Scripting & Computer Environments

Web2py: The Models

IIIT-H

Nov 6, 2013

...Previously & Today...

Previously: Web2py Intro

- The whats, whys and hows of web2py
- The `Model, View` & `Controller` (MVC) components
- The Architecture
- URL Parsing/mapping

Today: The Models

1. The Database Abstraction Layer (DAL) → The `db.py` file
2. Menus → The `menu.py` file

Recap: MVC

Each web2py app has the components:

{`Models, Controllers, Views, Languages, Modules, Static Files, Plugins`}

- `Models` - describe the *data representation* of the app.

  The `db.py` and `menu.py` files

- `Controllers` - describe *the logic & workflow* of the app.

  The `default.py` and `appadmin.py` files

- `Views` - describe the *data presentation* of the app.

  Formats: `html, xml, json, rss, csv, rtf ...`

Brainstorm

- Database?
- Database Management Systems (DBMS)?
- Database systems?

Brainstorm

- Database?
- Database Management Systems (DBMS)?
- Database systems?

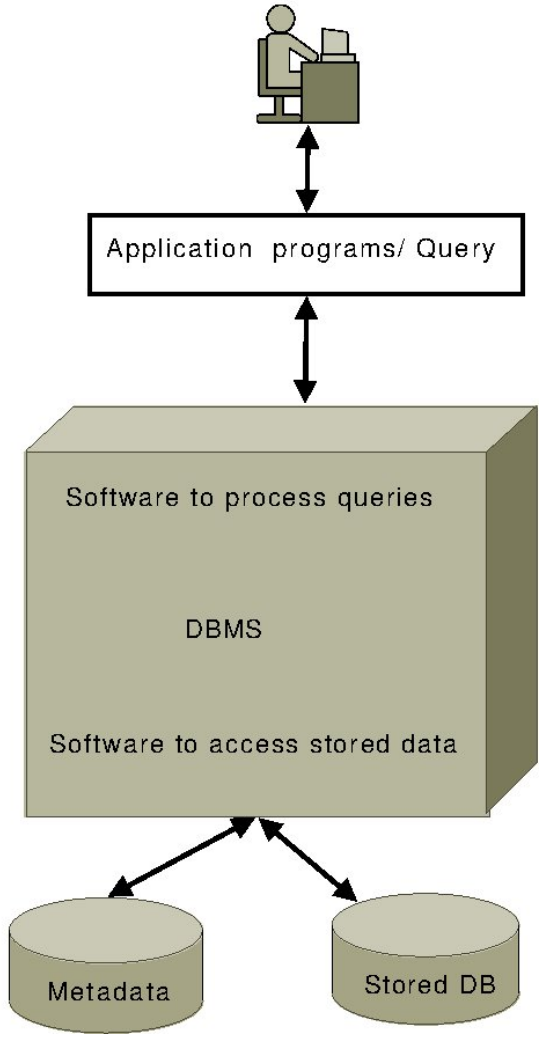
`Database`: A logically coherent collection of related data.

`DBMS`: Software designed to help create & maintain computerized databases.

e.g. `MSSQL`, `MySQL`, `Oracle`, `DB2`, `PostgreSQL`, `Informix`, etc

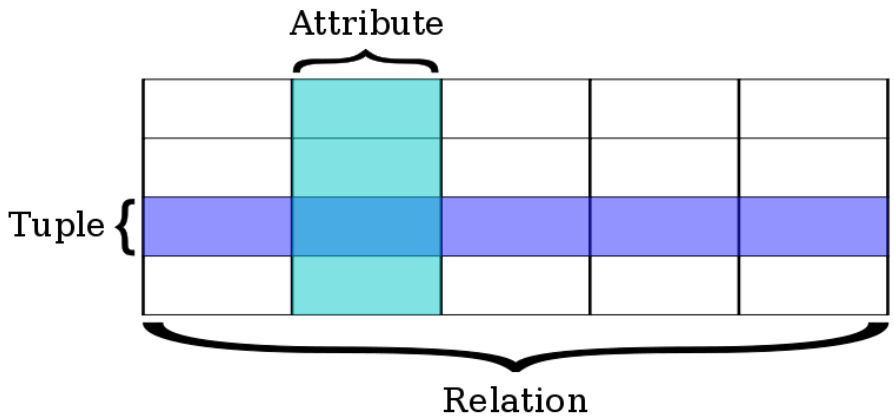
`Database System` = DBMS + the stored data + applications

Simplified database system environment



Relational DBMS

- The most common type of DBMS is *relational DBMS*.
- Models a DB as a collection of entities/objects with relationships.
- *Relation*: a table or flat file with columns and rows.
- The *rows*: a.k.a. tuples/records/instances of the relation
- The *columns*: a.k.a fields/attributes

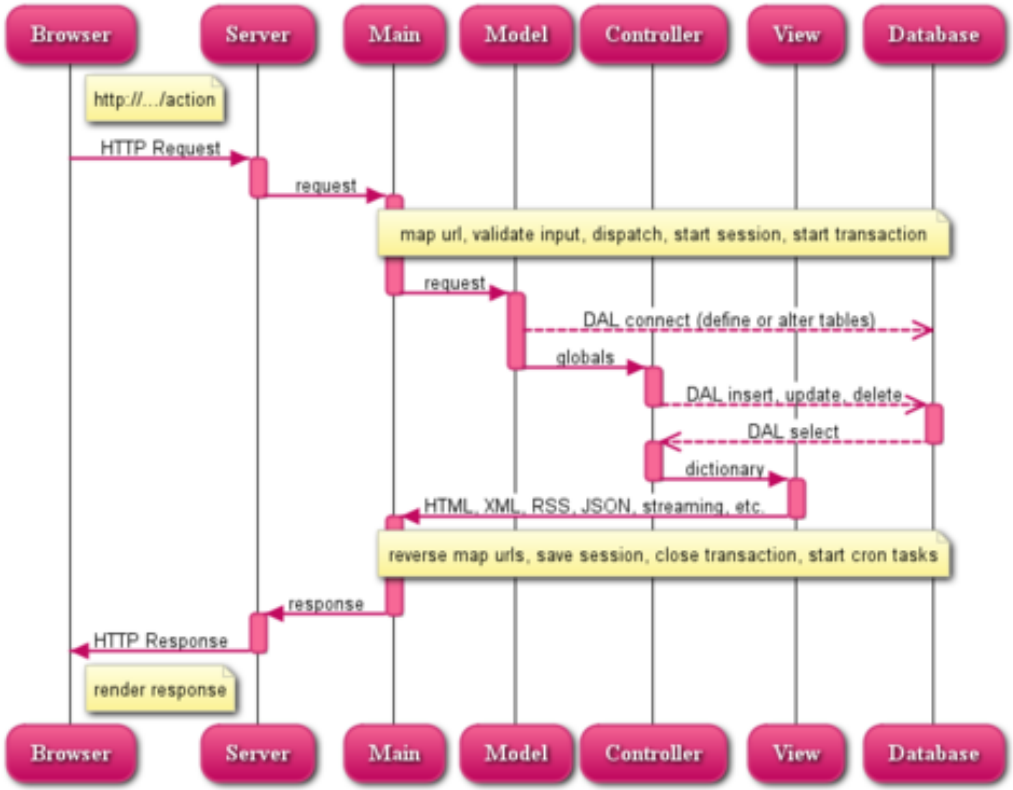


- DAL ≡ Database Abstraction Layer
- An abstraction layer is a way of hiding implementation details of a specific functionality.
- DAL is an API that maps python objects into DB objects (tables, records, queries ...).
- Generates the SQL code for the back-end database dynamically.
- The ramification: no need to write the SQL code (though possible).
- DB drivers be installed for most DBs.

- SQLite (default)
- MySQL
- MSSQL
- PostgreSQL
- Oracle
- Firebird
- DB2
- Informix
- Ingress
- Google App Engine (GAE)

etc ...

The DAL Interaction Model



The DAL Classes

DAL houses the following major web2py classes:

- `DAL` - represents connection to a DB
- `Table` - represents a table in the DB
- `Field` - represents a field/column of a table
- `Query` - represents a SQL `where` clause.
- `DAL Rows` - Represents a list of rows returned by a query.

DAL Operations: Connection

- Before any operation, a connection must be established with the back-end DB engine. How?
- By creating an instance of the DAL class (instantiation).

```
var = DAL('connection string')
```

- For convenience, often the global variable `var=db`.
- The connection string depends on a specific back-end database.
- For `SQLite`, it is `sqlite://storage.sqlite`
- For `MySQL`, `mysql://username:password@localhost/<db name>`

- The `db.define_table(table name, fields)` method instantiates a DB table.
- The `Field` argument passed defines the table fields.
- Field Types:
  - `string`, `text`, `password`, `boolean`, `integer`, `double`, `date`, `time`, `datetime`, `upload` (store file name), `blob` (base64 encoded), `list:string`, `list:integer`, `reference <table name>`, etc

```
db.define_table('student', Field('name'), Field('birthdate','date') )

db.define_table('contacts', Field('name'),
                            Field('PhoneNo'),
                            Field('Photo', 'upload')
                )
```

- All tables have `ID` field by default.

- `Web2py` migration (i.e. behaviors when a table is created):
  - If a table doesn't exist, created.
  - If exists but doesn't match with the definition, altered.
  - If exists and matches, `web2py` does nothing.

- Once tables are defined, `web2py` also generates a fully functional web-based DB admin interface to access the DB and tables.

DAL Validators

- Validators do input validation on table fields, forms ...

```
db.define_table('tablename', Field('fieldname', requires = validator))     # 1

db.tablename.fieldname.requires = validator                                # 2
```

- Some validators:

IS_NOT_EMPTY()	IS_IN_DB()	IS_LENGTH()
IS_EMAIL()	IS_MATCH()	IS_IN_SET()
IS_DATE ()	IS_ALPHANUMERIC()	IS_LOWER()

Some DAL Notations

- `db.tablename` → `db['tablename']`
- `db.tablename.fieldname` → `db.tablename['fieldname']`
- `db.tablename.fieldname` → `db['tablename']['fieldname']`
- Useful Variables:
  - `db._uri` - the connection string
  - `db._dbname` - the DB name
  - `db.tables` - list of tables
  - `db.tablename.fields` - list of fields

- Via the DB admin interface
- The `insert()` method of `Table`

```
db.student.insert(name="Bob", birthdate=datetime.date(1999,12,1)))
```

```
db[‘student’].insert(name="Don")
```

```
db.student[0]=dict(name=‘Eve’)
```

- Using the `select()` method. Returns an iterable object.

```
rows = db(tablename).select()                    # usage 1
e.g. rows = db(db.contacts).select()

rows = db().select(fields)                       # usage 2
e.g. rows = db().select(db.student.ALL)

rows = db(query).select(fields)                  # usage 3
e.g. rows = db(db.student.id==2).select()

row = db.tablename[id]                          # select by ID
```

- Consider:

```
q1 = db.student.name=='Alice'
q2 = db.contacts.phoneNo=='1234567890'
q3 = db.student.name.like('m%')
rows = db(q2).select()
```

- Combining queries using $\sim$ (not), & (and), | (or) is possible.

```
rows = db(q1 & q2).select()
rows = db(q1 | q2 ~ q3).select()
```

- The `orderby='field'`, `groupby`, `distinct={True,False}`, `like`, ... attributes

- Multiple table can be joined together.

Consider the following table definitions:

```
db.define_table('company', Field('name', requires=IS_NOT_EMPTY())
```

```
db.define_table('contacts',
                   Field('name', requires=IS_NOT_EMPTY()),
                   Field('company', 'reference company'),
                   Field('picture', 'upload'),
                   Field('email', requires=IS_EMAIL()),
                   Field('phoneNo', requires=IS_MATCH('\d{10}')),
                   Field('address', 'text')
                )
```

DAL Operations: Update

- Using the `update()` method.

```
db(query).update(field='newvalue')

db.tablename[id]=dict(field='newvalue')          # update by id
```

e.g.

```
db(db.student.name=='Eve').update(name='Adam')

db.contacts[1]=dict(phoneNo='1122334455')
```

- Using the `delete()` method.

```
db(query).delete()

del db.tablename[id]                                    # delete by id
```

- The `truncate()` and the `drop()` methods.

```
db.student.truncate()          # delete records + reset counter

db.contacts.drop()
```

DAL Operations: Display

Suppose `rows` is the object returned by `select()`:

- `print str(rows)` `# CSV output`
- `print rows.xml()` `# XML output`
- `SQLFORM(tablename)` `# A form from a model`
  - `SQLFORM.grid()`
  - `SQLFORM.smartgrid()`
- `SQLTABLE(rows)` `# rows in a tabular format`

Demo

- *http://127.0.0.1:8000/ContactBook*

MENUS

- The other model file is `models/menu.py`.
- It defines menus for the scaffolding app.
- You can write your own custom menu for your app.
- Look into the `menu.py` file for the default menu & its syntax.
- The `URL()` function
  - `URL('function')` is mapped into `/[application]/[controller]/function`
  - `URL('static', 'image.png')` into `/[application]/static/image.png`

- The `response.menu` global object displays the `menu`.
- `menu` is a nested list of menu items.

Syntax

```
response.menu = [[‘name’, False, URL(...), [submenu]], ...]
```

- The 4th argument `submenu` is optional.
- `True/False` indicates if the link is the current one or not.

```
response.menu = [  ['item1', False, 'link1'],
                   ['item2', False, 'link2']
                ]
```

```
response.menu = [  ['item1', False, 'link1',
                     [ ['item2', False, 'link2'] ]
                   ]
                ]                                  # item2 is submenu
```

```
response.menu = [ ('Homepage', False, URL('index')),
                  ('Anotherpage', True, 'another'),
                  ('Search', False, 'http://www.google.com')
                ]
```