web2py for Scientific Applications

The needs of scientists to communicate effectively, collaborate over the Internet, and organize and present data in a structured and accessible manner have become critically important. The web2py framework offers rapid development and prototyping of secure database-driven Web applications and was created specifically with the scientific and academic communities in mind.

Discussions about scientific applications often refer to large-scale numerical computations. In recent times, things have changed, and the scientific community's needs have broadened. In many sectors, scientists' need to communicate effectively, increase access, and organize and present data are more important than their need to write fast code.

For example, using Secure Copy (SCP) and Remote Sync (RSync) protocols to transfer data from one place to another was a good solution when a handful of scientists were working on it, but it's no longer effective when hundreds are collaborating. Data must be tagged, access tracked, and errors logged. Applications that access the data must be given a face in the form of an accessible Web-based user interface or a Web service interface. Through the Web interface, users can more easily and efficiently interact with the data. Grid tools provide a solution to some of these problems, but such tools have a steep learning curve. Moreover, there's not one solution that fits all problems.

Web frameworks are a relatively new class of tools that have come to the rescue of scientists. These frameworks are libraries that allow rapid development of Web-based applications. Most of the frameworks provide APIs to handle concurrency and scalability (to serve multiple users simultaneously), security (authentication, authorization, cross-site scripting, and injection prevention), storage (cookies, sessions, and database access via a database abstraction layer), and interface with third-party programs via standard communication protocols.

A Web framework can turn a scientific software library into a complex interactive application. Web frameworks already find wide applications in science. They've been used to build Web interfaces for data acquisition systems, computing clusters, and computer algebra systems. Different communities have developed Web frameworks in almost every programming language. Common open source ones include Struts for Java, Symphony for PHP, Ruby on Rails for Ruby, and Django for Python.

Here, I describe web2py (http://web2py.com),[1] a new Web framework created in Python and programmable in Python. The only framework born in an academic environment, web2py is used, for example, in projects supported by the US Department of Energy,[2] and it counts more than 1,700 registered users. Here, I'll use web2py to show how we can build a Web application that stores a database of DNA strings and looks for similarities using Needleman-Wunsh plots.[3] This article is based on a tutorial I gave at Supercomputing 2009 in Portland; the code presented is a complete web2py application.

1521-9615/11/$26.00 © 2011 IEEE
Copublished by the IEEE CS and the AIP

Massimo Di Pierro
DePaul University

Framework Overview

Three things distinguish web2py from many existing frameworks. First, its main objective is not simply to provide a utility for Web development,

but to make Web development as easy as possible. To do this, web2py forces developers to follow good practice (such as form self-submission), relieves developers from security-related decisions, and provides everything in one package, including a Web-based integrated development environment (IDE).

Second, web2py is the only framework to provide a database abstraction layer that works on both relational databases and Google App Engine (support for other nonrelational databases is in the works). Third, web2py is written in Python, which lets users leverage power scientific libraries, including NumPy, SciPy, matplotlib, PyTable/HDF5, and others.

Because web2py is a model-view-controller framework, applications built with the framework are divided into files organized by role. The three most important roles are

- *models*, which describe how data should be represented in the system;
- *views*, which describe how data should be presented to users; and
- *controllers*, which describe the application workflow and run the core algorithms.

web2py also handles other types of files:

- regular Python extension modules;
- static files, including images and other files that don't change;
- sessions, which store each system user's current state;
- cache;
- databases, for long-term memory;
- uploads (that is, static files uploaded by visitors);
- crontab files, which describe tasks that must be executed on a fixed schedule; and
- languages, which contain the application translation to support internationalization.

Files are grouped under subfolders of the application folder. The web2py framework enforces this structure and other good software engineering practices. It also provides a Web-based IDE and Web-based testing and debugging tools.

Example Application

As an example, let's say we need a Web application that lets users post and retrieve DNA strings, compare them using the Needleman-Wunsh algorithm,[3] and plot the result. Users must be authenticated to use the system. For plotting, we'll use the matplotlib library.

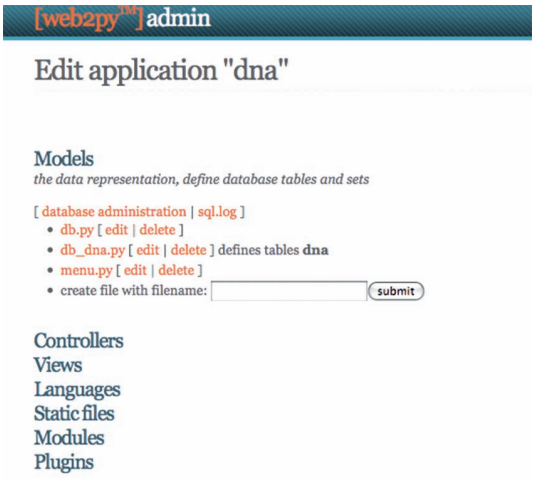

Figure 1. First steps in creating the example `dna` application. The process starts by creating an empty Web-based integrated development environment application that's prepopulated with scaffolding files that are editable.

```
db.define_table('dna',
   Field('name',unique=True,notnull=True)
   Field('sequence', 'text',
     requires=IS_MATCH('[ATGC]*')))
```

Figure 2. A new model file `models/db_dna.py`. The `define_table` dynamically generates and executes the SQL code to create the `dna` table.

To create our `dna` application, we first easily create a new empty application via the Web-based IDE. This application comes prepopulated with scaffolding files that can be edited. (I'll make it clear when we're editing a scaffolding file or creating a new file.) Once web2py is installed and running, we can access the Web-based IDE at URL http://127.0.0.1:8000/admin. Figure 1 shows a screenshot.

The Model

We start by creating a new model file `models/db_dna.py` (see Figure 2).

The function `define_table` dynamically generates and executes the SQL code—for any of the supported back-ends, including SQLite, PostgreSQL, MySQL, MSSQL, FireBird, Oracle, Informix, DB2, SyBase, and the Google App Engine—to create the `dna` table (or alter the table if it has different field content). The `dna` table has two fields, `name` and `sequence`, that have requirements that the framework will enforce at the form level:

- `name` must be non-empty and not already in database, and

```
if not db(db.dna.id>0).count():
   import random, uuid
   genes = []
   for k in range(10):
     bases = ['ATGC'[random.randint(0,3)] for i in range(20)]
     genes.append(''.join(bases))
   for k in range(100):
     sequence = ''.join([genes[random.randint(0,9)] for i in range(50)])
     db.dna.insert(name=uuid.uuid4(), sequence=sequence)
```

Figure 3. The code to append to the `models/db_dna.py` file. The `if` statement checks whether the table contains any record; if not, the code inserts 100 new random DNA sequences in the table.

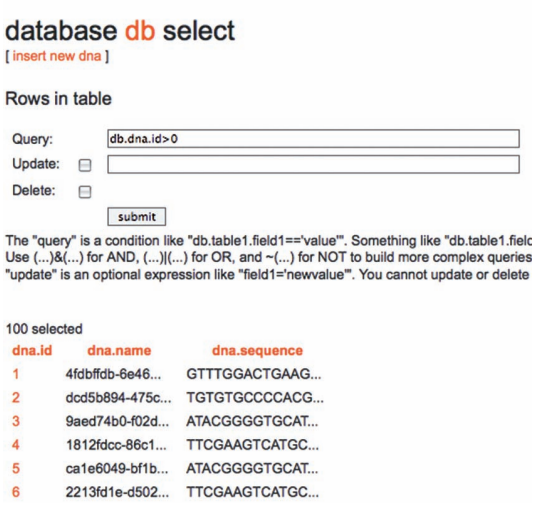

Figure 4. The `appadmin` interface. This interface lets users browse through the generated sequences.

- `sequence` must contain only the `ATGC` symbols.

When users simply type this text in the model file, web2py triggers, creates the table, and generates a Web-based interface (`appadmin`) to perform database inserts, updates, deletes, and selects. Users can access `appadmin` at http://127.0.0.1:8000/and/appadmin; Figure 2 shows a screenshot (I assume web2py is running on 127.0.0.1:8000).

Next, we need to generate dummy data to populate the database. We'll assume each DNA sequence is comprised of 50 genes, each picked at random from 10 possible variations. Each gene consists of 20 random bases (`ATGC`). We then name each sequence with a random universal unique identifier (`uuid`). Figure 3 shows the code to append to the `models/db_dna.py` file.

The `if` statement checks whether the table contains any record; if not, the code inserts 100 new random DNA sequences in the table. Ideally, real data should be used to populate the database. Users can browse generated sequences through the `appadmin` interface (see Figure 4).

The Algorithm

Figure 5 shows the core of our application: the Needleman-Wunsh algorithm.

The Needleman-Wunsh algorithm is implemented in the `model/db_dna.py` file. The algorithm takes two strings, `a` and `b` (which in our case are two DNA sequences), and returns a table `z`,

```
def needleman_wunsch(a,b,p=0.97):
   z=[]
   for i,r in enumerate(a):
       z.append([])
       for j,c in enumerate(b):
           if r==c: z[-1].append(z[i-1][j-1]+1 if i*j>0 else 1)
           else: z[-1].append(p*max(z[i-1][j] if i>0 else 0,
                                    z[i][j-1] if j>0 else 0))
   return z
```

Figure 5. The Needleman-Wunsh algorithm. This algorithm forms the application's core and is implemented in the `model/db_dna.py` file.

where `z[i][j]` represent a similarity factor between the substrings `a[:i]` and `b[:j]`. This empirical algorithm works by defining the similarity between `a[:i]` and `b[:j]` as the similarity between `a[:i-1]` and `b[:j-1]` times `p = 0.97`, if `a[i] = = b[j]` or, otherwise, as the maximum of the similarities between `a[:i],b[:j-1]` and `a[:i-1],b[:j]`. The application visualizes `z` so users can locate similar genes in each couple of DNA sequences (see Figure 6).

The Controller

The controller's role is to implement the application workflow, decide which functions and algorithms to expose, and call these algorithms. In our case, we edit the scaffolding controller `controllers/default.py` and replace the `index` function with the function shown in Figure 7.

This functions defines a `form` that contains two fields (`s1` and `s2`). Each of them is a reference to `db.dna` record. It must be represented by its `%(name)s` as opposed to its `id`. The function, which returns a form, is called by visiting http://127.0.0.1:8000/dna/default/index (see Figure 6). When we pick two sequences and submit the form, the form is accepted (`if form.accepts(...)`) and returns the variable `image` set to the dynamically generated URL calling another function (`needleman_wunsch_plot`) that generates the Needleman-Wunsh plot (which we've not yet implemented). The form variables `s1` and `s2` are also passed to that function via `vars=form.vars`.

The decorator (`@auth.requires_login`) guarantees that only users who have logged in can visit the page. All other visitors must register and go through a login page provided automatically by the framework (web2py implements the role-based access control mechanism with multiple authentication modules).

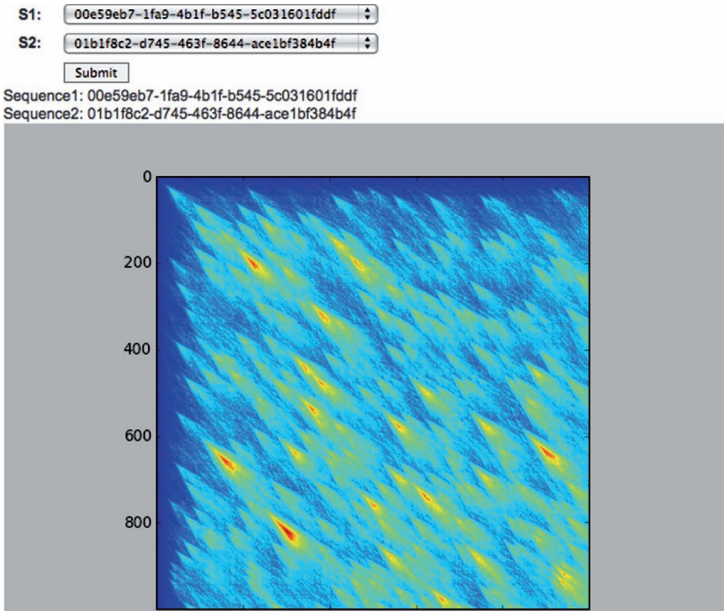

Figure 6. Visualizing `z`. Offering this visualization lets users locate similar genes in DNA sequences.

The View

web2py provides a default presentation for the controller functions' output, but we can customize it in multiple ways. The easiest way is to provide a view for the `index` function by editing the file `views/default/index.html` (see Figure 8).

In this view, everything in `{{ ... }}` is pure Python code and everything outside is HTML (because this is a `.html` view). Here, we extended the default `layout.html`. This is a generic HTML file that ships with the scaffolding application. Its role is to give all pages a consistent look and feel, while also allowing those pages to be customized. We embedded the form in the page `{{=form}}` and also embed the image pointing to the image's URL `{{if image:}}...{{pass}}`.

```
@auth.requires_login()
def index():
    form = SQLFORM.factory(
        Field('s1',db.dna,requires=IS_IN_DB(db,'dna.id','%(name)s')),
        Field('s2',db.dna,requires=IS_IN_DB(db,'dna.id','%(name)s')))
    if form.accepts(request.vars):
        image = URL(r=request,f='needleman_wunsch_plot',vars=form.vars)
    else:
        image = None
    return dict(form=form, image=image)
```

Figure 7. The function that replaces the `index` function. This new function defines a form that contains the two fields that reference the `db.dna` record.

```
{{extend 'layout.html'}}
{{=form}}
{{if image:}}
Sequence1: {{=db.dna[request.vars.s1].
  name}}<br/>
Sequence2: {{=db.dna[request.vars.s2].
  name}}<br/>
<img src="{{=image}}" alt="loading..."/>
{{pass}}
```

Figure 8. Customizing the controller functions' output. The easiest way to achieve this is to edit the file `views/default/index.html` to provide a view for the `index` function.

```
@auth.requires_login()
def needleman_wunsch_plot():
  import cStringIO
  from matplotlib.backends.backend_agg import *
  from matplotlib.figure import *
  ### 1) fetch sequences
  dna1 = db.dna[request.vars.s1].sequence
  dna2 = db.dna[request.vars.s2].sequence
  ### 2) run algorithm
  z = needleman_wunsch(dna1,dna2)
  ### 3) make figure and plot on canvas
  fig = Figure()
  fig.add_subplot(111).imshow(z)
  ### 4) print canvas to memory mapped file
  stream = cStringIO.StringIO()
  FigureCanvasAgg(fig).print_png(stream)
  ### 5) return content of memory mapped file
  response.headers['Content-Type']='image/png'
  return stream.getvalue()
```

Figure 9. The complete implementation. The `needleman_wunsch_plot` is implemented in `controllers/default.py` and has several tasks, including to retrieve actual sequences from the database and call the Needleman-Wunsh algorithm.

Plotting

Finally, we implement the function `needleman_wunsch_plot` in `controllers/default.py`. This function has five primary tasks:

- retrieve the actual sequences from the database,
- call the Needleman-Wunsh algorithm,
- print the data as a plot on a canvas in memory,
- print the canvas in a memory-mapped PNG file (`cStringIO`), and
- return the PNG content.

Figure 9 shows the complete implementation.

The matplotlib APIs are described in detail at http://matplotlib.sourceforge.net. The yellow and red patterns in Figure 6 let us easily find gene pairs (identified by their x and y coordinates) that contain similar DNA subsequences.

Web Services

At this point, our program is complete and fully functional, so it's time to add a Web service. Specifically, we want to expose the ability to retrieve a DNA sequence from its `uuid` name via an XML Remote Procedure Calling (RPC) service. We can do this simply by placing the following code in `controllers/default.py`:

```
@service.xmlrpc
def get_sequence(uuid):
    return db.dna(name=uuid).sequence
```

The decorator (`@service.xmlrpc`) exposes the function (the function's name, `get_sequence`, is arbitrary). The service is available at http://127.0.0.1:8000/dna/default/call/xmlrpc and can be called from any programming language supporting XML-RPC. Because all database access is performed via the database abstraction layer's API, there's no need to write any SQL.

Linked Data

Linked Data (LD; http://linkeddata.org) is a set of best practices for exposing, sharing, and connecting pieces of data, information, and knowledge on the Semantic Web using Universal Resource Identifiers and a Resource Description Framework (RDF).

RDF is an XML-based document file format. World Wide Web inventor Tim Berners-Lee proposed the LD, which consists of tagging database tables, fields, and relations using a Web Ontology Language (OWL) and exposing the data together with metadata that describe its meaning and relations. To support LD, web2py lets users tag data with OWL and automatically create an RDF service. More details are available at http://web2py.com/semantic.

By providing tools to help them expose their data and algorithms to the Web for both human and machine consumption, Web frameworks can help scientists increase transparency, awareness, and efficiency in their work. The web2py framework, which is released under the GPL2 license, was developed specifically with the scientific and academic communities in mind. CiSE

Acknowledgments

I thank David Angulo for reviewing this article and for insightful conversations about the algorithm used here.

References

1. M. Di Pierro, *web2py Manual*, 3rd ed., lulu.com, 2010.
2. M. Di Pierro, "mc4qcd," *Proc. Advanced Computing and Analysis Techniques, Proc. Science*, 2010; http://pos.sissa.it/archive/conferences/093/054/ACAT2010_054.pdf.
3. S.B. Needleman and C.D Wunsch, "A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins," *J. Molecular Biology*, vol. 48, no. 3, 1970, pp. 443–53.

Massimo Di Pierro *is an associate professor at the School of Computing of DePaul University in Chicago. His main expertise is in high-performance numerical algorithms for scientific and financial applications. Di Pierro has a PhD in high energy physics from the University of Southhampton, UK. Contact him at mdipierro@cs.depaul.edu.*