# Cheshire3 Documentation

***Release 1.1.1***

**Rob Sanderson, John Harrison, et al.**

July 24, 2014

Contents


**1 Cheshire3 Installation** **3**
- 1.1 Requirements / Dependencies . . . 3
- 1.2 Installation . . . 3

**2 Cheshire3 Tutorials** **5**
- 2.1 Cheshire3 Tutorials - Command-line UI . . . 5
- 2.2 Cheshire3 Tutorials - Python API . . . 6
- 2.3 Cheshire3 Tutorials - Configuring Databases . . . 12
- 2.4 Cheshire3 Tutorials - Configuring Indexes . . . 13
- 2.5 Cheshire3 Tutorials - Configuring Stores . . . 15
- 2.6 Cheshire3 Tutorials - Configuring Workflows . . . 17

**3 Cheshire3 Commands Reference** **21**
- 3.1 Introduction . . . 21
- 3.2 `cheshire3` . . . 21
- 3.3 `cheshire3-init` . . . 21
- 3.4 `cheshire3-register` . . . 22
- 3.5 `cheshire3-load` . . . 22
- 3.6 `cheshire3-search` . . . 23
- 3.7 `cheshire3-serve` . . . 23

**4 Cheshire3 Configuration** **25**
- 4.1 Common Configurations . . . 25
- 4.2 Cheshire3 Configuration - Indexes . . . 26
- 4.3 Cheshire3 Configuration - Protocol Map . . . 29
- 4.4 Cheshire3 Configuration - Workflows . . . 30
- 4.5 Introduction . . . 32
- 4.6 Configuration Elements . . . 33
- 4.7 Example . . . 35

**5 Cheshire3 Object Model** **37**
- 5.1 Cheshire3 Object Model - Abstract Base Class . . . 37
- 5.2 Cheshire3 Object Model - Database . . . 38
- 5.3 Cheshire3 Object Model - Document . . . 39
- 5.4 Cheshire3 Object Model - DocumentFactory . . . 40
- 5.5 Cheshire3 Object Model - DocumentStore . . . 41
- 5.6 Cheshire3 Object Model - Extractor . . . 41
- 5.7 Cheshire3 Object Model - Index . . . 42

5.8 Cheshire3 Object Model - IndexStore . . . 45
5.9 Cheshire3 Object Model - ObjectStore . . . 46
5.10 Cheshire3 Object Model - Parser . . . 47
5.11 Cheshire3 Object Model - PreParser . . . 48
5.12 Cheshire3 Object Model - Record . . . 49
5.13 Cheshire3 Object Model - RecordStore . . . 50
5.14 Cheshire3 Object Model - ResultSet . . . 51
5.15 Cheshire3 Object Model - ResultSetStore . . . 52
5.16 Cheshire3 Object Model - Selector . . . 52
5.17 Cheshire3 Object Model - Server . . . 53
5.18 Cheshire3 Object Model - TokenMerger . . . 54
5.19 Cheshire3 Object Model - Tokenizer . . . 54
5.20 Cheshire3 Object Model - Transformer . . . 56
5.21 Cheshire3 Object Model - User . . . 57
5.22 Cheshire3 Object Model - Workflow . . . 58
5.23 Overview . . . 59
5.24 Miscellaneous . . . 59
5.25 Summary Objects . . . 59
5.26 `ResultSetStore` . . . 60
5.27 Data Objects . . . 60
5.28 Processing Objects . . . 61
5.29 Other Notable Modules . . . 63

**6 Troubleshooting** **65**
6.1 Introduction . . . 65
6.2 Common Run-time Errors . . . 65
6.3 Apache Errors . . . 68

**7 Capabilities** **69**

**8 Indices and tables** **71**

**Python Module Index** **73**

Contents:

CHAPTER 1

# Cheshire3 Installation

## 1.1 Requirements / Dependencies

Cheshire3 requires Python 2.6.0 or later. It has not yet been verified as Python 3 compliant.

As of the version 1.0 release Cheshire3's core dependencies *should* be resolved automatically by the standard Python package management mechanisms (e.g. pip, easy_install, distribute/setuptools).

However on some systems, for example if installing on a machine without network access, it may be necessary to manually install some 3rd party dependencies. In such cases we would encourage you to download the necessary Cheshire3 bundles from the Cheshire3 download site and install them using the automated build scripts included. If the automated scripts fail on your system, they should at least provide hints on how to resolve the situation.

If you experience problems with dependencies, please get in touch via the GitHub issue tracker or wiki, and we'll do our best to help.

### 1.1.1 Additional / Optional Features

Certain features within the Cheshire3 Information Framework will have additional dependencies (e.g. web APIs will require a web application server). We'll try to maintain an accurate list of these in the README file for each sub-package.

The bundles available from the Cheshire3 download site should continue to be a useful place to get hold of the source code for these pre-requisites.

## 1.2 Installation

The following guidelines assume that you have administrative privileges on the machine you're installing on, or that you're installing in a local Python install or a virtual environment created using `virtualenv`. If this is not the case, then you might need to use the `--local` or `--user` option . For more details, see: http://docs.python.org/install/index.html#alternate-installation

### 1.2.1 Users

Users (i.e. those not wanting to actually develop Cheshire3) have several choices:

- pip: `pip install cheshire3`
- easy_install: `easy_install cheshire3`

- Install from source:
  1. Download a source code archive from one of:

     http://pypi.python.org/pypi/cheshire3

     http://cheshire3.org/download/lastest/src/

     http://github.com/cheshire3/cheshire3
  2. Unpack it:

     ```
     tar -xzf cheshire3-1.0.8.tar.gz
     ```
  3. Go into the unpacked directory:

     ```
     cd cheshire3-1.0.8
     ```
  4. Install:

     ```
     python setup.py install
     ```

### 1.2.2 Developers

1. In GitHub, fork the Cheshire3 GitHub repository
2. Locally clone your Cheshire3 GitHub fork
3. Run `python setup.py develop`

CHAPTER 2

# Cheshire3 Tutorials

This section contains tutorials on configuring objects within the *Cheshire3 Object Model* and then using them within a Python scripting environment.

Tutorials:

## 2.1 Cheshire3 Tutorials - Command-line UI

Cheshire3 provides a number of command-line utilities to enable you to get started creating databases, indexing and searching your data quickly. All of these commands have full help available, including lists of available options which can be accessed using the `--help` option. e.g.

```
``cheshire3 --help``
```

### 2.1.1 Creating a new Database

**`cheshire3-init [database-directory]`** Initialize a database with some generic configurations in the given directory, or current directory if absent

Example 1: create database in a new sub-directory

```
$ cheshire3-init mydb
```

Example 2: create database in an existing directory

```
$ mkdir -p ~/dbs/mydb
$ cheshire3-init ~/dbs/mydb
```

Example 3: create database in current working directory

```
$ mkdir -p ~/dbs/mydb
$ cd ~/dbs/mydb
$ cheshire3-init
```

Example 4: create database with descriptive information in a new sub-directory

```
$ cheshire3-init --database=mydb --title="My Database" \
--description="A Database of Documents" mydb
```

### 2.1.2 Loading Data into the Database

**cheshire3-load data** Load data into the current Cheshire3 database

Example 1: load data from a file:

```
$ cheshire3-load path/to/file.xml
```

Example 2: load data from a directory:

```
$ cheshire3-load path/to/directory
```

Example 3: load data from a URL:

```
$ cheshire3-load http://www.example.com/index.html
```

### 2.1.3 Searching the Database

**cheshire3-search query** Search the current Cheshire3 database based on the parameters given in query

Example 1: search with a single keyword

```
$ cheshire3-search food
```

Example 2: search with a complex CQL query

```
$ cheshire3-search "cql.anywhere all/relevant food and \
rec.creationDate > 2012-01-01"
```

### 2.1.4 Exposing the Database via SRU

**cheshire3-serve** Start a demo HTTP WSGI application server to serve configured databases via SRU

*Please Note* the HTTP server started is probably not sufficiently robust for production use. You should consider using something like mod_wsgi.

Example 1: start a demo HTTP WSGI server with default options

```
$ cheshire3-serve
```

Example 2: start a demo HTTP WSGI server, specifying host name and port number

```
$ cheshire3-serve --host myhost.example.com --port 8080
```

## 2.2 Cheshire3 Tutorials - Python API

The *Cheshire3 Object Model* defines public methods for each object class. These can be used within Python, for embedding Cheshire3 services within a Python enabled web application framework, such as Django, CherryPy, mod_wsgi etc. or whenever the command-line interface is insufficient. This tutorial outlines how to carry out some of the more common operations using these public methods.

### 2.2.1 Initializing Cheshire3 Architecture

Initializing the Cheshire3 Architecture consists primarily of creating instances of the following types within the *Cheshire3 Object Model*:

**Session** An object representing the user session. It will be passed around amongst the processing objects to maintain details of the current environment. It stores, for example, user and identifier for the database currently in use.

**Server** A protocol neutral collection of databases, users and their dependent objects. It acts as an inital entry point for all requests and handles such things as user authentication, and global object configuration.

The first thing that we need to do is create a Session and build a Server:

```
>>> from cheshire3.baseObjects import Session
>>> session = Session()
```

The Server looks after all of our objects, databases, indexes ... everything. Its constructor takes session and one argument, the filename of the top level configuration file. You could supply your own, or you can find the filename of the default server configuration dynamically as follows:

```
>>> import os
>>> from cheshire3.server import SimpleServer
>>> from cheshire3.internal import cheshire3Root
>>> serverConfig = os.path.join(cheshire3Root, 'configs', 'serverConfig.xml')
>>> server = SimpleServer(session, serverConfig)
>>> server
<cheshire3.server.SimpleServer object...
```

Most often you'll also want to work within a Database:

**Database** A virtual collection of Records which may be interacted with. A Database includes Indexes, which contain data extracted from the Records as well as configuration details. The Database is responsible for handling queries which come to it, distributing the query amongst its component Indexes and returning a ResultSet. The Database is also responsible for maintaining summary metadata (e.g. number of items, total word count etc.) that may be need for relevance ranking etc.

To get a database.:

```
>>> db = server.get_object(session, 'db_test')
>>> db
<cheshire3.database.SimpleDatabase object...
```

After this you MUST set session.database to the identifier for your database, in this case 'db_test':

```
>>> session.database = 'db_test'
```

This is primarily for efficiency in the workflow processing (objects are cached by their identifier, which might be duplicated for different objects in different databases).

Another useful path to know is the database's default path:

```
>>> dfp = db.get_path(session, 'defaultPath')
```

#### Using the `cheshire3` command

One way to ensure that Cheshire3 architecture is initialized is to use the Cheshire3 interpreter, which wraps the main Python interpreter, to run your script or just drop you into the interactive console.

**cheshire3 [script]** Run the commands in the script inside the current cheshire3 environment. If script is not provided it will drop you into an interactive console (very similar the the native Python interpreter.) You can also tell it to drop into interactive mode after executing your script using the `--interactive` option.

When initializing the architecture in this way, `session` and `server` variables will be created, as will a `db` object if you ran the script from inside a Cheshire3 database directory, or provided a database identifier using the `--database` option. The variable will correspond to instances of Session, Server and Database respectively.

### 2.2.2 Loading Data

In order to load data into your database you'll need a document factory to find your documents, a parser to parse the XML and a record store to put the parsed XML into. The most commonly used are defaultDocumentFactory and LxmlParser. Each database needs its own record store.:

```
>>> df = db.get_object(session, "defaultDocumentFactory")
>>> parser = db.get_object(session, "LxmlParser")
>>> recStore = db.get_object(session, "recordStore")
```

Before we get started, we need to make sure that the stores are all clear:

```
>>> recStore.clear(session)
<cheshire3.recordStore.BdbRecordStore object...
>>> db.clear_indexes(session)
```

First you should call db.begin_indexing() in order to let the database initialise anything it needs to before indexing starts. Ditto for the record store:

```
>>> db.begin_indexing(session)
>>> recStore.begin_storing(session)
```

Then you'll need to tell the document factory where it can find your data:

```
>>> df.load(session, 'data', cache=0, format='dir')
<cheshire3.documentFactory.SimpleDocumentFactory object...
```

DocumentFactory's load function takes session, plus:

**data** this could be a filename, a directory name, the data as a string, a URL to the data and so forth.

If data ends in [(numA):(numB)], and the preceding string is a filename, then the data will be extracted from bytes numA through to numB (this is pretty advanced though - you'll probably never need it!)

**cache** setting for how to cache documents in memory when reading them in. This will depend greatly on use case. e.g. if loading 3Gb of documents on a machine with 2Gb memory, full caching will obviously not work very well. On the other hand, if loading a reasonably small quantity of data over HTTP, full caching would read all of the data in one shot, closing the HTTP connection and avoiding potential timeouts. Possible values:

**0** no document caching. Just locate the data and get ready to discover and yield documents when they're requested from the documentFactory. This is probably the option you're most likely to want.

**1** Cache location of documents within the data stream by byte offset.

**2** Cache full documents.

**format** The format of the data parameter. Many options, the most common are:

**xml** xml file. Can have multiple records in single file.

**dir** a directory containing files to load

**tar** a tar file containing files to load

**zip** a zip file containing files to load

**marc** a file with MARC records (library catalogue data)

**http** a base HTTP URL to retrieve

**tagName** the name of the tag which starts (and ends!) a record. This is useful for extracting sections of documents and ignoring the rest of the XML in the file.

**codec** the name of the codec in which the data is encoded. Normally 'ascii' or 'utf-8'

You'll note above that the call to load returns itself. This is because the document factory acts as an iterator. The easiest way to get to your documents is to loop through the document factory:

```
>>> for doc in df:
...     rec = parser.process_document(session, doc)   # [1]
...     recStore.create_record(session, rec)          # [2]
...     db.add_record(session, rec)                   # [3]
...     db.index_record(session, rec)                 # [4]
recordStore/...
```

In this loop, we:

1. Use the Lxml Etree Parser to create a record object.
2. Store the record in the recordStore. This assigns an identifier to it, by default a sequential integer.
3. Add the record to the database. This stores database level metadata such as how many words in total, how many records, average number of words per record, average number of bytes per record and so forth.
4. Index the record against all indexes known to the database - typically all indexes in the indexStore in the database's 'indexStore' path setting.

Then we need to ensure this data is committed to disk:

```
>>> recStore.commit_storing(session)
>>> db.commit_metadata(session)
```

And, potentially taking longer, merge any temporary index files created:

```
>>> db.commit_indexing(session)
```

#### Pre-Processing (PreParsing)

As often than not, documents will require some sort of pre-processing step in order to ensure that they're valid XML in the schema that you want them in. To do this, there are PreParser objects which take a document and transform it into another document.

The simplest preParser takes raw text, escapes the entities and wraps it in a element:

```
>>> from cheshire3.document import StringDocument
>>> doc = StringDocument("This is some raw text with an & and a < and a >.")
>>> pp = db.get_object(session, 'TxtToXmlPreParser')
>>> doc2 = pp.process_document(session, doc)
>>> doc2.get_raw(session)
'<data>This is some raw text with an &amp; and a &lt; and a &gt;.</data>'
```

### 2.2.3 Searching

In order to allow for translation between query languages (if possible) we have a query factory, which defaults to CQL (SRU's query language, and our internal language):

```
>>> qf = db.get_object(session, 'defaultQueryFactory')
>>> qf
<cheshire3.queryFactory.SimpleQueryFactory object ...
```

We can then use this factory to build queries for us:

```
>>> q = qf.get_query(session, 'c3.idx-text-kwd any "compute"')
>>> q
<cheshire3.cqlParser.SearchClause ...
```

And then use this parsed query to search the database:

```
>>> rs = db.search(session, q)
>>> rs
<cheshire3.resultSet.SimpleResultSet ...
>>> len(rs)
3
```

The 'rs' object here is a result set which acts much like a list. Each entry in the result set is a ResultSetItem, which is a pointer to a record:

```
>>> rs[0]
Ptr:recordStore/1
```

### 2.2.4 Retrieving

Each result set item can fetch its record:

```
>>> rec = rs[0].fetch_record(session)
>>> rec.recordStore, rec.id
('recordStore', 1)
```

Records can expose their data as xml:

```
>>> rec.get_xml(session)
'<record>...
```

As SAX (Simple API for XML) events:

```
>>> rec.get_sax(session)
["4 None, 'record', 'record', {}...
```

Or as DOM nodes, in this case using the Lxml Etree API:

```
>>> rec.get_dom(session)
<Element record at ...
```

You can also use XPath expressions on them:

```
>>> rec.process_xpath(session, '/record/header/identifier')
[<Element identifier at ...
>>> rec.process_xpath(session, '/record/header/identifier/text()')
['oai:CiteSeerPSU:2']
```

### 2.2.5 Transforming Records

Records can be processed back into documents, typically in a different form, using Transformers:

```
>>> dctxr = db.get_object(session, 'DublinCoreTxr')
>>> doc = dctxr.process_record(session, rec)
```

And you can get the data from the document with get_raw():

```
>>> doc.get_raw(session)
'<?xml version="1.0"?>...
```

This transformer uses XSLT, which is common, but other transformers are equally possible.

It is also possible to iterate through stores. This is useful for adding new indexes or otherwise processing all of the data without reloading it.

First find our index, and the indexStore:

```
>>> idx = db.get_object(session, 'idx-creationDate')
```

Then start indexing for just that index, step through each record, and then commit the terms extracted:

```
>>> idxStore.begin_indexing(session, idx)
>>> for rec in recStore:
...     idx.index_record(session, rec)
recordStore/...
>>> idxStore.commit_indexing(session, idx)
```

### 2.2.6 Indexes (Looking Under the Hood)

Configuring Indexes, and the processing required to populate them requires some further object types, such as Selectors, Extractors, Tokenizers and TokenMergers. Of course, one would normally configure these for each index in the database and the code in the examples below would normally be executed automatically. However it can sometimes be useful to get at the objects and play around with them manually, particularly when starting out to find out what they do, or figure out why things didn't work as expected, and Cheshire3 makes this possible.

Selector objects are configured with one or more locations from which data should be selected from the Record. Most commonly (for XML data at least) these will use XPaths. A selector returns a list of lists, one for each configured location:

```
>>> xp1 = db.get_object(session, 'identifierXPathSelector')
>>> rec = recStore.fetch_record(session, 1)
>>> elems = xp1.process_record(session, rec)
>>> elems
[[<Element identifier at ...
```

However we need the text from the matching elements rather than the XML elements themselves. This is achieved using an Extractor, which processes the list of lists returned by a Selector and returns a dictionary a.k.a an associative array or hash:

```
>>> extr = db.get_object(session, 'SimpleExtractor')
>>> hash = extr.process_xpathResult(session, elems)
>>> hash
{'oai:CiteSeerPSU:2 ': {'text': 'oai:CiteSeerPSU:2 ', ...
```

And then we'll want to normalize the results a bit. For example we can make everything lowercase:

```
>>> n = db.get_object(session, 'CaseNormalizer')
>>> h2 = n.process_hash(session, h)
>>> h2
{'oai:citeseerpsu:2 ': {'text': 'oai:citeseerpsu:2 ', ...
```

And note the extra space on the end of the identifier...:

```
>>> s = db.get_object(session, 'SpaceNormalizer')
>>> h3 = s.process_hash(session, h2)
>>> h3
{'oai:citeseerpsu:2': {'text': 'oai:citeseerpsu:2',...
```

Now the extracted and normalized data is ready to be stored in the index!

This is fine if you want to just store strings, but most searches will probably be at word or token level. Let's get the abstract text from the record:

```
>>> xp2 = db.get_object(session, 'textXPathSelector')
>>> elems = xp2.process_record(session, rec)
>>> elems
[[<Element {http://purl.org/dc/elements/1.1/}description ...
```

Note the {...} bit ... that's lxml's representation of a namespace, and needs to be included in the configuration for the xpath in the `Selector`.:

```
>>> extractor = db.get_object(session, 'ProxExtractor')
>>> hash = extractor.process_xpathResult(session, elems)
>>> hash
{'The Graham scan is a fundamental backtracking...
```

`ProxExtractor` records where in the record the text came from, but otherwise just extracts the text from the elements. We now need to split it up into words, a process called tokenization:

```
>>> tokenizer = db.get_object(session, 'RegexpFindTokenizer')
>>> hash2 = tokenizer.process_hash(session, hash)
>>> h
{'The Graham scan is a fundamental backtracking...
```

Although the key at the beginning looks the same, the value is now a list of tokens from the key, in order. We then have to merge those tokens together, such that we have 'the' as the key, and the value has the locations of that type:

```
>>> tokenMerger = db.get_object(session, 'ProxTokenMerger')
>>> hash3 = tokenMerger.process_hash(session, hash2)
>>> hash3
{'show': {'text': 'show', 'occurences': 1, 'positions': [12, 41]},...
```

After token merging, the multiple terms are ready to be stored in the index!

## 2.3 Cheshire3 Tutorials - Configuring Databases

### 2.3.1 Introduction

`Database`s are primarily collections of `Record`s and `Index`es along with the associated metadata and objects required for processing the data.

Configuration is typically done in a single file, with all of the dependent components included within it and stored in a directory devoted to just that database. The file is normally called, simply, `config.xml`.

### 2.3.2 Example

An example `Database` configuration:

```
1  <config type="database" id="db_ead">
2      <objectType>cheshire3.database.SimpleDatabase</objectType>
3      <paths>
4          <path type="defaultPath">dbs/ead</path>
5          <path type="metadataPath">metadata.bdb</path>
6          <path type="indexStoreList">eadIndexStore</path>
7          <object type="recordStore" ref="eadRecordStore"/>
8      </paths>
9      <subConfigs>
10     ...
11     </subConfigs>
12     <objects>
13     ...
14     </objects>
15 </config>
```

### 2.3.3 Explanation

In line 1 we open a new object, of type `Database` with an identifier of `db_ead`. You should replace `db_ead` with the identifier you want for your `Database`.

Line 2 defines the *<objectType>* of the `Database` (which will normally be a class from the `cheshire3.database` module). There is currently only one recommended implementation, `cheshire3.database.SimpleDatabase`, so this line should be copied in verbatim, unless you have defined your own sub-class of `cheshire3.baseObjects.Database` (in which case you're probably more advanced than the target audience for this tutorial!)

Lines 4 and 7 define three *<path>*s and one *<object>*. To explain each in turn:

**defaultPath** the path to the directory where the database is being stored. It will be prepended to any further paths in the database or in any subsidiary object.

**metadataPath** the path to a datastore in which the database will keep its metadata. This includes things like the number of records, the average size or the records and so forth. As it's a file path, it would end up being `dbs/ead/metdata.bdb` – in other words, in the same directory as the rest of the database files.

**indexStoreList** a space separated list of references to all `IndexStore`s the `Database` will use. This is needed if we intend to index any `Record`s later, as it tells the `Database` which `IndexStore`s to register the `Record` in.

The *<object>* element refers to an object called `eadRecordStore` which is an instance of a `RecordStore`. This is important for future `Workflow`s, so that the `Database` knows which `RecordStore` it should put `Record`s into by default.

Line 10 would be expanded to contain a series of *<subConfig>* elements, each of which is the configuration for a subsidiary object such as the `RecordStore` and the `Index`es to store in the `IndexStore`, `eadIndexStore`.

Line 13 could be expanded to contain a series of *<path>* elements, each of which has a reference to a Cheshire3 object that has been previously configured. This lines instruct the server to actually instantiate the object in memory. while this is not strictly necessary it may occasionally be desirable, see *<objects>* for more information.

## 2.4 Cheshire3 Tutorials - Configuring Indexes

### 2.4.1 Introduction

`Index`es are the primary means of locating `Record`s in the system, and hence need to be well thought out and

specified in advance. They consist of one or more *<paths>* to tags in the `Record`, and how to process the data once it has been located.

### 2.4.2 Example

Example index configurations:

```
1  <subConfig id = "xtitle-idx">
2      <objectType>index.SimpleIndex</objectType>
3      <paths>
4          <object type="indexStore" ref="indexStore"/>
5      </paths>
6      <source>
7          <xpath>/ead/eadheader/filedesc/titlestmt/titleproper</xpath>
8          <process>
9              <object type="extractor" ref="SimpleExtractor"/>
10             <object type="normalizer" ref="SpaceNormalizer"/>
11             <object type="normalizer" ref="CaseNormalizer"/>
12         </process>
13     </source>
14     <options>
15         <setting type="sortStore">true</setting>
16     </options>
17 </subConfig>
18
19 <subConfig id = "stemtitleword-idx">
20     <objectType>index.ProximityIndex</objectType>
21     <paths>
22         <object type="indexStore" ref="indexStore"/>
23     </paths>
24     <source>
25         <xpath>titleproper</xpath>
26         <process>
27             <object type="extractor" ref="ProxExtractor" />
28             <object type="tokenizer" ref="RegexpFindOffsetTokenizer"/>
29             <object type="tokenMerger" ref="OffsetProxTokenMerger"/>
30             <object type="normalizer" ref="CaseNormalizer"/>
31             <object type="normalizer" ref="PossessiveNormalizer"/>
32             <object type="normalizer" ref="EnglishStemNormalizer"/>
33         </process>
34     </source>
35 </subConfig>
```

### 2.4.3 Explanation

Lines 1 and 2, 19 and 20 should be second nature by now. Line 4 and the same in line 22 are a reference to the `IndexStore` in which the `Index` will be maintained.

This brings us to the *<source>* section starting in line 6. It must contain one or more xpath elements. These XPaths will be evaluated against the record to find a node, nodeSet or attribute value. This is the base data that will be indexed after some processing. In the first case, we give the full path, but in the second only the final element.

If the records contain XML Namespaces, then there are two approaches available. If the element names are unique between all the namespaces in the document, you can simply omit them. For example /srw:record/dc:title could be written as just /record/title. The alternative is to define the meanings of 'srw' and 'dc' on the xpath element in the normal xmlns fashion.

After the XPath(s), we need to tell the system how to process the data that gets pulled out. This happens in the process section, and is a list of objects to sequentially feed the data through. The first object must be an extractor. This may be followed by a `Tokenizer` and a `TokenMerger`. These are used to split the extracted data into tokens of a particular type, and then merge it into discreet index entries. If a `Tokenizer` is used, a `TokenMerger` must also be used. Generally any further processing objects in the chain are `Normalizers`.

The first `Index` uses the `SimpleExtractor` to pull out the text as it appears exactly as a single term. This is followed by a `SpaceNormalizer` on line 10, to remove leading and trailing whitespace and normalize multiple adjacent whitespace characters (e.g. newlines followed by tabs, spaces etc.) into single whitespaces The second `Index` uses the `ProxExtractor`; this is a special instance of `SimpleExtractor`, that has been configured to also extract the position of the XML elements from which is extracting. Then it uses a `RegexpFindOffsetTokenizer` to identify word tokens, their positions and character offsets. It then uses the necessary `OffsetProxTokenMerger` to merge identical tokens into discreet index entries, maintaining the word positions and character offsets identified by the Tokenizer. Both indexes then send the extracted terms to a `CaseNormalizer`, which will reduce all characters to lowercase. The second `Index` then gives the lowercase terms to a `PossessiveNormalizer` to strip off 's and s' from the end, and then to `EnglishStemNormalizer` to apply linguistic stemming.

After these processes have happened, the system will store the transformed terms in the `IndexStore` referenced in the *<paths>* section.

Finally, in the first example, we have a setting called `sortStore`. When this is provided and set to a true value, it instructs the system to create a map of `Record` identifier to terms enabling the `Index` to be used to quickly re-order `ResultSets` based on the values extracted.

For detailed information about available settings for `Indexes` see the *Index Configuration, Settings section*.

## 2.5 Cheshire3 Tutorials - Configuring Stores

### 2.5.1 Introduction

There are several trpyes of Store objects, but we're currently primarily concerned with `RecordStores`. `DocumentStores` are practically identical to `RecordStores` in terms of configuration, so we'll talk about the two together.

`Database` specific stores will be included in the *<subConfigs>* section of a database configuration file [`Database Config Tutorial`, *Database Config Reference*].

### 2.5.2 Example

Example store configuration:

```
1  <subConfig type="recordStore" id="eadRecordStore">
2      <objectType>recordStore.BdbRecordStore</objectType>
3      <paths>
4          <path type="databasePath">recordStore.bdb</path>
5          <object type="idNormalizer" ref="StringIntNormalizer"/>
6      </paths>
7      <options>
8          <setting type="digest">sha</setting>
9      </options>
10 </subConfig>
```

### 2.5.3 Explanation

Line 1 starts a new `RecordStore` configuration for an object with the identifier `eadRecordStore`.

Line 2 declares that it should be instantiated with *<objectType>* `cheshire3.recordStore.BdbRecordStore`. There are several possible classes distributed with Cheshire3, another is `cheshire3.sql.recordStore.PostgresRecordStore` which will maintain the data and associated metadata in a PostgreSQL relational database (this assumes that you installed Cheshire3 with the optional `sql` features enabled - see */install* for details). The default is the much faster BerkeleyDB based store.

Then we have two fields wrapped in the :ref:config-paths section. Line 4 gives the filename of the database to use, in this case `recordStore.bdb`. Remember that this will be relative to the current *defaultPath*.

Line 5 has a reference to a `Normalizer` object – this is used to turn the `Record` identifiers into something appropriate for the underlying storage system. In this case, it turns integers into strings (as Berkeley DB only has string keys.) It's safest to leave this alone, unless you know that you're always going to assign string based identifiers before storing `Record`s.

Line 8 has a setting called `digest`. This will configure the `RecordStore` to maintain a checksum for each `Record` to ensure that it remains unique within the store. There are two checksum algorithms available at the moment, 'sha' and 'md5'. If left out, the store will be slightly faster, but allow (potentially inadvertant) duplicate records.

There are some additional possible objects that can be referenced in the *<paths>* section not shown here:

**inTransformer** A `Transformer` to run the `Record` through in order to transform (serialize) it for storing.

If configured, this takes priority over `inWorkflow` which will be ignored.

If not configured reverts to `inWorkflow`.

**outParser** A `Parser` to run the stored data through in order to parse (deserialize) it back into a `Record`.

If configured, this takes priority over `outWorkflow` which will be ignored.

If not configured reverts to `outWorkflow`.

**inWorkflow** A `Workflow` to run the `Record` through in order to transform (serialize) it for storing.

The use of a `Workflow` rather than a `Transformer` enables chaining of objects, e.g. a `XmlTransformer` to serialize the `Record` to XML, followed by a `GzipPreParser` to compress the XML before storing on disk. In this case one would need to configure an `outWorkflow` to reverse the process.

If not configured a `Record` will be serialized using its method, `get_xml(session)()`.

**outWorkflow** A `Workflow` to run the stored data through in order to turn it back into a `Record`.

The use of a `Workflow` rather than a `Parser` enables chaining of objects, e.g. a `GunzipPreParser` to decompress the data back to XML, followed by a `LxmlParser` to parse (deserialize) the XML back into a `Record`.

If not configured, the raw XML data will be parsed (deserialized) using a `LxmlParser`, if it can be got from the `Server`, otherwise a `BSLxmlParser`.

#### DocumentStores

For `DocumentStore`s, instead all we would change would be the identifier, the *<objectType>*, and probably the *databasePath*. Everything else can remain pretty much the same. `DocumentStore`s have slightly different additional objects that can be referenced in the paths section however:

**inPreParser** A `PreParser` to run the `Document` through before storing its content.

For example a `GzipPreParser` to compress the `Document` content before storing on disk. In this case one would need to configure a `GunzipPreParser` as the `outPreParser`.

If configured, this takes priority over `inWorkflow` which will be ignored.

If not configured reverts to `inWorkflow`.

**`outPreParser`** A `PreParser` to run the stored data through before returning the `Document`.

For example a `GunzipPreParser` to decompress the data from the disk to trun it back into the original `Document` content.

If configured, this takes priority over `outWorkflow` which will be ignored.

If not configured reverts to `outWorkflow`.

## 2.6 Cheshire3 Tutorials - Configuring Workflows

### 2.6.1 Introduction

`Workflow`s are first class objects in the Cheshire3 system - they're configured at the same time and in the same way as other objects. Their function is to provide an easy way to define a series of common steps that can be reused by different Cheshire3 databases/systems, as opposed to writing customised code to achieve the same end result for each.

Build `Workflow`s are the most common type as the data must generally pass through a lot of different functions on different objects, however as explained previously the differences between `Database`s are often only in one section. By using `Workflow`s, we can simply define the changed section rather than writing code to do the same task over and over again.

The disadvantage, currently, of `Workflow`s is that it is very complicated to find out what is going wrong if something fails. If your data is very clean, then a `Workflow` is probably the right solution, however if the data is likely to have XML parse errors or has to go through many different `PreParser`s and you want to verify each step, then hand written code may be a better solution for you.

The distribution comes with a generic build workflow object called `buildIndexWorkflow`. It then calls `buildIndexSingleWorkflow` to handle each individual `Document`, also supplied. This second `Workflow`s then calls `PreParserWorkflow`, of which a trivial one is supplied, but this is very unlikely to suit your particular needs, and should be customised as required. An example would be if you were trying to build a `Database` of legacy SGML documents, your `PreParserWorkflow` would probably need to call an: `SgmlPreParser`, configured to deal with the non-XML conformant parts of that particular SGML DTD.

For a full explanation of the different tags used in `Workflow` configuration, and what they do, see the *Configuration section dealing with workflows*.

### 2.6.2 Example 1

Simple workflow configuration:

```
<subConfig type="workflow" id="PreParserWorkflow">
    <objectType>workflow.SimpleWorkflow</objectType>
    <workflow>
        <!-- input type:  document -->
        <object type="preParser" ref="SgmlPreParser"/>
        <object type="preParser" ref="CharacterEntityPreParser"/>
    </workflow>
</subConfig>
```

### 2.6.3 Example 2

Slightly more complex workflow configurations:

```
<subConfig type="workflow" id="buildIndexWorkflow">
    <objectType>workflow.SimpleWorkflow</objectType>
    <workflow>
        <!-- input type:  documentFactory -->
        <log>Loading records</log>
        <object type="recordStore" function="begin_storing"/>
        <object type="database" function="begin_indexing"/>
        <for-each>
            <object type="workflow" ref="buildIndexSingleWorkflow"/>
        </for-each>
        <object type="recordStore" function="commit_storing"/>
        <object type="database" function="commit_metadata"/>
        <object type="database" function="commit_indexing"/>
    </workflow>
</subConfig>

<subConfig type="workflow" id="buildIndexSingleWorkflow">
    <objectType>workflow.SimpleWorkflow</objectType>
    <workflow>
        <!-- input type:  document -->
        <object type="workflow" ref="PreParserWorkflow"/>
        <try>
            <object type="parser" ref="LxmlParser"/>
        </try>
        <except>
             <log>Unparsable Record</log>
        </except>
        <object type="recordStore" function="create_record"/>
        <object type="database" function="add_record"/>
        <object type="database" function="index_record"/>
        <log>Loaded Record</log>
    </workflow>
</subConfig>
```

### 2.6.4 Explanation

The first two lines of each configuration example are exactly the same as all previous objects. Then there is one new section - *<workflow>*. This contains a series of instructions for what to do, primarily by listing objects to handle the data.

The workflow in Example 1 is an example of how to override the `PreParserWorkflow` for a specific database. In this case we start by giving the document input object to the `SgmlPreParser` in line 5, and the result of that is given to the `CharacterEntityPreParser` in line 6. Note that lines 4 and 20 are just comments and are not required.

The workflows in Example 2 are slightly more complex with some additional constructions. Lines 5, 26, 31 use the log instruction to get the `Workflow` to log the fact that it is starting to load `Record`s.

In lines 6 and 7 the object tags have a second attribute called `function`. This contains the name of the function to call when it's not derivable from the input object. For example, a `PreParser` will always call `process_document()`, however you need to specify the function to call on a `Database` as there are many available. Note also that there isn't a 'ref' attribute to reference a specific object identifier. In this case it uses the current session to determine which `Server`, `Database`, `RecordStore` and so forth should be used. This allows the `Workflow` to be used in multiple contexts (i.e. if configured at the server level it can be used by several `Database`s).

The for-each block (lines 8-10) then iterates through the Documents in the supplied DocumentFactory, calling another Workflow, buildIndexSingleWorkflow (configured in lines 17-33), on each of them. Like the PreParser objects mentioned earlier, Workflow objects called don't need to be told which function to call - the system will always call their process() function. Finally the Database and RecordStore have their commit functions called to ensure that everything is written out to disk.

The second workflow in Example 2 is called by the first, and in turn calls the PreParserWorkflow configured in Example 1. It then calls a Parser, carrying out some error handling as it does so (lines 22-27), and then makes further calls to the RecordStore (line 28) and Database (lines 29-30) objects to store and Index the record produced.

CHAPTER 3

# Cheshire3 Commands Reference

## 3.1 Introduction

This page describes the Cheshire3 command line utilities, their intended purpose, and options.

Examples of their use can be found in the *Command-line UI Tutorial*.

## 3.2 cheshire3

Cheshire3 interactive interpreter.

This wraps the main Python interpreter to ensure that Cheshire3 architecture is initialized. It can be used to execute a custom *script* or just drop you into the interactive console.

`session` and `server` variable will be created automatically, as will a `db` object if you ran the script from inside a Cheshire3 database directory, or provided a database identifier using the *cheshire3 --database* option. These variables will correspond to instances of `Session`, `Server` and `Database` respectively.

**script**
: Run the commands in the script inside the current cheshire3 environment. If script is not provided it will drop you into an interactive console (very similar the the native Python interpreter.) You can also tell it to drop into interactive mode after executing your script using the *--interactive* option.

**-h, --help**
: show help message and exit

**-s** <PATH>**, --server-config** <PATH>
: Path to `Server` configuration file. Defaults to the default `Server` configuration included in the distribution.

**-d** <DATABASE>**, --database** <DATABASE>
: Identifier of the `Database`

**--interactive**
: Drop into interactive console after running *script*. If no *script* is provided, interactive mode is the default.

## 3.3 cheshire3-init

Initialize a Cheshire3 `Database` with some generic configurations.

**DIRECTORY**
: name of directory in which to init the `Database`. Defaults to the current working directory.

**-h, --help**
show help message and exit

**-s** <PATH>, **--server-config** <PATH>
Path to Server configuration file. Defaults to the default Server configuration included in the distribution.

**-d** <DATABASE>, **--database** <DATABASE>
Identifier of the Database to init. Default to db_<database-directory-name>.

**-t** <TITLE>, **--title** <TITLE>
Title for the Cheshire3 Database to init. This wil be inserted into the *<docs>* section of the generated configuration, and the *CQL Protocol Map configuration*.

**-c** <DESCRIPTION>, **--description** <DESCRIPTION>
Description of the Database to init. This wil be inserted into the *<docs>* section of the generated configuration, and the *CQL Protocol Map configuration*.

**-p** <PORT>, **--port** <PORT>
Port on which Database will be served via SRU (Search and Retrieve via URL).

## 3.4 cheshire3-register

Register a Cheshire3 Database config file with the Cheshire3 Server.

**CONFIGFILE**
Path to configuration file for a Database to register with the Cheshire3 Server. Default: config.xml in the current working directory.

**--help**
show help message and exit

**--server-config** <PATH>
Path to Server configuration file. Defaults to the default Server configuration included in the distribution.

## 3.5 cheshire3-load

Load data into a Cheshire3 Database.

**data**
Data to load into the Database.

**-h, --help**
show help message and exit

**-s** <PATH>, **--server-config** <PATH>
Path to Server configuration file. Defaults to the default Server configuration included in the distribution.

**-d** <DATABASE>, **--database** <DATABASE>
Identifier of the Database

**-l** <CACHE>, **--cache-level** <CACHE>
Level of in memory caching to use when reading documents in. For details, see *Loading Data*

**-f** <FORMAT>, **--format** <FORMAT>
Format of the data parameter. For details, see *Loading Data*

**-t** <TAGNAME>, **--tagname** <TAGNAME>
The name of the tag which starts (and ends!) a record. This is useful for extracting sections of documents and ignoring the rest of the XML in the file.

**-c** <CODEC>, **--codec** <CODEC>
The name of the codec in which the data is encoded. Commonly `ascii` or `utf-8`.

## 3.6 cheshire3-search

Search a Cheshire3 `Database`.

**query**
Query to execute on the `Database`.

**-h, --help**
show help message and exit

**--server-config** <PATH>
Path to `Server` configuration file. Defaults to the default `Server` configuration included in the distribution.

**-d** <DATABASE>, **--database** <DATABASE>
Identifier of the `Database`

**-f** <FORMAT>, **--format** <FORMAT>
Format/language of query. default: CQL (Contextual Query Language)

**-m** <MAXIMUM>, **--maximum-records** <MAXIMUM>
Maximum number of hits to display

**-s** <START>, **--start-record** <START>
Point in the resultSet to start from (enables result paging) first record in results = 1 (not 0)

## 3.7 cheshire3-serve

Start a demo server to expose Cheshire3 `Database`s. via SRU.

**-h, --help**
show help message and exit

**-s** <PATH>, **--server-config** <PATH>
Path to `Server` configuration file. Defaults to the default `Server` configuration included in the distribution.

**--hostname** <HOSTNAME>
Name of host to listen on. Default is derived by inspection of local system

**-p** <PORT>, **--port** <PORT>
Number of port to listen on. Default: 8000

CHAPTER 4

# Cheshire3 Configuration

Contents:

## 4.1 Common Configurations

### 4.1.1 Introduction

Below are the most commonly required or used paths, objects and settings in Cheshire3 configuration files.

### 4.1.2 Paths

**defaultPath** A default path to be prepended to all other paths in the object and below

**metadataPath** Used to point to a database file or directory for metadata concerning the object

**databasePath** Used in store objects to point to a database file or directory

**tempPath** For when temporary file(s) are required (e.g. for an `IndexStore` .)

**schemaPath** Used in Parsers to point to a validation document (eg xsd, dtd, rng)

**xsltPath** Used in `LxmlXsltTransformer` to point to the XSLT (Extensible Stylesheet Language Transformations) document to use.

**sortPath** Used in an `IndexStore` to refer to the local unix **sort** utility.

### 4.1.3 Settings

**log** This contains a space separated list of function names to log on invocation. The *functionLogger* object referenced in *<paths>* will be used to do this.

**digest** Used in recordStores to name a digest algorithm to determine if a record is already present in the store. Currently supported are 'sha' (which results in sha-1) and 'md5'.

## 4.2 Cheshire3 Configuration - Indexes

### 4.2.1 Introduction

Indexes need to be configured to know where to find the data that they should extract, how to process it once it's extracted and where to store it once processed.

### 4.2.2 Paths

**indexStore** An object reference to the default indexStore to use for extracted terms.

**termIdIndex** Alternative index object to use for termId for terms in this index.

**tempPath** Path to a directory where temporary files will be stored during batch mode indexing.

### 4.2.3 Settings

The value for any true/false type settings must be 0 or 1.

**sortStore** If the value is true , then the indexStore is instructed to also create an inverted list of record Id to value (as opposed to value to list of records) which should be used for sorting by that index.

**cori_constant[0-2]** Constants to be used during CORI relevance ranking, if different from the defaults.

**lr_constant[0-6]** Constants to be used during logistic regression relevance ranking, if different from the defaults.

**okapi_constant_[b|k1|k3]'** Constants to be used for the OKAPI BM-25 algorithm, if different from the defaults. These can be used to fine tune the behavior of relevance ranking using this algorithm.

**noIndexDefault** If the value is true, the `Index` should not be called from `index_record()` method of `Database`.

**noUnindexDefault** If the value is true, the `Index` should not be called from `unindex_record()` method of `Database`.

**vectors** Should the index store vectors (doc -> list of termIds)

**proxVectors** Should the index store vectors that also maintain proximity for their terms

**minimumSupport** TBC

**vectorMinGlobalFreq** TBC

**vectorMaxGlobalFreq** TBC

**vectorMinGlobalOccs** TBC

**vectorMaxGlobalOccs** TBC

**vectorMinLocalFreq** TBC

**vectorMaxLocalFreq** TBC

**longSize** Size of a long integer in this index's underlying data structure (e.g. to migrate between 32 and 64 bit platforms)

**recordStoreSizes** Use average record sizes from recordStores when calculating relevances. This is useful when a database includes records from multiple recordStores, particularly when recordStores contain records of varying sizes.

**maxVectorCacheSize** Number of terms to cache when building vectors.

### 4.2.4 Index Configuration Elements

#### <source>

An index configuration must contain at least one source element. Each source block configures a way of treating the data that the index is asked to process.

It's worth mentioning here that the index object will be asked to process incoming search terms as well as data from records being indexed. A <source> element may have a mode attribute to specify when the processing configured within this source block should be applied. To clarify, the mode attribute may have the value of any of the relations defined by CQL (any, all, =, exact, etc.), indicating that the processing in this source should be applied when the index is searched using that particular relation.

The mode attribute may also have the value 'data', indicating that the processing in the source block should be applied to the records at the time they are indexed. Multiple modes can be specified for a single source block by separating the with a vertical pipe | character within the value of the mode attribute. If no mode attribute is specified, the source will default to being a data source. *Example 2'_ demonstrates the use of the ''mode'* attribute to apply a different Extractor object when carrying out searches using the 'any', 'all' or '=' CQL relation, in this case to preserve masking/wildcard characters.

Each data mode source block configures one or more XPaths to use to extract data from the record, a workflow of objects to process the results of the XPath evaluation and optionally a workflow of objects to pre-process the record to transform it into a state suitable for XPathing. Each data mode source block will be processed in turn by the system for each record during indexing.

For source blocks with modes other than data, only the element configuring the workflow of objects to process the incoming term with is required. *<xpath> or <selector>* and :ref:config-indexes-elements-preprocess' elements will be ignored.

#### <xpath> or <selector>

These elements specify a way to select data from a Record. They can contain either a simple XPath expression as CDATA (as in Example 1) or have a ref attribute containing a reference to a configured Selector object within the configuration hierarchy (as in Example 2).

While it is possible to use either element in either way, it is considered best practice to use the convention of <xpath> for explicit CDATA XPaths and <selector> when referencing a configured Selector.

These elements may not appear more than once within a given *<source>* , however a Selector may itself specify multiple <xpath> or <location> elements. When the a configured Selector contains multiple <xpath> or <location> elements, the results of each expression will be processed by the *process chain* (as described below).

If an XPath makes use of XML namespaces, then the mappings for the namespace prefixes must be present on the XPath element. This can be seen in Example 1.

#### <process> and <preprocess>

These elements contain an ordered list of objects. The results of the first object is given to the second and so on down the chain.

The first object in a process chain must be an Extractor, as the input data is either a string, a DOM (Document Object Model) node or a SAX event list as appropriate to the XPath evaluation. The result of a process chain must be a hash, typically from an Extractor or a Normalizer . However if the last object is an IndexStore , it will be used to store the terms rather than the default.

The input to a preprocess chain is a Record , so the first object is most likely to be a Transformer. The result must also be a Record , so the last object is most likely to be a Parser .

For existing processing objects that can be used in these fields, see the object documentation.

### 4.2.5 Example 1

```
<subConfig type="index" id="zrx-idx-9">
    <objectType>index.ProximityIndex</objectType>
    <paths>
        <object type="indexStore" ref="zrxIndexStore"/>
    </paths>
    <source>
        <preprocess>
            <object type="transformer" ref="zeerexTxr"/>
            <object type="parser" ref="SaxParser"/>
        </preprocess>
        <xpath>name/value</xpath>
        <xpath xmlns:zrx="http://explain.z3950.org/dtd/2.0">zrx:name/zrx:value</xpath>
        <process>
            <object type="extractor" ref="ExactParentProximityExtractor"/>
            <object type="normalizer" ref="CaseNormalizer"/>
        </process>
    </source>
    <options>
        <setting type="sortStore">true</setting>
        <setting type="lr_constant0">-3.7</setting>
    </options>
</subConfig>
```

### 4.2.6 Example 2

```
<subConfig type="selector" id="indexXPath">
    <objectType>cheshire3.selector.XPathSelector</objectType>
    <source>
        <xpath>/explain/indexInfo/index/title</xpath>
        <xpath>/explain/indexInfo/index/description</xpath>
    </source>
</subConfig>

<subConfig type="index" id="zrx-idx-10">
    <objectType>index.ProximityIndex</objectType>
    <paths>
        <object type="indexStore" ref="zrxIndexStore"/>
    </paths>
    <source mode="data">
        <selector ref="indexXPath"/>
        <process>
            <object type="extractor" ref="ProximityExtractor"/>
            <object type="normalizer" ref="CaseNormalizer"/>
            <object type="normalizer" ref="PossessiveNormalizer"/>
        </process>
    </source>
    <source mode="any|all|=">
        <process>
            <object type="extractor" ref="PreserveMaskingProximityExtractor"/>
            <object type="normalizer" ref="CaseNormalizer"/>
            <object type="normalizer" ref="PossessiveNormalizer"/>
```

```
        </process>
    </source>
</subConfig>
```

## 4.3 Cheshire3 Configuration - Protocol Map

### 4.3.1 Introduction

ZeeRex is a schema for service description and is required for SRU but it can also be used to describe Z39.50, OAI-PMH (Open Archives Initiative Protcol for Metadata Harvesting) and other information retrieval protocols.

As such, a ZeeRex description is required for each database. The full ZeeRex documentation is available at http://explain.z3950.org/ along with samples, schemas and so forth. It is also being considered as the standard service description schema in the NISO Metasearch Initiative, so knowing about it won't hurt you any.

In order to map from a CQL (the primary query language for Cheshire3 and SRU) query, we need to know the correlation between CQL index name and Cheshire3's `Index` object. Defaults for the SRU handler for the database are also drawn from this file, such as the default number of records to return and the default record schema in which to return results. Mappings between requested schema and a `Transformer` object are also possible. These mappings are all handled by a ProtocolMap.

### 4.3.2 ZeeRex Elements/Attributes of Particular Significance for Cheshire3

#### `<database>`

If you plan to make your database available over SRU, then the contents of the field MUST correspond with that which has been configured as the mount point for the SRU web application in Apache (or an alternative Python web framework), i.e. if you configured with mapping /api/sru/ to the `sruApacheHandler` code, then the first part of the database MUST be api/sru/.

Obviously the rest of the information in serverInfo should be correct as well, but without the database field being correct, it won't be available over SRU.

#### `c3:index`

This attribute may be present on an index element, or a supports element within `<configInfo>` within an `<index>`. It maps that particular index, or the use of the index with a `<relation>` or `<relationModifier>`, to the `Index` object with the given id. `<relationModifiers>` and `<relations>` will override the index as appropriate.

#### `c3:transformer`

Similar to c3:index, this can be present on a `<schema>` element and maps that schema to the `Transformer` used to process the internal schema into the requested one. If the schema is the one used internally, then the attribute should not be present.

### 4.3.3 Paths

**zeerexPath** In the configuration for the ProtocolMap object, this contains the path to the ZeeRex file to read.

### 4.3.4 Examples

<subConfig> within the main Database configuration (see *Cheshire3 Configuration* for details.):

```
<subConfig type="protocolMap" id="l5rProtocolMap">
    <objectType>protocolMap.CQLProtocolMap</objectType>
    <paths>
        <object type="zeerexPath">sru_zeerex.xml</path>
    </paths>
</subConfig>
```

Contents of the sru_zeerex.xml file:

```
<explain id="org.o-r-g.srw-card" authoritative="true"
    xmlns="http://explain.z3950.org/dtd/2.0/"
    xmlns:c3="http://www.cheshire3.org/schemas/explain/">
    <serverInfo protocol="srw/u" version="1.1" transport="http">
        <host>srw.cheshire3.org</host>
        <port>8080</port>
        <database numRecs="3492" lastUpdate="2002-11-26 23:30:00">srw/l5r</database>
    </serverInfo>
    [...]
    <indexInfo>
        <set identifier="http://srw.cheshire3.org/contextSets/ccg/1.0/" name="ccg"/>
        <set identifier="http://srw.cheshire3.org/contextSets/ccg/l5r/1.0/" name="ccg_l5r"/>
        <set identifier="info:srw/cql-context-set/1/dc-v1.1" name="dc"/>

        <index c3:index="l5r-idx-1">
            <title>Card Name</title>
            <map>
                <name set="dc">title</name>
            </map>
            <configInfo>
                <supports type="relation" c3:index="l5r-idx-1">exact</supports>
                <supports type="relation" c3:index="l5r-idx-15">any</supports>
                <supports type="relationModifier" c3:index="l5r-idx-15">word</supports>
                <supports type="relationModifier" c3:index="l5r-idx-1">string</supports>
                <supports type="relationModifier" c3:index="l5r-idx-16">stem</supports>
            </configInfo>
        </index>
    </indexInfo>
    <schemaInfo>
        <schema identifier="info:srw/schema/1/dc-v1.1"
            location="http://www.loc.gov/zing/srw/dc.xsd"
            sort="false" retrieve="true" name="dc"
            c3:transformer="l5rDublinCoreTxr">
            <title>Dublin Core</title>
        </schema>
    </schemaInfo>
</explain>
```

## 4.4 Cheshire3 Configuration - Workflows

### 4.4.1 Introduction

Workflow can be configured to define a series of processing steps that are common to several Cheshire3 Database

or `Server`, as an alternative to writing customized code for each.

### 4.4.2 Workflow Configuration Elements

#### <workflow>

Base wrapping tags for workflows; analagous to *<process> and <preprocess>* in *Index configurations*.

Contains an ordered list of *<object>s*. The results of the first object is given to the second and so on down the chain. It should be apparent that subsequent objects must be able to accept as input, the result of the previous.

#### <object>

A call to an object within the system. `<object>` s define the following attributes:

**type [ mandatory ]** Specifies the type of the object within the Cheshire3 framework. Broadly speaking this may be a:

- preParser
- parser
- database
- recordStore
- index
- logger
- transformer
- workflow

**ref** A reference to a configured object within the system. If unspecified, the current `Session` is used to determine which `Server`, `Database`, `RecordStore` and so forth should be used.

**function** The name of the method to call on the object. If unspecified, the default function for the particular type of object is called.

For existing processing objects that can be used in these fields, see the *object documentation*.

#### <log>

Log text to a `Logger` object. A reference to a configured `Logger` may be provided using the `ref` attribute. If no `ref` attribute is present, the `Database` 's default logger is used.

#### <assign>

Assign a specified value to a variable with a given name. Requires both of the following attributes:

**from [ *mandatory* ]** the value to assign

**to [ *mandatory* ]** a name to refer to the variable

#### <fork>

Feed the current input into each processing fork. [ more details to follow in v1.1]

#### <for-each>

Iterate/loop through the items in the input object. Like *<workflow>* contains an ordered list of *<object>s* . Each of the items in the input is run through the chain of processing objects.

#### <try>

Allows for error catching. Any errors that occur within this element will not cause the `Workflow` to exit with a failure. Must be followed by one *<except>* elements, which may in turn also be followed by one *<else>* element.

#### <except>

Enables error handling. This element may only follow a *<try>* element. Specifies action to take in the event of an error occurring during the work executed within the preceding *<try>*.

#### <else>

Success handling. This element may follow a *<try>* / *<except>* pair.

Specifies the action to take in the event that no errors occur within the preceding *<try>*.

#### <continue/>

Skip remaining processing steps, and move on to next iteration while inside a *<for-each>* loop element. May not contain any further elements or attributes. This can be useful in the error handling *<except>* element, e.g. if a document cannot be parsed, it cannot be indexed, so skip to next `Document` in the `DocumentFactory`.

#### <break/>

Break out of a *<for-each>* loop element, skipping all subsequent processing steps, and all remaining iterations. May not contain any further elements or attributes.

#### <raise/>

Raise an error occurring within the preceding *<try>* to the calling script or `Workflow`. May only be used within an *<except>* element. May not contain any further elements or attributes.

#### <return/>

Return the result of the previous step to the calling script or `Workflow`. May not contain any further elements or attributes.

## 4.5 Introduction

As Cheshire3 is so flexible and modular in the way that it can be implemented and then the pieces fitted together, it requires configuration files to set up which pieces to use and in which order. The configuration files are also very modular, allowing as many objects to be defined in one file as desired and then imported as required. They are put together from a small number of elements, with some additional constructions for specialized objects.

A very basic default configuration for a Database can be obtained using the **cheshire3-init** command described in *Cheshire3 Commands Reference*. The generated default configuration can then be used as a base on which to build.

Every object in the system that is not instantiated from a request or as the result of processing requires a configuration section. Many of these configurations will just contain the object class to instantiate and an identifier with which to refer to the object. Object constructor functions are called with the top DOM node of their configuration and another object to be used as a parent. This allows a tree hierarchy of objects, with a Server at the top level. It also means that objects can handle their own specialized configuration elements, while leaving the common elements to the base configuration handler.

The main elements will be described here, the specialized elements and values will be described in object specific pages.

## 4.6 Configuration Elements

XML namespace is optional, but if used it must be:

`http://www.cheshire3.org/schemas/config/`

If you wish to generate configurations in Python and have Cheshire3 installed, then you can import the configuration namespace from `cheshire3.internal.CONFIG_NS`

### 4.6.1 <config>

The top level element of any configuration file is the config element, and contains at least one object to construct. It should have an `id` attribute containing an identifier for the object in the system, and a `type` attribute specifying what sort of object is being created.

If the configuration file is not for the top level Server, this element must contain an <objectType> element. It may also contain one of each of <docs>, <paths>, <subConfigs>, <objects> and <options> .

### 4.6.2 <objectType>

This element contains the module and class to use when instantiating the object, using the standard `package.module.class` Python syntax.

When using classes defined by external packages/modules it is expected that they will inherit from a base class in the Cheshire3 Object Model (specifically from a class in `cheshire3.baseObjects`), and conform to the public API defined therein.

### 4.6.3 <docs>

This element may be used to provide configured object level documentation.

e.g. to explain that a particular Tokenizer splits data into sentences based on some pre-defined pattern.

### 4.6.4 <paths>

This element may contain <path> and/or <object> elements to be stored when building the object in the system.

### 4.6.5 <path>

This element is used to refer to a path to a resource (usually a filepath) required by the object and has several attributes to govern this:

- It **must** have a 'type' attribute, saying what sort of thing the resource is. This is somewhat context dependent, but is either an object type (e.g. 'database', 'index') or a description of a file path (e.g. 'defaultPath', 'metadataPath').
- For configurations which are being included as an external file, the path element should have the same `id` attribute as the included configuration.
- For references to other configurations, a `ref` attribute is used to contain the identifier of the referenced object.
- Finally, for configuration files which are held in a `ObjectStore` object, the document's identifier within the store (rather than the identifier of the object it contains) should be put in a `docid` attribute.

**Note:** A `<path>` element may only occur within a <paths> , <subConfigs> or <objects> element.

### 4.6.6 <object>

Object elements are used to create references to other objects in the system by their identifier, for example the default `RecordStore` used by the `Database`.

There are two mandatory attributes, the `type` of object and `ref` for the object's identifier.

### 4.6.7 <options>

This section may include one or more <setting> (a value that can't be changed) and/or <default> (a value that can be overridden in a request) elements.

### 4.6.8 <setting> and <default>

`<setting>` and `<default>` have a `type` attribute to specify which setting/default the value is for and the contents of the element is the value for it.

Each class within the Cheshire3 Object Model will have different setting and default types.

**Note:** `<setting>` and `<default>` may only occur within an <options> element.

### 4.6.9 <subConfigs>

This wrapper element contains one or more <subConfig> elements. Each <subConfig> has the same model as the <config>, and hence a nested tree of configurations and subConfigurations can be constructed. It may also contain <path> elements with a file path to another file to read in and treat as further subConfigurations.

Cheshire3 employs 'Just In Time' instantiation of objects. That is to say they will be instantiated when required by the system, or when requested from their parent object in a script.

### 4.6.10 <subConfig>

This element has the same model as the <config> element to allow for nested configurations. `id` and `type` attributes are mandatory for this element.

### 4.6.11 <objects>

The objects element contains one or more path elements, each with a reference to an identifier for a <subConfig> ). This reference acts as an instruction to the system to actually instantiate the object from the configuration.

---

**Note:** while this is no longer required (due to the implementation of 'Just In Time' object instantiation) it remains in the configuration schema as there are still situation in which this may be desirable, e.g. to instantiate objects with long spin-up times at the server level.

---

## 4.7 Example

```
<config type="database" id="db_l5r">
    <objectType>database.SimpleDatabase</objectType>
    <paths>
        <path type="defaultPath">/home/cheshire/c3/cheshire3/l5r</path>
        <path type="metadataPath">metadata.bdb</path>
        <object type="recordStore" ref="l5rRecordStore"/>
    </paths>
    <options>
        <setting type="log">handle_search</setting>
    </options>
    <subConfigs>
        <subConfig type="parser" id="l5rAttrParser">
            <objectType>parser.SaxParser</objectType>
            <options>
                <setting type="attrHash">text@type</setting>
            </options>
        </subConfig>
        <subConfig id = "l5r-idx-1">
            <objectType>index.SimpleIndex</objectType>
            <paths>
                <object type="indexStore" ref="l5rIndexStore"/>
            </paths>
            <source>
                <xpath>/card/name</xpath>
                <process>
                    <object type="extractor" ref="ExactExtractor"/>
                    <object type="normalizer" ref="CaseNormalizer"/>
                </process>
            </source>
        </subConfig>
        <path type="index" id="l5r-idx-2">configs/idx2-cfg.xml<path>
    </subConfigs>
    <objects>
        <path ref="l5RAttrParser"/>
        <path ref="l5r-idx-1"/>
    </objects>
</config>
```

CHAPTER 5

# Cheshire3 Object Model

## 5.1 Cheshire3 Object Model - Abstract Base Class

### 5.1.1 API

**class** cheshire3.baseObjects.**C3Object**(*session*, *config*, *parent=None*)
Abstract Base Class for Cheshire3 Objects.

**add_auth**(*session*, *name*)
Add an authorisation layer on top of a named function.

**add_logging**(*session*, *name*)
Set a named function to log invocations.

**get_config**(*session*, *id*)
Return a configuration for the given object.

**get_default**(*session*, *id*, *default=None*)
Return the default value for an option on this object

**get_object**(*session*, *id*)
Return the object with the given id.

Searches first within this object's scope, or search upwards for it.

**get_path**(*session*, *id*, *default=None*)
Return the named path

**get_setting**(*session*, *id*, *default=None*)
Return the value for a setting on this object.

**remove_auth**(*session*, *name*)
Remove the authorisation requirement from the named function.

**remove_logging**(*session*, *name*)
Remove the logging from a named function.

### 5.1.2 Implementations

There are no pre-configured out-of-the-box ready objects of this type.

## 5.2 Cheshire3 Object Model - Database

### 5.2.1 API

**class** cheshire3.baseObjects.**Database**(*session*, *config*, *parent=None*)
A Database is a collection of Records and Indexes.

It is responsible for maintaining and allowing access to its components, as well as metadata associated with the collections. It must be able to interpret a request, splitting it amongst its known resources and then recombine the values into a single response.

**accumulate_metadata**(*session*, *obj*)
Accumulate metadata (e.g. size) from and object.

**add_record**(*session*, *rec*)
Ensure that a Record is registered with the database.

This method does not ensure persistence of the Record, nor index it, just perform registration, and accumulate its metadata.

**begin_indexing**(*session*)
Prepare to index Records.

Perform tasks before Records are to be indexed.

**commit_indexing**(*session*)
Finalize indexing, commit data to persistent storage.

Perform tasks after Records have been sent to all Indexes. For example, commit any temporary data to IndexStores

**commit_metadata**(*session*)
Ensure persistence of database metadata.

**index_record**(*session*, *rec*)
Index a Record, return the Record.

Send the Record to all Indexes registered with the Database to be indexed and then return the Record (for the sake of Workflows).

**reindex**(*session*)
Reindex all Records registered with the database.

**remove_record**(*session*, *rec*)
Unregister the Record.

This method does not delete the Record, nor unindex it, just de-registers the Record and subtracts its metadata from the whole.

**scan**(*session*, *clause*, *nTerms*, *direction='>='*)
Scan (browse) through an Index to return a list of terms.

Given a single clause CQL query, resolve to the appropriate Index and return an ordered term list with document frequencies and total occurrences with a maximum of nTerms items. Direction specifies whether to move backwards or forwards from the term given in clause.

**search**(*session*, *query*)
Search the database, return a ResultSet.

Given a CQL query, execute the query and return a ResultSet object.

**sort** (*session*, *resultSets*, *sortKeys*)
Merge, sort and return one or more ResultSets.

Take one or more resultSets, merge them and sort based on sortKeys.

**unindex_record** (*session*, *rec*)
Unindex a Record, return the Record.

Sends the Record to all Indexes registered with the Database to be removed/unindexed.

### 5.2.2 Implementations

**class** cheshire3.database.**SimpleDatabase** (*session*, *config*, *parent*)
Default database implementation

**class** cheshire3.database.**OptimisingDatabase** (*session*, *config*, *parent*)
Experimental query optimising database

### 5.2.3 Configurations

There are no pre-configured databases as this is totally application specific. Configuring a database it your primary task when beginning to use Cheshire3 for your data. There are some example databases including configuration available in the Cheshire3 Download Site.

You can also obtain a default `Database` configuration using **cheshire3-init** (see *Cheshire3 Commands Reference* for details.)

## 5.3 Cheshire3 Object Model - Document

### 5.3.1 API

**class** cheshire3.baseObjects.**Document** (*data*, *creator=''*, *history=[]*, *mimeType=''*, *parent=None*, *filename=''*, *tagName=''*, *byteCount=0*, *byteOffset=0*, *wordCount=0*)
A Document is a wrapper for raw data and its metadata.

A Document is the raw data which will become a Record. It may be processed into a Record by a Parser, or into another Document type by a PreParser. Documents might be stored in a DocumentStore, if necessary, but can generally be discarded. Documents may be anything from a JPG file, to an unparsed XML file, to a string containing a URL. This allows for future compatability with new formats, as they may be incorporated into the system by implementing a Document type and a PreParser.

**get_raw** (*session*)
Return the raw data associated with this document.

### 5.3.2 Implementations

The following implementations are included in the distribution by default:

**class** cheshire3.document.**StringDocument** (*data*, *creator=''*, *history=[]*, *mimeType=''*, *parent=None*, *filename=None*, *tagName=''*, *byteCount=0*, *byteOffset=0*, *wordCount=0*)

## 5.4 Cheshire3 Object Model - DocumentFactory

### 5.4.1 API

**class** cheshire3.baseObjects.**DocumentFactory**(*session*, *config*, *parent=None*)
A DocumentFactory takes raw data, returns one or more Documents.

A DocumentFacory can be used to return Documents from e.g. a file, a directory containing many files, archive files, a URL, or a web-based API.

**get_document**(*session*, *n=-1*)
Return the Document at index n.

**load**(*session*, *data*, *cache=None*, *format=None*, *tagName=None*, *codec=''*)
Load documents into the document factory from data.

Returns the DocumentFactory itself which acts as an iterator DocumentFactory's load function takes session, plus:

**data := the data to load. Could be a filename, a directory name,** the data as a string, a URL to the data etc.

**cache := setting for how to cache documents in memory when reading** them in.

**format := format of the data parameter. Many options, most common:**

- xml – XML file. May contain multiple records
- dir – a directory containing files to load
- tar – a tar file containing files to load
- zip – a zip file containing files to load
- marc – a file with MARC records (library catalogue data)
- http – a base HTTP URL to retrieve

tagName := name of the tag which starts (and ends!) a Record.

codec := name of the codec in which the data is encoded.

**classmethod register_stream**(*session*, *format*, *cls*)
Register a new format, handled by given DocumentStream (cls).

Class method to register an implementation of a DocumentStream (cls) against a name for the format parameter (format) in future calls to load().

### 5.4.2 Implementations

The following implementations are included in the distribution by default:

**class** cheshire3.documentFactory.**SimpleDocumentFactory**(*session*, *config*, *parent*)

**class** cheshire3.documentFactory.**ClusterExtractionDocumentFactory**(*session*, *config*, *parent*)
Load lots of records, cluster and return the cluster documents.

## 5.5 Cheshire3 Object Model - DocumentStore

### 5.5.1 API

**class** cheshire3.baseObjects.**DocumentStore**(*session*, *config*, *parent=None*)
A persistent storage mechanism for Documents and their metadata.

**create_document**(*session*, *doc=None*)
Create an identifier, store and return a Document

Generate a new identifier. If a Document is given, assign the identifier to the Document and store it using store_document. If Document not given create a placeholder Document. Return the Document.

**delete_document**(*session*, *id*)
Delete the Document with the given identifier from storage.

**fetch_document**(*session*, *id*)
Fetch and return Document with the given identifier.

**store_document**(*session*, *doc*)
Store a Document that already has an identifier assigned.

### 5.5.2 Implementations

The following implementations are included in the distribution by default:

**class** cheshire3.documentStore.**BdbDocumentStore**(*session*, *config*, *parent*)

**class** cheshire3.documentStore.**FileSystemDocumentStore**(*session*, *config*, *parent*)

In addition to the default implementation, the cheshire3.sql provides the following implementations:

## 5.6 Cheshire3 Object Model - Extractor

### 5.6.1 API

**class** cheshire3.baseObjects.**Extractor**(*session*, *config*, *parent=None*)
An Extractor takes selected data and returns extracted values.

An Extractor is a processing object called by an Index with the value returned by a Selector, and extracts the values into an appropriate data structure (a dictionary/hash/associative array).

Example Extractors might extract all text from within a DOM node / etree Element, or select all text that occurs between a pair of selected DOM nodes / etree Elements.

Extractors must also be used on the query terms to apply the same keyword processing rules, for example.

**process_eventList**(*session*, *data*)
Process a list of SAX events serialized in C3 internal format.

**process_node**(*session*, *data*)
Process a DOM node.

**process_string**(*session*, *data*)
Process and return the value of a raw string.

e.g. from an attribute value or the query.

**process_xpathResult**(*session*, *data*)
Process the result of an XPath expression.

Convenience function to wrap the other process_* functions and do type checking.

### 5.6.2 Implementations

The following implementations are included in the distribution by default:

**class** cheshire3.extractor.**SimpleExtractor**(*session*, *config*, *parent*)
Base extractor, extracts exact text.

**class** cheshire3.extractor.**TeiExtractor**(*session*, *config*, *parent*)

**class** cheshire3.extractor.**SpanXPathExtractor**(*session*, *config*, *parent*)
Select all text that occurs between a pair of selections.

## 5.7 Cheshire3 Object Model - Index

### 5.7.1 API

**class** cheshire3.baseObjects.**Index**(*session*, *config*, *parent=None*)
An Index defines an access point into the Records.

An Index is an object which defines an access point into Records and is responsible for extracting that information from them. It can then store the information extracted in an IndexStore.

The entry point can be defined using one or more Selectors (e.g. an XPath expression), and the extraction process can be defined using a Workflow chain of standard objects. These chains must start with an Extractor, but from there might then include Tokenizers, PreParsers, Parsers, Transformers, Normalizers, even other Indexes. A processing chain usually finishes with a TokenMerger to merge identical tokens into the appropriate data structure (a dictionary/hash/associative array)

An Index can also be the last object in a regular Workflow, so long as a Selector object is used to find the data in the Record immediately before an Extractor.

**begin_indexing**(*session*)
Prepare to index Records.

Perform tasks before indexing any Records.

**clear**(*session*)
Clear all data from Index.

**commit_indexing**(*session*)
Finalize indexing.

Perform tasks after Records have been indexed.

**construct_resultSet**(*session*, *terms*, *queryHash={}*)
Create and return a ResultSet.

Take a list of the internal representation of terms, as stored in this Index, create and return an appropriate ResultSet object.

**construct_resultSetItem**(*session*, *term*, *rsiType=''*)
Create and return a ResultSetItem.

Take the internal representation of a term, as stored in this Index, create and return a ResultSetItem from it.

**delete_record**(*session*, *rec*)
Delete a Record from the Index.

Identify terms from the Record and delete them from IndexStore. Depending on the configuration of the Index, it may be necessary to do this by repeating the extracting the terms from the Record, finding and removing them. Hence the Record must be the same as the one that was indexed.

**deserialize_term**(*session*, *data*, *nRecs=-1*, *prox=1*)
Deserialize and return the internal representation of a term.

Return the internal representation of a term as recreated from a string serialization from storage. Used as a callback from IndexStore to take serialized data and produce list of terms and document references.

data := string (usually retrieved from indexStore) nRecs := number of Records to deserialize (all by default) prox := boolean flag to include proximity information

**extract_data**(*session*, *rec*)
Extract data from the Record.

Deprecated?

**fetch_proxVector**(*session*, *rec*, *elemId=-1*)
Fetch and return a proximity vector for the given Record.

**fetch_summary**(*session*)
Fetch and return summary data for all terms in the Index.

e.g. for sorting, then iterating. USE WITH CAUTION! Everything done here for speed.

**fetch_term**(*session*, *term*, *summary*, *prox*)
Fetch and return the data for the given term.

**fetch_termById**(*session*, *termId*)
Fetch and return the data for the given term id.

**fetch_termFrequencies**(*session*, *mType*, *start*, *nTerms*, *direction*)
Fetch and return a list of term frequency tuples.

**fetch_termList**(*session*, *term*, *nTerms*, *relation*, *end*, *summary*)
Fetch and return a list of terms from the index.

**fetch_vector**(*session*, *rec*, *summary*)
Fetch and return a vector for the given Record.

**index_record**(*session*, *rec*)
Index and return a Record.

Accept a Record to index. If begin indexing has been called, the index might not commit any data until commit_indexing is called. If it is not in batch mode, then index_record will also commit the terms to the indexStore.

**merge_term**(*session*, *currentData*, *newData*, *op='replace'*, *nRecs=0*, *nOccs=0*)
Merge newData into currentData and return the result.

Merging takes the currentData and can add, replace or delete the data found in newData, and then returns the result. Used as a callback from IndexStore to take two sets of terms and merge them together.

currentData := output of deserialize_terms newData := flat list op := replace | add | delete nRecs := total records in newData nOccs := total occurrences in newdata

**scan** (*session*, *clause*, *nTerms*, *direction='>='*)
Scan (browse) through an Index to return a list of terms.

Given a single clause CQL query, return an ordered term list with document frequencies and total occurrences with a maximum of nTerms items. Direction specifies whether to move backwards or forwards from the term given in clause.

**search** (*session*, *clause*, *db*)
Search this Index, return a ResultSet.

Given a CQL query, execute the query and return a ResultSet object.

**serialize_term** (*session*, *termId*, *data*, *nRecs=0*, *nOccs=0*)
Return a string serialization representing the term.

Return a string serialization representing the term for storage purposes. Used as a callback from IndexStore to serialize a list of terms and document references to be stored.

termId := numeric ID of term being serialized data := list of longs nRecs := number of Records containing the term, if known nOccs := total occurrences of the term, if known

**sort** (*session*, *rset*)
Sort and return a ResultSet object.

Sort and return a ResultSet object based on the values extracted according to this index.

**store_terms** (*session*, *data*, *rec*)
Store the indexed Terms in the configured IndexStore.

### 5.7.2 Implementations

The following implementations are included in the distribution by default:

**class** cheshire3.index.**SimpleIndex** (*session*, *config*, *parent*)

**class** cheshire3.index.**ProximityIndex** (*session*, *config*, *parent*)
Index that can store term locations to enable proximity search.

An Index that can store element, word and character offset location information for term entries, enabling phrase, adjacency searches etc.

Need to use an Extractor with prox setting and a ProximityTokenMerger

**class** cheshire3.index.**XmlIndex** (*session*, *config*, *parent*)
Index to store terms as XML structure.

e.g.:

```
<rs tid="" recs="" occs="">
    <r i="DOCID" s="STORE" o="OCCS"/>
</rs>
```

**class** cheshire3.index.**XmlProximityIndex** (*session*, *config*, *parent*)
ProximityIndex to store terms as XML structure.

e.g.:

```
<rs tid="" recs="" occs="">
  <r i="DOCID" s="STORE" o="OCCS">
    <p e="ELEM" w="WORDNUM" c="CHAROFFSET"/>
  </r>
</rs>
```

**class** cheshire3.index.**RangeIndex**(*session*, *config*, *parent*)
Index to enable searching over one-dimensional range (e.g. time).

Need to use a RangeTokenMerger

**class** cheshire3.index.**BitmapIndex**(*session*, *config*, *parent*)

**class** cheshire3.index.**RecordIdentifierIndex**(*session*, *config*, *parent=None*)

**class** cheshire3.index.**PassThroughIndex**(*session*, *config*, *parent*)
Special Index pull in search terms from another Database.

## 5.8 Cheshire3 Object Model - IndexStore

### 5.8.1 API

**class** cheshire3.baseObjects.**IndexStore**(*session*, *config*, *parent=None*)
A persistent storage mechanism for terms organized by Indexes.

Not an ObjectStore, just looks after Indexes and their terms.

**begin_indexing**(*session*, *index*)
Prepare to index Records.

Perform tasks as required before indexing begins, for example creating batch files.

**clean_index**(*session*, *index*)
Remove all the terms from an Index, but keep the specification.

**commit_centralIndexing**(*session*, *index*, *filePath*)
Finalize indexing for given index in single process context.

Commit data from the indexing process to persistent storage. Called automatically unless indexing is being carried out in distributed context. In this case, must be called in only one of the processes.

**commit_indexing**(*session*, *index*)
Finalize indexing for the given Index.

Perform tasks after all Records have been sent to given Index. For example, commit any temporary data to disk.

**construct_resultSetItem**(*session*, *recId*, *recStoreId*, *nOccs*, *rsiType=None*)
Create and return a ResultSetItem.

Take the internal representation of a term, as stored in this Index, create and return a ResultSetItem from it.

**contains_index**(*session*, *index*)
Does the IndexStore currently store the given Index.

**create_index**(*session*, *index*)
Create an index in the store.

**create_term**(*session*, *index*, *termId*, *resultSet*)
Take resultset and munge to Index format, serialise, store.

**delete_index**(*session*, *index*)
Completely delete an index from the store.

**delete_terms** (*session*, *index*, *terms*, *rec=None*)
Delete the given terms from Index.

Optionally only delete terms for a particular Record.

**fetch_proxVector** (*session*, *index*, *rec*, *elemId=-1*)
Fetch and return a proximity vector for the given Record.

**fetch_sortValue** (*session*, *index*, *item*)
Fetch a stored value for the given Record to use for sorting.

**fetch_summary** (*session*, *index*)
Fetch and return summary data for all terms in the Index.

e.g. for sorting, then iterating. USE WITH CAUTION! Everything done here for speed.

**fetch_term** (*session*, *index*, *term*, *summary=0*, *prox=0*)
Fetch and return data for a single term.

**fetch_termById** (*session*, *index*, *termId*)
Fetch and return data for a single term based on term identifier.

**fetch_termFrequencies** (*session*, *index*, *mType*, *start*, *nTerms*, *direction*)
Fetch and return a list of term frequency tuples.

**fetch_termList** (*session*, *index*, *term*, *nTerms=0*, *relation=''*, *end=''*, *summary=0*, *reverse=0*)
Fetch and return a list of terms for an Index.

**Parameters**

- **numReq** (*integer*) – how many terms are wanted.
- **relation** – which order to scan through the index.
- **end** – a point to end at (e.g. between A and B)
- **summary** – only return frequency info, not the pointers to

matching records. :type summary: boolean (or something that can be evaluated as True or False) :param reverse: use the reversed index if available (eg 'xedni' not 'index'). :rtype: list

**fetch_vector** (*session*, *index*, *rec*, *summary=0*)
Fetch and return a vector for the given Record.

**store_terms** (*session*, *index*, *terms*, *rec*)
Store terms in the index for a given Record.

### 5.8.2 Implementations

The following implementations are included in the distribution by default:

**class** cheshire3.indexStore.**BdbIndexStore** (*session*, *config*, *parent*)

In addition to the default implementation, the cheshire3.sql provides the following implementations:

## 5.9 Cheshire3 Object Model - ObjectStore

### 5.9.1 API

**class** cheshire3.baseObjects.**ObjectStore** (*session*, *config*, *parent=None*)
A persistent storage mechanism for configured Cheshire3 objects.

**create_object**(*session*, *obj=None*)
Create a slot for and store a serialized Cheshire3 Object.

Given a Cheshire3 object, create a serialized form of it in the database. Note: You should use create_record() as per RecordStore to create an object from a configuration.

**delete_object**(*session*, *id*)
Delete an object.

**fetch_object**(*session*, *id*)
Fetch and return an object.

**store_object**(*session*, *obj*)
Store an object, potentially overwriting an existing copy.

### 5.9.2 Implementations

The following implementations are included in the distribution by default:

**class** cheshire3.objectStore.**BdbObjectStore**(*session*, *config*, *parent*)
BerkeleyDB based implementation of an ObjectStore.

Store XML records in RecordStore, retrieve and instantiate when requested.

In addition to the default implementation, the cheshire3.sql provides the following implementations:

## 5.10 Cheshire3 Object Model - Parser

### 5.10.1 API

**class** cheshire3.baseObjects.**Parser**(*session*, *config*, *parent=None*)
A Parser takes a Document and parses it to a Record.

Parsers could be viewed as Record Factories. They take a Document containing some data and produce the equivalent Record.

Often a simple wrapper around an XML parser, however implementations also exist for various types of RDF data.

**process_document**(*session*, *doc*)
Take a Document, parse it and return a Record object.

### 5.10.2 Implementations

The following implementations are included in the distribution by default:

**class** cheshire3.parser.**MinidomParser**(*session*, *config*, *parent=None*)
Use default Python Minidom implementation to parse document.

**class** cheshire3.parser.**SaxParser**(*session*, *config*, *parent*)
Default SAX based parser. Creates SaxRecord.

**class** cheshire3.parser.**StoredSaxParser**(*session*, *config*, *parent=None*)

**class** cheshire3.parser.**LxmlParser**(*session*, *config*, *parent*)
lxml based Parser. Creates LxmlRecords

**class** cheshire3.parser.**LxmlHtmlParser**(*session*, *config*, *parent*)
lxml based parser for HTML documents.

**class** cheshire3.parser.**PassThroughParser**(*session*, *config*, *parent=None*)
Take a Document that already contains parsed data and return a Record.

Copy the data from a document (eg list of sax events or a dom tree) into an appropriate record object.

**class** cheshire3.parser.**MarcParser**(*session*, *config*, *parent=None*)
Creates MarcRecords which fake the Record API for Marc.

## 5.11 Cheshire3 Object Model - PreParser

### 5.11.1 API

**class** cheshire3.baseObjects.**PreParser**(*session*, *config*, *parent=None*)
A PreParser takes a Document and returns a modified Document.

For example, the input document might consist of SGML data. The output would be a Document containing XML data.

This functionality allows for Workflow chains to be strung together in many ways, and perhaps in ways which the original implemention had not foreseen.

**process_document**(*session*, *doc*)
Take a Document, transform it and return a new Document object.

### 5.11.2 Implementations

The following implementations are included in the distribution by default:

**class** cheshire3.preParser.**NormalizerPreParser**(*session*, *config*, *parent*)
Calls a named Normalizer to do the conversion.

**class** cheshire3.preParser.**UnicodeDecodePreParser**(*session*, *config*, *parent*)
PreParser to turn non-unicode into Unicode Documents.

A UnicodeDecodePreParser should accept a Document with content encoded in a non-unicode character encoding scheme and return a Document with the same content decoded to Python's Unicode implementation.

**class** cheshire3.preParser.**CmdLinePreParser**(*session*, *config*, *parent*)

**class** cheshire3.preParser.**FileUtilPreParser**(*session*, *config*, *parent*)
Call 'file' util to find out the current type of file.

**class** cheshire3.preParser.**MagicRedirectPreParser**(*session*, *config*, *parent*)
Map to appropriate PreParser based on incoming MIME type.

**class** cheshire3.preParser.**HtmlSmashPreParser**(*session*, *config*, *parent*)
Attempts to reduce HTML to its raw text

**class** cheshire3.preParser.**RegexpSmashPreParser**(*session*, *config*, *parent*)
Strip, replace or keep only data which matches a given regex.

**class** cheshire3.preParser.**HtmlTidyPreParser**(*session*, *config*, *parent*)

**class** cheshire3.preParser.**SgmlPreParser**(*session*, *config*, *parent*)
Convert SGML into XML

**class** cheshire3.preParser.**AmpPreParser**(*session*, *config*, *parent*)
Escape lone ampersands in otherwise XML text.

**class** cheshire3.preParser.**MarcToXmlPreParser**(*session*, *config*, *parent=None*)
Convert MARC into MARCXML

**class** cheshire3.preParser.**MarcToSgmlPreParser**(*session*, *config*, *parent=None*)
Convert MARC into Cheshire2's MarcSgml

**class** cheshire3.preParser.**TxtToXmlPreParser**(*session*, *config*, *parent=None*)
Minimally wrap text in <data> XML tags

**class** cheshire3.preParser.**PicklePreParser**(*session*, *config*, *parent=None*)
Compress Document content using Python pickle.

**class** cheshire3.preParser.**UnpicklePreParser**(*session*, *config*, *parent=None*)
Decompress Document content using Python pickle.

**class** cheshire3.preParser.**GzipPreParser**(*session*, *config*, *parent*)
Gzip a not-gzipped document.

**class** cheshire3.preParser.**GunzipPreParser**(*session*, *config*, *parent=None*)
Gunzip a gzipped document.

**class** cheshire3.preParser.**B64EncodePreParser**(*session*, *config*, *parent=None*)
Encode document in Base64.

**class** cheshire3.preParser.**B64DecodePreParser**(*session*, *config*, *parent=None*)
Decode document from Base64.

**class** cheshire3.preParser.**PrintableOnlyPreParser**(*session*, *config*, *parent*)
Replace or Strip non printable characters.

**class** cheshire3.preParser.**CharacterEntityPreParser**(*session*, *config*, *parent*)
Change named and broken entities to numbered.

Transform latin-1 and broken character entities into numeric character entities. eg &something; –> {

## 5.12 Cheshire3 Object Model - Record

### 5.12.1 API

**class** cheshire3.baseObjects.**Record**(*data*, *xml=''*, *docId=None*, *wordCount=0*, *byteCount=0*)
A Record is a wrapper for parsed data and its metadata.

Records in the system are commonly stored in an XML form. Attached to the record is various configurable metadata, such as the time it was inserted into the database and by which user. Records are stored in a RecordStore and retrieved via a persistent and unique identifier. The record data may be retrieved as a list of SAX events, as regularised XML, as a DOM tree or ElementTree.

**fetch_vector**(*session*, *index*, *summary=False*)
Fetch and return a vector for the Record from the given Index.

**get_dom**(*session*)
Return the DOM document node for the record.

**get_sax**(*session*)
Return the list of SAX events for the record

SAX events are serialized according to the internal Cheshire3 format.

**get_xml** (*session*)
Return the XML for the record as a serialized string.

**process_xpath** (*session*, *xpath*, *maps={}*)
Process and return the result of the given XPath

XPath may be either a string or a configured XPath, perhaps with some supplied namespace mappings.

### 5.12.2 Implementations

The following implementations are included in the distribution by default:

**class** cheshire3.record.**LxmlRecord** (*data*, *xml=''*, *docId=None*, *wordCount=0*, *byteCount=0*)

**class** cheshire3.record.**MinidomRecord** (*data*, *xml=''*, *docId=None*, *wordCount=0*, *byteCount=0*)

**class** cheshire3.record.**SaxRecord** (*data*, *xml=''*, *docId=None*, *wordCount=0*, *byteCount=0*)

**class** cheshire3.record.**MarcRecord** (*data*, *xml=''*, *docId=0*, *wordCount=0*, *byteCount=0*)
For dealing with Library MARC Records.

The class that you interact with will almost certainly depend on which Parser you used.

In addition to the default implementation, the cheshire3.graph provides the following implementations:

**class** cheshire3.graph.record.**GraphRecord** (*data*, *xml=''*, *docId=None*, *wordCount=0*, *byteCount=0*)

**class** cheshire3.graph.record.**OreGraphRecord** (*data*, *xml=''*, *docId=None*, *wordCount=0*, *byteCount=0*)

## 5.13 Cheshire3 Object Model - RecordStore

### 5.13.1 API

**class** cheshire3.baseObjects.**RecordStore** (*session*, *config*, *parent=None*)
A persistent storage mechanism for Records.

A RecordStore allows such operations as create, update, fetch and delete. It also allows fast retrieval of important Record metadata, for use in computing relevance rankings for example.

**create_record** (*session*, *rec=None*)
Create an identifier, store and return a Record.

Generate a new identifier. If a Record is given, assign the identifier to the Record and store it using store_record. If Record not given create a placeholder Record. Return the Record.

**delete_record** (*session*, *id*)
Delete the Record with the given identifier from storage.

**fetch_record** (*session*, *id*, *parser=None*)
Fetch and return the Record with the given identifier.

**fetch_recordMetadata** (*session*, *id*, *mType*)
Return the size of the Record, according to its metadata.

**replace_record**(*session*, *rec*)
Check for permission, replace stored copy of an existing Record.

Carry out permission checking before calling store_record.

**store_record**(*session*, *rec*, *transformer=None*)
Store a Record that already has an identifier assigned.

If a Transformer is given, use it to serialize the Record data.

### 5.13.2 Implementations

The following implementations are included in the distribution by default:

**class** cheshire3.recordStore.**BdbRecordStore**(*session*, *config*, *parent*)

**class** cheshire3.recordStore.**RedirectRecordStore**(*session*, *config*, *parent*)

**class** cheshire3.recordStore.**RemoteWriteRecordStore**(*session*, *config*, *parent*)
Listen for records and write

**class** cheshire3.recordStore.**RemoteSlaveRecordStore**(*session*, *config*, *parent*)

In addition to the default implementation, the cheshire3.sql provides the following implementations:

## 5.14 Cheshire3 Object Model - ResultSet

### 5.14.1 API

**class** cheshire3.baseObjects.**ResultSet**
A collection of results, commonly pointers to Records.

Typically created in response to a search on a Database. ResultSets are also the return value when searching an IndexStore or Index and are merged internally to combine results when searching multiple Indexes combined with boolean operators.

**combine**(*session*, *others*, *clause*)
Combine the ResultSets in 'others' into this ResultSet.

**deserialize**(*session*, *data*)
Deserialize string in data to return the populated ResultSet.

**order**(*session*, *spec*, *ascending=None*, *missing=None*, *case=None*, *accents=None*)
Re-order in-place based on the given spec and arguments.

**retrieve**(*session*, *nRecs*, *start=0*)
Return an iterable of nRecs Records starting at start.

**serialize**(*session*)
Return a string serialization of the ResultSet.

### 5.14.2 Implementations

The following implementations are included in the distribution by default:

**class** cheshire3.resultSet.**SimpleResultSet**(*session*, *data=None*, *id=''*, *recordStore=''*)

**class** cheshire3.resultSet.**SimpleResultSetItem**(*session*, *id=0*, *recStore=''*, *occs=0*, *database=''*, *diagnostic=None*, *weight=0.5*, *resultSet=None*, *numeric=None*)

[a SimpleResultSet consists of zero or more SimpleResultSetItems]

## 5.15 Cheshire3 Object Model - ResultSetStore

### 5.15.1 API

**class** cheshire3.baseObjects.**ResultSetStore**(*session*, *config*, *parent=None*)

A persistent storage mechanism for ResultSet objects.

**create_resultSet**(*session*, *rset=None*)

Create an identifier, store and return a ResultSet

Generate a new identifier. If a ResultSet is given, assign the identifier and store it using store_resultSet. If ResultSet is not given create a placeholder ResultSet. Return the ResultSet.

**delete_resultSet**(*session*, *id*)

Delete a ResultSet with the given identifier from storage.

**fetch_resultSet**(*session*, *id*)

Fetch and return Resultset with the given identifier.

**store_resultSet**(*session*, *rset*)

Store a ResultSet that already has an identifier assigned.

### 5.15.2 Implementations

The following implementations are included in the distribution by default:

**class** cheshire3.resultSetStore.**BdbResultSetStore**(*session*, *config*, *parent*)

In addition to the default implementation, the cheshire3.sql provides the following implementations:

## 5.16 Cheshire3 Object Model - Selector

### 5.16.1 API

**class** cheshire3.baseObjects.**Selector**(*session*, *config*, *parent=None*)

A Selector is a simple wrapper around a means of selecting data.

This could be an XPath or some other means of selecting data from the parsed structure in a Record.

**process_record**(*session*, *record*)

Process the given Record and return the results.

### 5.16.2 Implementations

The following implementations are included in the distribution by default:

**class** cheshire3.selector.**XPathSelector**(*session*, *config*, *parent*)

Selects data specified by XPath(s) from Records.

**class** cheshire3.selector.**TransformerSelector**(*session*, *config*, *parent*)
Selector that applies a Transformer to the Record to select data.

**class** cheshire3.selector.**MetadataSelector**(*session*, *config*, *parent*)
Selector specifying and attribute or function.

Selector that specifies an attribute or function to use to select data from Records.

**class** cheshire3.selector.**SpanXPathSelector**(*session*, *config*, *parent*)
Selects data from between two given XPaths.

Requires exactly two XPaths. The span starts at first configured XPath and ends at the second. The same XPath may be given as both start and end point, in which case each matching element acts as a start and stop point (e.g. an XPath for a page break).

## 5.17 Cheshire3 Object Model - Server

### 5.17.1 API

**class** cheshire3.baseObjects.**Server**(*session*, *configFile='serverConfig.xml'*)
A Server object is a collection point for other objects.

A Server is a collection point for other objects and an initial entry into the system for requests from a ProtocolHandler. A server might know about several Databases, RecordStores and so forth, but its main function is to check whether the request should be accepted or not and create an environment in which the request can be processed.

It will likely have access to a UserStore database which maintains authentication and authorization information. The exact nature of this information is not defined, allowing many possible backend implementations.

Servers are the top level of configuration for the system and hence their constructor requires the path to a local XML configuration file, however from then on configuration information may be retrieved from other locations such as a remote datastore to enable distributed environments to maintain synchronicity.

**get_config**(*session*, *id*)
Return a configuration for the given object.

**get_default**(*session*, *id*, *default=None*)
Return the default value for an option on this object

**get_object**(*session*, *id*)
Return the object with the given id.

Searches first within this object's scope, or search upwards for it.

**get_path**(*session*, *id*, *default=None*)
Return the named path

**get_setting**(*session*, *id*, *default=None*)
Return the value for a setting on this object.

### 5.17.2 Implementations

The following implementations are included in the distribution by default:

**class** cheshire3.server.**SimpleServer**(*session*, *configFile='serverConfig.xml'*)

## 5.18 Cheshire3 Object Model - TokenMerger

### 5.18.1 API

**class** cheshire3.baseObjects.**TokenMerger**(*session*, *config*, *parent=None*)
A TokenMerger merges identical tokens and returns a hash.

A TokenMerger takes an ordered list of tokens (i.e. as produced by a Tokenizer) and merges them into a hash. This might involve merging multiple tokens per key, while maintaining frequency, proximity information etc.

One or more Normalizers may occur in the processing chain between a Tokenizer and TokenMerger in order to reduce dimensionality of terms.

**process_hash**(*session*, *data*)
Merge and return tokens found in a hash.

**process_string**(*session*, *data*)
Merge and return tokens found in a raw string.

### 5.18.2 Implementations

The following implementations are included in the distribution by default:

**class** cheshire3.tokenMerger.**SimpleTokenMerger**(*session*, *config*, *parent=None*)

**class** cheshire3.tokenMerger.**ProximityTokenMerger**(*session*, *config*, *parent=None*)

**class** cheshire3.tokenMerger.**OffsetProximityTokenMerger**(*session*, *config*, *parent=None*)

**class** cheshire3.tokenMerger.**RangeTokenMerger**(*session*, *config*, *parent*)

**class** cheshire3.tokenMerger.**SequenceRangeTokenMerger**(*session*, *config*, *parent*)
Merges tokens into a range for use in RangeIndexes.

Assumes that we've tokenized a single value into pairs, which need to be concatenated into ranges.

**class** cheshire3.tokenMerger.**MinMaxRangeTokenMerger**(*session*, *config*, *parent*)
Merges tokens into a range for use in RangeIndexes.

Uses a forward slash (/) as the interval designator after ISO 8601.

**class** cheshire3.tokenMerger.**NGramTokenMerger**(*session*, *config*, *parent*)

**class** cheshire3.tokenMerger.**ReconstructTokenMerger**(*session*, *config*, *parent=None*)

**class** cheshire3.tokenMerger.**PhraseTokenMerger**(*session*, *config*, *parent*)

## 5.19 Cheshire3 Object Model - Tokenizer

### 5.19.1 API

**class** cheshire3.baseObjects.**Tokenizer**(*session*, *config*, *parent=None*)
A Tokenizer takes a string and returns an ordered list of tokens.

A Tokenizer takes a string of language and processes it to produce an ordered list of tokens.

Example Tokenizers might extract keywords by splitting on whitespace, or by identifying common word forms using a regular expression.

The incoming string is often in a data structure (dictionary / hash / associative array), as per output from Extractor.

**process_hash** (*session*, *data*)
Process and return tokens found in the keys of a hash.

**process_string** (*session*, *data*)
Process and return tokens found in a raw string.

### 5.19.2 Implementations

The following implementations are included in the distribution by default:

**class** cheshire3.tokenizer.**SimpleTokenizer** (*session*, *config*, *parent*)

**class** cheshire3.tokenizer.**RegexpSubTokenizer** (*session*, *config*, *parent*)
Substitute regex matches with a character, then split on whitespace.

A Tokenizer that replaces regular expression matches in the data with a configurable character (defaults to whitespace), then splits the result at whitespace.

**class** cheshire3.tokenizer.**RegexpSplitTokenizer** (*session*, *config*, *parent*)
A Tokenizer that simply splits at the regex matches.

**class** cheshire3.tokenizer.**RegexpFindTokenizer** (*session*, *config*, *parent*)
A tokenizer that returns all words that match the regex.

**class** cheshire3.tokenizer.**RegexpFindOffsetTokenizer** (*session*, *config*, *parent*)
Find tokens that match regex with character offsets.

A Tokenizer that returns all words that match the regex, and also the character offset at which each word occurs.

**class** cheshire3.tokenizer.**RegexpFindPunctuationOffsetTokenizer** (*session*, *config*, *parent*)

**class** cheshire3.tokenizer.**SentenceTokenizer** (*session*, *config*, *parent*)

**class** cheshire3.tokenizer.**LineTokenizer** (*session*, *config*, *parent*)
Trivial but potentially useful Tokenizer to split data on whitespace.

**class** cheshire3.tokenizer.**DateTokenizer** (*session*, *config*, *parent*)
Tokenizer to identify date tokens, and return only these.

Capable of extracting multiple dates, but slowly and less reliably than single ones.

**class** cheshire3.tokenizer.**DateRangeTokenizer** (*session*, *config*, *parent*)
Tokenizer to identify ranges of date tokens, and return only these.

e.g.

```
>>> self.process_string(session, '2003/2004')
['2003-01-01T00:00:00', '2004-12-31T23:59:59.999999']
>>> self.process_string(session, '2003-2004')
['2003-01-01T00:00:00', '2004-12-31T23:59:59.999999']
>>> self.process_string(session, '2003 2004')
['2003-01-01T00:00:00', '2004-12-31T23:59:59.999999']
>>> self.process_string(session, '2003 to 2004')
['2003-01-01T00:00:00', '2004-12-31T23:59:59.999999']
```

For single dates, attempts to expand this into the largest possible range that the data could specify. e.g. 1902-04 means the whole of April 1902.

```
>>> self.process_string(session, "1902-04")
['1902-04-01T00:00:00', '1902-04-30T23:59:59.999999']
```

**class** cheshire3.tokenizer.**PythonTokenizer**(*session*, *config*, *parent*)
Tokenize python source code into token/TYPE with offsets

## 5.20 Cheshire3 Object Model - Transformer

### 5.20.1 API

**class** cheshire3.baseObjects.**Transformer**(*session*, *config*, *parent=None*)
A Transformer transforms a Record into a Document.

A Transformer may be seen as the opposite of a Parser. It takes a Record and produces a Document. In many cases this can be handled by an XSLT stylesheet, but other instances might include one that returns a binary file based on the information in the Record.

Transformers may be used in the processing chain of an Index, but are more likely to be used to render a Record in a format or schema for delivery to the end user.

**process_record**(*session*, *rec*)
Take a Record, transform it and return a new Document object.

### 5.20.2 Implementations

The following implementations are included in the distribution by default:

**class** cheshire3.transformer.**XmlTransformer**(*session*, *config*, *parent=None*)
Return a Document containing the raw XML string of the record

**class** cheshire3.transformer.**Bzip2XmlTransformer**(*session*, *config*, *parent=None*)
Return a Document containing bzip2 compressed XML.

Return a Document containing the raw XML string of the record, compressed using the bzip2 algorithm.

**class** cheshire3.transformer.**SaxTransformer**(*session*, *config*, *parent=None*)

**class** cheshire3.transformer.**WorkflowTransformer**(*session*, *config*, *parent*)
Transformer to execute a workflow.

**class** cheshire3.transformer.**LxmlXsltTransformer**(*session*, *config*, *parent*)
XSLT transformer using Lxml implementation. Requires LxmlRecord.

Use Record's resultSetItem's proximity information to highlight query term matches.

**class** cheshire3.transformer.**LxmlOffsetQueryTermHighlightingTransformer**(*session*, *config*, *parent*)
Return Document with search hits higlighted based on character offsets.

Use character offsets from Record's resultSetItem's proximity information to highlight query term matches.

**class** cheshire3.transformer.**TemplatedTransformer**(*session*, *config*, *parent*)
Trasnform a Record using a Selector and a Python string.Template.

Transformer to insert the output of a Selector into a template string containing place-holders.

Template can be specified directly in the configuration using the template setting (whitespace is respected), or in a file using the templatePath path. If the template is specified in the configuration, XML reserved characters (<, >, & etc.) must be escaped.

This can be useful for Record types that are not easily transformed using more standard mechanism (e.g. XSLT), a prime example being GraphRecords

Example

config:

**<subConfig type="transformer" id="myTemplatedTransformer">** <objectType>cheshire3.transformer.TemplatedTransform
<paths>

<object type="selector" ref="mySelector"/> <object type="extractor" ref="SimpleExtractor"/>

</paths> <options>

**<setting type="template">** This is my document. The title is {0}. The author is {1}

</setting>

</options>

</subConfig>

selector config:

**<subConfig type="selector" id="mySelector">** <objectType>cheshire3.selector.XpathSelector</objectType>
<source>

<location type="xpath">//title</location> <location type="xpath">//author</location>

</source>

</subConfig>

**class** cheshire3.transformer.**MarcTransformer**(*session*, *config*, *parent*)
Transformer to converts records in marc21xml to marc records.

## 5.21 Cheshire3 Object Model - User

### 5.21.1 API

**class** cheshire3.baseObjects.**User**(*session*, *config*, *parent=None*)
A User represents a user of the system.

An object representing a user of the system to allow for convenient access to properties such as username, password, rights and permissions metadata.

Users may be stores and retrieved from an ObjectStore like any other configured or created C3Object.

**has_flag**(*session*, *flag*, *object=None*)
Does the User have the specified flag?

Check whether or not the User has the specified flag. This flag may be set regarding a particular object, for example write access to a particular ObjectStore.

### 5.21.2 Implementations

The following implementations are included in the distribution by default:

**class** cheshire3.user.**SimpleUser**(*session*, *config*, *parent*)

**check_password**(*session*, *password*)
Check the supplied en-clair password.

Check the supplied en-clair password by obfuscating it using the same algorithm and comparing it with the stored version. Return True/False.

## 5.22 Cheshire3 Object Model - Workflow

### 5.22.1 API

**class** cheshire3.baseObjects.**Workflow**(*session*, *config*, *parent=None*)
A Workflow defines a series of processing steps.

A Workflow is similar to the process chain concept of an index, but acts at a more global level. It will allow the configuration of a Workflow using Cheshire3 objects and simple code to be defined and executed for input objects.

For example, one might define a common Workflow pattern of PreParsers, a Parser and then indexing routines in the XML configuration, and then run each Document in a DocumentFactory through it. This allows users who are not familiar with Python, but who are familiar with XML and available Cheshire3 processing objects to implement tasks as required, by changing only configuration files. It thus also allows a user to configure personal workflows in a Cheshire3 system the code for which they don't have permission to modify.

**process**(*session*, **args*, ***kw*)
Executes the code as constructed from the XML configuration.

Executes the generate code on the given input arguments. The return value is the last object to be produced by the execution. This function is automatically written and compiled when the object is instantiated.

### 5.22.2 Implementations

The following implementations are included in the distribution by default:

**class** cheshire3.workflow.**SimpleWorkflow**(*session*, *config*, *parent*)
Default workflow implementation.

Translates XML to python and compiles it on object instantiation.

**class** cheshire3.workflow.**CachingWorkflow**(*session*, *config*, *parent*)
Slightly faster Workflow implementation that caches the objects.

Object must not be used in one database and then another database without first calling workflow.load_cache(session, newDatabaseObject).

## 5.23 Overview

## 5.24 Miscellaneous

### 5.24.1 Abstract Base Class

Abstract Base Class for all configurable objects within the Cheshire3 framework. It is not the base class for Data Objects :

- Document
- Record
- ResultSet
- User
- Query

See C3Object for details and API

### 5.24.2 Session

**class** cheshire3.baseObjects.**Session**(*user=None*, *logger=None*, *task=''*, *database=''*, *environment='terminal'*)

An object to be passed around amongst the processing objects to maintain a session. It stores, for example, the current environment, user and identifier for the database.

### 5.24.3 ProtocolMap

**class** cheshire3.baseObjects.**ProtocolMap**(*session*, *config*, *parent=None*)

A ProtocolMap maps incoming queries to internal capabilities.

A ProtocolMaps maps from an incoming query type to internal Indexes based on some specification.

## 5.25 Summary Objects

Objects that summarize and provide persistent storage for other objects and their metadata.

### 5.25.1 Server

A Server is a collection point for other objects and an initial entry into the system for requests from a ProtocolHandler. A Server might know about several Databases, RecordStore s and so forth, but its main function is to check whether the request should be accepted or not and create an environment in which the request can be processed.

It will likely have access to a UserStore database which maintains authentication and authorization information. The exact nature of this information is not defined, allowing many possible backend implementations.

Servers are the top level of configuration for the system and hence their constructor requires the path to a local XML configuration file, however from then on configuration information may be retrieved from other locations such as a remote datastore to enable distributed environments to maintain synchronicity.

### 5.25.2 Database

A Database is a collection of Records and Indexes.

It is responsible for maintaining and allowing access to its components, as well as metadata associated with the collections. It must be able to interpret a request, splitting it amongst its known resources and then recombine the values into a single response.

### 5.25.3 DocumentStore

A persistent storage mechanism for Document s and their metadata.

### 5.25.4 RecordStore

A persistent storage mechanism for Record s.

A RecordStore allows such operations as create, update, fetch and delete. It also allows fast retrieval of important Record metadata, for use in computing relevance rankings for example.

### 5.25.5 IndexStore

A persistent storage mechanism for terms organized by Indexes.

Not an ObjectStore, just looks after Indexes and their terms.

## 5.26 ResultSetStore

A persistent storage mechanism for ResultSet objects.

### 5.26.1 ObjectStore

A persistent storage mechanism for configured Cheshire3 objects.

## 5.27 Data Objects

Objects representing data to be stored, indexed, discovered or manipulated.

### 5.27.1 Document

A Document is a wrapper for raw data and its metadata.

A Document is the raw data which will become a Record. It may be processed into a Record by a Parser, or into another Document type by a PreParser. Documents might be stored in a DocumentStore, if necessary, but can generally be discarded. Documents may be anything from a JPG file, to an unparsed XML file, to a string containing a URL. This allows for future compatability with new formats, as they may be incorporated into the system by implementing a Document type and a PreParser.

### 5.27.2 Record

A Record is a wrapper for parsed data and its metadata.

Records in the system are commonly stored in an XML form. Attached to the Record is various configurable metadata, such as the time it was inserted into the Database and by which User. Records are stored in a RecordStore and retrieved via a persistent and unique identifier. The Record data may be retrieved as a list of SAX events, as serialized XML, as a DOM tree or ElementTree (depending on which *implementation* is used).

### 5.27.3 ResultSet

A collection of results, commonly pointers to Records.

Typically created in response to a search on a Database. ResultSets are also the return value when searching an IndexStore or Index and are merged internally to combine results when searching multiple Indexes combined with boolean operators.

### 5.27.4 User

A User represents a user of the system.

An object representing a user of the system to allow for convenient access to properties such as username, password, rights and permissions metadata.

Users may be stores and retrieved from an ObjectStore like any other configured or created C3Object.

## 5.28 Processing Objects

### 5.28.1 Workflow

A Workflow defines a series of processing steps.

A Workflow is similar to the process chain concept of an index, but acts at a more global level. It will allow the configuration of a Workflow using Cheshire3 objects and simple code to be defined and executed for input objects.

For example, one might define a common Workflow pattern of PreParsers, a Parser and then indexing routines in the XML configuration, and then run each Document in a DocumentFactory through it. This allows users who are not familiar with Python, but who are familiar with XML and available Cheshire3 processing objects to implement tasks as required, by changing only configuration files. It thus also allows a user to configure personal workflows in a Cheshire3 system the code for which they don't have permission to modify.

### 5.28.2 DocumentFactory

A DocumentFactory takes raw data, returns one or more Documents.

A DocumentFactory can be used to return Documents from e.g. a file, a directory containing many files, archive files, a URL, or a web-based API.

### 5.28.3 PreParser

A PreParser takes a Document and returns a modified Document.

For example, the input document might consist of SGML data. The output would be a Document containing XML data.

This functionality allows for Workflow chains to be strung together in many ways, and perhaps in ways which the original implemention had not foreseen.

### 5.28.4 Parser

A Parser takes a Document and parses it to a Record.

Parsers could be viewed as Record Factories. They take a Document containing some data and produce the equivalent Record.

Often a simple wrapper around an XML parser, however implementations also exist for various types of RDF data.

### 5.28.5 Index

An Index defines an access point into the Records.

An Index is an object which defines an access point into Records and is responsible for extracting that information from them. It can then store the information extracted in an IndexStore.

The entry point can be defined using one or more Selectors (e.g. an XPath expression), and the extraction process can be defined using a Workflow chain of standard objects. These chains must start with an Extractor, but from there might then include Tokenizers, PreParsers, Parsers, Transformers, Normalizers, even other Indexes. A processing chain usually finishes with a TokenMerger to merge identical tokens into the appropriate data structure (a dictionary/hash/associative array)

An Index can also be the last object in a regular Workflow, so long as a Selector object is used to find the data in the Record immediately before an Extractor.

### 5.28.6 Selector

A Selector is a simple wrapper around a means of selecting data.

This could be an XPath or some other means of selecting data from the parsed structure in a Record.

### 5.28.7 Extractor

An Extractor takes selected data and returns extracted values.

An Extractor is a processing object called by an Index with the value returned by a An Selector, and extracts the values into an appropriate data structure (a dictionary/hash/associative array).

Example An Extractors might extract all text from within a DOM node / etree Element, or select all text that occurs between a pair of selected DOM nodes / etree Elements.

Extractors must also be used on the query terms to apply the same keyword processing rules, for example.

### 5.28.8 Tokenizer

A Tokenizer takes a string and returns an ordered list of tokens.

A Tokenizer takes a string of language and processes it to produce an ordered list of tokens.

Example Tokenizers might extract keywords by splitting on whitespace, or by identifying common word forms using a regular expression.

The incoming string is often in a data structure (dictionary / hash / associative array), as per output from Extractor.

### 5.28.9 Normalizer

A Normalizer modifies terms to allow effective comparison.

Normalizer objects are chained after Extractors in order to transform the data from the Record or query.

Example Normalizers might standardize the case, perform stemming or transform a date into ISO8601 format.

Normalizers are also needed to transform the terms in a request into the same format as the term stored in the Index. For example a date index might be searched using a free text date and that would need to be parsed into the normalized form in order to compare it with the stored data.

### 5.28.10 TokenMerger

A TokenMerger merges identical tokens and returns a hash.

A TokenMerger takes an ordered list of tokens (i.e. as produced by a TokenMerger) and merges them into a hash. This might involve merging multiple tokens per key, while maintaining frequency, proximity information etc.

One or more Normalizers may occur in the processing chain between a Tokenizer and TokenMerger in order to reduce dimensionality of terms.

### 5.28.11 Transformer

A Transformer transforms a Record into a Document.

A Transformer may be seen as the opposite of a Parser. It takes a Record and produces a Document. In many cases this can be handled by an XSLT stylesheet, but other instances might include one that returns a binary file based on the information in the Record.

Transformers may be used in the processing chain of an Index, but are more likely to be used to render a Record in a format or schema for delivery to the end user.

## 5.29 Other Notable Modules

Other notable modules in the Cheshire3 framework:

- bootstrap
- dynamic Cheshire3 configured object dynamic creation.

  Module to support dynamic creation of Cheshire3 objects based on XML configurations.
- exceptions
- internal

- logger

CHAPTER 6

# Troubleshooting

## 6.1 Introduction

This page contains a list of common Python and Cheshire3 specific errors and exceptions.

It is hoped that it also offers some enlightenment as to what these errors and exception mean in terms of your configuration/code/data, and suggests how you might go about correcting them.

## 6.2 Common Run-time Errors

### 6.2.1 `AttributeError: 'NoneType' object has no attribute ...`

The object the system is trying to use is null i.e. of NoneType. There are several things that can cause this (.. hint:: The reported attribute might give you a clue to what type the object should be):

- The object does not exist in the architecture. This is often due to errors/omissions in the configuration file.

  **ACTION**

  Make sure that the object is configured (either at the database or server level).

  **Hint:** Remember that everything is configured hierarchically from the server, down to the individual subConfigs of each database.

- There is a discrepancy between the identifier used to configure the object, and that used to get the object for use in the script.

  **ACTION**

  Ensure that the identifier used to get the object in the script is the same as that used in the configuration.

  **Hint:** Check the spelling and case used.

- If the object is the result of a get or fetch operation (e.g., from a `DocumentFactory` or `ObjectStore`), it looks like it wasn't retrieved properly from the store.

  **ACTION**

  Afraid there's no easy answer to this one. Check that the requested object actually exists in the group/store.

- If the object is the result of a process request (e.g., to a `Parser`, `PreParser` or `Tranformer`), it looks like it wasn't returned properly by the processor.

**ACTION**

Afraid there's no easy answer here either. Check for any errors/exceptions raised during the processing operation.

### 6.2.2 AttributeError: x instance has no attribute 'y'

An instance of object type x, has neither an attribute or method called y.

**ACTION**

Check the API documentation for the object-type, and correct your script.

### 6.2.3 Cheshire3 Exception: 'x' referenced from 'y' has no configuration

An object referred to as 'x' in the configuration for object 'y' has no configuration.

**ACTION**

Make sure that object 'x' is configured in `subConfigs`, and that all references to object 'x' use the correct identifier string.

### 6.2.4 Cheshire3 Exception: Failed to build myProtocolMap: not well-formed ...

The `zeerex_srx.xml` file contains XML which is not well formed.

**ACTION**

Check this file at the suggested line and column and make the necessary corrections.

### 6.2.5 TypeError: cannot concatenate 'str' and 'int' objects

If the error message looks like the following:

```
File "../../code/baseStore.py", line 189, in generate_id
id = self.currentId +1
TypeError: cannot concatenate 'str' and 'int' objects
```

Then it's likely that your `RecordStore` is trying to create a new integer by incrementing the previous one, when the previous one is a string!

**ACTION**

This can easily be remedied by adding the following line to the `<paths>` section of the `<subConfig>` that defines the `RecordStore`:

```
<object type="idNormalizer" ref="StringIntNormalizer"/>
```

### 6.2.6 TypeError: some_method() takes exactly x arguments (y given)

The method you're trying to use requires x arguments, you only supplied y arguments.

**ACTION**

Check the API for the required arguments for this method.

**Hint:** All Cheshire3 objects require an instance of type `Session` as the first argument to their public methods.

### 6.2.7 UnicodeEncodeError: 'ascii' codec can't encode character u'\uXXXX' ...

Oh Dear! Somewhere within one of your `Documents` `Records` there is a character which cannot be encoded into ascii unicode.

**Tip:** Use a `UnicodeDecodePreParser` or `PrintableOnlyPreParser` to turn the unprintable unicode character into an XML character entity.

### 6.2.8 xml.sax._exceptions.SAXParseException: <unknown>:x:y: not well-formed ...

Despite the best efforts of the `PreParsers` there is badly formed XML within the document; possibly a malformed tag, or character entity.

**Hint:** Check the document source at line x, column y.

### 6.2.9 ConfigFileException: : Sort executable for indexStore does not exist

This means that the unix sort utility executable was not present at the configured location, and could not be found. You will need to configure it for your Cheshire3 server.

**ACTION**

Discover the path to the unix sort executable on your system by running the following command and making a note of the result:: of it:

```
which sort
```

Insert this value into the `sortPath` `<path>` in the `<paths>` sections of your server configuration file.

Removing the dependency on the unix sort utility is on the TODO list in our *issue tracker <https://github.com/cheshire3/cheshire3/issues/6>*.

## 6.3 Apache Errors

### 6.3.1 "No space left on device" Apache error

If there is space left on your hard drives, then it is almost certainly that the linux kernel has run out of semaphores for mod_python or Berkeley DB.

**ACTION**

You need to tweak the kernel performance a little. For more information, see *Clarens FAQ <http://clarens.sourceforge.net/index.php?docs+faq>*

CHAPTER 7

# Capabilities

What Cheshire3 can do:

- Create a `Database` of your documents, and put a search engine on top.
- Index the full text of the documents in your `Database`, and allow you to define your own `Index` of specific fields within each structured or semi-structured `Document` .
- Set up each `Index` to extract and normalize the data exactly the way you need (e.g. make an index of people's names as keywords, strip off possessive apostrophes, treat all names as lowercase)
- Search your `Database` to quickly find the `Document` you want. When searching the `Database` the user's search terms are treated the same way as the data, so a user doesn't need to know what normalization you've applied, they'll just get the right results!
- Advanced boolean search logic ('AND', 'OR', 'NOT') as well as proximity, phrase and range searching (e.g. for date/time periods).
- Return shared 'facets' of your search results to indicate ways in which a search could be refined.
- Scan through all terms in an `Index`, just like reading the index in a book.
- Add international standard webservice APIs to your database
- Use an existing Relation Database Management Systems as a source of documents.

[More Coming]

CHAPTER 8

# Indices and tables

- *genindex*
- *modindex*
- *search*

## C

cheshire3.bootstrap, 63
cheshire3.dynamic, 63
cheshire3.exceptions, 63
cheshire3.internal, 63
cheshire3.logger, 64