dbkit Documentation

Release 0.2.5

Keith Gaughan

May 14, 2016

Contents

dbkit is a library that abstracts away at least part of the pain involved in dealing with DB-API 2 compatible database drivers.

CHAPTER 1

Overview

Note: Like dbkit itself, this documentation is a work in progress. Unlike dbkit, it is nowhere near complete yet. Bear with me.

1.1 Introduction

dbkit is intended to be used in circumstances where it is impractical or overkill to use an ORM such as SQLObject or SQLAlchemy, but it would be useful to at least abstract away some of the pain involved in dealing with the database.

Features:

- Rather than passing around database connections, statements are executed within a database context, thus helping to decouple modules that interface with the database from the database itself and its connection details.
- Database contexts contain references to the exceptions exposed by the database driver, thus decoupling exception handling from the database driver.
- Easier to use transaction handling.
- Easier iteration over resultsets.
- Connection pooling. In addition, any code using pooled connections has no need to know connection pooling is in place.
- Query logging.

Non-aims:

- Abstraction of SQL statements. The idea is to get rid of the more annoying but necessary boilerplate code involved in dealing with DB-API 2 drivers, not to totally abstract away SQL itself.

1.2 Comparison with straight DB-API 2 code

Need a "Hello, World!" example? Here's how you'd set up a connection context, query a database table, and print out its contents with *dbkit*:

```
from dbkit import connect, query
from contextlib import closing
import sqlite3
```

```
with connect(sqlite3, 'counters.db') as ctx, closing(ctx):
    for counter, value in query('SELECT counter, value FROM counters'):
        print "%s: %d" % (counter, value)
```

And here's how you'd so it with a DB-API 2 (using *just* PEP 249, no driver-specific extensions):

```
import sqlite3
from contextlib import closing

with closing(sqlite3.connect('counters.db')) as conn:
    with closing(conn.cursor()) as cur:
        cur.execute('SELECT counter, value FROM counters')
        while True:
            row = cur.fetchone()
            if row is None:
                break
            print "%s: %d" % row
```

1.3 Download

The latest *development* version can be found in the *dbkit* Git repository:

```
git clone https://github.com/kgaughan/dbkit
```

The project has yet to be submitted to PyPI, but I'm hoping to do that as soon as I'm happy with the documentation. To build a source package for installation and subsequently install it, do:

```
python setup.py sdist
pip install dist/dbkit-0.2.5.tar.gz
```

Alternatively, you can install it directly, bypassing package creation:

```
python setup.py install
```

1.4 Requirements

dbkit will work with Python 2.6, 2.7, and 3.3+ without issue. It appears to have some minor issues with PyPy, but it ought to work fine. Since 0.2.3, it is no longer compatible with Python 2.5.

dbkit has no dependencies other than requiring a database driver and six to allow Python 2 and Python 3 support.

CHAPTER 2

Tutorial

2.1 A simple application

Let's start with an 'hello, world' example. It's a small application for manipulating an SQLite database of counter. Here's the schema:

```
CREATE TABLE counters (
    counter TEXT    PRIMARY KEY,
    value   INTEGER
);
```

You'll find that file in the *examples* directory, and it's called *counters.sql*. Let's create the database:

```
$ sqlite3 counters.sqlite < counters.sql
```

You should now have the database set up.

You'll find the file we'll be working on in the examples as *counters.py*.

Now let's import some of the libraries we'll be needing for this project:

```
"""

from contextlib import closing
from os import path
import sqlite3
import sys
```

There are a few different thing we want to be able to do to the counter, such as setting a counter, deleting a counter, listing counters, incrementing a counter, and getting the value of a counter. We'll need to implement the code to do those.

One of the neat things about *dbkit* is that you don't have to worry about passing around database connections. Instead, you create a context in which the queries are ran, and *dbkit* itself does the work. Thus, we can do something like this:

```
value = query_value(
    'SELECT value FROM counters WHERE counter = ?',
    (counter,),
    default=0)
```

And we don't need to worry about the database connection we're actually dealing with. With that in mind, here's how we'd implement getting a counter's value with *dbkit.query_value()*:

```
def get_counter(counter):
    """
    Get the value of a counter.
    """
    print query_value(
        'SELECT value FROM counters WHERE counter = ?',
        (counter,),
        default=0)
```

To perform updates, there's the *dbkit.execute()* function. Here's how we increment a counter's value:

```
def set_counter(counter, value):
    """
    Set a counter.
    """
    execute(
        'REPLACE INTO counters (counter, value) VALUES (?, ?)',
        (counter, value))
```

dbkit also makes dealing with transactions very easy. It provides two mechanisms: the *dbkit.transaction()* context manager, as demonstrated above, and *dbkit.transactional()* decorator. Let's implement incrementing the counter using the context manager:

```
def increment_counter(counter, by):
    """
    Modify the value of a counter by a certain amount.
    """
    execute(
        'UPDATE counters SET value = value + ? WHERE counter = ?',
        (by, counter))
```

With the decorator, we'd write the function like so:

```
@transactional
def increment_counter(counter, by):
    execute(
        'UPDATE counters SET value = value + ? WHERE counter = ?',
        (by, counter))
```

Both are useful in different circumstances.

Deleting a counter:

```
def delete_counter(counter):
    """
    Delete a counter.
    """
    execute(
        'DELETE FROM counters WHERE counter = ?',
        (counter,))
```

dbkit also has ways to query for result sets. Once of these is *dbkit.query_column()*, which returns an iterable of the first column in the result set. Thus, to get a list of counters, we'd do this:

```
def list_counters():
    """
    List the names of all the stored counters.
    """
    print "\n".join(query_column('SELECT counter FROM counters'))
```

One last thing that our tool ought to be able to do is dump the contents of the *counters* table. To do this, we can use *dbkit.query()*:

```
def dump_counters():
    """
    Query the database for all counters and their values.
    """
    return query('SELECT counter, value FROM counters')
```

This will return a sequence of result set rows you can iterate over like so:

```
def print_counters_and_values():
    """
    List all the counters and their values.
    """
    for counter, value in dump_counters():
        print "%s: %d" % (counter, value)
```

By default, query() will use tuples for each result set row, but if you'd prefer dictionaries, all you have to do is pass in a different row factory when you call *dbkit.query()* using the *factory* parameter:

```
def dump_counter_dict():
    return query(
        'SELECT counter, value FROM counters',
        factory=dict_set)
```

dbkit.dict_set() is a row factory that generates a result set where each row is a dictionary. The default row factory is *dbkit.tuple_set()*, which yields tuples for each row in the result set. Using *dbkit.dict_set()*, we'd print the counters and values like so:

```
def print_counters_and_values():
    for row in dump_counters_dict():
        print '%s: %d' % (row['counter'], row['value'])
```

Now we have enough for our counter management application, so lets start on the main function. We'll have the following subcommands: *set*, *get*, *del*, *list*, *incr*, *list*, and *dump*. The *dispatch()* function below deals with calling the right function based on the command line arguments, so all we need to create a database connection context with *dbkit.connect()*. It takes the database driver module as its first argument, and any parameters you'd pass to that module's *connect()* function to create a new connection as its remaining arguments:

```
def main():
    # This table tells us the subcommands, the functions to dispatch to,
    # and their signatures.
    command_table = {
        'set': (set_counter, str, int),
        'del': (delete_counter, str),
        'get': (get_counter, str),
        'list': (list_counters,),
        'incr': (increment_counter, str, int),
        'dump': (print_counters_and_values,),
    }
    with connect(sqlite3, 'counters.sqlite') as ctx:
        with closing(ctx):
            dispatch(command_table, sys.argv)
```

Finally, two utility methods, the first of which decides which of the functions to call based on a command dispatch table and the arguments the program was ran with:

```
def dispatch(table, args):
    """
    Dispatches to a function based on the contents of `args`.
    """
    # No arguments: print help.
    if len(args) == 1:
        print_help(args[0], table)
        sys.exit(0)

    # Bad command or incorrect number of arguments: print help to stderr.
    if args[1] not in table or len(args) != len(table[args[1]]) + 1:
        print_help(args[0], table, dest=sys.stderr)
        sys.exit(1)

    # Cast all the arguments to fit their function's signature to ensure
    # they're correct and to make them safe for consumption.
    sig = table[args[1]]
    try:
        fixed_args = [type_(arg) for arg, type_ in zip(args[2:], sig[1:])]
    except TypeError:
        # If any are wrong, complain to stderr.
        print_help(args[0], table, dest=sys.stderr)
        sys.exit(1)

    # Dispatch the call to the correct function.
    sig[0](*fixed_args)
```

And a second for displaying help:

```
def print_help(filename, table, dest=sys.stdout):
    """
    Print help to the given destination file object.
    """
    cmds = '|'.join(sorted(table.keys()))
    print >> dest, "Syntax: %s %s [args]" % (path.basename(filename), cmds)
```

Bingo! You now has a simple counter manipulation tool.

CHAPTER 3

Examples

3.1 counters.py

A command line tool for manipulating and querying bunch of counters stored in an SQLite database. This demonstrates basic use of *dbkit*.

```
"""
A counter management tool.
"""

from contextlib import closing
from os import path
import sqlite3
import sys

from dbkit import (
    connect,
    execute,
    query,
    query_column,
    query_value,
    transactional,
)


def get_counter(counter):
    """
    Get the value of a counter.
    """
    print query_value(
        'SELECT value FROM counters WHERE counter = ?',
        (counter,),
        default=0)


@transactional
def set_counter(counter, value):
    """
    Set a counter.
    """
    execute(
        'REPLACE INTO counters (counter, value) VALUES (?, ?)',
        (counter, value))
```

```
38
39
40  @transactional
41  def increment_counter(counter, by):
42      """
43      Modify the value of a counter by a certain amount.
44      """
45      execute(
46          'UPDATE counters SET value = value + ? WHERE counter = ?',
47          (by, counter))
48
49
50  @transactional
51  def delete_counter(counter):
52      """
53      Delete a counter.
54      """
55      execute(
56          'DELETE FROM counters WHERE counter = ?',
57          (counter,))
58
59
60  def list_counters():
61      """
62      List the names of all the stored counters.
63      """
64      print "\n".join(query_column('SELECT counter FROM counters'))
65
66
67  def dump_counters():
68      """
69      Query the database for all counters and their values.
70      """
71      return query('SELECT counter, value FROM counters')
72
73
74  def print_counters_and_values():
75      """
76      List all the counters and their values.
77      """
78      for counter, value in dump_counters():
79          print "%s: %d" % (counter, value)
80
81
82  def print_help(filename, table, dest=sys.stdout):
83      """
84      Print help to the given destination file object.
85      """
86      cmds = '|'.join(sorted(table.keys()))
87      print >> dest, "Syntax: %s %s [args]" % (path.basename(filename), cmds)
88
89
90  def dispatch(table, args):
91      """
92      Dispatches to a function based on the contents of `args`.
93      """
94      # No arguments: print help.
95      if len(args) == 1:
```

```
        print_help(args[0], table)
        sys.exit(0)

    # Bad command or incorrect number of arguments: print help to stderr.
    if args[1] not in table or len(args) != len(table[args[1]]) + 1:
        print_help(args[0], table, dest=sys.stderr)
        sys.exit(1)

    # Cast all the arguments to fit their function's signature to ensure
    # they're correct and to make them safe for consumption.
    sig = table[args[1]]
    try:
        fixed_args = [type_(arg) for arg, type_ in zip(args[2:], sig[1:])]
    except TypeError:
        # If any are wrong, complain to stderr.
        print_help(args[0], table, dest=sys.stderr)
        sys.exit(1)

    # Dispatch the call to the correct function.
    sig[0](*fixed_args)


def main():
    # This table tells us the subcommands, the functions to dispatch to,
    # and their signatures.
    command_table = {
        'set': (set_counter, str, int),
        'del': (delete_counter, str),
        'get': (get_counter, str),
        'list': (list_counters,),
        'incr': (increment_counter, str, int),
        'dump': (print_counters_and_values,),
    }
    with connect(sqlite3, 'counters.sqlite') as ctx:
        with closing(ctx):
            dispatch(command_table, sys.argv)


if __name__ == '__main__':
    main()
```

3.2 pools.py

A small web application, built using web.py, pystache, and psycopg2, to say that prints "Hello, *name*" based on the URL fetched, and which records how many times it's said hello to a particular name.

This demonstrates use of connection pools.

```
import web
import psycopg2
import pystache
from dbkit import (
    dict_set,
    execute,
    Pool,
    query,
```

```
    query_value,
    transactional,
)


urls = (
    '/(.*)', 'hello',
)
app = web.application(urls, globals())
pool = Pool(psycopg2, 2, "dbname=namecounter user=keith")


TEMPLATE = """<!DOCTYPE html>
<html>
    <head>
        <title>Hello!</title>
    </head>
    <body>
        <p>Hello, {{name}}!</p>
        <p>Previously, I've said hello to:</p>
        <ul>
        {{#hellos}}
            <li>{{name}}, {{n}} times</li>
        {{/hellos}}
        </ul>
    </body>
</html>"""


@transactional
def save_name(name):
    if query_value("SELECT n FROM greeted WHERE name = %s", (name,), 0) == 0:
        execute("INSERT INTO greeted (name, n) VALUES (%s, 1)", (name,))
    else:
        execute("UPDATE greeted SET n = n + 1 WHERE name = %s", (name,))


def get_names():
    return query("SELECT name, n FROM greeted ORDER BY n", factory=dict_set)


class hello(object):

    def GET(self, name):
        ctx = pool.connect()
        if not name:
            name = 'World'
        with ctx:
            hellos = list(get_names())
            save_name(name)
        return pystache.render(TEMPLATE, {'name': name, 'hellos': hellos})


if __name__ == '__main__':
    try:
        app.run()
    finally:
        pool.finalise()
```

CHAPTER 4

Design philosophy

The design philosophy of dbkit can be summed up as "decoupling through statefulness".

Many things about database interaction are already stateful, transactions for instance. While statelessness can, in many areas, help with scalability, it has a negative effect on ease of use: stateless code must always have the context it requires to do its work passed to it.

Implemented poorly, this means that stateless code can end up with assumptions about the kind of contextual information it needs to have to do its job and where it's much easier to do the wrong thing than to do the right thing.

dbkit aims to solve this, at least for relational database access, by providing an interface that makes the easy solution the right case while still making the difficult stuff possible[1]_.

It does this by decoupling the execution of SQL statements from the connection they're executed against. This might seem like a small thing, but it has significant consequences: it means that database code need have little if any awareness of the environment it executes in, and what context it does need to have can easily be introspected when needed.

CHAPTER 5

Reference

dbkit.**connect**(*module*, **args*, ***kwargs*)
Connect to a database using the given DB-API driver module. Returns a database context representing that connection. Any arguments or keyword arguments are passed the module's `connect()` function.

dbkit.**context**()
Returns the current database context.

5.1 Contexts

Contexts wrap a notional database connection. They're returned by the *dbkit.connect()* function. Methods are for the internal use of dbkit only though it does expose a method for closing the database connection when you're done with it and contains references for each of the exceptions exposed by the connection's database driver. For a list of these exceptions, see PEP-0249.

class dbkit.**Context**(*module*, *mdr*)
A database connection context.

default_factory
The row factory used for generating rows from *dbkit.query()* and *dbkit.query_row()*. The default is *dbkit.tuple_set()*.

The factory function should take a cursor an return an iterable over the current resultset.

logger
The function used for logging statements and their arguments.

The logging function should take two arguments: the query and a sequence of query arguments.

There are two supplied logging functions: *dbkit.null_logger()*, the default, logs nothing, while *dbkit.stderr_logger()* logs its arguments to stderr.

close()
Close the connection this context wraps.

5.2 Exceptions

class dbkit.**NoContext**
You are attempting to use dbkit outside of a database context.

class dbkit.**NotSupported**
You are attempting something unsupported.

class dbkit.**AbortTransaction**
Raised to signal that code within the transaction wants to abort it.

5.3 Transactions

dbkit.**transaction**()
Sets up a context where all the statements within it are ran within a single database transaction.

Here's a rough example of how you'd use it:

```
import sqlite3
import sys
from dbkit import connect, transaction, query_value, execute

# ...do some stuff...

with connect(sqlite3, '/path/to/my.db') as ctx:
    try:
        change_ownership(page_id, new_owner_id)
    catch ctx.IntegrityError:
        print >> sys.stderr, "Naughty!"

def change_ownership(page_id, new_owner_id):
    with transaction():
        old_owner_id = query_value(
            "SELECT owner_id FROM pages WHERE page_id = ?",
            (page_id,))
        execute(
            "UPDATE users SET owned = owned - 1 WHERE id = ?",
            (old_owner_id,))
        execute(
            "UPDATE users SET owned = owned + 1 WHERE id = ?",
            (new_owner_id,))
        execute(
            "UPDATE pages SET owner_id = ? WHERE page_id = ?",
            (new_owner_id, page_id))
```

dbkit.**transactional**(*wrapped*)
A decorator to denote that the content of the decorated function or method is to be ran in a transaction.

The following code is equivalent to the example for *dbkit.transaction()*:

```
import sqlite3
import sys
from dbkit import connect, transactional, query_value, execute

# ...do some stuff...

with connect(sqlite3, '/path/to/my.db') as ctx:
    try:
        change_ownership(page_id, new_owner_id)
    catch ctx.IntegrityError:
        print >> sys.stderr, "Naughty!"

@transactional
```

```
    def change_ownership(page_id, new_owner_id):
        old_owner_id = query_value(
            "SELECT owner_id FROM pages WHERE page_id = ?",
            (page_id,))
        execute(
            "UPDATE users SET owned = owned - 1 WHERE id = ?",
            (old_owner_id,))
        execute(
            "UPDATE users SET owned = owned + 1 WHERE id = ?",
            (new_owner_id,))
        execute(
            "UPDATE pages SET owner_id = ? WHERE page_id = ?",
            (new_owner_id, page_id))
```

5.4 Statement execution

These functions allow you to execute SQL statements within the current database context.

dbkit.**execute**(*stmt*, *args=()*)
Execute an SQL statement. Returns the number of affected rows.

dbkit.**query**(*stmt*, *args=()*, *factory=None*)
Execute a query. This returns an iterator of the result set.

dbkit.**query_row**(*stmt*, *args=()*, *factory=None*)
Execute a query. Returns the first row of the result set, or *None*.

dbkit.**query_value**(*stmt*, *args=()*, *default=None*)
Execute a query, returning the first value in the first row of the result set. If the query returns no result set, a default value is returned, which is *None* by default.

dbkit.**query_column**(*stmt*, *args=()*)
Execute a query, returning an iterable of the first column.

5.5 Stored procedures

These functions allow you to execute stored procedures within the current database context, if the DBMS supports stored procedures.

dbkit.**execute_proc**(*procname*, *args=()*)
Execute a stored procedure. Returns the number of affected rows.

dbkit.**query_proc**(*procname*, *args=()*, *factory=None*)
Execute a stored procedure. This returns an iterator of the result set.

dbkit.**query_proc_row**(*procname*, *args=()*, *factory=None*)
Execute a stored procedure. Returns the first row of the result set, or *None*.

dbkit.**query_proc_value**(*procname*, *args=()*, *default=None*)
Execute a stored procedure, returning the first value in the first row of the result set. If it returns no result set, a default value is returned, which is *None* by default.

dbkit.**query_proc_column**(*procname*, *args=()*)
Execute a stored procedure, returning an iterable of the first column.

5.6 Result generators

Result generators are generator functions that are used internally by dbkit to take the results from a database cursor and turn them into a form that's easier to deal with programmatically, such a sequence of tuples or a sequence of dictionaries, where each tuple or dictionary represents a row of the result set. By default, *dbkit.tuple_set()* is used as the result generator, but you can change this by assigning another, such as *dbkit.dict_set()* to `dbkit.Context.default_factory` function.

Some query functions allow you to specify the result generator to be used for the result, which is passed in using the *factory* parameter.

`dbkit.`**`dict_set`**(*cursor*, *mdr*)
Iterator over a statement's results as a dict.

`dbkit.`**`tuple_set`**(*cursor*, *mdr*)
Iterator over a statement's results where each row is a tuple.

5.7 Loggers

Loggers are functions that you can assign to `dbkit.Context.logger` to have dbkit log any SQL statements ran or stored procedures called to some sink. dbkit comes with a number of simple loggers listed below. To create your own logger, simply create a function that takes two arguments, the first of which is the SQL statement or stored procedure name, and the second is a sequence of arguments that were passed with it.

`dbkit.`**`null_logger`**(*_stmt*, *_args*)
A logger that discards everything sent to it.

`dbkit.`**`make_file_object_logger`**(*fh*)
Make a logger that logs to the given file object.

`dbkit.`**`stderr_logger`**(*stmt*, *args*)
A logger that logs to standard error.

5.8 Utilities

`dbkit.`**`to_dict`**(*key*, *resultset*)
Convert a resultset into a dictionary keyed off of one of its columns.

`dbkit.`**`make_placeholders`**(*seq*, *start=1*)
Generate placeholders for the given sequence.

5.9 Connection pools

Note: Connection pool support is currently considered pre-alpha.

Connection pooling is a way to share a common set of database connections over a set of contexts, each of which can be executing in different threads. Connection pooling can increase efficiency as it mitigates much of the cost involved in connecting and disconnecting from databases. It also can help lower the number of database connections an application needs to keep open with a database server concurrently, thus helping to lower server low.

As with contexts, pools have a copy of the driver module's exceptions. For a list of these exceptions, see PEP-0249.

The *acquire* and *release* methods are for internal use only.

class dbkit.**PoolBase**(*module*, *threadsafety*, *args*, *kwargs*)
Abstract base class for all connection pools.

acquire()
Acquire a connection from the pool and returns it.

This is intended for internal use only.

connect()
Returns a context that uses this pool as a connection source.

finalise()
Shut this pool down. Call this or have it called when you're finished with the pool.

Please note that it is only guaranteed to complete after all connections have been returned to the pool for finalisation.

release(*conn*)
Release a previously acquired connection back to the pool.

This is intended for internal use only.

class dbkit.**Pool**(*module*, *max_conns*, **args*, ***kwargs*)
A very simple connection pool.

connect()
Returns a context that uses this pool as a connection source.

finalise()

5.10 Connection mediators

Connection mediators are used internally within contexts to mediate connection acquisition and release between a context and a (notional) connection pool. They're an advanced feature that you as a developer will only need to understand and use if writing your own connection pool. All connection mediator instances are context managers.

Note: You might find the naming a bit odd. After all, wouldn't calling something like this a 'manager' be just as appropriate and less... odd? Not really. Calling something a 'manager' presupposes a degree of control over the resource in question. A 'mediator', on the other hand, simply acts as a middle man which both parties know. Introducing the mediator means that contexts don't need to know where their connections come from and pools don't need to care how they're used. The mediator takes care of all that.

---

class dbkit.**ConnectionMediatorBase**(*exceptions*)
Mediates connection acquisition and release from/to a pool.

Implementations should keep track of the times they've been entered and exited, incrementing a counter for the former and decrementing it for the latter. They should acquire a connection when entered with a counter value of 0 and release it when exited with a counter value of 0.

close()
Called to signal that any resources can be released.

commit()
Commit the current transaction.

cursor()
Get a cursor for the current connection.

rollback()
Rollback the current transaction.

class dbkit.**SingleConnectionMediator**(*module*, *connect_*)
Mediates access to a single unpooled connection.

class dbkit.**PooledConnectionMediator**(*pool*)
Mediates connection acquisition and release from/to a pool.

CHAPTER 6

Change history

6.1 0.2.5 (2016-04-15)

- Minor fixes and cleanup, including getting rid of a nose dependency.
- Wheel support.

6.2 0.2.4 (2015-11-30)

- Python 3 support.

6.3 0.2.3 (2015-11-26)

- *Context.cursor()* now always creates a transaction. The lack of this outer transaction meant that PostgreSQL would end up with a large number of idle transactions that had neither been committed or rolled back.
- This is the last version that will work on Python 2.5.

6.4 0.2.2 (2013-04-04)

- Scrap *unindent_statement()*.
- Derive all dbkit exceptions from *Exception*.
- Clean up connection pinging code.
- Add *make_placeholders()* for generating statement placeholders safely.
- Add *to_dict()* for converting resultsets to dicts mapped off of a particular field.

6.5 0.2.0 (2012-10-16)

- Add *last_row_id()*.
- Pools now can have custom mediators.
- Cursors are now tracked.

- Pooled connections are no longer closed prematurely.
- Row factories are now usable outside of context safely.

6.6 0.1.4 (2012-10-11)

- *execute*()* now returns the number of affected rows.
- Add *last_row_count* and *last_row_id* to *Context*.
- Remove *DummyPool* and *ThreadAffinePool*, though the latter may be returning.
- Stablise the behaviour of *Pool* when dealing with expired connections.
- Documentation version is now pegged directly to the library.

6.7 0.1.2 (2012-09-02)

- Initial revision with a changelog.

CHAPTER 7

Indices and tables

- genindex
- search