XRecord Documentation

Release 0.1.8

Jakub Wroniecki

January 27, 2013

CONTENTS

Contents:

CHAPTER

ONE

MANAGING DATABASE CONNECTIONS

The basic `XRecordDatabase` functionality is not very different from other *connection* objects provided by RDBMS backend drivers. It connects to the backend, and serves as a proxy for sending queries and receiving results.

It also provides an API to query the database and receive results in object format (`Record` and `XRecord`), but some backend drivers also have this functionality built-in.

The main difference comes from the way XRecord ORM was utilized, before it was called *XRecord ORM* - it was used in long-running daemons and system services. All of these are vulnerable to database backend failures, restarts, network downtime etc., so graceful re-connection had to be made easy at the *database API* level. It's nothing fancy, but it does its job, a primitive example could look like this

```
while True:
    try:
       arr = db.XArray("blog_entry")
       for e in arr:
          do_something()
    except db.Error:
       while not db.Test():
          time.sleep(10)
          db.Reconnect()
```

XRecord was also used in short-lived programs, some of which required speed, and the additional overhead caused by fetching meta-data for each session was simply not acceptable. This is why we decided for the meta-data fetching functions to be *lazy*, ie. fetching it only when it is needed (when XRecords for a specific table are instantiated), so when not using the *XRecord.X?????* functions, no hidden hits to the database are made.

1.1 XRecordDatabase

class `xRecord.`**`XRecordDatabase`**`(`**args*, ***kwargs*`)`

This class represents a database.

classmethod **`getInstance`**`(`**args*, ***kwargs*`)`

This class method should be used to retrieve an instance of this class, or instantiate a new object if it does not exist. Using this method ensures that only one connection is used throughout the whole process.

If you want a new instance, call the constructor directly.

multithreading/multiprocessing: NOTE: As most backend drivers are not thread safe, each new thread should have its own instance, or protect the access to its methods. **YOU HAVE BEEN WARNED**.

`Test``()`

Check if the connection is still active

Return type boolean

Returns True if connection is alive, False otherwise

Close()
Close the backend connection

Reconnect()
Reconnect to the back-end, using last known parameters

CheckConnection()
Check if the connection to the backend is alive, and reconnect if necessary.

Connection
Return the backend driver's connection object.

Return type instance

Manager
The `Manager` attribute provides a way to access the generated classes for database tables. This may come in handy if you defined custom class methods for your table proxy.:

```
>> db.Manager.blog_entry.getByCategory ( "programming", "python" )
[<xrecord::blog_entry(1)>, <xrecord::blog_entry(2)>]
```

SQLLog(*stream*)
Set the output stream object, to which all SQL queries run by this database instance are logged.

Parameters **stream** – file object or None

CommandQuery(*sql*, **args*)
Run an SQL query, returning the number of affected rows.

Best used for UPDATE and DELETE queries.

InsertQuery(*sql*, **args*)
Run an SQL query. If it succeeds, return the id of the *last inserted row*, otherwise return the number of affected rows.

SingleValue(*sql*, **args*)
Run the query, and return the value of the first column in the first row of the returned result set.

SingleObject(*sql*, **args*, ***kwargs*)
Run the query, and return the first row of the returned result set as a `Record` object.

ArrayObject(*sql*, **args*, ***kwargs*)
Run the query, and return the result set as an array of `Record` objects.

ArrayObjectIndexed(*sql*, *index_column*, **args*, ***kwargs*)
Run the query, and return the result set as dictionary with the key set to the value of the *index_column* of each row of the returned result set, and the value set to the corresponding `Record` object.

If values of *index_column* are not unique, each subsequent record overwrites the previous key-value mapping for the given key.

Return type ordereddict

ArrayObjectIndexedList(*sql*, *index_column*, **args*, ***kwargs*)
Run the query, and return the result set as dictionary with the key set to the value of the *index_column* of each row of the returned result set, and the value set to a list of the corresponding :class:‘Record‘objects.

If values of *index_column* are unique, this function returns a key=>value mapping where all values are lists of length 1.

Return type ordereddict

`SingleAssoc`(*sql*, **args*, ***kwargs*)
: Run the query, and return the first row of the returned result set as a dictionary.

`ArrayAssoc`(*sql*, **args*)
: Same as *ArrayObject*, but returns dicts instead of `Record` objects;

`ArrayAssocIndexed`(*sql*, *index_column*, **args*, ***kwargs*)
: Same as *ArrayObjectIndexed*, but returns dicts instead of `Record` objects;

`ArrayAssocIndexedList`(*sql*, *index_column*, **args*, ***kwargs*)
: Same as *ArrayObjectIndexedList*, but returns dicts instead of `Record` objects;

`XRecord`(*tablename*, **args*, ***kwargs*)
: Create a new instance of XRecord subclass for the given table. If there are any unnamed arguments, they are treated as primary key value, and a Fetch is performed on the record after initialization.

  The keyword arguments are used as default values for attributes, but only if they appear in the table schema as columns.

  Parameters

  - **tablename** – name of the table
  - ***args** – primary key value
  - ****kwargs** – default attribute values

  Returns new record

  Return type XRecord

`XSingle`(*table*, *sql=None*, **args*)
: Same as *SingleObject*, but returns `XRecord` objects for the given *table* instead of `Record` objects.

  If *sql* is None returns the object with its primary key value equal to the unnamed arguments.

`XArray`(*table*, *sql=None*, **args*, ***kwargs*)
: Same as *ArrayObject*, but returns `XRecord` objects for the given *table* instead of `Record` objects.

  If *sql* is None (default) returns all records in the table.

`XArrayIndexed`(*table*, *index_column*, *sql=None*, **args*, ***kwargs*)
: Same as *ArrayObjectIndexed*, but returns `XRecord` objects for the given *table* instead of `Record` objects.

  If *sql* is None (default) returns all records in the table.

`XArrayIndexedList`(*table*, *index_column*, *sql=None*, **args*, ***kwargs*)
: Same as *ArrayObjectIndexedList*, but returns `XRecord` objects for the given *table* instead of `Record` objects.

  If *sql* is None (default) returns all records in the table.

`XRecordRefCacheEnable`(*tablename*, *key_column*, *cache={}*)
: Enable reference cache for a Foreign Key (key_column) in table 'tablename' The optional 'cache' argument may be initialized to a dictionary like object containing pairs of (pk : XRecord)

`XRecordRefCacheDisable`(*tablename*, *key_column*, *cache={}*)

`Initialize`()
: Called after the contructor is finished, may be overloaded to define custom XRecord and XSchema classes

1.2 XRecordMySQL

class `xRecord.`**`XRecordMySQL`**

The named attributes accepted by the constructor of this class are:

name database name

host server host name

port server tcp port

user user name

password user's password

1.3 XRecordSqlite

class `xRecord.`**`XRecordSqlite`**

The named attributes accepted by the constructor of this class are:

name path to the file containing the database

1.4 XRecordPostgreSQL

class `xRecord.`**`XRecordPostgreSQL`**

The named attributes accepted by the constructor of this class are:

name database name

host server host name

port server tcp port

user user name

password user's password

CHAPTER

TWO

WORKING WITH RECORDS / DATA ROWS

Working with relational databases, in majority of applications, comes down to working with *rows of data*, also known as *records*. Therefore, for a library used in the database-abstraction layer , whether it wants to be called an ORM or not, it is most important to make using these records as comfortable as possible.

The most commonly used standard for such libraries is DBAPI (currently v2.0). It's working, it's complete, optimized and it has a well designed, widely accepted interfaces. The problem is, that to run a database task using DBAPI, you usually have to:

1. Create a cursor
2. Execute the query
3. Iterate over the cursor
4. Extract data from the cursor for each row
5. Run a seperate query to, for eg. update related data.
6. Close the cursor

This becomes tedious in larger applications, which start to look like majority of their code is related to data fetching / saving.

ORMs on the other hand provide a simpler mechanism:

1. Execute the query
2. Use/modify the returned objects

What is hidden from you is the fact that for each row you want to work with, the ORM has to instantiate a class, and fill its attributes - something you'd have to do one way or another.

XRecord provides you with two alternative ways to run database queries. The first - using basic record objects, is more suited for running complex SQL queries and working with the results. The functions used for this method, are the documented methods of XRecordDatabase, that have *Object* in their name. The objects they return have no reference to the database, table or row they came from, cannot be saved, updated or deleted without writing additional SQL. They also do not follow intra-table relationships. In fact, the only difference between them and the return values from DBAPI queries is possibility of accessing values via object attributes, and that there is no need to create and use a cursor object.

The second way XRecord lets you access data is the reason why we call it an ORM. The functions using this method are the ones with names starting with 'X'. These functions return instances of classes derived from the `XRecord` class. Each such instance represents a row of data in a specified table, and can easily fetch referenced rows, child rows and rows related via many-to-many relationships. You may update, delete and insert rows without writing a single line of SQL.

The `XRecord` subclasses for each table are generated on-the-fly from the metadata in your RDBMS. It means you have to specify all the primary and foreign keys in your DDL scripts. More about this can be found in the *Handling meta-data* section.

The `XRecord` subclasses may be further extended to provide a richer object interface to your data.

2.1 Basic record objects - Record

class `xRecord.`**`Record`**`(`*`**kwargs`*`)`

Simple container object, for storing rows of database data in a serializable form. Objects of this class are returned by XXXObject, methods of XRecordDatabase. This is the simplest possible ORM - it takes whatever is returned by a query, looks at the signature and creates objects on the fly.

XRecord.Serialized also returns an instance of this class, since it’s easily processed by most common python serializers.

Attributes (column values) may be accessed like attributes and dictionary items alike:

```
>>> for r in db.ArrayObject ( "SELECT * FROM blog_entry" ):
...     print r.title, r.author
...     print r['title'], r['author']
...
Article 1  1
Article 1  1
Article 2  1
Article 2  1
```

2.2 Active record objects - XRecord

class `xRecord.`**`XRecord`**`(`*`*args`*`, `*`**kwargs`*`)`

Base class for all XRecords (active records).

There numerous ways to instantiate an XRecord:

```
>>> e1 = db.XRecord("blog_entry", 1)
>>> e2 = db.Manager.blog_entry(1)
>>> assert e1 == e2
>>> e3 = db.XSingle("blog_entry", "SELECT * FROM blog_entry WHERE id=1" )
>>> e4 = db.XSingle("blog_entry", "SELECT *, CONCAT('<h1>', title, '</h1>') as html_title FROM b
>>> assert e3 == e4
>>> print e4.html_title
<h1>Article 1</h1>
```

`fetch``(`*`*args`*`, `*`**kwargs`*`)`

Fetch a row of data to this record. May raise XRecordDatabase.NotFound.

```
>>> e = db.XRecord("blog_entry")
>>> e.Fetch(1)
>>> print e
<xrecord::blog_entry(1)>
```

Parameters ***args** – primary key value of the row, as unnamed arguments

Returns nothing

`reload()`
Fetch this record's data again, losing all changes made since last Save/Fetch.

Returns nothing

`save()`
UPDATE the database with this record's data, or INSERT if the primary key is empty.

Returns number of affected rows, should by 1 or 0

`insert()`
INSERT a new row into the database.

`delete()`
Remove this row from the database. The row must be Fetched or otherwise initialized prior to this.

Returns number of affected rows, should be 1 or 0 (if row was already deleted)

`nullify()`
Make this record NULL (containing no data).

`serialized`(*depth=1*)
Generate a simple Record object with this records data, following foreign keys, children references and mtm references up to the given depth.

The references must be fetched prior to the call to this function.

Parameters **depth** – the maximum recursion depth

Returns a serializable representation of *self*

Return type Record

`PK`
A tuple containing this records primary key value.

`Table`
Name of the table this record belongs to

`SCHEMA`
The XSchema object this record was derived from.

2.3 Extending XRecord

When XRecord subclasses are generated from meta-data, they provide a set of basic functions for working with records of a specified table (described above). It is also possible to further subclass them to extend data row objects with custom functionality. An example:

```
@db.CustomXRecord("blog_entry")
class blog_articles:
      def __repr__(self):
          return "Entry: '" + self.title + "'"

      def last_comments(self, number=10):
          return self.DB.XArray ("comment",
              "SELECT c.* FROM comment WHERE entry=? ORDER BY when DESC LIMIT ?",
              (self.id, number) )

      @classmethod
```

```
    def last_entries(cls, number=10):
        return self.DB.XArray ( "blog_entry",
            "SELECT * FROM blog_entry ORDER BY when DESC LIMIT ?",
            (number, ) )
```

What we've done here is we customized the `blog_articles` class, so that each subsequent instance will have a custom string representation, and will provide a `last_comments` method to fetch a given number of most recent comments. We also added a class method, to fetch an array of a given number of most recent blog entries.

Now we may use the new functions like this:

```
>>> e = db.XRecord("blog_entry", 1)
>>> print e
Entry: 'Article 1'
>>> print e.last_comments(2)
[<xrecord::comment(2)>, <xrecord::comment(3)>]
>>> print db.Manager.blog_entry.last_entries (2)
['Entry: \'Article 1\'', 'Entry: \'Article 2\'']
```

The piece of code that makes this happen is the class decorator: `db.CustomXRecord`. It takes the default class for a given table (`blog_entry` in this case) and derives a new class which inherits it, together with the decorated class.

For this to work the XRecordDatabase object must be instantiated and the connection to the database must be active. Therefore it is recommended that all XRecord subclass customizations be made inside the `Intialize` method of a XRecordDatabase subclass, like this:

```
class MyDatabase(XRecordDatabase):
   def Initialize(self):
       @self.CustomXRecord("blog_entry")
       class blog_entry:
           """Do your customizations here"
           pass
       #Or:
       self.CustomXRecord("category") (some_other_class)
   pass
```

CHAPTER

THREE

HANDLING META-DATA

TODO :)

3.1 XSchema

class `xRecord.`**`XSchema`**(**args*, ***kwargs*)

> **`rename_mtm`**(*old_name*, *new_name*)
>
> **`rename_child_reference`**(*old_name*, *new_name*)
>
> **`has_child`**(*key*)
>
> **`has_mtm`**(*via_table*)
>
> **`has_column`**(*column_name*)
>
> **`get_child`**(*key*)
>
> **`get_mtm`**(*via_table*)
>
> **`column_list`**()
>
> **`columns`**()
>
> **`pre_update`**(*xrec*, *where_condition_dict*, *update_values_dict*)
>
> **`post_update`**(*xrec*)
>
> **`pre_insert`**(*xrec*, *insert_values_dict*)
>
> **`post_insert`**(*xrec*)
>
> **`pre_delete`**(*xrec*, *where_condition_dict*)
>
> **`post_delete`**(*xrec*, *old_record*)
>
> **`initialize`**()
>
> **`null`** = **<unbound method XSchema.null>**
>
> **`default`** = **<unbound method XSchema.default>**
>
> **`verbose_info`**

3.2 Customizing XSchema

```
@db.CustomXSchema("author")
class blog_entry:
      def __repr__(self):
          return self.name

      def initialize(self):
          self.rename_child_reference ( "blog_entry_author", "blog_entries")
```

CHAPTER

FOUR

TUTORIAL

For this tutorial we will create a simple database for a blog system. While it may be the most *cliche* example there is (perhaps with the exception of an address book), it will allow us to demonstrate all features of the *XRecord ORM*. We will use the Sqlite driver, so it is easier to reproduce on the reader's machine.

4.1 Sample database

We begin by creating the database, and populating it with some example data.

Its complexity, and the number of triggers is due to the fact that SQLite does not enforce foreign key contraints. This schema was generated with the excellent SQLite foreign key trigger generator, which made the job as easy as copy & paste :).

4.2 Connecting to the database

Whew! Now we're ready to start-up python

```
>>> import XRecord
>>> db = XRecord.connect("sqlite", name = "blog.db" )
```

4.3 Meta data

Now that we're connected to the database, let's see some debugging info about one of our tables

```
>>> print db.XSchema("blog_entry").verbose_info
Table `blog_entry`.
Columns:
- content <text>
- author <integer>
- id <integer>
- ts <timestamp>
- title <text>
References:
- author -> author (id)
Referenced by:
- id <- entry_category (entry)
Many-To-Many
- `entry_category` to category (id) via entry_category
```

This is what it tell us, about what was read from the database meta-data:

- We have 5 columns of the show type
- The table references the *author* table via the *author* column
- The table is referenced by the *entry_category* table's column *entry*
- The table has a many-to-many relationship with the table *category*, via the *entry_category_table*

4.4 Querying the database and child references

Let's look at some data:

```
>>> print db.XArray ( "author" )
[<xrecord:author(1)>, <xrecord:author(2)>, <xrecord:author(3)>]
```

We've fetched the contents of the *author* table as a list of `XRecord` objects. The default python display isn't very informative, we'll see later how to fix that (*tutorial2*).

Now let's get an *author* record and play with it for a while :)

```
>>> hemingway = db.XSingle ( "author", "SELECT * FROM author WHERE name like '%hemingway%'" )
>>> hemingway2 = db.XSingle ( "author", 1 )
>>> hemingway == hemingway2
True
>>> print hemingway
<xrecord:author(1)>
>>> print hemingway.id, hemingway.PK, hemingway.SCHEMA.pk
1 (1,) (u'id',)
>>> print hemingway.name
Ernest Hemingway
>>> print hemingway.blog_entry_author
[<xrecord:blog_entry(1)>, <xrecord:blog_entry(2)>, <xrecord:blog_entry(3)>]
```

What happened here? First we retrieved a specific *author* `XRecord` using 2 different methods - with pure SQL, and using its primary key value of *1* (which we happen to know, by chance ;)). The two records, although different instances, compare as equal with the standard python operator. Next we printed the primary key information and the value of the *name* attribute.

Next we accessed the *blog_entry_author* attribute which is a list of referencing records in the *blog_entry* table. The attribute name is generated using a template: <referencing table name>_<referencing column name>, and also may be customized, which will be discussed later (*tutorial2*).

Let's take a look at the author's blog entries

```
>>> for entry in hemingway.blog_entry_author:
...     print entry, entry.id
...     print entry.title
...     print entry.entry_category
...
<xrecord:blog_entry(1)> 1
How I killed myself.
[<xrecord:category(1)>, <xrecord:category(2)>, <xrecord:category(3)>]
<xrecord:blog_entry(2)> 2
How I said "Farewell!" to arms
[<xrecord:category(2)>]
<xrecord:blog_entry(3)> 3
```

```
The day I heard the bell toll
[<xrecord:category(2)>]
```

4.5 Modifying data

We've iterated over Hemingway's blog entries, show their attributes, and a list of categories assigned to each one.

Let's put some random garbage as the entries' content

```
>>> for entry in hemingway.blog_entry_author:
...     entry.content = hashlib.md5(str(random.random())).hexdigest()
...     entry.save()
...
1
1
1
```

4.6 Adding many-to-many relationships

The *1* are the return value of the **:method:'XRecord.Save'** method, which returns the number of affected rows. Now let's create a new *category* and assign it to one of Hemingway's entries

```
>>> entry = hemingway.blog_entry_author[0]
>>> new_category = db.XRecord ( "category", name="Everything else!")
>>> new_category.save()
>>> entry.entry_category.add(new_category)
```

We took an entry from the author's list, created the new category, saved it (important) and put it in relationship with the entry using the virtual method *entry_category.add*. Try it again

```
>>> entry.entry_category.add(new_category)
```

The database backend complains about a contraint violation. Now let's see the entry's category list

```
>>> print entry.entry_category
[<xrecord:category(1)>, <xrecord:category(2)>, <xrecord:category(3)>]
```

The new category does not appear in it. The reason for this is that XRecord instances cache the foreign key, child and many-to-many relationships. When a new related object is added in a mtm relationship, the cached list remains the same unless explicitly purged like this:

```
>>> del entry.entry_category
>>> print entry.entry_category
[<xrecord:category(1)>, <xrecord:category(2)>, <xrecord:category(3)>, <xrecord:category(10)]
```

Now we delete the new category:

```
>>> new_category.delete()
>>> del entry.entry_category
>>> print entry.entry_category
[<xrecord:category(1)>, <xrecord:category(2)>, <xrecord:category(3)>]
```

and the database triggers take care of the rest.

4.7 Accessing Foreign Key references

It is also easy to access records referenced by a record we are working with

```
>>> entry
<xrecord:blog_entry(1)>
>>> entry.author
1
>>> entry.author.ref
<xrecord:author(1)>
>>> entry.author.ref.id
1
>>> entry in entry.author.ref.blog_entry_author
True
```

Word of caution. The `entry.author` attribute is an object of class XRecordFK. Even though, when converted to its string representation it looks like the actual value of the corresponding column, it is safer to access this value by using `entry.author.value`. That way you can be certain your are working with the value returned by the backend. When setting this attributes value, you can use `entry.author = new_val` as well as `entry.author.value = new_val` - they are equivalent. `new_val` may be the actual value for the column, or an instance of `XRecord` for the referenced table.

CHAPTER

FIVE

EXTENDING AND CUSTOMIZING XRECORDS AND XSCHEMAS

When an XRecord object is created a some things are happening behind the scenes.

First, an XSchema definition for the given table is looked for in the XRecordDatabase internal cache. If it's not found, the database metadata is fetched (from the INFORMATION_SCHEMA) and the XSchema instance is built.

Next, the library looks for an auto-generated class (a subclass of XRecord) for the given table, generating it if needed.

Finally an object of this class is instantiated, and, if its primary key value is known a record is fetched from the backend

```
#new empty record with all default values
rec1 = db.XRecord ("blog_entry")
#new record with user given values
rec2 = db.XRecord ( "blog_entry", title="new post", content="blabla" )

#another notation
rec1 = db.Manager.blog_entry()
rec2 = db.Manager.blog_entry(title="new post", content="blalbla")

assert isinstance(rec1, db.XRecordClass)
assert isinstance(rec1, db.Manager.blog_entry)

#another way of accessing the class
assert db.XRecordCurrentClass ( "blog_entry" ) is db.Manager.blog_entry
```

5.1 Subclassing XRecord

By default the XRecord classes for the tables in your database have the plain functionality of XRecord. To take advantage of object-oriented nature of the ORM, it is possible to extend these classes to add-in your custom string representation, properties, methods and class methods. This is done using the `db.CustomXRecord` decorator and it must be done after the connection to the database is established. It is therefore recommended that subclassing is done inside the overloaded `Initialize` method in your own XRecordDatabase subclass

```
class MyDatabase(XRecordSqlite):

      def Initialize(self):

          @self.CustomXRecord("author")
          class any_name_will_be_ok:
                def __repr__(self):
                    return self.name

                def instance_method(self):
                    do_something_with(self)

                @classmethod
                def class_method(cls):
                    do_something_else()
```

Now we may do the following

```
db = MyDatabase (name='blog.sqlite')
author = db.Manager.author(1)
author.instance_method()

db.Manager.author.class_method()

print author
#prints author.name
```

5.2 Subclassing XSchema

XSchema objects store the table meta data, specifically - column information, primary keys, foreign keys, child references, many-to-many references and unique indices. They are used when new XRecords are instantiated, when data is saved and fetched, and when special attributes are accessed.

In our example database to fetch blog entries related to an author we had to write

```
for entry in author.blog_entry_author:
    print e
```

This is because the default name of an attribute used to access child references, is build using the <referencing table>_<referencing column> template. To change this we may subclass the XSchema for the author table and rename this attribute

```
class MyDatabase(XRecordSqlite):

      def Initialize(self):

          @self.CustomXSchema("author")
          class any_name_will_be_ok:

                def initialize(self):
                    self.rename_child_reference ( "blog_entry_author", "entries" )
                    do_something_with(self)
```

Note that we used the `CustomXSchema` decorator, instead of the `CustomXRecord` used for subclassing XRecords.

Now it's possible to write

```
for entry in author.entries:
    print entry
```

Other method that may be used in an XSchema subclass initialization is `rename_mtm`, used to rename the attribute under which a mtm relationship is stored.

XSchema subclasses may also define following methods, to emulate *trigger* behaviour:

- pre_update (xrecord, where_conditions, update_values)
- post_update (xrecord)
- pre_insert (xrecord, insert_values)
- post_insert (xrecord)
- pre_delete (xrecord)
- post_delete (new_xrecord, old_record)

CHAPTER

SIX

DATABASE VIEWER

The database viewer is a small web application which ships with the XRecord package. Currently its function is limited to a simple review of the database structure with all its table relationships.

6.1 Starting the web app

Running it is as simple as:

```
from XRecord import viewer, connection_factory

viewer.run ( connection_factory ( "sqlite", name="filename" ), address="127.0.0.1", port=3000 )
```

The run function accepts a connection_factory as its first argument - a function yielding new database connections each time it's called. Its arguments match thos of the connect function.

If you wish to run the viewer on your subclass of XRecordDatabase, you may do so like this:

```
from XRecord import viewer
from myapp import myXRecordDatabase

viewer.run ( myXRecordDatabase.getFactory ( [arguments] ), address = "127.0.0.1", port=3000 )
```

CHAPTER

SEVEN

INTEGRATING WITH DJANGO

XRecord integrates seamlessly with the Django Web framework.

7.1 How?

Just use it, there is no magic to it, no tricks. You'd probably want to subclass XRecordDatabase, to customize your objects behaviour.

```
from XRecord.mysql import XRecordMySQL

class AppDatabase(XRecordMySQL):
      connection_defaults = { 'name' : 'blog', 'user' : 'blogger' }
      pass

      def Initialize(self):
          ### customization....
          ### customization....
          pass
```

Then use it inside a view function:

```
from django.shortcuts import render_to_response
from django.template import RequestContext
from mydatabase import AppDatabase

def view_function (request):
  try:
    db = AppDatabase.getInstance()
    authors = db.XArray ( "author" )
    return render_to_response ( 'view_template.html', {'authors' : authors }, context_instance = Requ
  except db.Error, e:
    return render_to_response ( 'view_error.html', {'error' : e }, context_instance = RequestContext
```

And your template `view_template.html`, could look a little like this:

```
<pre>
{% for author in authors %}
   {{ author.name }}
   {% for blog_entry in author.blog_entry_authors %}
   entry: {{ blog_entry.title }}
      categories: {% for category in blog_entry.entry_category %} {{ category.name }} {% endfor %}
   {% endfor %}
{% endfor %}
</pre>
```

7.1.1 Performance issues

There is something about the way XRecord works, that raises questions about its performance in a high-load web environment: every time XRecordDatabase is instantiated, it reads the meta-data from the backend. For normal, long-running applications this is has negligable impact on performance, but when it is happening once for every single web request, it can be significant.

XRecord has a solution for this problem - *meta-data caching*. The XRecordDatabase class has two methods `ReadMetaDataCache` and `WriteMetaDataCache`, which read and write the meta-data information from a file on the disk. The call to first may be placed inside the `Initialize` method, the second has to be run every time something in your database structure changes.

```python
def Initialize(self):
  #....
  self.ReadMetaDataCache('/var/lib/blog_database.metadata')

#eg. inside some "update" script:
db.WriteMetaDataCache ('/var/lib/blog_database.metadata')
```

7.2 Why?

In fact, integration with Django was one of our main concerns, when we designed and implemented XRecord. When we first attempted to port some of our applications to use Django, the situation was as follows:

- we had a big, complex MySQL database, with a frequently changing structure,
- we had a number of Python applications that used this database,
- we had a big, ugly PHP web app, which also used this database,
- we had a simple thin db-api Python library named XRecord used by the Python applications.

We decided to port the web app to Django, so it seemed what we needed to do was:

1. use Django's *inspectdb* feature to generate the model from our db,
2. rewrite the web app
3. later rewrite the Python applications to use the Django model, so the project code is clean.

Step `1` turned out to be problematic, but not impossible. The Django introspection engine had some issues detecting all the relationships between tables, so they had to be completed by hand.

Step `2` seemed to be going fine, some working prototypes were produced, but then we had to modify the database definition, and there was no other way, but to

- modify the mysql database
- make the corresponding changes to the model, by hand

As lovers of the DRY principle, we were totally dissatisfied with the way this was turning out. So we quickly moved to step `3`, to see if any other problems would surface. Without going into details - we understood that Django was simply not a good tool to write applications that are not meant to run in a web environment. We also understood, that it does not have to be such a tool, and probably, should not be, since it had "Web framework" in the name.

So we decided to take a different approach:

1. modify XRecord so it can be used inside Django
2. rewrite the web app
3. leave the other Python apps as they are, tested and working

which is good because we have a single database layer for both the web-application and the non-web-applications. DRY.

CHAPTER

EIGHT

INDICES AND TABLES

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

PYTHON MODULE INDEX