Webarchiv Documentation

Release 1.0

Christopher Pahl, Christoph Piechula

June 16, 2012

CONTENTS

Contents:

CHAPTER

ONE

COMMANDLINEINTERFACE

1.1 Quick-Reference

```
Usage:
  archive.py [--loglevel=<severity>] init [<path>]
  archive.py [--loglevel=<severity>] crawler
  archive.py [--loglevel=<severity>] javadapter
  archive.py [--loglevel=<severity>] db (--rebuild|--remove)
  archive.py [--loglevel=<severity>] repair
  archive.py config (--get=<confurl>|--set=<confurl><arg>)
  archive.py -h | --help
  archive.py --version

General Options:
  -h --help                Show this screen.
  --version                Show version.
  --loglevel=<loglevel>    Set the loglevel to any of debug, info, warning, error, critical.

DB Options:
  --rebuild                Rebuild Databse completely from XML Data.
  --remove                 Remove the Database completely.

Config Options:
  --set=<confurl><value>   Set a Value in the config permanently.
  --get=<confurl>          Acquire a Value in the config by it’s url.
```

1.2 Additional Notes

- The Commandline interfaces relies on submodules like *crawler*, *config* ...
- Submodules may have own options
- Before stating the submodule common options may be set (e.g. *–loglevel*)
- The submodules *javadapter* and *crawler* start a special shell
- In order to locate the config you either have to pass it explicitly, or your current working directory is at the archive root.

1.3 Implementation

class `cli.cmdparser.`**`Cli`**

Archive commandline intepreter

`cmd_loop``(`*shell*, *i*, *cv*`)`

The cmdloop runs in a seperate thread.

`handle_config``()`

Invokes Config Handler operations

`handle_crawler``()`

Starts and controls crawler commandline

`handle_db``()`

Handle “db” submodule

`handle_init``()`

Initializes archive paths

`handle_javadapter``()`

Starts javadapter commandline

`handle_repair``()`

Invokes archive rapair tool

CHAPTER

TWO

PYTHON TO JAVA INTERFACE (AKA JAVADAPTER)

2.1 Description

The Javadapter is a simple TCPServer that will listen by default on port `42421` on `localhost`. One may connect to this server and send one of the commands below. On success the server will send a reponse that is terminated with `OK`, otherwise `ACK Some Error Description.` is send.

The Server may be started via:

```
$ archive.py javadapter --start
# This will enter a special shell.
# Use 'help' to see what you can do there
```

2.2 List of commands

lock

description: Lock a domain and wait to a maximal time of 5 minutes, will return a timeout then

usage: `lock [domain]`

- domain is e.g. www.heise.de
- Returns nothing

examples:

```
$ lock www.heise.de
OK
$ lock www.heise.de
(... will timeout after 5 mins ...)
ACK Timeout occured.
$ unlock www.heise.de
OK
$ lock www.heise.de
OK
```

---

try_lock

description: As `lock`, but return immediately with `ACK Already locked.` if already locked previously.

usage: `try_lock [domain]`

- domain is e.g. www.heise.de
- Returns nothing

examples:

```
$ try_lock www.heise.de
OK
$ try_lock www.heise.de
ACK Already locked.
```

---

unlock

description: Unlock a previous lock

usage: `unlock [domain]`

- domain is e.g. www.heise.de
- Returns nothing

examples:

```
$ unlock www.heise.de
OK
$ unlock www.youporn.com
ACK Invalid Domain.
```

---

checkout

description: Checkout a certain branch (usually a commitTag or `master`) You do not need to manually set a lock for this.

usage: `checkout [domain] {branch_name}`

- domain is e.g. www.heise.de
- branch_name the entity to checkout, if omitted only the path is returned (if valid) and no git work is done.
- Returns: The Path to the checkout'd domain

warning: **Note:** You should always checkout `master` when done!

examples:

```
$ checkout www.hack.org 2012H06H15T19C08C15
/tmp/archive/content/www.hack.org
OK
$ checkout www.youporn.com
ACK Invalid Domain.
$ checkout www.hack.org no_branch_name
ACK checkout returned 1
```

---

commit

description: Make a commit on a certain domain.

usage: `commit [domain] {message}`

- domain is e.g. www.heise.de
- message is the commit message (optional, `edit` by default)
- Returns nothing

examples:

```
$ commit www.hack.org HelloWorld
ACK commit returned 1
# Uh-Oh, nothing to commit - add some content manually
user@arc $ touch /tmp/archive/content/www.hack.org/new_file
# Now commiting works:
$ commit www.hack.org
OK
```

---

list_commits:

description: List all commits on a certain domain and its current branch.

usage: `list_commits [domain]`

- domain is e.g. www.heise.de
- Returns a newline seperated list of commithashes

examples:

```
$ list_commits www.hack.org
6309b01f5b04b4e60c19f5dd147f935f40d94840
942f9a1da172592228d22ca638dd3f5ae583d285
OK
```

---

list_branches:

description: List all branches on a certain domain.

usage: `list_branches [domain]`

- domain is e.g. www.heise.de
- Returns a newline seperated list of branchnames

examples:

```
$ list_branches www.hack.org
2012H06H13T23C02C18
2012H06H15T19C07C46
2012H06H15T19C08C15
2012H06H15T21C57C35
2012H06H15T21C57C43
# (..snip..)
OK
```

2.3 Implementation

Actual function to start the server:

`javadapter.server.`**`start`**(*host='localhost'*, *port=42421*)
Start the Javadapter server, and exit once done

Host the host to start the server on (does anythinh but localhost work?)

Port the port on which the server listens on

Returns a server, on which shutdown() can be called

Convienience class to show a servershell:

class `javadapter.server.`**`ServerShell`**(*host='localhost'*, *port='42421'*, *server_instance=None*)
Command shell to manage javadapter

`do_EOF`(*arg*)
Shortcut for quit (Press CTRL+D)

`do_quit`(*arg*)
Quits the server

`do_start`(*arg*)
Start a server if not already active

`do_status`(*arg*)
Print current status of the Server

`do_stop`(*arg*)
Stop a running Server

GIT-HANDLING

3.1 Overview

Initialization:

```
$ git init
$ git checkout -b empty
# At least one commit is needed for a valid branch
$ echo 'This is Empty' > EMPTY
$ git add EMPTY
$ git commit -a -m 'Init'
# master will be used to track
# the most recent branch
```

Synchronization:

```
... lock ...
# Gehe zum leeren stand zurück,
# da sonst der neue branch die history
# des aktuellen erbt
$ git checkout empty
# Neuer branch mit ehem. Tagnamen
$ git checkout -b 24052012T1232
... rsync ins Archiv ...
$ git add .
$ git commit -am 'Seite xyz.com wurde gekrault'
# Nun ist 'master' mit dem aktuellsten Stand identisch
$ git branch -d master
$ git checkout -b master
... unlock ...
```

Reading/Writing on most recent data:

```
# Not git-work required
... lock ...
... read ...
... unlock ...

# Lesen / Editieren von alten Ständen
# Hierfür muss das Datum des alten Standes gegeben sein
# -f falls jemand unerlaubt änderungen gemacht hat
... lock ...
$ git checkout -f old_date
... lesen / schreiben ...
```

```
# Im Falle von schreiben:
$ git add .
$ git commit -am 'Edited old Kraul'
# Der Kopf des neuen branches zeigt nun auf den neuen commit
$ git checkout master
... unlock ...
```

Rough schema as ASCII-Art:

```
          -- Kraul1 -> edit <- branch '03052012T1232'
        /
Init -- ---- Kraul2 <- branch '15052012T1232'
|       \
|         -- Kraul3 <- branch '24052012T1232' <- branch 'master'
|
|
\-> branch 'empty'
```

Previously, with the `tag` approach:

```
Kraul1 -> Kraul2 -> Kraul3 -> Kraul4 <- branch 'master'
|         |         |         |
|         |         |          \
|         |         \           -- Tag 04
|         \           -- Tag 03
\           -- Tag 02
  -- Tag 01


# Vorteile:
# - Alte Stände editierbar
# - Überprüfung ob aktuell fällt weg
# - (Seltsamerweise) weniger Platzverbrauch

# Nachteile:
# - Alte Stände auschecken (vermutlich) langsamer als normale git tag checkouts
# - Traversieren über alle Stände (für DB Recover wird etwas schwieriger) - aber ist möglich.
```

3.2 Implementation

Wrapper for Git

This is highly simplified, and may be replaced by a faster, native implementation using Dunwhich. But that's not on the plan due our limited time.

Git commands (init e.g.) are tailored for use in this archive, less for general use.

class `crawler.git.`**`Git`**(*domain*)

A (overly-simple) Wrapper for the git binary

`branch`(*branch_name='empty'*)

create a new named branch

Branch_name the name of the new branch, may not exist yet

Returns 0 on success, another rcode on failure

`checkout`(*target='master'*)

checkout a certain point (tag, branch or commit)

Target the target to visit

Returns 0 on success, another rcode on failure

`commit`(*message='edit'*)
commit any changes made

git add . and git commit -am <message> is done

Message The commit message

Returns 0 on success, another rcode on failure

classmethod `convert_branch_name`(*date_string*)
Convert a datestring suitably to a branch name

Git does not allow special characters such as : or - in branchnames for whatever reason

Parameters **date_string** – the string to convert

Returns the new, converted string

`domain`
Return the domain, to which this wrapper belongs

`init`()
Create a new archive at specified domain path

The target directory does not need to exit yet

Returns 0 on success, another rcode on failure

`list_branches`()
List all branches in this repo, which conform to the 'date'-regex.

This means, Empty and master branch are not mentioned. If you want to checkout those, just checkout 'empty' or 'master'

Returns a list of branchestrings

`list_commits`()
List all commits in this repo and branch

Returns a list of commithashestrings

`recreate_master`()
A very special helper.

It deletes the current master branch, and recreates it. So, the master always points to the most recently created branch

CHAPTER

FOUR

DATABASE GENERATION

4.1 Overview

On the very end of every run of the crawler an update is done on the database, by iterating over all data in the internal metadata-list and builiding SQL Statements from this.

Insertion, for every:

1. ... domain a new row is inserted to the *domain* table. (Already existent domains are ignored)
2. ... mimeType a new row is inserted to the *mimeType* table.
3. ... url and path a new row is inserted to the *metaData* table.
4. ... commit a new row is inserted to the *commitTag* table, with a reference to the corresponding domain.
5. ... new file committed to the archive a new row is inserted into the *history* table.

If a row with this data already exists it is ignored.

4.2 Implementation

For Peformance-reasons only very simple insert-statements are used in combination with as simple select statements, instead of insert-statements with sub-selects.

DBGenerator is capable of generating an sqlite database from a list of metadictionaries.

class `crawler.dbgen.`**`DBGenerator`**(*meta_list=None*)

DBGenerator module

`batch`()

Start db creating procedure

Returns a truthy value on success

`close`()

Close connection and commit.

`execute_statement`(*source_name*, *arglist=None*)

Exececute a previously loaded statement by name

Source_name Sourcename to execute (e.g. 'create')

Arglist You may pass an additional list of variable elements

`insert_history``()`
Fill history table

`insert_mdata_ctag``()`
Fill metadata and committag table

`insert_mime_domain``()`
Fill mimeType and domain table

`load_statements``()`
(Re-)Load Sql Files from Disk

This is already called in init

Returns a dictionary with statements, indexed by name (e.g. 'create')

`select``(`*table*, **columns*`)`
Internal helper for collecting data

Table Table on which a SELECT shall be performed

Columns a list of columns to select

Returns A dictionary of column[0]: column[1:]

CHAPTER

FIVE

RECOVERING OF THE DATABASE

5.1 Strategies

Currently, there are two strategies to re-generate the Database:

Reading all XML Files: With this method the whole archive is traversed like this:

- **Iterate over all domains.**
  - **Iterate over all branches of this domain (excluding `empty` branch)**
    - **Iterate over all commits (excluding `Init` commit)** Iterate over all XML Files in there and build metadata-dicts from them

From the generated metadata-list a new Database can be generated.

Advantages:

- Works always, unless the archive is not totally broken
- Also works for XML-Files that were modified somehow (also their baseattribs shouldn't)

Disadvantages:

- May not be fast enough.

Using Cached .pickle files:

Instead of converting each XML costly to the internal representation, an object dump of the metadata-list is written to `/archive-root/pickle_cache/` on each crawl-run. If a recover is desired all of these *pickled* lists are joined, and the DB is regnerated.

```
# Files are named like this:
# <system-date-on-write>_<uuid>.pickle
2012-06-15T22:10:29_7cc2292a-80a6-4fcf-98fc-376953b387ca.pickle
2012-06-15T22:10:41_e2b1ebb2-1b13-4fb4-bd1c-7fe06aff2758.pickle
2012-06-15T23:04:35_59dc7790-5f65-47af-99fe-099610099ea4.pickle
2012-06-15T23:04:36_e58bf4c4-2639-4950-a788-6c84e1c4d1a6.pickle
2012-06-15T23:04:51_360107b5-d946-4c66-95c8-0d6ceb7a8c8a.pickle
...
```

Advantages:

- Much faster.

Disadvantages:

- Changes in the internal representation may break things

- If Base-Attributes of the XML Files are changed manually, they will not be found.

5.2 Implementation

Actual functions to use:

`dbrecover.recover.`**`rebuild`**`()`
Rebuilds the db either by using PickleDBRecover or XMLDBRecover

`dbrecover.recover.`**`remove`**`()`
Removes db

`dbrecover.repair.`**`repair`**`()`
Walks through domain hierarchy invoking repair() and clear_locks()

---

class `dbrecover.xml_recover.`**`XMLDBRecover`**
XMLDBRecover submodule class

`description`

Returns module description

`load``()`
Invokes threaded xml recovery

`recover_domain``(`*domain*`)`
Iterates through given domain trying to recover metadata

---

class `dbrecover.pickle_recover.`**`PickleDBRecover`**
Recovers database from previously generated pickle files

`description`

Returns description

`load``()`
Loads pickle files and regenerates metalist

Returns metalist object

`save``(`*metalist*`)`
Dumps given metalist as pickle file

CHAPTER

SIX

INDICES AND TABLES

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

PYTHON MODULE INDEX

INDEX