Introduction

Application programming

Scalability

Maintenance

Future ideas

Epilogue

Seamless object persistence in Python — ZODB 3

Christian Theune

gocept gmbh & co. kg

July 9, 2007

About me

- Christian Theune, Software Developer
- Co-founder of “gocept” a Zope service provider and strong proponent of free software
- Core developer of various zope.org projects including Zope 3 and ZODB
- Developer of various Zope-related projects spawned by customer projects

Contents

Introduction

Application programming

Scalability

Maintenance

Future ideas

Epilogue

1 Introduction

2 Application programming

3 Scalability

4 Maintenance

5 Future ideas

6 Epilogue

Introduction

Seamless object persistence in Python — ZODB 3

Christian Theune

Introduction

Application programming

Scalability

Maintenance

Future ideas

Epilogue

ZODB is the “Zope Object Database” — an object database for Python.

Introduction

An object-oriented application creates objects. The objects and their data are lost when the application's process ends.

Objects have to be re-created from scratch when the application starts again.

The application needs to store the data somewhere before the process ends (and load it again when it is started).

Introduction

The ZODB automates the mechanics of preserving the objects (and their data) past the lifetime of an application process.

Application developers can simply create and modify the objects and not have to worry about when and how to load and save data physically.

Introductory example

Let's use the interactive Python interpreter to show how the ZODB stores and loads objects. Assume we have the following class:

```
class Account(Persistent):
    def __init__(self):
        self.balance = 0.0
    def deposit(self, amount):
        assert amount > 0
        self.balance += amount
    def cash(self, amount):
        assert amount < self.balance
        self.balance -= amount
```

Example — Review

What happened in the example?

- The database has a dictionary-like root object.
- We instantiated a new persistent object and stored it in the root object.
- “transaction.commit()” makes any changes persistent.
- “transaction.abort()” will revert any changes – as does ending the process without committing.
- Remember: it’s a native object database. No RDBMS is hidden behind the scenes.

Example — The boilerplate

The example was simplified. The “root” object was “just there”. Here’s the boilerplate:

```
import ZODB.FileStorage.FileStorage
from ZODB.DB import DB
import transaction

storage = ZODB.FileStorage.FileStorage(
    '/tmp/example1.fs')
db = DB(storage)
connection = db.open()
root = connection.root()
```

Architecture — or: Why the boilerplate?

We used 7 lines of boilerplate to setup the ZODB to use within our program. Why?

Two aspects:

- database architecture
- transaction management

Architecture — Database architecture

The database architecture involves three elements: a **root object**, a **database** and a **storage**.

The **root object** is the entry point to the database that references our objects. From the root we can access all other objects by interacting with them: accessing attributes or calling methods.

The **database** contains the root object and represents the logical layer of the DBMS model: it stores objects and serializes them into data.

The **storage** is the physical layer of the DBMS model and acts as a backend for the database. Storages store serialized objects (data) physically.

Architecture — Transaction management

The transaction management architecture involves two important elements: a **transaction** and a **connection**.

The root object can not be retrieved directly from a database object. We first have to open a **connection** on which we then call the ‘root‘ method.

A connection is associated with a **transaction** and any object retrieved from a connection is therefore associated with the same transaction.

Note: There is some more internal architecture involved which allows coordinating transactions with objects from multiple databases and even different transaction systems like relational databases.

Summary overview of features

- Can store arbitrary Python objects, but provides special types for optimal support (e.g. BTrees)
- Full ACID compatible transactions
- Different storage implementations: FileStorage, MappingStorage, DemoStorage, ZEO
- In-transaction savepoints
- Undo / Time Travel
- Application-level conflict resolution

Application programming

Let's take a closer look at what working with ZODB is like:

- Installation
- Configuration
- Persistent objects
- Transaction API
- Testing

Installation

The ZODB is available as a Python egg and can be installed using setuptools' "easy_install":

```
$ easy_install ZODB3
...
$ python
>>> import ZODB
```

Configuration

The “manual” way of opening a database from Python code makes your application depend on a specific configuration:

```
>>> from ZODB.FileStorage.FileStorage \
...     import FileStorage
>>> storage = FileStorage("example.fs")
>>> from ZODB.DB import DB
>>> database = DB(storage)
>>> connection = database.open()
>>> connection.root()
{}
```

Configuration

For better maintenance and control, you can use a config file:

```
<zodb>
    <filestorage>
        path /tmp/example.fs
    </filestorage>
</zodb>
```

Then use “databaseFromURL()” to configure your application’s database:

```
>>> from ZODB.config import databaseFromURL
>>> db = databaseFromURL('.../zodb.conf')
>>> connection = db.open()
>>> connection.root()
{}
```

Persistent objects

(Almost) any object can be placed in the database.

However, if we have mutable objects, the changes aren't detected and don't get picked up by the transaction.

The base class "Persistent" detects attribute changes.

Sub-objects have to derive from "Persistent" as well if they are mutable. For lists and dictionaries we have "PersistentList", "PersistentDict" and various BTree flavours.

(Demo)

Persistent objects - Blobs

One of the kinds of objects that can't be pickled are files.

Historically there have been various approaches to represent file data in the ZODB like using long strings (memory inefficient) or linked lists of data chunks (caching inefficient).

Since ZODB 3.8 (currently beta) Blobs allow to store file data efficiently and with a "natural" interface.

(Demo)

Transaction API

The transaction API has five methods:

- begin()
- commit()
- abort()
- doom()
- savepoint(optimistic=False)

Transactions – Beginning

Any connection is always associated with a transaction, even if we do not call “begin”.

We can also explicitly begin a transaction before our application starts making changes (and to make sure we see the latest version of the database). If a transaction already existed then it will be aborted and your previous changes are lost.

```
>>> transaction.begin()
```

Transactions – Committing

If all of our changes in a transaction were successfull and we want to make then durable, we can commit the transaction:

```
>>> transaction.commit()
```

Transactions – Aborting

Instead of committing the transaction we can also abort it and discard all changes that happened during the transaction:

```
>>> root['banana'].price
1.2
>>> root['banana'].price = 1.5
>>> root['banana'].price
1.5
>>> transaction.abort()
>>> root['banana'].price
1.2
```

Transactions – Dooming

When aborting a transaction, we immediately loose all changes. Sometimes we want to abort a transaction but create some status message or log before actually aborting the data.

The “doom()” function makes sure that a transaction can not be committed, but keeps the data around:

```
>>> root['banana'].price = 2.0
>>> transaction.doom()
>>> root['banana'].price
2.0
>>> transaction.commit()
Traceback (most recent call last):
  ...
DoomedTransaction
>>> transaction.abort()
```

Transactions – Savepoints

Transactions are by definition atomic.

In some situations, like batch processing, it is worthwhile to be able to process some data, remember that state, process some more and revert to the previous state without having to commit the whole transaction immediately.

Savepoints help with this.

(Demo)

Undo

ZODB storages can support saving old versions of objects. Those storages also can provide an API for undoing (and re-doing) transactions:

The API includes:

- IDatabase.undoInfo()
- IDatabase.undoLog()
- IDatabase.undo()

(Demo)

Scalability

The ZODB can be scaled up very well to:

- process many transactions
- store and handle many objects

Scalability — Processing many transactions

Processing many transactions is mainly a problem of CPU power.

Zope Enterprise Objects (ZEO) are a solution to this. They provide a network server for the ZODB that is used as an alternative storage and allows multiple Python processes to work against a single database.

ZEO includes distributed transaction management so that your applications can still be integrated with other transaction systems.

(Demo, if time)

Scalability — Handling many objects

Handling many objects is a matter of providing data structures that efficiently allow **handling large collections** and ways to **query objects** without having to actually load all objects "manually".

Scalability — Handling large collections

The ZODB provides various flavours of **BTrees** which work like dictionaries to allow efficient handling of large collections.

Typical flavours are IOBTree (keys are integers) or OOBTree (keys are generic objects) and the *Set families (only store keys).

The efficiency comes from not having to load the whole collection of objects into memory when talking about subsets of the collection.

Scalability — Querying objects

The ZODB itself does not provide specific mechanics for querying objects, but the Zope project provides various solutions to this: **zope.index**, **zope.catalog**.

The general approach is to provide a tabular view on the object data and use efficient indexes based on our BTrees to look up queries.

Maintenance

The ZODB is a very low-maintenance database, but a few things need to be considered:

- The storage that is used — packing might be required
- Upgrading — Software needs to take care when upgrading from old databases
- Backups — FileStorage allows hot snapshots
- Reliability — ZRS to remove SPOF from the ZEO server

Future ideas

- Abstracted network layer to simplify testing
- Size-based caches

My personal projects include:

- ZEORaid
- ZODB Index

Questions

Any questions?

Thank you

Thank you for listening!

I'll be at the Zope booth if you have further questions.