Towards Unified Invenio Configuration Database Manager

Author: Jiří Kunčar

jiri.kuncar@gmail.com

CERN Openlab

26/08/2011

Contents

Abstract

Managing of configuration and data of large projects requires handy configuration tool for deployment and backups. The runtime configuration and data mostly live in databases, hence the configuration tool should kind of enrich basic SQL dumper and loader functionality. However, the structure of stored data can be very complex and have complicated relationships and recursive associations. This particular project has the objective of defining database independent model for replacing hand-written SQL schema and create universal dumper and loader for all modules of Invenio software suite enabling to run a digital library or document repository on the web.

Chapter 1

Introduction

Managing of configuration and data of large projects requires dependable tool for deployment and backups. The runtime configuration and data mostly live in databases, hence the configuration tool should kind of enrich basic SQL dumper and loader functionality. However, the structure of stored data can be very complex and have complicated relationships and recursive associations.

This technical report attempts to provide description of the database management tool developed primarily for the Invenio project, however the core of tool should be universal for any project. The Invenio software suite provides the technology that *cover all aspects of digital library management from document ingestion through classification, indexing, and curation to dissemination. Invenio has been originally developed at CERN to run the CERN document server, managing over 1,000,000 bibliographic records in high-energy physics since 2002, covering articles, books, journals, photos, videos, and more* [1]. Currently it is being used by about thirty[1] scientific institutions worldwide.

The developed software is usually deployed to several machines in order to do proper testing. Afterwards, it can distributed to many institutions, thus it is necessary to transfer the database schema and fill it with initial data. The main challenge is to integrate universal database models for manipulating and providing data represented in human-friendly format, so that the intermediate files can be easily editable before being loaded to the database.

[1]in July 2011

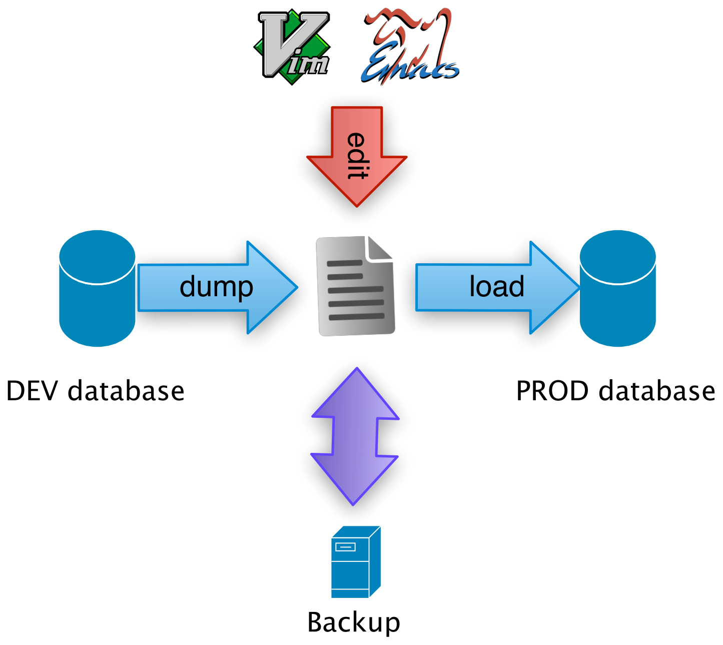

Figure 1.1: Work-flow

The tool will be used by administrators of Invenio for transferring of data between development and production servers or between different instances of the same software (see Figure 1.1). The reasons that let to development of this program, are described in following section.

1.1 Problem definition

The `inveniocfg`[2] has been developed to ease the installation process, demo site set-up, configuration files editing, and database tables update. Most of the database actions is defined as SQL queries and run from `Makefiles`. This approach has several disadvantages:

- hard to define selective data dump with all related rows;
- difficult to maintain create/alter tables; and
- only MySQL database management system (DBMS) is supported.

[2]Invenio configuration and administration CLI tool.

A basic command line tool for dumping and loading data has been already developed using specific syntax for defining model relationships and queries. The goal of this project was to create unified database models and define human friendly queries in standard configuration (ini) file.

This is the first step to database independence and time for review of database schema, because most of missing relationships between tables will be captured in the created model. On top of that the tool will also facilitate database schema creation with newly added constraints for multiple database management systems using Object Relation Mapper (ORM).

In addition, it is going to simplify database configuration tool, which should also support selective data export and import based on defined queries. The selected classes with all related objects will be serialized into human-readable text file, that should be easily editable. It can be eventually backed up and transferred to other servers.

1.2 Outline

The remaining part of this report comprises these chapters: Chapter 2 presents design and implementation and gives an overview of the technologies used when developing software for the database dumper/loader as well as the design and structure of the tool. Chapter 3 concludes the report and brings new ideas for possible future improvements.

Chapter 2

Design and Implementation

Data management tasks and database schema operation in object-oriented programming are typically implemented by manipulating objects corresponding to tables in relational database. However, there is a dependency drawback between demanded abstraction level and performance while using typical Object Relation Mapper (ORM) hence it should give application developers the full power and flexibility of SQL queries.

One of the most famous Python SQL toolkit is SQLAlchemy with an optional ORM component that provides the data mapper pattern, where classes can be mapped to the databases in open ended, multiple ways — allowing the object model and database schema to develop in a cleanly decoupled way from the beginning. SQLAlchemy allows to express wide range of SQL statements; any of these units can be composed into a larger structure.

In this section, we will describe features of the developed management tool. The implementation takes advantages of SQLAlchemy models for easy iterations and manipulations of database tables. The typical work–flow with developed tool is demonstrated in Figure 1.1 and it corresponds to the initial project plan. The program uses standard Python library `ArgumentParser`[1] for parsing command line options and `ConfigObj`[2] for simple reading and writing of configuration (ini) files.

[1]`http://docs.python.org/library/argparse.html`
[2]`http://code.google.com/p/configobj/`

2.1 SQLAlchemy Models

The SQLAlchemy Object Relational Mapper presents variety of methods of associating user-defined Python classes with database tables, and instances of those classes (objects) with rows in their corresponding tables. The usual ORM configuration process starts by describing the database tables, and then by defining classes which will be mapped to those tables. The SQLAlchemy allows to perform these steps together, using a system known as *Declarative, which allows us to create classes that include directives to describe the actual mapped database table* [2].

The Listing 2.1 shows typical example of declarative model definition describing the `user` table of Invenio with columns of various data types. The SQLAlchemy keeps all columns transparently synchronized with changes that have been made on mapped objects. It also allows to define and to handle relationships between models (including self-referenced variant).

```
class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True, autoincrement=
       True)
    email = Column(String(255), nullable=False)
    password = Column(Binary, nullable=False)
    note = Column(String(255), nullable=True)
    settings = Column(Binary, nullable=True)
    nickname = Column(String(255), nullable=False)
    last_login = Column(DateTime, nullable=False,
       server_default='0000-00-00␣00:00:00')
```

Listing 2.1: User Model

The following example 2.2 represents a One-to-Many relationship between `Collections` and their names (`Collectionnames`). First we define foreign key (`ForeignKey()`) on column `id_collection` referring to `Collection.id`. A second directive, known as `relationship()`, tells the ORM that the `Collectionname` class itself should be linked to the `Collections` class, using the attribute `Collectionname.collection`. A subdirective of `relationship()` called `backref` is placed inside of `relationship()`, providing details about the relationship as expressed in reverse, that of a set of `Collectionname` objects on `Collection` referenced by `Collection.names`. The **bidirectional relationship** is a key feature of the SQLAlchemy and it can be used also for Many-to-Many relations.

```
class Collection(Base):
    __tablename__ = 'collection'
    id = Column(Integer(9), primary_key=True)
    name = Column(String(255), unique=True, index=True)
  # ...

class Collectionname(Base):
    __tablename__ = 'collectionname'
    id_collection = Column(Integer, ForeignKey(Collection.
       id), primary_key=True)
    ln = Column(String(5), primary_key=True)
    type = Column(String(3), primary_key=True)
    value = Column(String(255))
    collection = relationship(Collection, backref='names')
```

Listing 2.2: One-to-Many Relation

The above `Collectionname` class contains details about the table `collectionname` with multiple columns denoted as a primary (composite) keys. Note that ORM in order to actually map to a particular table needs there to be a**t least one column denoted as a primary key column**.

2.2 Queries Definition

To retrieve desired data it is necessary to enter a query into the system. Queries are formal statements identifying an object or a collection of relevant information in a database. Queries should allow the user to describe desired data in easy way without the knowledge of their physical location. However, SQL[3] — the most used database language, requires certain knowledge of database structure and also database management system. Moreover the description of complex data with many relations may lead to complicated expressions. Hence we introduce simplified query definition taking advantage from SQLAlchemy models.

Our query definition relies on defined models and relations included in the class property definition that we explained in previous section. Thus the entire query can be simplified to the list of class names with information about blacklisted columns and requested related objects. The comparison predicate (know from SQL as `WHERE` clause) is uncouple and the query restriction is done in the tool.

Let us suppose that all table models have been already created. Now we describe a way how to define queries in configuration file for a part of Invenio module call `WebSearch` (see Figure B.7). The configuration file has to have a root section `[Queries]`. Then each subsection is name of a query that should have defined `import` modules and parent `class`.

```
[Queries]
    [[websearch]]
        import = WebSearch,
        class = Collection
```

Listing 2.3: Queries configuration

The following subsection `[[[options]]]` includes extended features for querying individual classes. The default setting tells to print all class column properties. Some or all of them can be blacklisted by adding prefix '`-`' to column name. If there are some relationships defined that should be exported or imported, it is necessary to explicitly append their names to configuration list. The default behaviour ensures only export of primary keys of associated models. Relationship names prefixed with sign '`+`' enable to recursively include associated classes.

```
[[[options]]]
    Collection = -nbrecs, +names, +examples, +sons
    Collectionname = ,
```

[3]Structured Query Language

```
    CollectionCollection = +son,
    CollectionExample = +example,
    Example = ,
```

Listing 2.4: Query options

We believe that the introduced query definition is more intuitive and offers users to specify their needs much faster and without deep knowledge of SQL.

2.3 Schema Manipulation - `create`, `drop`, `truncate`

In order to replace old installation process we need a way to easily create database tables. After the models have already been created, it is very convenient to use the SQLAlchemy feature to automatically generate SQL `create` or `drop` statements for database schema. However, in some cases it can be also useful to only create or drop indexes. Therefore we added parameter `-i` to answer such a need. User can also print only SQL queries that are going to be executed when he specifies the "dry run" option `-d`. The last option `truncate` allows to delete all data from defined tables.

2.3.1 MySQL Specific Problem (`error #1170`)

While we were creating models definitions, we found a problem[4] with creating indexes with defined length on `TEXT` or `BINARY` columns in MySQL. The syntax helps database management system to create better index with improved performance, but it is not supported by SQLAlchemy query builder out of the box.

Fortunately, SQLAlchemy supports redefinition of dialect templates for creating indexes (`CreateIndex()`) and primary key constrains (`PrimaryKeyConstraint()`) thus we could create index length definition for indexed `TEXT` columns by redefining SQLAlchemy these default MySQL dialect templates.

[4] `http://dev.mysql.com/doc/refman/5.0/en/error-messages-server.html#error_er_blob_key_without_length`

2.4 Schema Visualization - schema

Visualization of created classes can be very beneficial for future authors of queries for modules so they can be familiar with the database structure much faster. The tool is based on `SchemaDisplay`[5] extension that loads the model classes and analyses its properties and relationships (model associations like one/many-to-many, etc.). Schema visualization uses a wrapper for `pydot` along with `graphviz` for generating png class diagrams from all loaded tables in selected module. There is also option (`-t`) to display only tables defined in `options` section in configuration file for the used module query.

2.5 Data Export - dump

After having defined models and queries the tables content can be easily exported to an intermediate text file that can be transferred from development to production server later.

The export starts with filtering data according primary key value(s) in table mapped to parent class. Then each object in found tuple is serialized according to query options (described in Section 2.2) and stored in dictionary. The associated objects are automatically loaded by SQLAlchemy and the export function is recursively called. The filled dictionary object is transformed by `ConfigObj` parser to intermediate file with configuration syntax and printed to file defined in argument `-f` or to the standard output. The Listing 2.5 shows shortened output of `dump`ed data.

```
[Collection.20d29966]
   dbquery = None
   name = New Books & Reports
   [[sons]]
      [[[CollectionCollection.20ce285e]]]
         score = 0
         type = r
         [[[[Collection.20ce23e0]]]]
            dbquery = collection:THESIS
```

[5] http://www.sqlalchemy.org/trac/wiki/UsageRecipes/SchemaDisplay

```
            name = New Theses
            [[[[[examples]]]]]
               [[[[[[CollectionExample.20cdf8de]]]]]]
                  score = 0
                  [[[[[[[Example.20cdf5dc]]]]]]]
                     body = quark -sigma +dense
                     type = boolean search
               ...
   [[[CollectionCollection.20c740a2]]]
         score = 0
         type = r
         [[[[Collection.20c7248c]]]]
            dbquery = collection:BOOK
            name = New Books
            [[[[[examples]]]]]
               [[[[[[CollectionExample.20c1fc6e]]]]]]
                  score = 0
                  [[[[[[[Example.20c17c94]]]]]]]
                     body = "author:ellis␣-muon*␣+abstract
                        :'dense␣quark␣matter'"
                     type = complex boolean search
               ...
[[examples]]
   [[[CollectionExample.20d24bc8-cf2e-11e0-bae8-0800274
      e827d]]]
      score = 0
      [[[[Example.20d24970-cf2e-11e0-bae8-0800274e827d
         ]]]]
         body = quantum
         type = word search
...
```

Listing 2.5: Example of Exported Data

2.5.1 Column Data Transformation

In some situations, it is quite problematic to map stored column data onto the same format in intermediate text file. The typical example is a column with binary data where it is necessary to encode the data before saving them in the intermediate text file. SQLAlchemy allows to define TypeDecorator class that associates callback functions `process_bind_param()` and `process_result_value()`.

```
class _PrintBinary(types.TypeDecorator):
    def process_bind_param(self, value, dialect):
        return (value != None) and base64.decodestring(
            value) or None

    def process_result_value(self, value, dialect):
        return (value != None) and base64.encodestring(
            value) or None
```

Listing 2.6: Example of Binary Data Manipulation

2.6 Data Import - `import`

The ability to import data is very important because it means that the administrator does not have to manually insert data that already exists and were tested in another Invenio instance. The other useful scenario comprises easier adding or editing of existing object relations in schematic way using favourite text editor.

The import process is inverse to the export phase, hence the same query configuration should be used. Possibly some additional blacklisted columns (i.e. they will be computed by the business login after import) can be added to the query definition, but if the recursive field is newly defined and there are no related models in intermediate file the associations will be lost.

Note: it is full responsibility of editor to keep the syntax of the intermediate file compatible with `ConfigObj` parser.

Chapter 3

Conclusion

The goal of this project was to develop and implement tool that provides necessary support for the manipulation of tables stored Invenio database. The major interest concerns the universal models that is the first step to database independence.

To summarize, the main contribution of the present work consists in analysis of various possibilities of the SQLAlchemy ORM model to represent current Invenio database model, and implementation of the basic tool supporting database schema and data manipulation. The substantial part of the work was dedicated to examine the database model, improving table structures and defining simple query definitions. We have created 442 models covering the whole current version of Invenio database. During the process we have also suggested several improvements of missing primary keys and indexes on foreign key columns.

The key issue tackled in the design of the intermediate file is to preserve correspondence between object structure, and reasonable simplicity for human beings readability and ease of editing. There are two possible suggestions to improve readability even more: (1) association proxies[1] seem to be good start to reduce nesting of intermediate configuration files in case where many-to-many relationships contain additional attributes, and (2) ordered lists[2] could be an option to request ordered relationships.

[1]`http://www.sqlalchemy.org/docs/orm/extensions/associationproxy.html`
[2]`http://www.sqlalchemy.org/docs/orm/extensions/orderinglist.html`

Acknowledgement

I would like to thank Tibor Simko for his advices and invaluable input to my work during my stay at CERN. I also very appreciate the support from Jean-Yves le Meur and the whole IT/UDS/CDS group.

I would also like to thank Melissa Gaillard, Amalia Boari and Morag Hickman for devoted help with the administrative matters.

At last but not less important, I would like to leave my gratitude to all the other Openlab Summer Students 2011. They are Ioan Bucur, Ruggero Caravita, Goran Cetusic, Daniela Dorneanu, Joao Faria, Urs Fassler, Georgi Hah, Martin Hellmich, Hallgeir Lien, Jurand Nogiec, Wojciech Ozga, Arsalaan Shaikh, Sonia Stan and Grace Young.

See you friends!

Bibliography

[1] Invenio web site
http://invenio-software.org/

[2] SQLAlchemy web site
http://www.sqlalchemy.org/

Appendix A

Modules

In this section we provide full list of modules with all 442 model classes.

- BibAuthorID
  - AidAUTHORNAMES
  - AidAUTHORNAMESBIBREFS
  - AidCACHE
  - AidDOCLIST
  - AidPERSONID
  - AidREALAUTHORDATA
  - AidUSERINPUTLOG
  - AidVIRTUALAUTHORS
  - AidVIRTUALAUTHORSCLUSTERS
  - AidVIRTUALAUTHORSDATA
  - AidREALAUTHORS
- BibCirculation
  - CrcBORROWER
  - CrcLIBRARY
  - CrcITEM
  - CrcILLREQUEST

- CrcLOAN
- CrcLOANREQUEST
- CrcVENDOR
- CrcPURCHASE

- BibClassify
  - ClsMETHOD
- BibExport
  - ExpJOB
  - UserExpJOB
  - ExpJOBRESULT
  - ExpJOBExpQUERY
  - ExpQUERY
  - ExpQUERYRESULT
  - ExpJOBRESULTExpQUERYRESULT
- BibHarvest
  - OaiHARVEST
  - OaiREPOSITORY
  - OaiHARVESTLOG
- BibIndex
  - IdxINDEX
  - IdxINDEXNAME
  - Field
  - IdxINDEXField
  - Fieldname
  - Tag
  - FieldTag
  - IdxPAIR{01..17}F
  - IdxPAIR{01..17}R
  - IdxPHRASE{01..17}F

  - IdxPHRASE{01..17}R
  - IdxWORD{01..17}F
  - IdxWORD{01..17}R

- BibKnowledge
  - KnwKB
  - KnwKBDDEF
  - KnwKBRVAL

- BibRank
  - RnkMETHOD
  - RnkMETHODDATA
  - RnkMETHODNAME
  - RnkCITATIONDATA
  - RnkCITATIONDATAEXT
  - RnkAUTHORDATA
  - RnkDOWNLOADS
  - RnkPAGEVIEWS
  - RnkWORD01F
  - RnkWORD01R

- Bibrec
  - Bibrec
  - Bibfmt
  - BibHOLDINGPEN
  - Bibdoc
  - BibdocBibdoc
  - BibrecBibdoc
  - HstDOCUMENT
  - HstRECORD
  - Bib[0-9]{2}x
  - BibrecBib[0-9]{2}x

- BibSched
  - HstTASK
  - SchTASK
- BibSword
  - SwrREMOTESERVER
  - SwrCLIENTDATA
- BibUpload
  - HstBATCHUPLOAD
- ErrorLib
  - HstEXCEPTION
- misc
  - Publreq
- User
  - User
  - Usergroup
  - UserUsergroup
- WebAccess
  - AccACTION
  - AccARGUMENT
  - AccMAILCOOKIE
  - AccROLE
  - AccAssociation
  - UserAccROLE
- WebAlert
  - UserQueryBasket

- WebBasket
  - BskBASKET
  - BskEXTREC
  - BskEXTFMT
  - BskREC
  - BskRECORDCOMMENT
  - UserBskBASKET
  - UsergroupBskBASKET
- WebComment
  - CmtRECORDCOMMENT
  - CmtACTIONHISTORY
  - CmtSUBSCRIPTION
- WebJournal
  - JrnJOURNAL
  - JrnISSUE
- WebMessage
  - MsgMESSAGE
  - UserMsgMESSAGE
- WebSearch
  - Collection
  - Collectionname
  - Collectiondetailedrecordpagetabs
  - CollectionCollection
  - Example
  - CollectionExample
  - Portalbox
  - CollectionPortalbox
  - Externalcollection

  - CollectionExternalcollection
  - Format
  - CollectionFormat
  - Formatname
  - Field
  - Fieldvalue
  - Fieldname
  - Tag
  - FieldTag
  - WebQuery
  - UserQuery
  - CollectionFieldFieldvalue
  - CollectionClsMETHOD
  - CollectionRnkMETHOD

- WebSession
  - Session

- WebStat
  - StaEVENT

- WebSubmit
  - SbmACTION
  - SbmALLFUNCDESCR
  - SbmAPPROVAL
  - SbmCATEGORIES
  - SbmCHECKS
  - SbmCOLLECTION
  - SbmCOLLECTIONSbmCOLLECTION
  - SbmDOCTYPE
  - SbmCOLLECTIONSbmDOCTYPE
  - SbmCOOKIES

- SbmCPLXAPPROVAL
- SbmFIELD
- SbmFIELDDESC
- SbmFORMATEXTENSION
- SbmFUNCTIONS
- SbmFUNDESC
- SbmGFILERESULT
- SbmIMPLEMENT
- SbmPARAMETERS
- SbmPUBLICATION
- SbmPUBLICATIONCOMM
- SbmPUBLICATIONDATA
- SbmREFEREES
- SbmSUBMISSIONS

Appendix B

Schemas

The list of figures with module schemas follows.

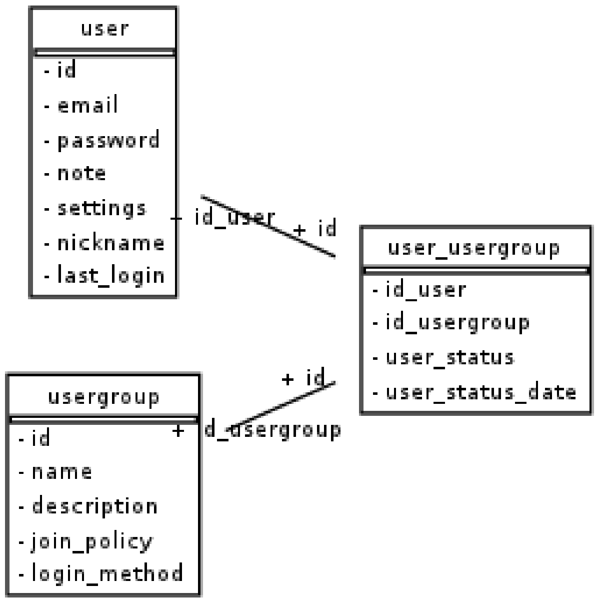

Figure B.1: User

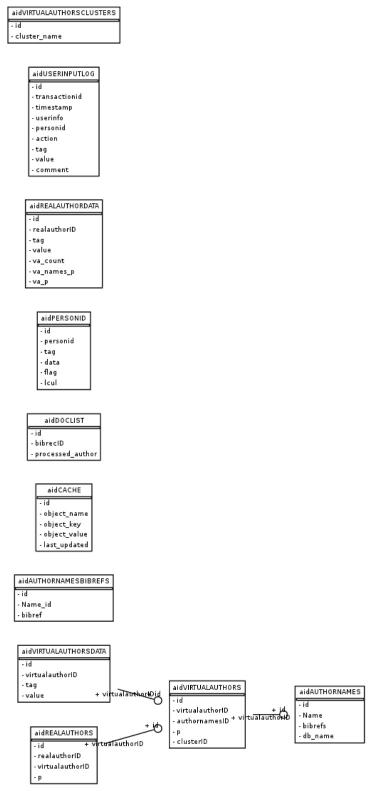

Figure B.2: BibAuthorID

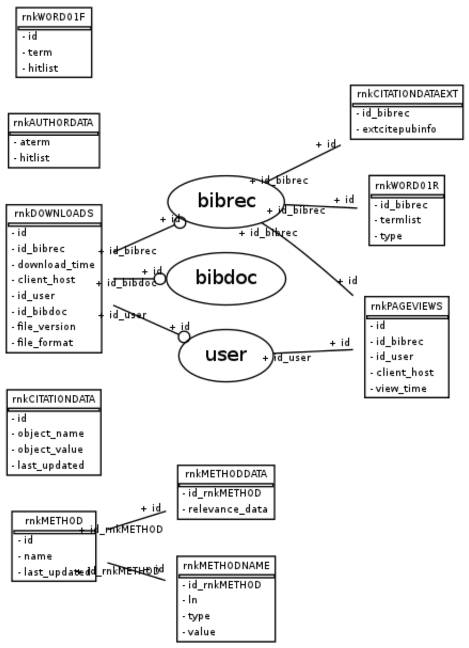

Figure B.3: BibRank

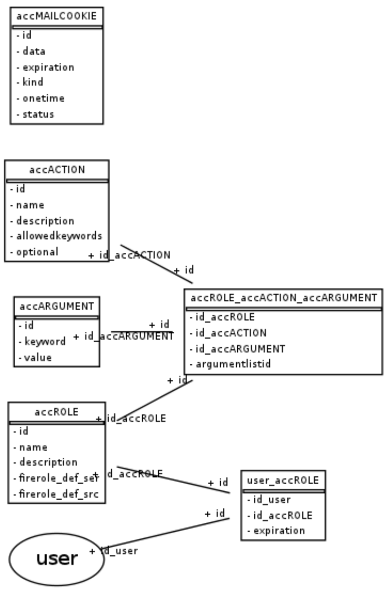

Figure B.4: WebAccess

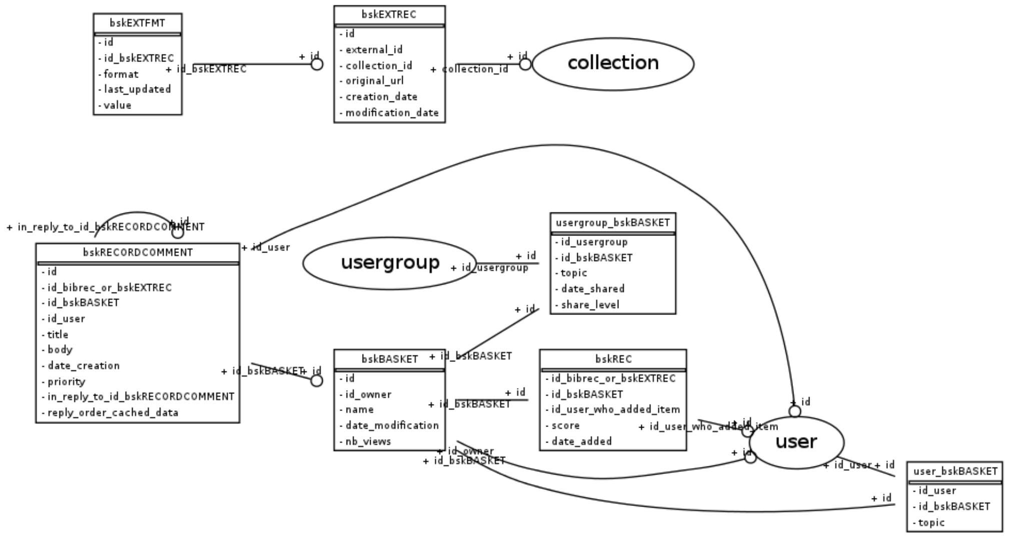

Figure B.5: WebBasket

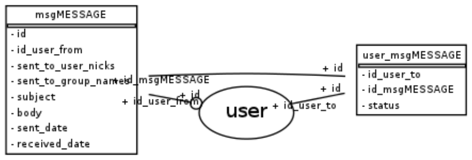

Figure B.6: WebMessage

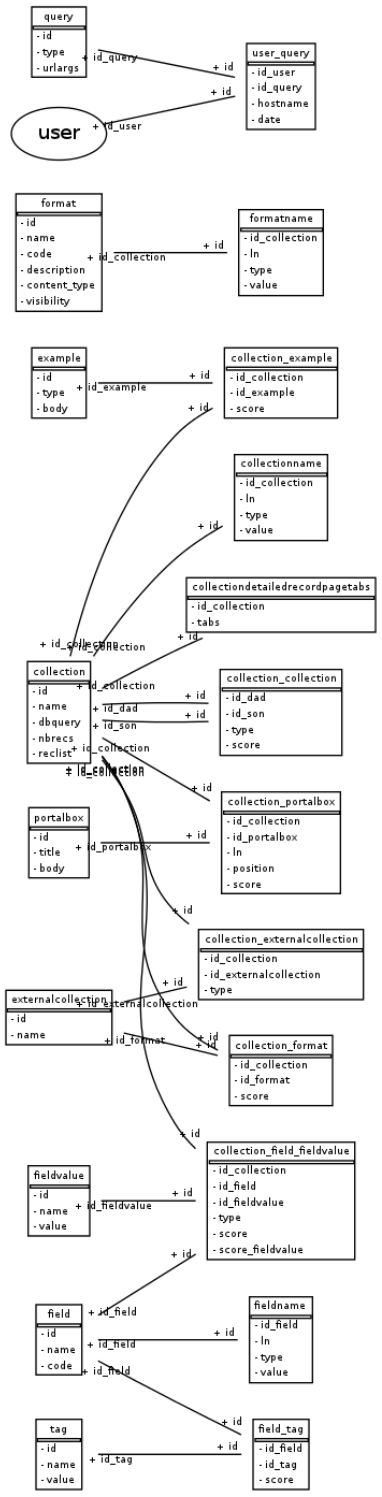

Figure B.7: WebSearch