Laboratorio di Basi di Dati e Programmazione Web

Programmazione web
DB access e framework MVC

Stefano Zacchiroli
zack@pps.jussieu.fr

Programmazione Web

DB abstraction

Eterogeneità dei DBMS

- l'eterogeneità dei DBMS è problematica per gli sviluppatori
  - l'estremo (da evitare se possibile) è scrivere backend per ogni DBMS che si intende supportare
    - caso non infrequente, ma solo per grossi applicativi che necessitano di feature avanzate
- eterogeneità da considerare:
  1. eterogeneità di feature
     - induce eterogeneità a livello di linguaggio DML/DDL/DCL
  2. eterogeneità di accesso
     - diversi client (per shell script)
     - diverse librerie (per lo sviluppo in vari linguaggi)

Astrazione sui DBMS

- soluzione informatica proverbiale:

  « just add an extra level of indirection »

- per l'eterogeneità di feature (1.)
  - soluzioni solo parziali :-(
  - gli standard non sono sufficienti: sono implementati parzialmente e male nei DBMS
  - scrivere query nell'intersezione dei linguaggi dei DBMS che si vogliono supportare
  - scrivere in linguaggi intermedi (non SQL) che vengono poi “compilati” verso DBMS specifici
    - e.g. ORM (Object Relational Mapper), li vedremo in seguito

Astrazione sui DBMS (cont.)

- per l'eterogeneità di accesso (2.)
  - soluzioni “accettabili”
  - ODBC (language independent, ma bloated)
  - librerie language-specific di astrazione sui DBMS
    - una API comune
      - definita con processi simili alla standardizzazione, con la descrizione di feature opzionali e obbligatorie
    - una implementazione per ogni DBMS supportato
    - codice scritto per tali librerie è solitamente portabile da un DBMS ad un altro cambiando “poco”
      - i.e. una stringa o l'import di un modulo
    - esempi notevoli:
      - DBI (Perl), DB-API 2.0 (Python), JDBC (Java), ...

Python DB API in a nutshell

- Python DB API è una API (2.0) per accedere uniformemente a DBMS etoregenei
  - Standardizzata secondo le procedura PEP (Python Enhancement Proposal)
    - in particolare Python DB API è PEP249: http://www.python.org/dev/peps/pep-0249/
- Feature
  - Connection management
  - Cursori
  - Transazioni
  - Query preparation
- Backend disponibili:
  - MySQL, Postgres, SQLite (nella stdlib), Oracle, ODBC, Firebird, Ingres, DB2, ...

DB API – architettura

- Python DB API vs Perl's DBI
  - A differenza di DBI, Python DB è una API
    - Ovvero un insieme di convenzioni (di naming, di typing, di semantica) su metodi
    - I moduli compatibili con Python DB si possono usare uniformemente, semplicemente cambiando il nome del modulo
- Pro:
  - Python DB API non aggiunge overhead dovuto all'indirezione
  - I moduli che implementano i backend non hanno dipendenze extra
  - Esempio di “duck-typing”: « *it it walks like a duck and quacks like a duck, I would call it a duck* »
- Cons:
  - La API deve essere abbastanza generale da adattarsi a tutti i backend (che hanno partecipato alla sua standardizzazione)
    - Convenzioni diverse, sebbene dichiarate, di parameter passing
    - Connection management è database-specific

DB API – connection management

- Main concept: *connection object*
- Globals
  - `connect(...)`
    - Si collega al database e restituisce un connection object
    - Parameters sono DBMS-specific
      - Tipicamente connection string, o kwargs (database, host, port, user, password, …)
      - RTFM !
  - `apilevel` = “2.0”
  - `paramstyle`
    - Tipologia di marker utilizzati per riempire query preparate
    - Stay tuned …
- Classe `Error`
  - Base class, per tutti le eccezioni DBMS-related
  - Sub-class hierarchy:
    - `Error`
      - `InterfaceError`
      - `DatabaseError`
        - `DataError`
        - `OperationalError`
        - `IntegrityError`
        - `InternalError`
        - `ProgrammingError`
        - `NotSupportedError`

DB API – connection example

```
import sys
import psycopg2 as db   # or import sqlite3, or …
# note: do not use psycopg2 all over the code!
try:
      conn = db.connect(
                database=”test”,
                host=”localhost”,  # or via FS socket?
                user=”myuser”,
                password=”secret”)
except db.DbError, exn:
      print “Can't connect to database: %s” % exn
      sys.exit(3)
```

DB API – connection object

- I connection object offrono i metodi
  - `close()`
  - `commit()`
  - `rollback()`
  - `cursor()`
    - L'interazione principale con il DBMS, via DQL, avviene tramite cursor object
    - Il metodo cursor restituisce un cursor object *fresh*
    - Se il DBMS non supporta nativamente il concetto di cursore (e.g., SQLite), l'implementazione DB API ne emula le funzionalità

DB API – cursor objects

- Metodi e attributi dei cursor object
  - `execute(query, [params])`
    - Prepara ed esegue una query. I risultati sono accessibili con altri metodi; esiste un concetto di “risultato corrente”
  - `rowcount` (attribute) numero di righe nel risultato corrente
  - `fetchone()` ritorna la prossima riga come sequenza
    - Ritorna `None` se non ci sono più righe disponibili
  - `fetchmany(size=`*n*`)` ritorna una lista lunga n di righe
  - `fetchall()`
  - `close()` dealloca il cursore, libera risorse (DBMS-side)
  - `description` (attribute) DB schema inspection

DB API – query example

```
# … continued from previous example
curs = conn.cursor()
try:
   curs.execute(“””CREATE TABLE atable (
                    name char(8),
                    value float)”””)
   curs.execute(“INSERT INTO atable VALUES ('one', 1.0)”)
   curs.execute(“INSERT INTO atable VALUES ('two', 2.0)”)
   conn.commit()   # general case: transactions
except DbError, exn:
   print “cannot execute some query: %s” % exn
   # conn.rollback()
```

DB API – query example (cont.)

```
# … continued from previous example
try:
   curs.execute(“SELECT * FROM atable”)
   tuple = curs.fetchone()
   while tuple: # cursors don't implement __iter__ :-(
      for field in tuple:
         print field, “ “,
      print '----'
except DbError, exn:
   print “cannot execute some query: %s” % exn
```

Query reuse

- Necessità frequente: ri-effettuare query SQL simili
  - Rationale: SQL non è Turing completo, i “loop” vanno gestiti esternamente
- È particolarmente inefficiente ricreare le query SQL ad ogni iterazione
  - Garbage collection delle query (viste come stringhe)
  - Impossibilità di riutilizzare le ottimizzazioni lato DBMS
- Molti layer di astrazione su DBMS, DB API incluso, permettono di fattorizzare le query
  - Distinzione tra *preparazione* ed *esecuzione*
  - Il DBMS, se lo supporta, *può* pre-compilare le query: risparmia parsing, rete e può riutilizzare le ottimizzazioni

DB API – query placeholders

- L'uso è semplice
  - *Placeholder* sintattici nelle query a prepare-time
  - Riempimento (*filling*) degli stessi a execution-time
- DB API “offre” diversi tipi di placeholder, quelli usati da una particolare implementazione dipendono da `paramstyle`

```
'qmark'    Question mark style, e.g. '...WHERE name=?'
'numeric'  Numeric, positional style, e.g. '..WHERE name=:1'
'named'    Named style, e.g. '...WHERE name=:name'
'format'   ANSI C printf format codes, e.g. '..WHERE name=%s'
'pyformat' Python extended format codes, e.g.'..WHERE name=%(name)s'
```

- Nota
  - “prepare” può essere emulata a livello di implementazione se non supportato dal DBMS
  - Scrivere codice DB API portabile richiede gestire i diversi tipi di paramstyle :-(

DB API – query reuse example

```
# … same database as before
# assume paramstyle = “qmark”

q = “INSERT INTO atable VALUES (?, ?)”
# note: no quotes around markers, they are added as
#       (and if) needed

curs.execute(q, (“one”, 1.0));     # q's memoization
curs.execute(q, (“two”, 2.0));
curs.executemany(q, [(“three”, 3.0), (“four”, 4.0)])

conn.commit()
```

Esercizi

- Implementare una sorta di “ls -R” per database usando Python e DB API
  - Dato i parametri di connessione ad un database ed il nome di una tabella, lo script lista tutte le sue tuple
  - Per ogni tupla che ha un campo foreign key su un'altra tabella, inizia il listing ricorsivo su quella tabella; tale listing include soltanto le righe che corrispondono al valore della foreign key per la tupla corrente
- Effettuare un porting a database del semplice wiki realizzato in precedenza
  - Testarlo durante il debugging con sqlite3, poi portarlo a mysql/postgresql per il deployment finale

Programmazione Web

Object Relational Mapping

ORM – Object Relational Mapping

- Reprisal sull'astrazione (deja-vu?):
  - soluzione (parziale) per l'eterogeneità di feature

    1. scrivere query (SQL) nell'intersezione dei DBMS che si vogliono supportare *oppure*
    2. scrivere in *linguaggi intermedi* (non SQL) che vengono poi "compilati" verso DBMS specifici

    Perl DBI e Python DB API 2.0 sono un esempio di (1.)
- Caso particolare di (2.):
  - il *linguaggio intermedio* coincide con il (oppure è un sottoinsieme del) linguaggio di sviluppo dell'applicazione

ORM (cont.)

- ORM è una tecnica di programmazione per convertire dati tra il sistema di tipi dei *linguaggi ad oggetti* e quello dei DBMS
- concetti chiave di ORM:
  - restrizione al paradigma di *programmazione ad oggetti* (non applicabile ad altri paradigmi)
  - inferire *entità* e *relazioni* (dell'universo relazionale) da *classi* e *attributi* (dell'universo della programmazione ad oggetti)
- il linguaggio intermedio nel caso di ORM è il linguaggio di definizione delle classi

ORM (cont.)

- risultato per il programmatore
  - un database “virtuale” di oggetti che usa un DBMS come data storage
  - molte delle azioni esprimibili in SQL sono mascherate da azioni su oggetti/classi
    - creazione di tabelle → definizione di classe (runtime)
    - inserimento di tuple → istanziazione
    - proiezione/update → accesso ad attributi
    - join → definizione di classe + accesso ad attributi
    - cancellazione → uso di class method
    - selezione di tuple → uso di class method

[Intermezzo: *DBMS]

(i.e.: tantissime buzzword in 2 slide)

- bestiario:
  - RDBMS
    - ormai li conosciamo :-)
  - ORDBMS (poco accordo sul termine...)
    - DBMS estensibile che permette l'integrazione di tipi di dati e metodi forniti dall'utente
    - Non c'entra nulla con la programmazione OO
  - OODBMS (Object-Oriented DBMS)
    - DBMS in cui l'informazione è rappresentata come istanze (nel senso della programmazione ad oggetti)
    - Non c'entra nulla con gli ORM ...

[Intermezzo: *DBMS (cont.)]

- in un OODBMS il programmatore sviluppa ad oggetti e stabilisce quali oggetti siano *persistenti*
  - i.e. quali vivono nel data storage del ODBMS
- esempi di ODBMS
  - db4o (database for objects), per Java e .NET
  - Zope Object Database (ZODB), per Python
- una implementazione di ORM *non è* un OODBMS. È piuttosto un OODBMS “virtuale”, in quanto l'astrazione è la stessa, ma il data storage è un RDBMS, per nulla OO-specific

ORM (cont.)

- effetti collaterali:
  - astrazione sul DBMS: essendo il linguaggio di definizione degli oggetti scorrelato dal DBMS, il mapping verso il mondo relazionale può essere istanziato più volte per diversi DBMS
  - diventa inoltre possibile (grazie al maggiore livello di astrazione) implementare a livello ORM feature non disponibili nel DBMS sottostante
    - e.g. integrità referenziale, trigger, ...
    - case study: SQLObject implementa l'integrità referenziale per SQLite

ORM (cont.)

- ORM è una *tecnica* di programmazione language-neutral ...
- … le sue *implementazioni* sono librerie language-specific
- idee comuni a molte implementazioni
  - classi specifiche da cui ereditare per stabilire un mapping tra una classe ed una tabella del DBMS
  - uso dell'introspezione
  - convenzioni su nomi / metodi di classe per definire campi di una tabella e join

SQLObject

- SQLObject è un diffuso ORM open-source per il linguaggio di programmazione Python
- feature
  - database independence
    - MySQL, Postgres, SQLite, Firebird, MS SQL Server ...
  - non tenta di “nascondere” completamente il DBMS
  - sqlbuilder: astrazione ad AST per costruire query SQL
    - mappabili anche verso DBMS non-SQL
- success stories
  - framework web: Turbogears, Pylons, …
- concorrenti
  - SQL Alchemy, Storm, Axiom
- http://www.sqlobject.org

SQLObject – dichiarazioni

```
from sqlobject import *

sqlhub.processConnection = \
  connectionForURI('sqlite:/:memory:')


class Person(SQLObject):
    fname = StringCol()
    mi = StringCol(length=1, default=None)
    lname = StringCol()
    ...


Person.createTable()
```

SQLObject – SQL

```
class Person(SQLObject):
    fname = StringCol()
    mi = StringCol(length=1, default=None)
    lname = StringCol()
```

```
CREATE TABLE person (          --- sintassi: SQLite / MySQL
    id INT PRIMARY KEY AUTO_INCREMENT,
    fname TEXT,
    mi CHAR(1),
    lname TEXT
);
```

SQLObject – tipi di colonne

- alcuni tipi di colonne supportate e tipi equivalenti Python
  - BLOBCol (string), BoolCol (boolean), DecimalCol / CurrencyCol (float), StringCol / UnicodeCol (string)
  - DateCol / DateTimeCol (datetime.{date/datetime})
  - EnumCol (string), SetCol (set)
- il mapping automatico tra tipi di colonne e tipi del linguaggio rimuove la necessità di marshalling/unmarshalling

SQLObject – INSERT

```
>>> p = Person(fname="John", lname="Doe")
>>> p
<Person 1 fname='John' mi=None lname='Doe'>
>>> p.fname
'John'
>>> p.mi = 'Q'
>>> p2 = Person.get(1)
>>> p2
<Person 1 fname='John' mi='Q' lname='Doe'>
>>> p is p2
True
```

SQLObject – SELECT

```
>>> peeps = Person.select(Person.q.fname=="John")
>>> list(peeps)
[<Person 1 fname='John' mi='Q' lname='Doe'>]
peeps = Person.select(
            AND(Address.q.personID == Person.q.id,
                Address.q.zip.startswith('504')))
...
peeps = Person.selectBy(firstName="John", lastName="Doe")
...
bob = User.byUsername('bob')
# richiede “alternateID=True” sul campo username
```

SQLObject – relazioni 1-*

```
class Address(SQLObject):
    street = StringCol()
    city = StringCol()
    state = StringCol(length=2)
    zip = StringCol(length=9)
    person = ForeignKey('Person')

class Person(SQLObject):
    addresses = MultipleJoin('Address')

# l'accesso all'attributo “addresses” di un oggetto
# Person restituisce una lista di oggetti di tipo Address
```

SQLObject – relazioni *-*

```
class Utente(SQLObject):
    nome = StringCol(alternateID=True, length=20)
    ruoli = RelatedJoin('Ruolo')

class Ruolo(SQLObject):
    nome = StringCol(alternateID=True, length=20)
    utenti = RelatedJoin('Utente')

# la tabella extra per il join viene creata implicitamente
# da SQLObject e non ha alcuna classe corrispondente
```

Esercizi

- Effettuare il porting a SQLObject degli script di accesso sviluppati in precedenza
- Implementare in SQLObject il data model per un applicativo web a vostra scelta, alcuni suggerimenti
  - BTS (Bug Tracking System)
  - Screenshot manager
  - Wiki
  - CMS

Programmazione Web

Web framework Model-View-Controller

Some stuff from:

- the 20 minute wiki tutorial http://docs.turbogears.org/1.0/Wiki20/Page1
- Crhistopher Arndt's talk at RuPy conference 2007 http://chrisarndt.de/talks/rupy/

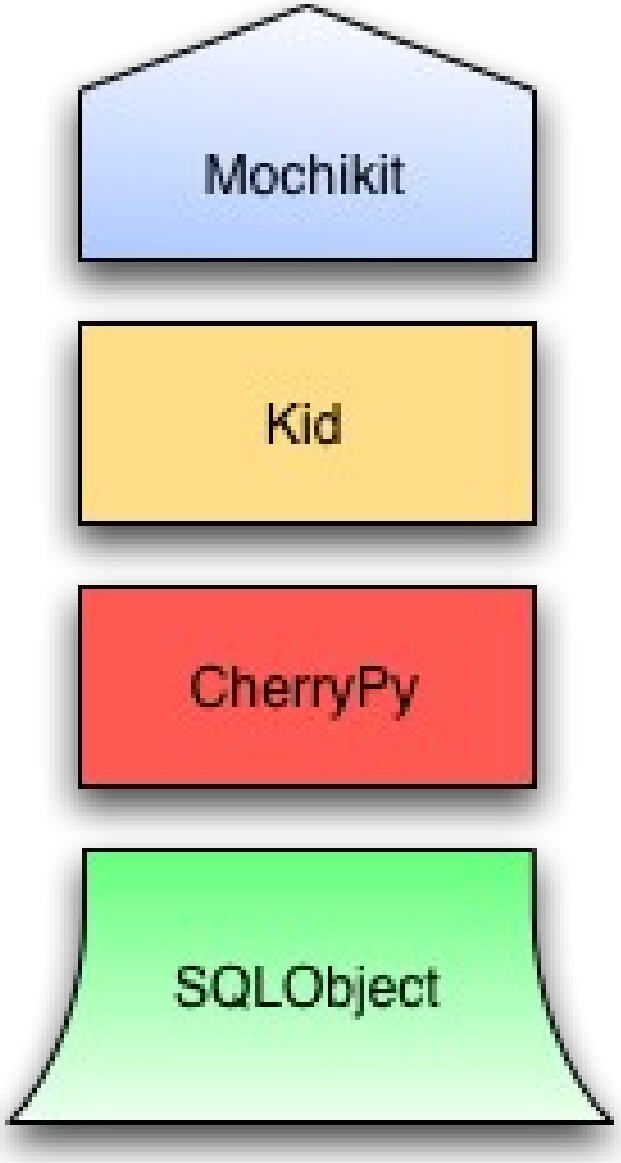
What is TurboGears?

- a Python web framework
  - comparable to Django and Ruby on Rails (the latter Ruby-based)
  - Open Source (MIT license)
  - still young (1st public version autumn 2005)
    - 2.0 got released May 2009
    - current version: 2.1 beta
  - buzzword compliant: MVC, REST, AJAX

What can it be used for?

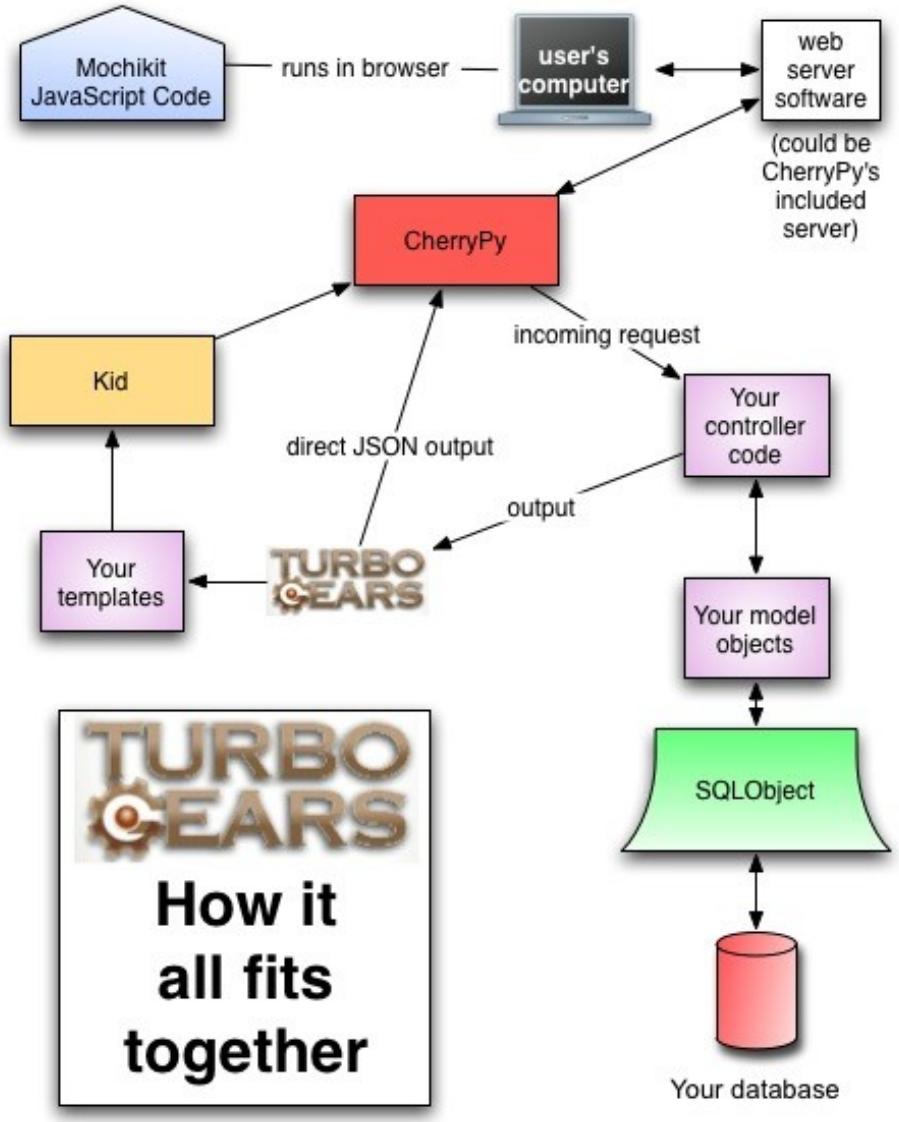
- “classic” web (1.0/2.0/whatever) apps ...
- http://docs.turbogears.org/1.0/SitesUsingTurboGears
  - aggregators
  - blogs
  - social neworking
  - ...
- ... i.e. database-frontends on the web!

Which components?



- philosophy: reuse
  - db abstraction
  - application server
  - template engine
  - javascript
- other bits:
  - formencode, nose, simplejson
- Replacement components
  - Kid → Genshi
  - SQLObject → SQL Alchemy

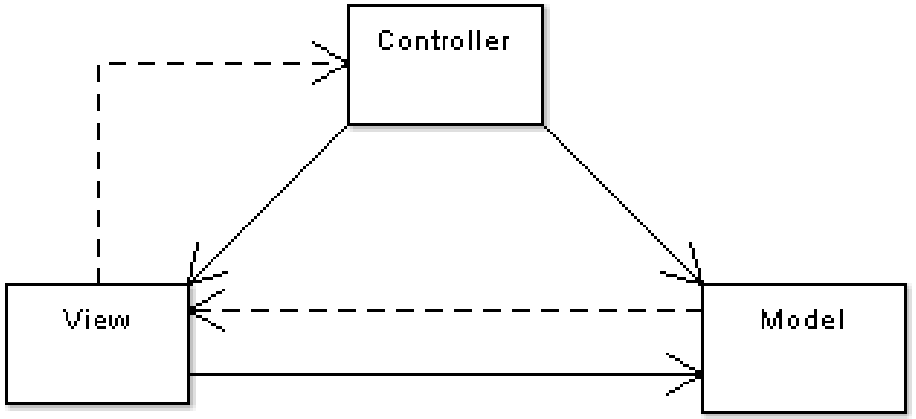
Putting it all together



- The big picture of component interactions
  - Figure from http://docs.turbogears.org

MVC: Model / View / Controller

- Buzzword ... but
  - helps separating concerns
- Encourage writing reusable components
- On the web:
  - db / template / data manipulation



- Model
  - *Represent* the data the application is working with
- Controller (in the web ...)
  - Map URL's to the model's API
  - Invoke API's methods
  - Associate templates to methods' results
- View
  - Invoke templates to render results

Recipe

1. Scaffold
   - (i.e. tg-admin quickstart)
2. Code the model
3. Create the DB
4. Fill DB with sample data
5. Design your URLs
6. Write controller methods
7. Write templates
8. Bells and whistles
   - CSS / JavaScript
9. Deploy
10. (Shuffle and iterate over and over again)

Scaffold

- create a skeleton project from scratch

```
tg-admin quickstart
Enter project name: Wiki 20
Enter package name [wiki20]: wiki20
Do you need Identity (usernames/passwords) in
this project? [no] no
```

… output …

```
cd Wiki-20
```

tg-admin

- `tg-admin` subcommands (overview)
  - `quickstart` → initial scaffolding
  - `sql` → ORM ↔ DB interaction
    - several sub-commands ...
  - `info` → Python package installation info
  - `i18n` → internationalization / localization support
  - `shell` → customized Python interpreter to play with model instances
    - Ask for DB change saving upon exit
    - Used for batch changes
  - `toolbox` → web based toolbox (poor's men `shell`)

Now serving ...

- let's test what we have

```
python start-wiki20.py

# or ./start-wiki20.py on *nix
```

- now just visit http://localhost:8080/
  - logs on stdout
  - try, for an error, http://localhost:8080/foo
- easy?
  - a working (yet useless) web app
- note: automatic reloading upon save (on the controller)
  - beware of the “5 second rules”

Code the model

- first of all we need a wiki page model
- edit wiki20/model.py

```
class Page(SQLObject):
    pagename = UnicodeCol(alternateID=True,
                          length=30)
    data = UnicodeCol()
```

- Highlights of directory layout
  - wiki20/model.py → model (SQLObject)
  - wiki20/controllers.py → controller, URL map (CherryPy)
  - wiki20/templates/*.kid → views, templates (Kid)

Create the DB

- database configuration in dev.cfg
  - per default relies on SQLite
- create the table (from the model) without a bit of SQL

```
tg-admin sql create
```

`tg-admin sql`

- `tg-admin sql` subcommands (overview):
  - `create` → exec CREATE statements
  - `sql` → show CREATE statements
  - `drop` → exec DROP statements
  - `status` → DB status wrt model
  - `list` → list known classes in the model
  - Versioning: record / upgrade
- Reference
  - just a wrapper around `sqlobject-admin`
    http://sqlobject.org/sqlobject-admin.html

Create a template

- create a template

```
cp wiki20/templates/welcome.kid
wiki20/templates/page.kid
```

- edit wiki20/templates/page.kid, bits:

```
<title> ${page.pagename} - 20 Minute Wiki </title>
...
<div class="main_content">
<div style="float:right; width: 10em">
 Viewing <span py:replace="page.pagename">Page Name Goes
Here</span>
 <br/>
 You can return to the <a href="/">FrontPage</a>.
</div>
...
<div py:replace="XML(data)">Page text goes here.</div>
```

Kid templating

- Kid templates are real XHTML documents
  - Kid directives are given via attributes in the separate “py:” namespace
  - Templates can be opened directly in browsers …
  - … and edited directly via web authoring apps
    - e.g. Dreamweaver for the non-dev web designers
- Just be careful
  - provide “sample” data in the template:

```
<div py:replace="XML(data)">Page text goes here.</div>
```

Kid templates – basics

```
<?xml version='1.0' encoding='utf-8'?>
<?python              <!-- Python snippet processing instruction -->
  import time          <!-- … don't do that!, rather: pass data -->
  title = "A Kid Template"
?>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:py="http://purl.org/kid/ns#"> <!-- kid's namespace -->
  <head>
    <title py:content="title">      <!-- replace content with -->
      PAGE TITLE
    </title>
  </head>
  <body>                          <!-- expression interpolation -->
    <p> The current time is ${time.strftime('%C %c')}. </p>
  </body>
</html>        <!-- want to batch test?    $ kid template.kid -->
```

Kid templates – basics (cont.)

```
<?python
  title = "A Kid Test Document"
  fruits = ["apple", "orange", "kiwi", "M&M"]
  from platform import system
?>
<html xmlns:py="http://purl.org/kid/ns#">
  <body>
    <p>These are some of my favorite fruits:</p>
    <ul>
      <li py:for="fruit in fruits">   <!-- for-each looping -->
        I like ${fruit}s
      </li>
    </ul>
    <p py:if="system() == 'Linux'">   <!-- conditional -->
      Good for you!
    </p> </body> </html>
```

Kid templates – basics (cont.)

```
<?python
  x = 10
?>
<html xmlns:py="http://purl.org/kid/ns#">
  <body>
    <!-- content replacement … -->
    <p> <span py:content="x"> … </span> </p>

    <!-- … vs element replacement -->
    <p> <span py:replace="x">...</span> </p>

    <!--! non-serialized comment -->
  </body>
</html>
```

Kid templates – basics (cont.)

```
<?python
def hello():
    return "<hello>world</hello>"
?>
<html xmlns:py="http://purl.org/kid/ns#">
  <body>
    <p>Defeating interpolation: $${gotcha!}</p>

    <p>${hello()}</p>
      <!-- the above will be escaped, e.g: &lt; … -->

    <p>${XML(hello())}</p>
      <!-- the above is, probably, what we want →
  </body>
</html>
```

Kid templates – basics (cont.)

- And many more:
  - Named template definition / invocation
  - XSLT/XPath-like pattern matching
  - Template inheritance
  - Layout templates
- Reference: concise presentation of the Kid language available at http://www.kid-templating.org/language.html

Write a controller

- in wiki20/controllers.py

```
import turbogears
from turbogears import controllers, expose
from wiki20.model import Page
from docutils.core import publish_parts

class Root(controllers.RootController):
    @expose(template="wiki20.templates.page")   #1
    def index(self , pagename="FrontPage"):     #2
        page = Page.byPagename(pagename)        #3
        content = publish_parts(page.data,
            writer_name="html")['html_body']    #4
        return dict(data=content, page=page)    #5
```

CherryPy

“ a pythonic, object-oriented HTTP framework ”

- A set of libraries for structuring web apps as “normal” OO apps: sets of *classes* and *methods*
- Basic principles:
  1. URL paths are mapped to methods
  2. “CGI” query strings are decoded and passed as keyword arguments
- Other features
  - WSGI-compatible
    - Deployment as CGI, standalone HTTP, mod_python, …
    - PEP333 http://www.python.org/dev/peps/pep-0333/
- http://www.cherrypy.org/

CherryPy – basics

```
import cherrypy

class WelcomePage:
    @cherrypy.expose
    def greetUser(self, name=None):

    # GET/POST variables as keyword
    # arguments, uniformly.
    #
    # Default parameter should be given.

        if name: # "name" was passed
            # Greet the user!
            return "Hey %s, what's up?" % \
                name
        else
            return 'Please enter your ' \
             'name <a href="./">here</a>.'
```

```
    @cherrypy.expose
    def index(self):
        # Ask for the user's name.
        return '''
          <form action="greetUser"
                method="GET">
          What is your name?
          <input type="text"
                 name="name" />
          <input type="submit" />
          </form>'''

# end of WelcomePage

if __name__ == '__main__':
    cherrypy.root = WelcomePage()
    cherrypy.server.start()
```

#
http://www.cherrypy.org/browser/trunk/cherrypy/tutorial/tut03_get_and_post.py

CherryPy – basics (cont.)

- URL → method mapping, rules:
  1. Break URLs at steps
     - http://host/foo/bar/baz/… → foo, bar, baz, …
  2. Start from the root class (given to CherryPy at bootstrap)
  3. If the step list is empty → invoke `index()`
  4. Lookup the (current) first step on the (current) class, if it is found:
     - If it is an attribute → continue the lookup on it using the remaining steps
     - If it is a method → invoke it and do the same
  5. If it is not found: invoke `default()`
  6. Anyway, explode query string and pass the result as keyword arguments
- Enable implementing REST architecture non-intrusively

Add sample data

- What will happen in visiting our wiki right now?
  - .... error!
  - SQLObject backtrace, FrontPage not found
- sample data can be easily provided using

```
tg-admin toolbox
```

- visit http://localhost:7654
  - choose “CatWalk”
  - and add a “FrontPage” page
- More programmatic way of adding instances

```
$ tg-admin shell
>>> p = Page(pagename="TestPage",
             data="Nothing to see here. Go away.")
```

Add page editing

- wiki20/templates/edit.kid, bits:

```
...
<form action="save" method="post">
 <input type="hidden" name="pagename"
        value="${page.pagename}" />
 <textarea name="data" py:content="page.data"
           rows="10" cols="60" />
 <input type="submit" name="submit" value="Save" />
</form>
...
```

- ... and its controllers.py method:

```
@expose("wiki20.templates.edit")
def edit(self, pagename):
    page = Page.byPagename(pagename)
    return dict(page=page)
```

Add page editing (cont.)

- the link in wiki20/templates/page.kid:

```
...
<div py:replace="XML(data)">Page text goes here.</div>
<p><a href="${tg.url('/edit',
      pagename=page.pagename)}">Edit this page</a>
</p>
...
```

- save action backend (controller method)

```
@expose()
def save(self, pagename, data, submit):
    page = Page.byPagename(pagename)
    page.data = data
    turbogears.flash("Changes saved!")
    raise turbogears.redirect("/", pagename=pagename)
```

Add page listing

- wiki20/templates/pagelist.kid

```
<h1>All Of Your Pages</h1>
<ul>
 <li py:for="pagename in pages">
  <a href="${tg.url('/' + pagename)}"
     py:content="pagename">Page Name Here.</a>
 </li>
</ul>
```

- controller method:

```
@expose("wiki20.templates.pagelist")
def pagelist(self):
    pages = [ page.pagename for page in
              Page.select(orderBy=Page.q.pagename)]
    return dict(pages=pages)
```

You want AJAX? We got AJAX!

- let's AJAX-ify page listing
- step 1: expose the data in an JavaScript friendly method: JSON
- @expose pagelist via JSON too

```
@expose("wiki20.templates.pagelist")
@expose("json")
def pagelist(self):
   ...
```

- visit: http://localhost:8080/pagelist?tg_format=json

We got AJAX! (cont.)

- MochiKit: client-side JavaScript framework
- add to wiki20/config/app.cfg:

```
tg.include_widgets = ['turbogears.mochikit']
```

  - (then restart the app)
- add id-s to what we want to change client-side later on, in master.kid:

```
<p>View the <a id="pagelist"
                href="${tg.url('/pagelist')}">
    complete list of pages.</a>
</p>
<div id="pagelist_results"></div>
```

We got AJAX! (cont.)

- JavaScript show time, still in master.kid:

```
<script type="text/javascript">
addLoadEvent(function(){
connect($('pagelist'),'onclick', function (e) {
  e.preventDefault();
  var d = loadJSONDoc("${std.url('/pagelist',
                          tg_format='json')}");
  d.addCallback(showPageList); });
});
</script>
```

- i.e. connect an onclick callback “showPageList”

We got AJAX! (cont.)

- JavaScript show time, here is the callback:

```
<script type="text/javascript">
function showPageList(result) {
    var currentpagelist = UL(null,
       map(row_display, result["pages"]));
    replaceChildNodes("pagelist_results",
       currentpagelist);
}
function row_display(pagename) {
    return LI(null, A({"href" :
      "${std.url('/')}" + pagename}, pagename))
}
</script>
```

Esercizi

- Implementate, nel framework TurboGears, l'applicativo web del quale è stato sviluppato il data model in precedenza (BTS, …)
  - Ignorate per il momento le problematiche di autorizzazione / autenticazione (i.e., l'accesso è sempre in modalità “anonima”)

User management in TurboGears

- User management in TG is provided by the “identity” framework, features:
  - Authentication
  - User roles
- Support:
  - In the *controller*: by adding decorators to methods (or classes) to require specific users before executing method's code
  - In *views*: for conditionally showing/hiding template parts
- Reference: http://docs.turbogears.org/1.0/Identity

Identity – basics

- `tg-admin quickstart -i`
  - Pre-populate the model with identity-specific classes
    - Classes: User, Group, Permission
    - Classes for the Visit framework (user/browser accounting)
- Test at http://localhost:8080/login

- Create user by using CatWalk ...
- … or instantiating User
- SQLobject columns:
  - user_name=UnicodeCol(length=16, ...)
  - email_address = \ UnicodeCol(length=255, ...)
  - display_name = \ UnicodeCol(length=255)
  - password = UnicodeCol(length=40)
  - created = \ DateTimeCol(default=datetime.now)
  - groups = RelatedJoin('Group', …)

```
>>> u = User(user_name=u'jdoe',
email_address=u'jdoe@example.com',
display_name=u'Jane Doe', password=u'xxx')
```

Identity – basics (cont.)

```
# using identities in the model
class Root(controllers.RootController):
    @expose()
    @identity.require(identity.in_group("admin"))
    def index(self):
        …
class MySecureController(turbogears.Controller,
        identity.SecureResource):
    require = identity.in_group("admin")


# decorator's bestiarium:
# - @identity.require(identity.not_anonymous())
# - @identity.require(identity.in_all_groups("admin", "editor"))
# - @identity.require(identity.in_any_groups("admin", "editor"))
# - @identity.require(identity.from_host("127.0.0.1"))
# - @identity.require(identity.from_any_host(("127.0.0.1", "10.0.0.1")))
```

Identity – basics (cont.)

```
<!-- using identities in the views -->

<a py:if="'admin' in tg.identity.groups" href="/admin">
    This is a link for admins</a>
…

<div py:if="'write' in tg.identity.permissions">This is a write
    permissions area</div>
…

<div py:if="tg.identity.anonymous">Welcome, guest!</div>
<div py:if="not tg.identity.anonymous">Welcome,
   ${tg.identity.user.display_name}!</div>
```

Identity – basics (cont.)

- “Current user” information are available via the `identity.current` value
  - Noteworthy attributes
    - `user` – current `User` object
    - `user_name`
    - `anonymous` – `True` for unauthenticated visitors
    - `permissions` – `Permission` sequence
    - `groups` – `Group` sequence
  - Noteworthy methods
    - `logout()` – close current session

Esercizi

- Aggiungete all'applicativo web sviluppato in precedenza il supporto per l'autenticazione degli utenti