is an inner loop over *Atom.States* in the generator, remember? In detail:
  - Line 19: We pull a single column out of the Transitions model, the key that links to the Species model. We put that into a *set()* to throw out duplicates.
  - Line 20: We use this set to query for all our Species.
  - Line 21: We count them and save the result for later.
  - Line 22: We make a new variable for the number of states which we will increase in the coming loop.
  - Line 23: Start a loop over our selected *species*.
  - Line 24: Make a sub-selection on our previously selected transitions, now only selecting the ones that belong to the current species.
  - Lines 25-26: As for the species IDs before, we now pull the keys to the upper and lower states out of our Transition model.
  - Line 27: We concatenate the two lists of IDs and put them in a *set()* to get rid of duplicates. *sids* is now a list of IDs of all the states within the current species that are used in the selected transtions.
  - Line 28: Use this list to make the query on the State model. And, **most importantly**, attach it to the current species object. This way we have constructed the nested structure for the generator.
  - Line 29: For the statistics, we now increase the state count with the number for the current species.
- Lines 31-36: Put the statistics into a key-value structure where the keys are the header names as definded by the VAMDC-TAP standard and the values are the strings/numbers that we calculated above.
- Lines 39-41: Return the QuerySets and the headers, again as key-value pairs. The keys are the names from the first column of the table above, so that the generator recognizes them and loops over them at the right place.

Note: As you might have noticed, all restrictions are passed to the Transitions model in the above example. This does not mean that we cannot put constraints on e.g. the species here. We simply use the models ForeignKey in that case in the RESTRICTABLES. An entry there could e.g. be *'AtomIonCharge':'species__ion'* which will use the *ion* field of the species model. Depending on your database layout, it might not be possible to pass all restrictions to a single model. Then you need to write a more advanced query than the shortcuts in Lines 7-8.

Note: We are well aware that adapting the above example to your data is a non-trivial task unless you know Python and Django reasonably well. There is a more complete example in `ExampleNode/node/queryfunc.py` and you can also have a look at the other nodes' `queryfunc.py` which are included in the NodeSoftware. And, of course, we are willing to assist you in this step, so feel free to contact us about this.

More comprehensive information on how to run queries within Django can be found at http://docs.djangoproject.com/en/1.3/topics/db/queries/.

6.6 The dictionaries

As the last important step before the new node works, we need to define how the data relates to the VAMDC *dictionary*. If you have not done so yet, please read *The VAMDC dictionary* before continuing.

What needs to be put into the file `node/dictionaries.py` is the definition of two variables that map the individual fields of the data model to the names from the dictionary, like this:

```
RESTRICTABLES = {\
'AtomSymbol':'species__name',
'AtomIonCharge':'species__ion',
'RadTransWavelength':'wavelength',
}

RETURNABLES={\
'NodeID':'YourNodeName', # constant strings work
'AtomIoncharge':'Atom.ion',
'AtomSymbol':'Atom.name',
'AtomStateEnergy':'AtomState.energy',
'RadTransWavelength':'RadTran.wavelength',
}
```

Note: Note for example the use of the names *Atom* and *AtomState* on the right-hand side of the dictionary definition. These are examples of the "loop variables" mentioned in the table above and act as shortcuts to the nested data you are storing.

6.6.1 About the RESTRICTABLES

As we have learned from writing the query function above, we can use the RESTRICTABLES to match the VAMDC dictionary names to places in our data model. The key in each key-value-pair is a name from the VAMDC dictionary and the values are the field names of the model class that you want to query primarily (Transition, in the example above, line 10).

The RESTRICTABLES example give fits our query function from above, so we know that the "main" model is the Transitions. Now if a query like "AtomIonCharge > 1" comes along, this can be translated into *Transition.objects.filter(species__ion__gt=1)* without further ado, which is exactly what *where2q()* does. Note that we here used a ForeignKey to the Species model; the values in the RESTRICTABLES need to be written from the perspective of the queried model.

Note: Even if you chose to not use the RESTRICTABLES in your setupResults() and treat the incoming queries manually, you are still encouraged to fill the keys (with the values being empty), because they are automatically provided to the VAMDC registry so that external services can figure out which names make sense to query at this node.

6.6.2 About the RETURNABLES

Equivalent to how the RESTRICTABLES take care of translating from global names to your custom data model when the query comes in, the RETURNABLES do the opposite on the way back, i.e. when the data reply is sent

by the generator, as we have already seen above.

Again the keys of the key-value-pairs are the global names from the VAMDC dictionary. The values now are their corresponding places in the QuerySets that are constructed in setupResults() above. This means that the XML generator will loop over the QuerySet, getting each element, and try to evaluate the expression that you put in the RETURNABLES.

Continuing our example from above, where the State model has a field called *energy*, so each object in the QuerySet will have that value accessible at *AtomState.energy*. Note that the first part before the dot is not the name of your model, but the *loop variable* inside the generator as it is listed in the second (or forth, in the case of an inner loop) column of the table above.

There is only one keyword that you **must** fill, all the others depend on your data. The obligatory one is *NodeID* which you should set to a short string that is unique to your node. It will be used in the internal reference keys of an XSAMS document. By including the NodeID, we make these keys globally unique within VAMDC which will facilitate the merging of data that come from different nodes.

http://dictionary.vamdc.org/returnables/ is where you can browse all the available keywords.

Note: Again, at least the keys of the RETURNABLES should be filled (even if you use your own generator for the XML output) because this allows the registry to know what kind of data your node holds before querying it.

6.7 Testing the node

Now you should have everything in place to run your node. If you still need to fill your database with the import tool, now is the time to do so according to *How to get your data into the database*.

Django comes with a built-in server for testing. You can start it with:

```
$ ./manage.py runserver
```

This will use port 8000 at your local machine which means that you should be able to browse to http://127.0.0.1:8000/tap/availability and hopefully see a positive status message.

You should also be able to run queries by accessing URLS like:

```
http://127.0.0.1:8000/tap/sync?LANG=VSS1&FORMAT=XSAMS&QUERY=SELECT ALL WHERE AtomIonCharge > 1
```

replacing the last part by whatever restriction makes sense for your data set.

Note: The URL has to be URL-encoded when testing from a script or similar. Web browsers usually do that for you. To also see the statistics headers, you can use *wget -S -O output.xml "<URL>"*.

You should run several different test queries to your node, using all the Restrictables that you defined. Make sure that the output values matches your expectations.

There is a very convenient software called **TAPvalidator** (see http://www.vamdc.org/software) which can be used to query a node, browse the output and check that it is valid with respect to the xsams standard.

Once your node does what it should do with the test server, you can start thinking about *deploying it*.

CHAPTER 7

How to get your data into the database

In the previous chapter, we have learned how to define the database layout and tell the framework to create the database accordingly. The following describes how to fill this database with data that previously resided in one or many ascii tables.

Note: There are many ways to achieve this and you are certainly free to fill the database in any way you want, if you already know how to do it.

The strategy we adopt is to use the database's own import mechanisms which are many times faster for large amounts of data than manually inserting data row by row.

The import thus becomes a two-step process:

1. create one ascii file per data model, each of which has columns that exactly will match the columns in the database.
2. run one SQL command for each of these files to load it into the matching database table.

Since you might already have step 1 finished or might be able to get it with your own data handling tools, let's have a look at step 2 first.

7.1 Loading ascii data into the database

In the following, we assume that you use MySQL as your database engine (this is also our recommendation when a new database is set up for the first time). Other engines have similar mechanisms for bulk loading data.

The mysql command we use looks like this:

```
mysql> LOAD DATA INFILE '/path/to/data.file' into table <TAB>;
```

where <TAB> is the name of the database table corresponding to the file being loaded.

Note: The table names have a prefix *node_*, i.e. the table for a model called *State* will be called *node_state*, unless you specify the table name in the model's definition. You can see a list of all tables by giving mysql the command *SHOW TABLES;*.

The LOAD DATA command has several more options and switches for setting the column delimiter, skipping header lines and the like. Mathematical or logical operations can be run on the columns too, before the data get inserted into the database.

You can read all about LOAD DATA at http://dev.mysql.com/doc/refman/5.1/en/load-data.html

A more complete example would look like:

```
mysql> LOAD DATA INFILE '/path/transitions.dat' IGNORE INTO TABLE transitions COLUMN$ TERMINATED
```

7.2 Preparing the input files

In the not so unlikely case that the data are not yet in a format exacly matching the database layout, the Node Software ships with a *rewrite tool* to convert your data into such a format. The output will be ascii files that can be loaded as described in the previous section and will fulfill the following criteria:

- One file per database table. LOAD DATA cannot update existing rows.
- Same number of columns in the file as in the table and in the right order. Although LOAD DATA can take a list of columns to circumvent this restriction, it makes sense to get this right.
- Links between the tables are in place. The key values that link tables (e.g. states and transitions) should be already in the ascii files (even though they *can* still be generated with LOAD DATA by using some SQL magic).
- A consistent delimiter between the columns (no fixed record length) and consistent quoting.
- Empty (NULL) values are written as \N, not 0 or anything else. (Can also be fixed later if this is the only thing missing)

The tool can be used to convert almost any format of file. It's easiest to convert files with its records stored as *lines* (one line per record), but the tool also supports blocks of data stretching several lines.

To use the rewrite tool, you need to tell it how your original data files are named and how they are structured. This is done in something called a *mapping file*. The mapping file describes how the rewriter should extract data from your custom text files. It will then use your data models (which you should have defined by now) to create output files in a format the database can import.

7.2.1 Starting the rewrite

Once you have defined the mapping file as described in the following section, you need to place yourself in the *imptools/* directory (this is so the rewriter can find all its dependencies) and then give the mapping file as an argument to the *imptools/run_rewrite.py* program:

```
$ python run_rewrite.py ../nodes/MyNode/mapping_mynode.py
```

The result will be a set of ascii output files on the right form.

Note: For large amounts of raw data, the rewrite operation can be very time consuming. We have found that a speed-up of as much as five times can be achieved by not using standard Python but an alternative implementation of Python called *pypy*. If installed you run the rewrite program just as above except you replace *python* with *pypy*. See http://pypy.org for further details on *pypy* usage.

7.3 The mapping file

The mapping file is a standard Python file and describes how the rewriter reads the raw data so it can be converted on the form needed for database import. *imptools/mapping_sample.py* is a minimal mapping file one can build from. A much more extensive example is found in the *nodes/ExampleNode* directory.

The mapping file must define a variable called *mapping* which contains a list of definitions that describe how the rewriter should parse each text file and correlate the data to the data models.

Let's start a sample mapping file. It starts by defining some convenient variables storing input/output filenames (just to make it easier to refer to them further down). We also include *imptools/linefuncs.py* which holds helper methods for parsing data. The only mandatory part is the *mapping* list:

```
from imptools.linefuncs import *

# the names of the input files
basepath = "/path/to/your/raw_data/"
```

```
outpath = "/path/to/store/rewritten/files/"
file1 = basepath + 'raw_file1.txt'
file2 = basepath + 'raw_file2.txt'
file3 = basepath + 'raw_file3.txt'
outfile1 = outpath + 'references.dat'
outfile2 = outpath + 'species.dat'

mapping = [ ... ]  # described below
```

7.3.1 The `mapping` list

The `mapping` variable is a list of Python *dictionaries*. A standard python dictionary is written as `{key:value, key2:value2, ... }` and is a very efficient means of storing data. Each of the dictionaries in `mapping` describes how to output data to exactly one output file and thus correspons to one database table (described by a model in *node/models.py*). It can use any number of raw input data files to get this data.

Each mapping is executed in parallel, using multiple processors if available. This means that you should not have any mapping write to the same output file as any other mapping. For example, one could be tempted to have two mappings both write to an outfile *states.dat*, writing upper- and lower- states into the file respectively. Due to the parallel operation, this will likely lead to file lock clashes. Output instead to two files (e.g. *states_upper.dat* and *states_lower.dat*) and read them separately into the same table later.

Only certain key names are allowed in each mapping dictionary. One of these keys, *linemap* holds a list with further dictionaries since it details exactly how to read each line/block of data from the input. The structure of the `mapping` variable looks like this:

```
mapping = [
   {key : value,
    key : value,
    ...
    linemap : [
        {linemap_key : value,
         linemap_key : value},
        {linemap_key : value,
         linemap_key : value}] },
   {...},
   {...},
    ...
   ]
```

And so on, continuing with more dictionaries. The *key* s and *value* s of each dictionary describes all aspects of the parsing, although not all options are mandatory depending on your structure.

key	value
Mandatory	
outfile	The name of the file that should be created. Each such output file will later be read into one database table/ model.
infiles	Input file(s). This may be a single file name or a list of multiple file names. More than one file may be relevant if the raw data is stored in multiple files related to each other by line number only.
linemap	A list of dictionaries defining how to parse each line/block of the file(s) into its components (see the next table below for the keys relevant when defining the linemap list)
Optional	
headlines	Number of header lines at the top of the input file(s) (default: 0). If more than one infile is used, this must be a list of headlines in the same order, as many as there are input files.
commentchar	Which comment symbol is used in the input file(s) to indicate a line to ignore (default is: '#'). As above, this must be a list if more than one filename is used.
errlines	Whole lines in the input file(s) that should be considered non-valid and ignored (no default). As above, this must be a list if more than one filename is read.
linestep	A step length (in number of lines) when reading the input file. Default (0) means stepping one line at a time. A linestep of 1 means skipping every other line. If more than one file is read at a time, this must be a list of the same length as there are files. So a lineoffset of [0,2] would mean that while every line is read in the first file, only every third is used in the second file.
lineoffset	A starting offset when reading a file, after headers have been skipped. So a lineoffset of 3 would first skip the header (if any), then another 3 lines. This is most useful in combination with linestep, to make sure the first line of data is read from the right start point. If many files are read, this must be given as a list of offsets, as many as there are files.
startblock	This is a string or a list of strings to be interpreted as *starting sentinels* for data records stretching over more than one line. So if every data block is wrapped in BEGIN ... END clauses, you should put "BEGIN" here. (default is the line break character, making each "block" equivalent to a line). The variables *linestep* and *lineoffset* will step through full blocks instead of lines if this is given.
endblock	This is a string or list of strings to be interpreted as *ending sentinels* for data records stretching over more than one line. So if every data block is wrapped in BEGIN ... END clauses, you should put "END" here. (default is the line break character, making each "block" equivalent to a line). If blocks are only separated by a single sentinel (e.g. ... RECORD ... RECORD ...), simply put the same sentinel ("RECORD" in this example) as both startblock and endblock.

A note about reading multiple files at the same time: The only main use for this is really if your raw data is related to data in other files by *record number only* (i.e. by counting line number or maybe block number). If you cannot use line numbers since you use, say, an ID string to relate data in one file to that in another, you cannot correlate them to each other this way. You should then instead read the files as separate reads. Exactly how the read will looks depend on your planned database layout and the models you need to populate. */nodes/vald/mapping_vald3.py* contains an advanced example of reading upper and lower atomic States from a file in two passes, using ID hashes to relate them to a second model (Transitions).

The *linemap* key points to another list with dictionaries. This is the actual operating piece of code and describes exactly how to parse each line or block (or lines/blocks, if more than one input file is read simultaneously). Each dictionary works for a single database field in your current model (that is, the model your output file will be read to down the line) and describes exactly how to parse the current line/block so as to produce a value in that field.

linemap_key	value
Mandatory	
cname	The name of the field in your database model to populate.
cbyte	A tuple `(linefunction, arguments)`. This names a function capable of parsing the line(s) to produce the data needed to feed to the field *cname*. The only provision of a linefunction is that it should take an argument *linedata* as its first argument. This will contain the current line/block to parse, or a list of lines/blocks if more than one input file were read simultaneously. You can define your own linefunctions directly in the mapping file. A host of commonly needed line functions (such as reading a particular index range or the Nth separated section etc) come with the package and can be used directly by importing from *imptools/linefuncs.py*.
Optional	
filenum	This is an integer or a list of integers used only when more than one file is read simultaneously. It allows you to specify the index/indices of the file/files to be parsed. Default is file 0. Note: If you need to somehow merge data from two or more files to produce one value, you need to write a custom line function for this and then use this setting to specify which file(s) should be used.
cnull	Indicates what should be interpreted as NULL data. If this string is found, the *N* symbol will be stored in the output file instead.
debug	This will activate verbose error messages for this parsing only. Useful for finding problems with the mapping.

Continuing our example, here's how this could look in the mapping file (the line breaks are technically not needed, but make things easier to read). Note also that we imported linefuncs.py earlier, making the line functions *bySepNr* and *charrange* available (among many others):

```
mapping = [
  # first dictionary, writing into outfile1 (defined above) from an
  # input file file1.
  {
    'outfile': outfile1,
    'infiles': file1,
    'headlines' : 3,
    'commentchar' : '#',
    'linemap' : [
        {'cname':'dbref',
         'cbyte':(bySepNr, 0, '||')}, # get 0th part of record separated by ||
        {'cname':'author',
         'cbyte':(bySepNr, 1, '||')}, # get 1st part of record separated by ||
            # ...
                ]
  }
  # next model dictionary, writing species.dat
  {
    'outfile' : outfile2,
    'infiles' : (file2, file3), # using more than one file!
    'commentchar' : (';', '#'),
    'headliens' : (1, 3),
    'lineoffset' : (0, 1),
    'linemap' : [
       {'cname':'pk',
        'cbyte':(charrange, 23, 25)}, # pick a range by index
       {'cname':'mass',
        'cbyte'(charrange, 45, 45, 1)}, # retrieved from file3!
          # ...
       {'cname':'source',
        'filenum':1,     # read from current line of second file!
        'cbyte':(charrange, 0, 10),
                ]
     }]
```

7.3.2 The line functions

Since the mapping file is a normal Python module, you are free to code your own line functions to extract the data from each line/block in your file. There are only three requirements for how a line function may look:

- The function must take at least one argument, which will hold the current line or block being processed, as a string. The import program will automatically send this to the function as it steps through the file. If you read multiple input files *and* supplied multiple *linenum* values in the mapping, this first argument will be a list with the corresponding lines/blocks. It's up to the custom function to handle this list properly.
- The function must return its extracted piece of data in a format suitable for the field it is to be stored in. So a function parsing data for a CharField should return strings, whereas one parsing for an IntegerField should return integer values.

Below is a simple example of a line function:

```
def charrange(linedata, start, end):
    """
    Simple extractor that cuts out part of a line
    based on string index.
    """
    return linedata[start:end].strip()
```

In the mapping dictionary we will call this with e.g. `'cbyte' : (charrange, 12, 17)`. The first element of the tuple is the function object, everything else will be fed to the function as arguments. The function should return the string to store.

The default line functions coming with the package will handle most common use cases. Just `import linefuncs *` from your mapping file to make them available. You can find more info in the *Linefuncs Documentation*.

More advanced line parsing

Sometimes you need more advanced parsing. Say for example that you need to parse two different sections of lines from one or more files and combine them into a unique identifier that you will then use as a key for connecting your model to another via a One-to-Many relationship. Or maybe you want to put a value in different fields depending on if they are bigger/smaller than a certain value. There is no way for the default line functions in *linefuncs.py* to account for all possibilities.

The solution is to write your own line function. You have the full power of Python at your command. Often you can use the default functions as "building blocks", linking them together to get what you want. Just code your custom line functions directly in the mapping file.

The mapping file will skip lines/blocks starting with the *commentchar* character or containing data matching the *errorline* key value. But sometimes you don't have enough information to know if the line/block should be skipped. You can then analyze this in your custom line function. If there is a problem raise *RuntimeError* - the import system will then cleanly skip that line/block for you.

Here is an example of a line function that wants to create a unique id by parsing different parts of lines from different files:

```
from imptools.linefuncs import *

def get_id_from_line(linedata, sepnr, index1, index2):
    """
    extracts id from several lines.
      sepnr - nth separator to pick from file 1
      index1, index2 - indices marking start/end index from file 2

      (file3 is (in this example) always used the same way,
       so we hard-code the indices for that file.)
    """
    l1 = bySepNr(linedata[0], sepnr, ',')
```

```
    l2 = charrange(linedata[1], index1, index2)
    l3 = charrange(linedata[2], 0, 3)
    if l3 == '000':
        l3 = 'unknown'
    # create unique id
    return "%s-%s-%s" % (l1, l2, l3)
```

Here we made use of the default line functions as building blocks to build a complex parsing using three different files. We also do some checking to replace data on the spot. The end result is a string combined from all sources.

This function assumes linedata is a list. It must thus be called from a mapping where at least three files are read (*inputfiles* is a list of at least three file names) and where *filenum* is given as a list specifying which files' lines/blocks are to be sent to the function. The the mapping dictionary could look something like this:

```
...
   {'outfile':outfile1,
    'infiles': [file1,file2,file3],
    'linemap': [
         {cname:'myidfield',
         filenum = (0,1,2)
         cbyte: (get_id_from_line, 3, 25, 29)},
         ...
               ]
  }
```

See *nodes/ExampleNode* for more examples of mappings and linefuncs.

CHAPTER 8

How to update an existing database

As long as your database schema has not changed, you can use this same rewrite mechanism to append new data to your database. Just run the rewriter on your new raw data, then use the *LOAD DATA INFILE* (MySQL) or equivalent again to import it into your database.

An important limitation of *LOAD DATA INFILE* is that it will not change already existing rows. So you cannot update data in-place with this method (it is also not the purpose of this import system).

For altering existing rows in the database, the standard SQL-command *UPDATE TABLE* will do the trick in most cases.

Adding data in the form of new columns to existing tables, can be done as follows. Add the empty column using SQL *ALTER TABLE*, fill it with *UPDATE TABLE* and then add the corresponding field definition in your `models.py` and `dictionaries.py` to make the NodeSoftware aware of it.

The underlying Django system comes with many third-party tools for helping you manage your database however. We recommend you look into Django-South (http://south.aeracode.org/). This Django-plugin allows you to write simple "migration" scripts for updating an existing database schema or do data conversions between different versions of a live database.

CHAPTER 9

Deployment of your node

Now that you have a node that runs nicely with Django's test server, the last remaining step is to configure the server that will run the node in a production setup.

How and on which server you set up your node to run permanently, is much dependent on your technical resources and the solution we give here is just one out of several possibilities (although we also quickly mention the most common alternative).

9.1 Gunicorn plus proxy

Our recommended way for hosting your node by yourself on a server is Gunicorn (http://gunicorn.org/, *apt-get install gunicorn* on a Debian system) which is aware of Django and understands its settings.

You would write a *gunicorn.conf* file (you find it in *nodes/ExampleNode*) like this:

```
import os
def numCPUs():
    if not hasattr(os, "sysconf"):
        raise RuntimeError("No sysconf detected.")
    return os.sysconf("SC_NPROCESSORS_ONLN")
workers = numCPUs() * 2 + 1

#bind = "127.0.0.1:8000"
bind = "unix:/tmp/gunicorn.sock"
pidfile = "/tmp/gunicorn.pid"
logfile = "/tmp/gunicorn.log"
loglevel = "info"
timeout = 60
daemon = True
```

and then simply start it from within your node directory with:

```
$ gunicorn_django -c gunicorn.conf
```

The example config makes Gunicorn listen at a unix-socket. Even though you can connect it to a TCP-port instead (see commented out line), you do not want external requests sent directly to Gunicorn, but to a proxy instead. This proxy takes care of the load balancing between the Gunicorn worker processes and can compress the XML output from your node before sending it.

9.1.1 Nginx as proxy

Nginx (http://nginx.org/en/, *apt-get install nginx* on a Debian system) is a fast and light-weight web server. To configure it to serve the running node with Gunicorn, according to the example above, you would configure it like this:

```
upstream app_server {
        server unix:/tmp/gunicorn.sock;
    }

server {
        listen   8080; ## listen for ipv4
        listen   [::]:8080 default ipv6only=on; ## listen for ipv6
        server_name your.server.domain.name;
        access_log  /var/log/nginx/vamdc.access.log;

        location /yournode/tap/ {
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header Host $http_host/yournode;
            proxy_pass http://app_server/tap/;
            proxy_redirect http://app_server/tap/ /yournode/tap/;

            gzip on;
            gzip_types       text/plain application/xml text/xml;
            gzip_proxied any;
        }
}
```

Note that you probably want to edit the port, server name and the location at which to serve the node (change */yournode/tap* at three places but make them match each other).

If you installed *nginx* with the debian/ubuntu package, you can place symbolic links to the config file into */etc/nginx/* like this to make it use the config above:

```
$ cd /etc/nginx/sites-available/
$ sudo ln -s $VAMDCROOT/nodes/YourNode/nginx.conf vamdcnode
$ cd ../sites-enabled/
$ sudo ln -s ../sites-available/vamdcnode
$ sudo /etc/init.d/nginx restart
```

9.1.2 Proxy Alternatives

What you choose as proxy for Gunicorn is somewhat arbitrary. Common alternatives to *nginx* are *lighttpd* or *Apache*. Especially if the server that is to run your node already has an Apache running for serving other websites, it makes sense to simply tell it how to proxy your Gunicorn server:

```
ProxyPass /yournode http://localhost:8000
ProxyPassReverse /yournode http://localhost:8000
```

9.2 Deployment in Apache

As an alternative to deplyment with Gunicorn plus proxy, the Apache webserver can not only act as a proxy but also replace Gunicorn by using its mod_wsgi plugin to run the Python code directly. The main disadvantage of this setup is that you cannot configure and restart the node independantly from Apache, so the likelyhood of interfering with any other sites that Apache serves is larger.

There are two example files in your node directory for setting this up:

- *apache.conf*: This is an Apache config file that defines a virtual server, bound to a certain host name. You will have to edit several things in that file before it will work in Apache: the server name and the path to the node software in a few places. On a Debian-like system you would then move this file to */etc/apache2/sites-available/vamdcnode* and run *a2ensite vamdcnode* to activate it.
- *django.wsgi*: This is the file that the previous one points to in its WsgiScriptAlias. Edit the path and your node's name.

Once you have set this up and re-started the Apache webserver, your node should deliver data at the configured URL.

9.3 Third party hosting

There are several upcoming hosting solutions that support Django directly so that you simply would upload the code and your database and everything is taken care of for you. Once these services mature, they are probably a very good solution for nodes with relatively small volumes of data.

Searching the web for "django hosting" will point you in the right direction, as does this list https://convore.com/django-community/django-hosting-explosion/

9.4 Logging

Finally, a few words on logging the access to your node. There are two basic ways:

- let the webserver do it.
- let the NodeSoftware do it.

The webserver/proxy, be it nginx or apache, keeps a log on when, how and by whom your node is accessed. Since the query itself is in the accessed URL, it also ends up in these logs. There are many tools to analyze and visualize this kind of logs. In the case of Apache/WSGI-deployment, errors in the NodeSoftware show up the webservers error-log since it is the former that executes the latter. With gunicorn, the webserver knows nothing about the NodeSoftware's errors since it only acts as a proxy. Gunicorn keeps its own logs.

However, the webserver logs usually contain no information about what happened inside the NodeSoftware. If you want to keep tabs on how much data was returned from each query, how long it took to process and so on, you need to tell the NodeSoftware to save this information for you - this is where the *logging*-facility comes into play.

Nodes will primarily use this in their `queryfunc.py` where you initialize it like this:

```
import logging
>>> log = logging.getLogger('vamdc.node.queryfu')
```

Then any of the following can be used to log messages of different levels:

```
>>> log.debug('some text with a variable: %s'%variable)
>>> log.info('bla')
>>> log.warning('bla')
>>> log.error('bla')
>>> log.critical('bla')
```

Where these messages end up is configured in `settings_default.py` and you can as usual override the default in the node's own `settings.py`. For example, you set the location and name of the log-file like this:

```
LOGGING['handlers']['logfile']['filename'] = '/path/to/yourlog.log'
```

Note: Critical errors (using *log.critical()*) are sent to the configured admin email address. You need to supply a valid address and make sure your server can send emails. The email address that these messages are sent from can bet set with `SERVER_EMAIL='vamdcnode@your.server'`.

If you want to turn off the logging of debug messages, you can add the following for turning them on or off, depending on your global DEBUG setting:

```
if not DEBUG:
    LOGGING['handlers']['logfile']['level'] = 'INFO'
```

For further information see https://docs.djangoproject.com/en/1.3/topics/logging/

CHAPTER 10

Miscellaneous

There are a few more bits and pieces that are both good to know and maybe necessary for a particular node setup.

10.1 Setting the deployment URL

The NodeSoftware tries to automatically find out the URL with which it is accessed and uses this to fill the URL-information in */tap/capabilities*, among other things. However, this does not always work (e.g. if you deploy behind a proxy) so there is a manual override. Simply set *DEPLOY_URL* in `settings.py`, ending with */tap/* like this:

```
DEPLOY_URL = 'http://your.server/some/path/tap/'
```

10.2 Filling the IDs

As you know, XSAMS is a hierarchical structure where certain parts reference other parts. For example, each (molecular or atomic) state has an ID, which can be used by a radiative transition to point to its initial and final states. Similarly, all species, bibliographic sources etc. have an ID that other parts use to point to them.

Here is a list of the most important Returnable names for IDs:

- **AtomSpeciesID** uniquely identifies an atomic spieces. Different isotopes and ions are considered different species.
- **AtomStateID** is the ID for the states within an atomic species.
- **CrossSectionID** identifies radiative crosssections.
- **EnvironmentID** identifies environments.
- **FunctionID** numbers functions.
- **MethodID** is for the defined methods.
- **MoleculeSpeciesID** identifies molecular species. As for atoms, different isotopologues are considered to be separate species.
- **MoleculeStateID**
- **ParticleSpeciesID** identifies particles.
- **SolidSpeciesID** identifies solids.
- **SourceID** identifies the bibliographical sources and is used in many places of the schema to connect data to its origin.

NodeID is "special" in the sense that it is not formally part of the schema. The XML generator uses it to make all the other IDs unique within VAMDC. Say, for example that you (in `dictionaries.py`) set your NodeID to "xyz" and fill the SourceID with numbers from your database. Then the XML output will look something like

<Source sourceID="Bxyz-1"> for your first source. This means that the generator takes care of adding the prefix "B" as mandated for sourceIDs by the schema, plus it inserts the NodeID to prevent clashed with IDs from other VAMDC nodes.

IDs are mandatory which means that you *have* to fill the Returnables from the list above, if you use the corresponding part of the schema.

Ideally the node's database layout roughly matches the XSAMS structure which means for example that you have separate tables for the atoms/molecules and their states. The linking indexes between the tables (usually an integer) are then directly suited to be used as the IDs above because the generator formats it as described.

In order to do this, it is good to be aware of the following Djangoism: Consider the example data model from *here* and that *s* is an instance of the *State* model. Then *s.energy* gives the value of the energy column in the database, as you expect. *s.species* however is, contrary to non-ForeignKey fields, not the key value of the corresponding species, but the actual instance of the species model because Django tries to be smart and convenient. Now we could use *s.species.id* to get the key value, but this would be slow since we would unnecessarily traverse into the species table to get it. The better way is to use *s.species_id* which is provided automatically, i.e. **for any ForeignKey field xyz there is a field xyz_id which holds the key value instead of the linked object.**

10.3 Using a custom model method for filling a Returnable

Sometimes it is necessary to do something with your data before returning them and then it is not possible to directly use the field name in the right-hand-side of the Returnable. Now remember that the string there simply gets evaluated and that your models can not only have fields but also custom methods. Therefore the easiest solution is to write a small method in your class that returns what you want, and then call this function though the returnable.

For example, assume you for some reason have two energies for your states and want them both returned into the Returnable *AtomStateEnergy* which can handle vectors as input. Then, in your `models.py`, you do:

```
class State(Model):
    energy1 = FloatField()
    energy2 = FloatField()

    def bothenergies(self):
        return [self.energy1, self.energy2]
```

And correspondingly in your RETURNABLES in `dictionaries.py`:

```
RETURNABLES = {\
    ...
    'AtomStateEnergy':'AtomState.bothenergies()',
    }
```

Note: Use this sparingly since it adds some overhead. For doing simple calculations like unit conversions it is usually better to do them once and for all in the database, instead of doing them for every query.

10.4 Handling the Requestables better

The XML generator is aware of the Requestables and it only returns the parts of the schema that are wanted. Therefore the nodes need in principle not care about this. However, there are two issues that can interfere:

- If a node imposes volume limitations, this can lead to false results. For example in a transition database, when a client asks for "SELECT SPECIES" without any restriction then a node's query function usually finds out the species for a set of transitions, which gets truncated to the volume limit, then only the species for the first few transitions in the database are returned.

- Again taking "SELECT SPECIES" as example, this can lead to performance issues if a node's query stategy is to impose the restrictions onto the most numerous model fist, since this query then corresponds to selecting everything and afterwards throwing everything away except the species information.

The solution is to make the queryfunction aware of the Returnables. These are attached to the object **sql** that comes as input. For example, one can test if the setup of atomic states is needed like this:

```
needAtomStates = not sql.requestables or 'atomstates' in sql.requestables
```

and then use the boolean variable **needAtomStates** to skip parts of the QuerySet building. This test checks first, if we have requestables at all (otherwise "ALL" is default) and then whether 'atomstates' is one of them.

Note: The query parser tries to be smart and adds the Requestables that are implied by another one. For example it adds 'atomstates' and 'moleculestates' when the client asks for 'states'. Therefore it is enough to test for the most explicit one in the query functions.

Note: The keywords in **sql.requestables** are all lower-case!

10.5 Setting the related name of a field

When you have a *ForeignKey* called *key1* in a *ModelB* which points *ModelA*, the fields from *ModelA* become accessible by *b.key1.fieldFromModelA* in a selection *b* of *ModelB*. This is using the ForeignKey in **forward direction**.

Django also automatically adds a field to *ModelA* that contains all the instances of *ModelB* that point to a specific instance *a* of *ModelA*. This field is by default called as the referenced model plus *_set*. So *a.modelb_set* would hold all the ModelBs that reference *a*. This is using the ForeignKey in **inverse direction**.

You can change the name of the inverse field by giving the argument *related_name='bla'* to the definition of the ForeignKey in the model. When you have more than one ForeignKey from one model to the same other model, you **must** set the related_name because the automatic naming cannot give the same name twice.

A typical example for this are the upper and lower states for a transition where it makes sense to have two ForeignKeys in the Transition model, e.g. called *upstate* and *lostate*, each pointing to an entry in the State model. Now one sets the related_names of these ForeignKeys to something like *'transitions_with_this_upstate'* and *'transitions_with_this_lostate'* respectively. Thereby, for any state *s* the transitions that have *s* as upper state can be retrieved by *s.transitions_with_this_upstate*.

10.6 Inserting custom XML into the generator

There can arise situations where it might be easier for a node to create a piece of XML itself than filling the Returnable and letting the generator handle this. This is allowed and the generator checks every time it loops over an object, if the loop variable, e.g. *AtomState* has an attribute called *XML*. If so, it returns *AtomState.XML()* instead of trying to extract the values from the Retunable for the current block of XSAMS. Note the *execution* of *.XML()* which means that this needs to be coded as a function/method in your model, not as an attribute.

10.7 Quick debugging and testing

Sometimes it is necessary to go manually go though the steps that happen when a query comes in in order to find out where omething goes wrong. A good tool for this is in interactive python session which you start from within your node directory with:

```
./manage.py shell
```

From within the Python shell, you can run:

```
# import the relevant part of the NodeSoftware
from vamdctap import views as V
# import your queryfunction
from node import queryfunc as Q
# set up a query
foo = {'LANG':'VSS2','FORMAT':'XSAMS',
    'QUERY':'select all where radtranswavelength < 1000 and radtranswavelength > 900'}
# run the parser
foo = V.TAPQUERY(foo)
# check basic validity
print foo.isvalid
...
# look at the parsed where clause
print foo.where
# put it into your query function and see what happens
Q.setupResults(foo)
```

You can also manually run the first step from the queryfunction:

```
from vamdctap import sqlparse as S
q = S.sql2Q(foo)
print q
```

10.8 Unit conversions for Restrictables

It is possible in `dictionaries.py` to apply a function to the values that come in the WHERE-clause of a query together with the Restrictables:

```
from vamdctap.unitconv import *
RESTRICTABLES = {\
'RadTransWavelength':'wave',
'RadTransWavenumber':('wave',invcm2Angstr),
...
```

Here we give a two-tuple as the right-hand-side of the Restrictable *RadTransWavenumber* where the first element is the name of the model field (as usual) and the second is the function that is to be applied.

Note: The second part of the tuple needs to be the function itself, not its name as a string. This allows you to write custom functions in the same file, just above where you use them.

Note: The common functions for unit conversion reside in `vamdctap/unitconv.py`. This set is far from complete and you are welcome to ask for additions that you need.

10.9 Treating a Restrictable as a special case

Perhaps a unit conversion (see above) is not enough to handle a Restrictable, e. g. because you do not have the quantity available in your database but know it anyway. Suppose a database has information on one atom only, say iron. For the output one would simply hardcode the information on iron in the Returnables as constant strings. For the query on the other hand, you would like to support AtomSymbol but have no field in your database to check against - after all it would be wasteful to have a database column that is the same everywhere.

10.9.1 Custom restrictable function

One way of handling this is to use a custom function as the value of the Restrictable in `dictionaries.py`:

```
'AtomSymbol':checkIron,
```

where *checkIron* would be a function, e.g. defined in the same file (before referencing it, of course) as:

```
def checkIron(restrictable,operator,value):
    value = string.strip('\'"')
    if value == 'Fe' and operator in ('=','=='):
        return return Q(pk=F('pk'))
    else:
        return ~Q(pk=F('pk'))
```

Note: *Q(pk=F('pk'))* is a restriction that is always true and should be fast. The operator ~ negates it.

Note: This (and the alternative below) do not cover all possible query cases, for example the operators LIKE or IN. In practice, some more lines of code will therefore be needed to manually handle a Restrictable.

Note: If this topic is relevant for you, please also have a look into `vamdctap/unitconv.py` where there are some examples.

10.9.2 Manipulatine the query

Another solution is to manipulate the set of restrictions by hand instead of letting *sql2Q()* handle it automatically. *sql2Q()* is a shorthand function that does these steps after each other:

1. Use *splitWhere(sql.where)* to split the WHERE statement in two:
   - a structure that represents the logical structure of the query.
   - a dictionary with numbers as keys and a list as values that each contain the Restrictable, the operator and the arument(s).
   - For example, the query *SELECT ALL WHERE RadTranswavelenth > 3000 and RadTranswavelenth < 3100 and (AtomSymbol = 'Fe' OR AtomSymbol = 'Mg')* would return the two variables like
   - *['r0', 'and', 'r1', 'and', '(', 'r2', 'or', 'r3', ')']*
   - *{'1': [u'RadTranswavelength', '<', u'3100'], '0': [u'RadTranswavelength', '>', u'3000'], '3': [u'AtomSymbol', '=', u"'Mg"'], '2': [u'AtomSymbol', '=', u"'Fe"']}*
2. Go through the Restrictables and apply the unit conversion functions that were specified with the mechanism above.
3. Make use of the information in `dictionaries.py` to rewrite the restrictions into the native field names, in the form of Django Q-objects.
4. Merge the individual restrictions together with their logic connection again and evaluate the whole shebang.

So, in summary, the call *q=sql2Q(sql)* at the start of the query function can be replaced by:

```
logic,restrictions,count = splitWhere(sql.where)
q_dict = {}
for i,restriction in restrictions.items():
    restriction = applyRestrictFu(restriction)
    q_dict[i] = restriction2Q(restriction)
q = mergeQwithLogic(q_dict, logic)
```

Now, depending on what you want to do, you can manipulate this process at any intermediate step. To continue the example with iron only, we could insert the following at the start of the loop over the restrictions:

```
if restriction[0].lower() == 'atomsymbol':
    if restriction[1] in ('=','==):
        if restriction[3] == 'Fe':
```

```
            q_dict[i] = Q(pk=F('pk'))
            continue
```

10.10 How to skip the XSAMS generator and return a custom format

Currently, only queries with *FORMAT=XSAMS* are officially supported. Since some nodes wanted to be able to return other formats (that are only useful for their community, for example to inculde binary data like an image of a molecule) there is a mechanism to to do this.

Whenever *FORMAT* is something else than *XSAMS*, the NodeSoftware checks whether there is a function called *returnResults()* in a node's `queryfunc.py`. If so, it completely hands the responsibility to assemble the output to this function.

Note: This means that you have to return a HttpResponse object from it and know a little more about Django views. In addition you are on your own to assembe your custom data format.

10.11 Making more use of Django

Django offers a plethora of features that we do not use for the purpose of a bare VAMDC node but that might be useful for adding custom funcitonality. For example you could:

- Use the included **admin-interface** to browse and manipulate the content of your database.
- Add a custom query form that is suited specifically for the most common use case of your data.
- Add a web-browsable view of your data.

For more information on all this have a look into Django's excellect documentation at https://docs.djangoproject.com/

For extending your node beyond the VAMDC-TAP interface, you would normally add a second *app* to your node directory, besides the existing one called *node*. Then you simply tell your `urls.py` to serve the new app at a certain URL.

CHAPTER 11

Known limitations

In general, the NodeSoftware tries to be forgiving with faulty input data from the nodes' databases and will do its best to return a valid and complete XML document. However it relies on the content of the connected dabase and the connection to the schema via the models and dictionaries. Errors in these cannot be compensated by the software itself and can result in invalid output data. All nodes are encouraged to check the validity of their XML output against the current XSAMS, for example with the help of the TAPValidator application.

The NodeSoftware does and will not offer the full possibilities of the XSAMS since choices and simplifications have to be made in the implementation. These deliberate limitations include:

- Treating isotopes and ions of atoms as different species, repeating the element information instead of nesting several ions within each isotope, and nesting the ions within each element.
- Only allowing one set of quantum numbers per atomic or molecular state. If a node wishes to return several different descriptions of the quantum numbers per state, this needs to be implemented in a custom fashion for this node.
- Only one set of line broadening parameters per transtion and per type (instrumental, natural, pressure, doppler) is allowed at this time. The next release of the software will include the possibility to give several pressure-broadenings per transition.

A full list of outstanding issues is available at the development repository at https://github.com/VAMDC/NodeSoftware/issues where anybody is welcome to file bugs or wishlist-items.

CHAPTER 12

Bugs and Contact

12.1 Report a bug

The NodeSoftware is in active development and there are some rough edges still. We very much appreciate your feedback and complaints are usually resolved quickly.

Please file an issue at https://github.com/VAMDC/NodeSoftware/issues

This can be used for bugs on both, the software itself and the documentation.

12.2 Contact information

You can write to the VAMDC developers email list: vamdc.developer <AT> sympa.obspm.fr

CHAPTER 13

The Code

You can download the NodeSoftware as *tar.gz* at these locations:

- Release branch: https://github.com/VAMDC/NodeSoftware/tarball/release
- Latest development branch: https://github.com/VAMDC/NodeSoftware/tarball/master

Please see *Upgrading*.

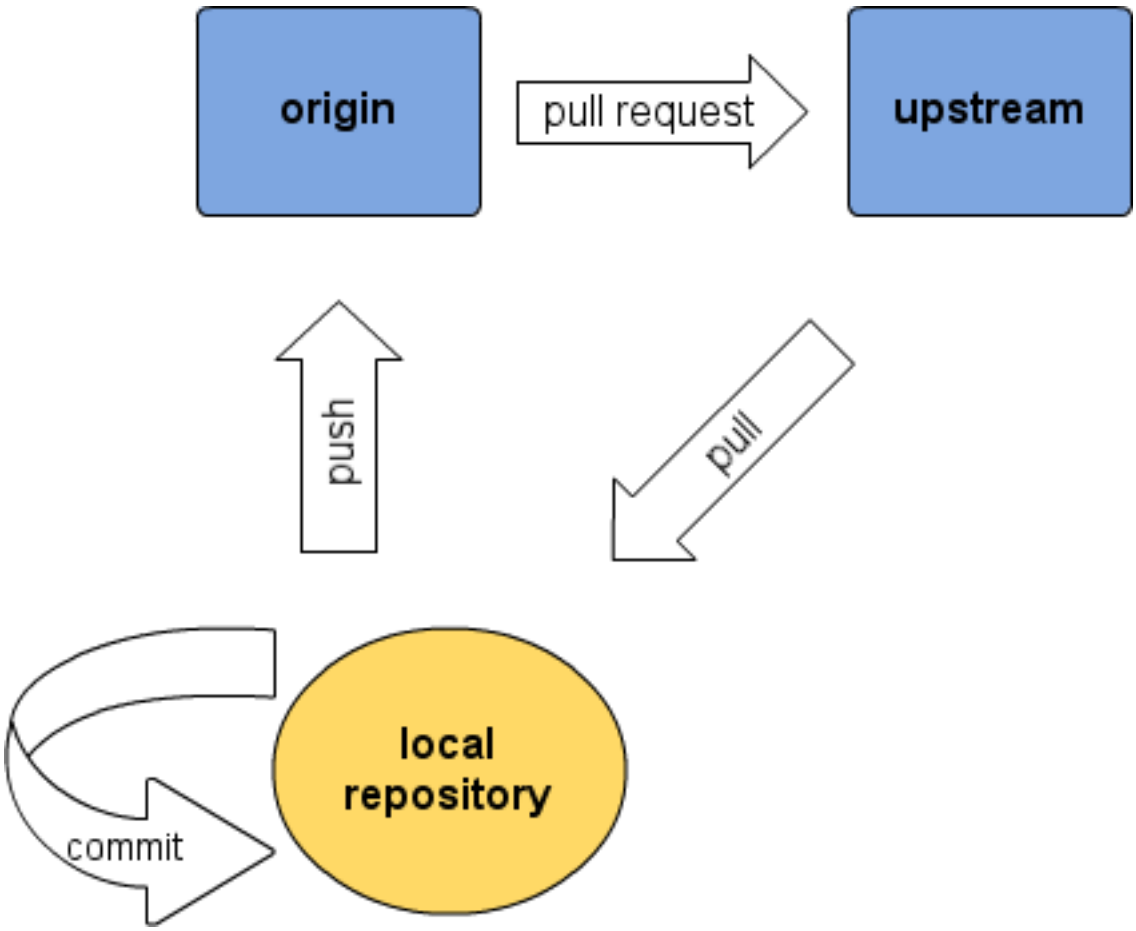
The development repository resides at https://github.com/VAMDC/NodeSoftware and you are welcome to use the version control software *git* to check out your own copy. This takes a few more minutes to set up but has the benefit of facilitating collaboration. After all, you might makes changes or extend the code for your needs and we would like to include your improvements into the main repository.

13.1 Collaborating with git and GitHub

Git is a decentralized version control system (http://git-scm.com/). This means among other things that:

- Each checked out copy of the code has the full version history.
- There is no central repository, all repositories ("repos") are equal (but some *can* be made more equal than others, as we'll see below).
- Commits happen locally into your working repo, no network connection needed.
- Repos are updated and synced with each other by pushing and pulling commits back and forth between them.
- There are web-platforms that offer free web-repositories which facilitates syncing and merging. We'll use *GitHub* (http://www.github.com/).

The setup that we want looks like this:



- The **local repository** (also known as your "working copy") is your own workspace. This is where you do all your work. It offers you full local version control without necessarily having to upload the changes anywhere. We'll get to how you create your local repo in a minute.
- Your **origin** is an online version of your repository, stored online at GitHub. When you want to sync the two you need to *push* your latest local changes to origin. Once online, others will also be able to see the changes.
- **Upstream** is a unique repository that serves as an online code "central" managed by VAMDC. It too is hosted on GitHub. Upstream serves as a convenient way to update your distribution; you should regularly *pull* the latest changes into your local repo to stay updated. Conversely, if you want your own changes to be incoorperated into the central distribution you can send a *pull request* to upstream. The relevant commit(s) in your **origin** repo will be reviewed and will, if accepted, be merged into upstream so that others will get the changes next time they do a pull.
- You can certainly have **several local repositories**, e.g. one on your laptop, one on your desktop and one on the server where the node runs. You then use the online **origin** repository to keep them in sync. For example: You work from your laptop and commit your changes locally. You then push them to your origin repository. Next all you need to do is to tell your other local repos to pull from origin and they will all be synced.

Now enough with theory, let's do this in practice. To create your own repositories (origin and local) do the following:

- Go to http://github.com and make an account. This includes that you (create and) upload an ssh-key to be able to pull and push securely and without typing your password all the time. Simply follow the instructions on GitHub.
- Visit the repository at https://github.com/VAMDC/NodeSoftware and klick "fork" in the upper right corner. This will make a copy of the original repository under your account. This is your **origin** (see above). For more information on forking, you can read http://help.github.com/forking/.

- Github will give you instructions on how to *clone* your origin to your own computer, thereby creating a local repo, your **local repository**, aka your "working copy".
- You can repeat the cloning on as many machines as you see fit.
- Tell your local repos where **upstream** is by running the following command in each of them: *git remote add upstream git://github.com/VAMDC/NodeSoftware.git*

Now that you are all set, a typical working session may look like this:

```
$ cd $VAMDCROOT               # got to your local repo
$ git status                  # should tell you you have a clean tree and are on the branch "maste
$ git pull origin             # pull from your origin, in case you pushed things there from anothe
$ git pull upstream           # fetch the latest from upstream and merge it with your tree.
$ git log                     # read the commit log about what is new.
$ ....                     # edit your files
$ git status                  # review which files have changed
$ git diff                    # review details of your changes
$ git diff <filename>         # see canges in one file only
$ git add <filename>          # add a file to be commited with the next commit, e.g. a new file
$ git commit -a -m "message"  # commit all changed files. ALWAYS check the status before you use
$ git commit -m "message" <filenames>   # commit, but include only the named files in the commit
$ ....                    # more edits, more commits. until, at the end of day:
$ git status                  # also tells you how many commits you are ahead of your origin
$ git push                    # push all commits to your origin, also the new ones that came from
```

Note: There are several graphical user interfaces available for git that will facilitate overview and some operations for the less command-line adept. Commonly used ones for Linux are *gitk* and *gitg*. Good editors also integrate with git so that you can handle the version control from within the editor.

After you pushed your work to your origin, you can go to the *GitHub* wesite and send a *pull request* to the upstream repository, if you want your changes to be propagated to everybody else. We will then look at your commits and merge them.

A few dos and don'ts that are worthwhile to keep in mind with git:

- Do commit often. It goes instantly.
- Pull and push less often, but often enough. You certainly want to pull from upstream before making changes, since you otherwise might work on outdated versions of files which will result in conflicts later. You also do no want to sit on your local commits for too long but push them frequently instead.
- Never pull into a dirty tree (i.e. one that has uncommitted changes). Commit first, then pull. Alternatively read *git help stash*.
- Do *not* commit data files that you have put in your node directory. (check `git status` on what will be committed before you use `git commit -a`.)
- *Git* trusts you know what you are doing. It will allow you to do stupid things, too.
- Don't panic. Yes, *git* may have a comparably steep learning curve, but it is a powerful tool and all problems can be resolved.

13.1.1 Situations that commonly arise and how to solve them

Merge conflicts. When you pull from Upstream into your repo, other's changes are merged with yours. It might however happen that someone else has changed the same line in the same file as you have in onw of your own commits, which results in a merge conflict. The pull commands warns you about this and *git status* shows the file in question as "both modified". The file itself contains both versions of the conflicting lines, clearly marked. Edit the file so that only one version remains and remove the markings. Then you simply commit the file (and push).

Undo a commit. To undo a commit means exactly that, *not* that any of the files change. For example, undoing the last commit leaves you with as much uncommitted changes as you had before your last commit. None of your edits is reversed. Undoing commits is practical e.g. when you have committed too many things at once or unwanted

files; or when you want to split one commit into several. You undo a commit with *git reset –soft <REF>* where <REF> is the commit that should be resetted to (i.e. the next-to-last one, if you want to undo your last commit). Common values for <REF> include:

- *HEAD^* - this is the next-to-last
- *HEAD^^* - the one before the next-to-last.
- *HEAD~5* - five commits back
- *111521cb9d3771e636f5f053d3d1048aa7c8852f* - each commit has a long hash number that uniquely identifies it. They can be seen in *git log* and you can give the hash number of the commit that you want to reset to to *git reset*.

Revert to an earlier version. If you want to *throw away* your edits since a certain commit, you use *git reset –hard*. For example, to revert all files to the state that they were in at the last commit (thow away uncommitted changes), you do *git reset –hard HEAD*. Similarly to the soft reset, you can also specify earlier commits that you want to reset to.

Look at an earlier version. You can check out any earlier version of any file at any time. For example, *git checkout "master@{1 month ago}" <filename>"* will give you the version of the file <filename> from a month ago. To go back to the latest, you do *git checkout master <filename>* ("master" is the name of the default branch where all you commits are). Note that the last command can also be used to thow away uncommitted changes in a specific file - a more gentle way than the reset described above.

You can also skip the <filename> to check out an earlier version of the whole repo (*git checkout master* brings you back to the latest). Instead of "master@{1 month ago}" you can use any of the <REF> mentioned above, or have a look at http://book.git-scm.com/4_git_treeishes.html.

Make a branch. Read *git help branch* for this.

13.1.2 Commit guidelines

One thing at a time. Please commit often and only include things in one commit that logically belong together. For example, changes to your node and changes to the common library should not be in the same commit but committed separately.

Meaningful commit messages. This goes together with the previous: If you cannot meaningfully summarize the changes you want to commit in onw or two lines, your commit is likely to be too large. Try to make the log messages meaningful!

Good code. Please try to avoid spaghetti-code, write modular, and follow http://www.python.org/dev/peps/pep-0008/

Pull first. Before you send a pull request, please make sure that you have pulled from upstream. This will make the merging of your code easier, since it will be you who needs to resolve potential conflicts before you push to your origin again.

The admin of *upstream* (aka the writer of these lines) might be bribed and/or convinced to turn a blind eye on violations against any of the above points, but he will be very happy if you try to follow them.

13.2 Source code documentation

The following is the automatically generated documentation from the source code. It lists and describes all functions, classes etc.

13.3 The VAMDC-TAP service library

13.3.1 Tapservice Documentation

This page contains the Tapservice Package documentation.

The `sqlparse` Module

The `generators` Module

The `views` Module

13.4 The import tool

13.4.1 Imptools Documentation

This page contains the Imptools Module documentation.

The `imptools` Module

This program implements a database importer that reads from ascii-input files to the django database. It's generic and is controlled from a mapping file.

class imptools.rewrite.**MappingFile**(*filepath*, *headblocks*, *commentchar*, *blockoffset*, *blockstep*, *errblock*, *startblock=None*, *endblock='n'*)

Bases: object

This class implements an object that represents an open file from which one can read blocks. The object keeps track of its own block-step speed and will return lines as defined by this speed. E.g. for a line-step speed of 0.5, it will return the same line twice in a row whereas for a step speed of 2, will return every second block etc.

If endblock is \n (default), the block will infact represent a line.

block_generator(*fileobj*, *startblock=None*, *endblock='\n'*)
generator, stepping through blocks

readblock()
Return a block from the file.

This method understand both slower block stepping (0 < blockstep < 1) and faster (> 1)

imptools.rewrite.**ftime**(*t0*, *t1*)
formats time to nice format.

imptools.rewrite.**get_value**(*linedata*, *column_dict*)
Process one line/block of data. Linedata is a tuple that always starts with the raw string for the line. The function with its arguments is read from the column_dict and applied to the linedata. The result is returned, after checking for the NULL value.

imptools.rewrite.**is_iter**(*iterable*)

imptools.rewrite.**log_trace**(*e*, *info=''*)
Intended to be called from inside a traceback exception with the exception object as first argument. Captures the latest traceback.

imptools.rewrite.**make_outfile**(*file_dict*, *global_debug=False*)
Process one file definition from a config dictionary by processing the file name stored in it and parse it according to the mapping.

file_dict - config dictionary representing one input file structure (for an example see e.g. mapping_vald3.py)

a function raising a RuntimeError exception will skip the current line being parsed.

imptools.rewrite.**parse_mapping**(*mapping*, *debug=False*)
Step through a list of mappings describing the relation between (usually ascii-)files and django database fields. This should ideally not have to be changed for different database types.

imptools.rewrite.**read_mapping**(*fname*)
Read the config dictionary from a file. Note: very unsafe, since the content gets executed. Have a look at createcfg() to see how it should look like.

imptools.rewrite.**validate_mapping**(*mapping*)
Check the mapping definition to make sure it contains a structure suitable for the parser.

13.4.2 Linefuncs Documentation

This page contains the Linefuncs Module documentation. See also *How to get your data into the database*.

The `linefuncs` Module

Line functions are helper functions available to use in the mapping file.

All importable line functions take 'linedata' as a first argument. This is either a line or a block of text data from the currently parsed input file.

Example of call from mapping dictionary:

{'cname' ['whatever_field_name',] 'cbyte' : (charrange, 56, 58)}

imptools.linefuncs.**bySepNr**(*linedata*, *number*, *sep=', '*)

imptools.linefuncs.**bySepNr2**(*linedata*, *number*, *sep=', '*)
Split a text line by sep argument and return the number:ed split section

Inputs: linedata (str or iterable) - current line(s) to operate on number (int) - nth section, separated by sep sep (str) - a separator to split by

imptools.linefuncs.**bySepNr2int**(*linedata*, *number*, *sep=', '*)
Split a text line by sep aargument and return the numbered split section. Always convert output to int. Inputs:

linedata (str) number (int) - nth section, separated by sep sep (str) - separator to split by

imptools.linefuncs.**charrange**(*linedata*, *start*, *end*)
Cut out part of a line of texts based on indices.

Inputs: linedata (str or iterable) - current line(s) to operate on start, end (int) - beginning and end indices of the line

imptools.linefuncs.**charrange2int**(*linedata*, *start*, *end*)
Cut out part of a line based on indices, return as integer

Inputs: linedata (str or iterable) - current line(s) to operate on start, end (int) - beginning and end indices of the line

imptools.linefuncs.**constant**(*linedata*, *value*)

imptools.linefuncs.**get_accur**(*linedata*, *range1*, *range2*)
extract accuracy

imptools.linefuncs.**get_alphawaals**(*linedata*, *sep1*, *sep2*)
extract alpha - van der waal value

imptools.linefuncs.**get_gammawaals**(*linedata*, *sep1*, *sep2*)
extract gamma - van der waal value

imptools.linefuncs.**get_publications**(*linedata*)
extract publication data. This returns a list since it is for a multi-reference.

imptools.linefuncs.**get_sigmawaals**(*linedata*, *sep1*, *sep2*)
extract sigma - van der waal value

imptools.linefuncs.**get_srcfile_ref**(*linedata*, *sep1*, *sep2*)
extract srcfile reference

imptools.linefuncs.**get_term_val**(*linedata*, *varname*)

extract configurations from term file.

varname is the value type we want (e.g. s or l); we search the identifyer field of the term-file to see if it exists and return the corresponding value, otherwise we return 'X'. Varname is case insensitive.

imptools.linefuncs.**is_iter**(*iterable*)
Helper function

Checks if the given argument is iterable or not, i.e. if it is a list or tuple. Strings are not considered iterable by this function.

imptools.linefuncs.**lineSplit**(*linedata*, *splitsep=''*, '')
Splits a line by splitsep, returns a list. The main use for this method is creating a many-to-many reference.

Inputs: linedata (str or iterable) - current line(s) to operate on splitsep (str) - string to split by

Returns a list!

imptools.linefuncs.**merge_cols**(*linedata*, **ranges*)

Merges data from several columns into one, separating them with '-'. ranges are any number of tuples (indexstart, indexend) defining the columns.

imptools.linefuncs.**merge_cols_by_sep**(*linedata*, **sepNr*)
Merges data from several columns (separated by ;) into one, separating them with '-'. sepNr are the nth position of the file, separated by 'sep'. Assumes a single line input.

i

B

C

F

G

I

L

M

P

R

V