Chapter 5
"A Photoblog Application"
In this package, you will find:
A Biography of the author of the book
A preview chapter from the book, Chapter 5 "A Photoblog Application"
A synopsis of the book’s content
Information on where to buy this book

About the Author

**Sylvain Hellegouarch** is an IT Software Consultant dedicated to the development of free software projects such as CherryPy. Since 2004, he has been coordinating and administrating the community efforts around the project providing support for newcomers and seasoned developers, alike. In 2006, he developed 'bridge' and 'amplee', two Python-based projects centered on XML and the upcoming Atom Publishing Protocol respectively.

He has also been deeply involved in The Viberavetions Project, a comprehensive grassroots solution for independent artists and musicians to better connect with consumers, as well as the nuXleus project, a platform designed for faster, more reliable inter- and intra-application and personal communication.

Born in France, Sylvain graduated with a degree in Computer Science from South Brittany University, Lorient, France in 2002. Since then he has been working as an IT consultant for a variety of companies, both small and large. He currently resides in the United Kingdom.

CherryPy Essentials: Rapid Python Web Application Development

Over the last few years, the boom that the World has experienced with the Internet breakthrough has pushed almost every programming language or platform to welcome the rise of web development toolkits, libraries, and frameworks.

The Python programming language has grown a rather large list of these environments though apart from a few of them such as Zope and Twisted most have a fairly small community. It is in this context that CherryPy came into existence when Rémi Delon, its creator, decided that he needed a tool that would work as he wanted for his own personal projects. He then released CherryPy under a free software license so that anyone could use, distribute, and contribute to the project.

CherryPy is a Python library implementing the HTTP protocol, which is at the very core of the Web, using common Python idioms. On top of that CherryPy offers its own view and concepts on how to help a developer to build web applications while being minimally intrusive through its own simple and straightforward API.

This book will guide you through the CherryPy library with the aim of giving you the key to make the best of it in your own web applications.

The first four chapters are dedicated to CherryPy, providing information ranging from its history to an in-depth presentation of its key features. The rest of the book will then take you into the development of a photoblog application. Each chapter tries to provide enough background to allow you to ponder the why and how of each decision made. Indeed writing software applications is not a precise science and compromises need to be undertaken for the better, however, the truth is that writing software usually does not go quite as planned. I have written this book with the hope that in the end you would have learnt much more than using a Python library.
What This Book Covers

Chapter 1 presents the story behind CherryPy and a high-level overview of the project.

Chapter 2 guides you through the installation and deployment of CherryPy via common strategies like using distutils, setuptools, or subversion.

Chapter 3 gives an overview of the main and the most common aspects of CherryPy, which will give you an understanding of what the library can do.

Chapter 4 goes into an in-depth review of the main aspects of the library such as its support for the HTTP protocol or the WSGI interface. It also extensively discusses the tool feature of the CherryPy API.

Chapter 5 introduces the application, which will be the unifying theme for the rest of the book. The chapter reviews the basic entities that the application will manipulate before moving onto explaining how we will map them into a relational database. This will allow us to explain the concept of ORM and perform a quick comparison between SQLAlchemy, SQLObject, and Dejavu.

Chapter 6 presents the idea behind web services by reviewing REST and the Atom Publishing Protocol.

Chapter 7 describes how to use a templating engine such as Kid to generate web pages dynamically. The chapter also introduces Mochikit a JavaScript toolkit to perform client-side development.

Chapter 8 extends chapter 7 by diving into the world of Ajax, which has reminded web developers that they can create extremely powerful applications by simply using the browser capabilities, the JavaScript language, and the HTTP protocol.

Chapter 9 makes a strong point that any application should be reasonably well tested and introduces some testing strategies like unit testing, functional testing, and load testing.

Chapter 10 ends the book by reviewing some methods to deploy a CherryPy application under a common web-server front end like Apache and lighttpd. The chapter also explains how to enable SSL from your CherryPy application.
In this chapter, we are going to explain what the next few chapters will put in place to develop a photoblog application. In the first half of this chapter, we will review the goals and features of this application from a high-level perspective without going into too much detail. In the second half, we will define the entities that our application will manipulate and introduce the concept of object-relational mappers, which aim at reducing the impedance mismatch between relational database and object-oriented software design. We will briefly present the most common Python ORMs and then develop our application data access layer based on the Dejavu ORM.

A Photoblog Application

In the previous chapters, we have reviewed CherryPy’s design and features in detail but we haven't demonstrated its use in the context of a web application. The next few chapters will undertake this task by going through the development of a photoblog application.

A photoblog is like a regular blog except that the principal content is not text but photographs. The main reason for choosing a photoblog is that the range of features to be implemented is small enough so that we can concentrate on their design and implementation.

The goals behind going through this application are as follows:

- To see how to slice the development of a web application into meaningful layers and therefore show that a web application is not very different from a rich application sitting on your desktop.
- To show that the separation of concerns can also be applied to the web interface itself by using principles grouped under the name of Ajax.
- To introduce common Python packages for dealing with common aspects of web development such as database access, HTML templating, JavaScript handling, etc.

Photoblog Entities

As mentioned earlier, the photoblog will try to stay as simple as possible in order to focus on the other aspects of developing a web application. In this section, we will briefly describe the entities our photoblog will manipulate as well as their attributes and relations with each other.

In a nutshell our photoblog application will use the following entities and they will be associated as shown in the following figure:

This figure is not what our application will look like but it shows the entities our application will manipulate. One photoblog will contain several albums, which in turn will host as many films as required, which will carry the photographs.

In other words, we will design our application with the following entity structure:

**Entity:** Photoblog

**Role:** This entity will be the root of the application.

**Attributes:**

- name: A unique identifier for the blog
- title: A public label for the blog

**Relations:**

- One photoblog will have zero to many albums
Entity: Album

Role: An album carries a story told by the photographs as an envelope.

Attributes:
- name: A unique identifier for the album
- title: A public label for the album
- author: The name of the album's author
- description: A simple description of the album used in feeds
- story: A story attached to the album
- created: A timestamp of when the album is being created
- modified: A timestamp of when the album is being modified
- blog_id: A reference to the blog handling the album

Relations:
- One album will reference zero to several films

Entity: Film

Role: A film gathers a set of photographs.

Attributes:
- name: A unique identifier for the film
- title: A public label for the film
- created: A timestamp of when the film is being created
- modified: A timestamp of when the film is being modified
- album_id: A reference to the album

Relations:
- A film will reference zero to several photographs

Entity: Photo

Role: The unit of our application is a photograph.
A Photoblog Application

Attributes:

- **name**: A unique identifier for the photo
- **legend**: A legend associated with the photograph
- **filename**: The base name of the photograph on the hard-disk
- **filesize**: The size in bytes of the photograph
- **width**: Width of the photograph in pixels
- **height**: Height of the photograph in pixels
- **created**: A timestamp of when the photograph is being created
- **modified**: A timestamp of when the photograph is being modified
- **film_id**: A reference to the film carrying the photograph

Relations: None

Functionally, the photoblog application will provide APIs to manipulate those entities via the traditional **CRUD** interface: Create, Retrieve, Update, and Delete. We will elaborate more on this in Chapter 6.

Now that we have briefly introduced what kind of application we will be developing throughout the following chapters we can move on to the next section and start reviewing our options to handle the database aspect of the application. But first a quick glossary of the terms this chapter will use.

**Vocabulary**

Here is a list of the terms we will be using:

- **Persistence**: Persistence is the concept of data items outliving the execution of programs manipulating them. Simply put, it is the process of storing data in long lasting memory medium such as a disk.
- **Database**: A database is a collection of organized data. There are different organization models: hierarchical, network, relational, object-oriented, etc. A database holds the logical representation of its data.
- **Database Management System (DBMS)**: A DBMS is a group of related software applications to manipulate data in a database. A DBMS platform should offer the following among other features:
  - Persistence of the data
  - A query language to manipulate data
  - Concurrency control

Security control
Integrity control
Transaction capabilities

We will use DBMSes as the plural of DBMS.

DBMSes Overview
In this section, we will quickly review the different kinds of existing DBMSes. The goal is to quickly introduce their main characteristics.

Relational Database Management System (RDBMS)
Of all DBMSes, the RDBMS is the most common, whether it is in small applications or multi-national infrastructure. An RDBMS comes with a database based on the concepts of the relational model, a mathematical model that permits the logical representation of a collection of data through relations. A relational database should be a concrete implementation of the relational model. However, modern relational databases follow the model only to a certain degree.

The following table shows the correlation between the terms of the relational model and the relational database implementation.

<table>
<thead>
<tr>
<th>Relational Model</th>
<th>Relational Database</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relation</td>
<td>Table</td>
</tr>
<tr>
<td>Attribute</td>
<td>Column</td>
</tr>
<tr>
<td>Tuple</td>
<td>Row</td>
</tr>
</tbody>
</table>

Relational databases support a set of types to define the domain of scope a column can use. However, there are only a limited number of supported types, which can be an issue with complex data types as allowed in object-oriented design.

Structure Query Language more commonly known as SQL is the language used to define, manipulate, or control data within a relational database.
A Photoblog Application

The following table is a quick summary of SQL keywords and their contexts.

<table>
<thead>
<tr>
<th>Context</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data manipulation</td>
<td>SELECT, INSERT, UPDATE, DELETE</td>
</tr>
<tr>
<td>Data definition</td>
<td>CREATE, DROP, ALTER</td>
</tr>
<tr>
<td>Data control</td>
<td>GRANT, REVOKE</td>
</tr>
<tr>
<td>Transaction</td>
<td>START, COMMIT, ROLLBACK</td>
</tr>
</tbody>
</table>

A construction of these keywords is called an SQL statement. When executed, an SQL statement returns a collection of rows of the data matching the query or nothing.

The relational model algebra uses the relation composition to compose operations across different sets; this is translated in the relational database context by **joins**. Joining tables allows complex queries to be shaped to filter out data.

SQL provides the following three kinds of joins:

<table>
<thead>
<tr>
<th>Union Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INNER JOIN</td>
<td>Intersection between two tables.</td>
</tr>
<tr>
<td>LEFT OUTER JOIN</td>
<td>Limits the result set by the left table. So all results from the left table will be returned with their matching result in the right table. If no matching result is found, it will return a NULL value.</td>
</tr>
<tr>
<td>RIGHT OUTER JOIN</td>
<td>Same as the LEFT OUTER JOIN except that the tables are reversed.</td>
</tr>
</tbody>
</table>

There is no RDBMS written in Python but most RDBMSes can be accessed via a corresponding Python library.

**Object-Oriented Database Management System (OODBMS)**

An OODBMS uses the object-oriented model to organize and store information. In other words, an OODBMS allows objects to be stored without having to be mapped into a different data structure like the relational database. This implies a great consistency between the database persisting the data and the application layers encapsulating it. In fact, the persistence mechanism is unobtrusive to the developer.
XML Database Management System (XMLDBMS)

Native XML Databases (NXDs) use XML documents as the unit of data they store and manipulate. XMLDBMSes on top of NXDs are optimized in this sense and provide native support for standard XML selection and querying languages such as XPath and XQuery. Some modern RDBMSes offer XML support through transparent conversion between the XML and relational data model leveraging the introduction of an XML database requirement.

Object-Relational Mapping

For the last fifteen years the software industry has moved towards a generalized use of the object-oriented modeling paradigm in the different layers of software application development. One of the last bastions that have resisted against this wave has been the database domain. Nonetheless, over the years quite an important amount of work has been conducted with success in order to develop OODBMSes for filling the gap of managing data. In spite of that OODBMSes have not taken off enough to steal RDBMS's thunder.

There are several factors behind this:

- Cost of changing the market. For decades RDBMSes have been the DBMSes of predilection to store and organize data. Most businesses have built their infrastructure around RDBMSes and changing that state is an immense task and only few are ready to pay for such a risk.
- Cost of migrating existing data. Even if a company is ready to step into that direction for new projects, it is unlikely it will for the existing infrastructure as the cost of migration and integration would be too high.
- Lack of unified query language.
- Lack of third-party software such as reporting tools based on OODBMS.
- Lack of experts. Finding a database administrator for a RDBMS is much easier than for an ODBMS.

Object-Relational Mappers (ORMs) succeeded because they were an efficient and cost-effective answer to some of the enumerated issues. The principle behind object-relational mapping is to reduce the impedance mismatch between the two models with minimum intrusion. ORMs allow the database designer and administrator to keep their beloved RDBMSes while proposing an object-oriented interface to a certain extent to the software developer. ORM is an extra layer between the database and the application which translates an object into a database row and vice versa.
It is important to bear in mind though that ORM can alleviate the problem only to a certain degree and that in some cases the differences between the relational and object design cannot be met without some compromise on both sides. For example, most ORMs correlate a database table into a class, which works fine when the number of entities and their relationships stay at a basic level. Unfortunately, this one-to-one relationship between a table and a class does not always work well in more complex object-oriented design. In such cases the impedance mismatch between the relational and object-oriented models may force designers to make concessions that could have negative impacts in the long run in regards to extending and maintaining the software.

Python Object-Relational Mappers

This section will introduce three ORMs in a very basic example in order to provide a basic understanding of how they work and their differences. The purpose is not to declare one of these ORMs the winner but to give you an idea of their design and features.

The three ORMs we will cover are:

- SQLObject from Ian Bicking
- SQLAlchemy from Michael Bayer
- Dejavu from Robert Brewer

Although great care has been taken in this section, it may happen that by the time you read this chapter these products might have changed a bit. You will have to refer to their official documentation.

In the following example we will map the following entities:

- Artist: An artist is made of a name. An artist can have zero or several albums.
- Album: An album is made of a title and a release year. An album is associated to an artist and can have zero or more songs.
- Song: A song is made of a name and a position within the album. A song is associated to an album.

This example should be seen as a stripped down version of our photoblog entity set that we defined at the start of this chapter in order to focus on the actual features of each ORM rather than on the entities themselves.
Step 1: Mapping the entities

**SQLObject**

```python
from sqlobject import *

class Song(SQLObject):
    title = StringCol()
    position = IntCol()
    album = ForeignKey('Album', cascade=True)

class Album(SQLObject):
    title = StringCol()
    release_year = IntCol()
    artist = ForeignKey('Artist', cascade=True)
    songs = MultipleJoin('Song', orderBy="position")

class Artist(SQLObject):
    # Using alternateID will automatically
    # create a byName() method
    name = StringCol(alternateID=True, unique=True)
    albums = MultipleJoin('Album')
```

The first point to note is that SQLObject does not require a separate declaration of the mapping that is done within the class itself. Each class must inherit from the unobtrusive `SQLObject` class to be manageable by SQLObject and the attributes will be mapped transparently by SQLObject into the table's columns. SQLObject automatically adds an attribute `id` to hold the unique identifier of each object. This means that every table mapped by SQLObject must have a primary key.

`ForeignKey` or `MultipleJoin` are examples of how to define the relationships between entities. Note that they need the name of the class as a string and not the class object itself. This allows the declaration of relationships without the prior existence of each class within the scope of the module. In other words `Artist` and `Album` could be declared in two different modules without the problem of cross imports.

SQLObject offers a useful feature when specifying `alternateID` as a parameter in one of the class attributes. By using it, SQLObject adds a new method to the class of the form `byAttributeName` as shown in the example above. Note also how you can specify at that level the way rows must be ordered when being retrieved.
Finally bear in mind that by default SQLObject auto-commits to the database each modification made to an attribute, which can increase the network overhead as well as break the database integrity if an error occurs. To work around this behavior SQLObject offers the `set` method on an SQLObject object that performs one single `UPDATE` query for all the modifications limiting the required bandwidth. Moreover, SQLObject supports the concept of transactions, allowing us to ensure that operations are atomic to the database and can then be rolled back if an error occurs. Note that the transactions have to be explicitly requested by the developer.

**SQLAlchemy**

```python
from sqlalchemy import *

artist_table = Table('Artist', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', String(), unique=True))

song_table = Table('Song', metadata,
    Column('id', Integer, primary_key=True),
    Column('title', String()),
    Column('position', Integer),
    Column('album_id', Integer,
        ForeignKey('Album.id')))  

album_table = Table('Album', metadata,
    Column('id', Integer, primary_key=True),
    Column('title', String()),
    Column('release_year', Integer),
    Column('artist_id', Integer,
        ForeignKey('Artist.id')))  

class Artist(object):
    def __init__(self, name):
        self.id = None
        self.name = name

class Album(object):
    def __init__(self, title, release_year=0):
        self.id = None
        self.title = title
        self.release_year = release_year

class Song(object):
    def __init__(self, title, position=0):
        self.id = None
```

self.title = title
self.position = position

song_mapper = mapper(Song, song_table)
album_mapper = mapper(Album, album_table,
                      properties = {'songs': relation(song_mapper,
                                                  cascade="all, delete-orphan")
                      })
artist_mapper = mapper(Artist, artist_table,
                       properties = {'albums': relation(album_mapper,
                                                 cascade="all, delete-orphan")
                       })

SQLAlchemy uses a declarative mapping style as you can see. The first step is to express tables into their Python syntax counterpart. Then we need to declare the class our application will manipulate. Note how they don't need to inherit from an SQLAlchemy class even though they must inherit from the built-in Python object class. Eventually, we map both aspects through the mapper function, which also allows us to inform SQLAlchemy of the relationships between entities.

You will notice how the identifier of each table is explicitly declared unlike SQLObject and Dejavu. Similarly you do not specify at that level how fetched rows must be ordered as this will be specified at the query level.

Dejavu

from dejavu import Unit, UnitProperty

class Song(Unit):
    title = UnitProperty(unicode)
    position = UnitProperty(int)
    album_id = UnitProperty(int, index=True)

class Album(Unit):
    title = UnitProperty(unicode)
    release_year = UnitProperty(int)
    artist_id = UnitProperty(int, index=True)

    def songs(self):
        return self.Song()
    songs = property(songs)

    def artist(self):
        return self.Artist()
    artist = property(artist)
A Photoblog Application

```python
def on_forget(self):
    for song in self.Song():
        song.forget()

class Artist(Unit):
    name = UnitProperty(unicode)

    def albums(self):
        return self.Album()
    albums = property(albums)

    def on_forget(self):
        for album in self.Album():
            album.forget()

    Album.one_to_many('ID', Song, 'album_id')
    Artist.one_to_many('ID', Album, 'artist_id')
```

Like SQLObject, Dejavu does lots of work under the hood. Each class participating in the mapping must inherit from `Unit`. The attributes of the class represent the columns of the table. Only the relationship between the entities is done through a more declarative interface.

One difference between Dejavu and the other two is that it does not provide the cascade delete feature. This means that this has to be accomplished from the class itself by defining an `on_forget()` method and specifying what tasks should be done when deleting a unit. This might look at first sight like a drawback but offers, in fact, a fine granularity on how you propagate a cascade delete.

Step 2: Setting up the access to the database

**SQLObject**

```python
# Create a connection to a SQLite 'in memory' database
sqlhub.processConnection = connectionForURI('sqlite:///:memory:?debug=True')
```

**SQLAlchemy**

```python
# Inform SQLAlchemy of the database we will use
# A SQLite 'in memory' database
# Mapped into an engine object and bound to a high level meta data interface
engine = create_engine('sqlite:///memory:', echo=True)
metadata = BoundMetaData(engine)
```
Dejavu

# Create the global arena object
arena = dejavu.Arena()
arena.logflags = dejavu.logflags.SQL + dejavu.logflags.IO

# Add a storage to the main arena object
conf = {'Database': ':memory:'}
arena.add_store("main","sqlite", conf)

# Register units the arena will be allowed to handle
# This call must happen after the declaration of the units
# and those must be part of the current namespace
arena.register_all(globals())

Step 3: Manipulating tables

SQLObject

def create_tables():
    Album.createTable()
    Song.createTable()
    Artist.createTable()

def drop_tables():
    Song.dropTable()
    Artist.dropTable()
    Album.dropTable()

SQLAlchemy

def create_tables():
    artist_table.create(checkfirst=True)
    album_table.create(checkfirst=True)
    song_table.create(checkfirst=True)

def drop_tables():
    artist_table.drop(checkfirst=False)
    song_table.drop(checkfirst=False)
    album_table.drop(checkfirst=False)

Dejavu

def create_tables():
    arena.create_storage(Song)
    arena.create_storage(Album)
    arena.create_storage(Artist)
def drop_tables():
    arena.drop_storage(Song)
    arena.drop_storage(Album)
    arena.drop_storage(Artist)

Step 4: Loading data

SQLObject

# Create an artist
jeff_buckley = Artist(name="Jeff Buckley")

# Create an album for that artist
grace = Album(title="Grace", artist=jeff_buckley, release_year=1994)

# Add songs to that album
dream_brother = Song(title="Dream Brother", position=10, album=grace)
mojo_pin = Song(title="Mojo Pin", position=1, album=grace)
lilac_wine = Song(title="Lilac Wine", position=4, album=grace)

SQLAlchemy

session = create_session(bind_to=engine)

jeff_buckley = Artist(name="Jeff Buckley")

grace = Album(title="Grace", release_year=1994)

dream_brother = Song(title="Dream Brother", position=10)
mojo_pin = Song(title="Mojo Pin", position=1)
lilac_wine = Song(title="Lilac Wine", position=4)

grace.songs.append(dream_brother)
grace.songs.append(mojo_pin)
grace.songs.append(lilac_wine)
jeff_buckley.albums.append(grace)
session.save(jeff_buckley)
session.flush()

Note the fact that each object is created independently from the other and their relationship is fulfilled in a second step, e.g. the append() method on the grace.songs object.

In the same declarative spirit as above SQLAlchemy does not commit automatically by default to the database. Instead it delays the operation until you flush the current session of work.
Dejavu

sandbox = arena.new_sandbox()

# Create an artist unit
jeff_buckley = Artist(name="Jeff Buckley")
sandbox.memorize(jeff_buckley)

grace = Album(title="Grace", release_year=1994)
sandbox.memorize(grace)

# Add the album unit to the artist unit
jeff_buckley.add(grace)

dream_brother = Song(title="Dream Brother", position=10)
sandbox.memorize(dream_brother)

mojo_pin = Song(title="Mojo Pin", position=1)
sandbox.memorize(mojo_pin)

lilac_wine = Song(title="Lilac Wine", position=4)
sandbox.memorize(lilac_wine)

# Add each song unit to the album unit
grace.add(dream_brother)
grace.add(mojo_pin)
grace.add(lilac_wine)

sandbox.flush_all()

Dejavu provides the concept of sandboxes in which you can isolate the entities you manipulate. Also note that newly created units do not exist for their relatives until you call the `sandbox.memorize()` method, which puts the unit into the sandbox.

Like SQLAlchemy, Dejavu delays the commit operation until you explicitly call the `sandbox.flush_all()` method.

Step 5: Manipulating data

First we define a function that will take an artist and display the albums' songs.

```python
def display_info(artist):
    for album in artist.albums:
        message = "
        %s released %s in %d
        It contains the following songs:
        %s
        %s
        %s
        
        It contains the following songs:
```

A Photoblog Application

```python
for song in album.songs:
    message = message + "       %s
" % (song.title, )
print message
```

**SQLObject**

```python
# Retrieve an artist by his name
buckley = Artist.byName('Jeff Buckley')
display_info(buckley)

# Retrieve songs containing the word 'la' from the given artist
# The AND() function is provided by the SQLObject namespace
songs = Song.select(AND(Artist.q.name=="Jeff Buckley",
                        Song.q.title.contains("la")))
for song in songs:
    print "  %s" % (song.title,)

# Retrieve all songs but only display some of them
songs = Song.select()
print "Found %d songs, let's show only a few of them:" % (songs.count(), )
for song in songs[1:-1]:
    print "  %s" % (song.title,)

# Retrieve an album by its ID
album = Album.get(1)
print album.title

# Delete the album and all its dependencies
# since we have specified cascade delete
album.destroySelf()
```

**SQLAlchemy**

```python
session = create_session(bind_to=engine)

# Retrieve an artist by his name
buckley = session.query(Artist).get_by(name='Jeff Buckley')
display_info(buckley)

# Retrieve songs containing the word 'la' from the given artist
songs = session.query(Song).select(and_(artist_table.c.name=="Jeff Buckley",
                                        song_table.c.title.like("%la%")))
```
for song in songs:
    print "  %s" % (song.title,)

# Retrieve all songs but only display some of them
# Note that we specify the order by clause at this level
songs = session.query(Song).select(order_by=[Song.c.position])
print "Found %d songs, let's show only a few of them:" % (len(songs),)
for song in songs[1:-1]:
    print "  %s" % (song.title,)

# Retrieve an album by its ID
album = session.query(Album).get_by(id=1)
print album.title

# Delete the album and all its dependencies
# since we have specified cascade delete
session.delete(album)
session.flush()

Dejavu

sandbox = arena.new_sandbox()

# Retrieve an artist by his name
buckley = sandbox.Artist(name="Jeff Buckley")
display_info(buckley)

# Retrieve songs containing the word 'la' from the given artist
# We will explain in more details the concepts of Expressions
f = lambda ar, al, s: ar.name == "Jeff Buckley" and "la" in s.title

# Note how we express the composition between the units
results = sandbox.recall(Artist & Album & Song, f)
for artist, album, song in results:
    print "  %s" % (song.title,)

# Retrieve all songs but only display some of them
songs = sandbox.recall(Song)
print "Found %d songs, let's show only a few of them:" % (len(songs),)
for song in songs[1:-1]:
    print "  %s" % (song.title,)

# Retrieve an album by its ID
album = sandbox.Album(ID=1)
print album.title
Choosing an object-relational mapper is a difficult task as it is usually after using one for a while that you can really measure the impact it has on the development design and process. As previously stated, it is critical to acknowledge that ORMs do not remove the impedance mismatch between the relational and object-oriented model.

SQLObject has a low-learning curve and quite a big community, which makes it suitable for developers debuting in the use of ORM. The project is aiming at its next version, which will fix a fair number of bad design decisions made in its early life while slowly abandoning the current version.

SQLAlchemy has based its design on the Hibernate ORM from the Java world and has thus avoided lots of pitfalls that SQLObject has not. Its declarative syntax will not please every pythoner but its flexibility and good documentation have placed SQLAlchemy as a very serious candidate in the field.

Dejavu is a fairly unknown ORM and thus has a small community. It is well documented and comes with relevant example cases. Its strength resides in its capacity to move away from the underlying relational database layer by providing a very high-level interface using common Python idioms.

For instance, SQLObject and SQLAlchemy do use terms such as `table`, `column`, or `select` while Dejavu refers to `storage` and `unit` providing a better abstraction to the underlying mechanisms.

This is also true when it comes to the process of building queries. Unlike SQLObject and SQLAlchemy, which stay very close to SQL by providing a Python interface to SQL statements, Dejavu provides an interface independent of SQL. Review the section on manipulating data for an example.

These are the reasons why our photoblog application will use Dejavu instead of SQLObject or SQLAlchemy. However, keep in mind that they are all good and powerful ORMs.

### Photoblog Application Entity Modeling

First we define what we will call a `storage` module providing a simple interface to some common operations like the connection to the database.

```python
import dejavu
arena = dejavu.Arena()

from model import Photoblog, Album, Film, Photo

def connect():
```

---

conf = {'Connect': "host=localhost dbname=photoblog user=test
password=test"}
arena.add_store("main", "postgres", conf)
arena.register_all(globals())

In this case, we import the dejavu module and we create one global instance of the Arena class. The arena will be our interface between the underlying storage manager and the business logic layer.

The connect function adds a storage manager to the arena object for a PostgreSQL RDBMS and then registers all imported entities so that the arena object knows what entities it will manage. (Please refer to the Dejavu documentation for a list of supported database managers and how to declare them within the add_store() method.) Once we have this module we can start mapping entities.

### Mapping Entities

Mapping the entities is done through the following process:

- Creating a class that inherits from Unit
- Adding attributes using the UnitProperty class
- Setting up the relationship between units

**Entity: Photoblog**

```python
from dejavu import Unit, UnitProperty
from engine.database import arena
from album import Album

class Photoblog(Unit):
    name = UnitProperty(unicode)
    title = UnitProperty(unicode)

def on_forget(self):
    for album in self.Album():
        album.forget()

Photoblog.one_to_many('ID', Album, 'blog_id')
```

**Entity: Album**

```python
import datetime
from dejavu import Unit, UnitProperty
from engine.database import arena
```
A Photoblog Application

from film import Film

class Album(Unit):
    name = UnitProperty(unicode)
    title = UnitProperty(unicode)
    author = UnitProperty(unicode)
    description = UnitProperty(unicode)
    content = UnitProperty(unicode, hints={u'bytes': 0})
    created = UnitProperty(datetime.datetime)
    modified = UnitProperty(datetime.datetime)
    blog_id = UnitProperty(int, index=True)

    def on_forget(self):
        for film in self.Film():
            film.forget()

    Album.one_to_many('ID', Film, 'album_id')

Entity: Film

import datetime

from dejavu import Unit, UnitProperty
from engine.database import arena

from photo import Photo

class Film(Unit):
    name = UnitProperty(unicode)
    title = UnitProperty(unicode)
    created = UnitProperty(datetime.datetime)
    modified = UnitProperty(datetime.datetime)
    album_id = UnitProperty(int, index=True)

    def on_forget(self):
        for photo in self.Photo():
            photo.forget()

    Film.one_to_many('ID', Photo, 'film_id')

Entity: Photo

import datetime

from dejavu import Unit, UnitProperty
from engine.database import arena
class Photo(Unit):
    name = UnitProperty(unicode)
    legend = UnitProperty(unicode)
    filename = UnitProperty(unicode)
    filesize = UnitProperty(int)
    width = UnitProperty(int)
    height = UnitProperty(int)
    created = UnitProperty(datetime.datetime)
    modified = UnitProperty(datetime.datetime)
    film_id = UnitProperty(int, index=True)

Units and UnitProperties
In the previous section, we mapped our entities into units that Dejavu will manage. All our classes inherit from the Unit base class. This class does not provide much apart from automatically adding an ID property to the class, which is the reason why we do not explicitly provide one in any of our units. Nonetheless by inheriting from the Unit class you allow Dejavu to register and handle your class.

The next step is obviously to add properties to your classes via the UnitProperty class, which has the following signature:

UnitProperty(type=unicode, index=False,
             hints=None, key=None, default=None)

- The type parameter is a Python type. Dejavu takes care of translating it into the appropriate SQL equivalent type transparently.
- The index parameter indicates whether the column should be indexed by the RDBMS if it supports it.
- The hints parameter is a dictionary to help Dejavu storage managers to optimize the creation of the column. Dejavu has three built-in hints but you can provide yours if you create your own storage manager:
  - bytes: Indicates the number of bytes to be used for a unicode property, 0 meaning unlimited.
  - scale: Number of digits to the right of the decimal point in a numeric column.
  - precision: Total number of digits in a numeric column.
- The key parameter is the property canonical name.
- The default parameter indicates the default value to be used.

Properties will map into the columns of a table in the relational database.
Associating Units

Associating units is the means of giving a shape to your design. Entities are bricks, relations are the mortar.

Dejavu supports the following common relationships:

- One to one (1, 1)
- One to many (1, n)
- Many to one (n, 1)

In each case you provide the signature as follows:

```
nearClass(nearKey, farClass, farKey)
```

Therefore the relation between `Film` and `Photo` is:

```
Film.one_to_many('ID', Photo, 'film_id')
```

The `nearClass` is `Film`, the `nearKey` is `ID` (property of the `nearClass`), the `farClass` is `Photo`, and the `farKey` is `film_id` (property of the `farClass`).

Dejavu does not provide a native many-to-many relationship but this is achievable through a third unit class and a one-to-one relationship.

The Sandbox Interface

The sandbox object manages memory dedicated to units in a protected way. A sandbox is where units spend their life. There are two ways to create sandboxes:

```
box = arena.create_sandbox()
box = dejavu.Sandbox(arena)
```

The former version is the most common and is the one we will use throughout this book.

Let's review a few key methods of the sandbox interface:

- `memorize`: When you create a new instance of a unit, it exists only in memory and is separated from the storage manager. You need to call the `memorize` method to make it part of the sandbox. This will also set the ID of the unit. Additionally this will reserve a place in the underlying database by issuing an `INSERT INTO` SQL statement.

- `forget`: In order to tell the store manager to stop managing a unit you must call the `forget` method. This will delete it from the sandbox and from the storage manager.
repress: In some cases you may wish to clear the unit from the sandbox but not from the store manager. In such cases you should use the repress method.

recall, xrecall: These two methods allow you to retrieve units based on filters (as we will explain in the section Querying Units). The difference between recall and xrecall is that the latter yields results in an iterating fashion whereas the former loads everything into a list at once.

unit: Both the previous methods are powerful ones to retrieve a set of data but they can be heavy when you simply look for one unit based on a value of its properties. This is what the unit method provides.

flush_all: Once you have manipulated your units you have to call flush_all in order to make those changes to the physical back end.

As you can see, the interface provided by the Sandbox class is quite simple, straightforward, and yet powerful as the next section will demonstrate.

### Querying Units

We have seen so far how to map our entities into units as well as how to manipulate those units. This section will explain in detail how to query the storage manager for units based on criteria.

Within Dejavu querying is done through an Expression instance. The Expression class is a filter for units. Let's take an example to explain how this works.

```python
# Search for all photographs with a width superior to 300 pixels
f = lambda x: x.width > 300
box.recall(Photo, f)
```

The first step is to create a function that returns a bool. That function is usually a lambda as there is no need to pollute the Python namespace with meaningless names. Then we pass it to one of the sandbox methods such as recall or xrecall, which will create a logic.Expression instance and apply it.

Expressions show their value when filtering against complex queries such as the one involving JOIN. For example, if you want to join between units you would use Python operators between the units themselves.

```python
# Search for all photographs of width superior to 300 pixels
# within albums created by Sylvain
box.recall(Album & Photo, lambda a, p: a.author == "Sylvain" and p.width > 300)
```
A Photoblog Application

As you can see, the first parameter of the method takes an aggregation of the unit classes that will take part in the join. Dejavu gives you the opportunity of using Python operators to declare aggregation between units.

When composing between units, the order is important while constructing the filter function. In the previous example the lambda function parameters will match the order of the composed units. This behavior is mirrored by the result returned by the recall() method, which will provide a list of Album and Photo items.

Below are the Dejavu representations of SQL JOINs.

<table>
<thead>
<tr>
<th>Join Type</th>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inner</td>
<td>&amp; or +</td>
<td>All related pairs of both classes will be returned.</td>
</tr>
<tr>
<td>Left Outer</td>
<td>&lt;&lt;</td>
<td>All related pairs of both classes will be returned. In addition, if any Unit in class1 has no match in class2, we return a single row with Unit1 and a null Unit (a Unit, all of whose properties are None).</td>
</tr>
<tr>
<td>Right Outer</td>
<td>&gt;&gt;</td>
<td>All related pairs of both classes will be returned. In addition, if any Unit in class2 has no match in class1, we return a single row with a null Unit (a Unit, all of whose properties are None) and Unit2.</td>
</tr>
</tbody>
</table>

There is no limitation from Dejavu regarding the aggregation you build. For instance you can write:

(Film << Album) & Photo

Extending the Data Access Layer

In the previous sections, we have defined the mapping between our entities and the classes our application will manipulate. As they stand these classes are not very useful; in this section we will see how to extend them to provide more functionalities. To keep this section concise, we will only discuss the Album class.

Methods to be added to the Album class:

```python
def films(self):
    """Returns all the attached films
    album = Album()
    ...
    for film in album.films:
        ...
```
return self.Film()
films = property(films)

def get_all(cls):
    """Returns all the existing albums
    
    for album in Album.albums:
        ...
    """
    sandbox = arena.new_sandbox()
    return sandbox.recall(Album)
albums = classmethod(get_all)

def fetch(cls, id):
    """Fetch one album by id"
    sandbox = arena.new_sandbox()
    return sandbox.unit(Album, ID=int(id))
fetch = classmethod(fetch)

def fetch_range(cls, start, end):
    """Fetch a range of albums which ID falls into the specified range.
    
    # This could return up to 5 albums
    albums = Album.fetch_range(4, 9)
    for album in albums:
        ...
    """
    sandbox = arena.new_sandbox()
    # dejavu's views change the capacity of dejavu to perform operations on a Unit
    # here we create a view of the Album unit so that only the created and ID properties appear in the result of the view. A view yields values not units unlike recall or xrecall.
    v = list(sandbox.view(Album, ['created', 'ID']))
    v.sort()
    size = len(v)
    if end > size and start >= size:
A Photoblog Application

    return None
    elif end > size and start < size:
        end = size
        # row[0] is the 'created' property value
        # row[1] is the 'ID' property value
        targets = [row[1] for row in v[start:end]]
        return sandbox.recall(Album, lambda x: x.ID in targets)
fetch_range = classmethod(fetch_range)

def create(self, photoblog, name, title, slug, author, description, content):
    """Instanciates the Album,
    adds it to the passed photoblog and
    persists the changes into the database""
    sandbox = photoblog.sandbox

    self.name = name
    self.title = title
    self.author = author
    self.description = description
    self.content = content
    self.created = datetime.datetime.now().replace(microsecond=0)
    self.modified = album.created
    self.blog_id = photoblog.ID
    sandbox.memorize(self)
    photoblog.add(self)
    sandbox.flush_all()

    def update(self, name, title, slug, author, description, content):
        """Updates the attributes of an album and
        persists the changes into the storage""
        self.title = title
        self.slug = slug
        self.author = author
        self.description = description
        self.content = content
        self.modified = datetime.datetime.now().replace(microsecond=0)
        self.sandbox.flush_all()
def delete(self):
    """Delete the album from the storage"""
    self.sandbox.forget(album)

def to_dict(self):
    """Return an album as a Python dictionary"""
    return {'id': self.ID,
            'uuid': self.uuid,
            'title': self.title,
            'author': self.author,
            'description': self.description,
            'content': self.content,
            'created': self.created.strftime('%d %b. %Y, %H:%M'),
            'modified': self.modified.strftime('%d %b. %Y, %H:%M')}

def to_json(self):
    """JSONify an album properties"""
    return simplejson.dumps(self.to_dict())

As you can see, the Album class now contains enough methods to allow manipulation of Album instances. The other photoblog entities share the same idea and will provide similar interfaces.

Summary
This chapter has introduced the backbone of our photoblog application through the description of its entities and how they are mapped in their Python counterparts. Our next chapter will review how to manipulate those entities from our CherryPy handlers to build the external interface of our application.
**Where to buy this book**

You can buy CherryPy from the Packt Publishing website:


Free shipping to the US, UK, Europe, Australia, New Zealand and India.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.