Web application development using Django

John Pinney

December 16, 2016

Introduction

Web applications (webapps) can make an important contribution to getting new bioinformatics software more widely used by the research community.

The Django framework (`www.djangoproject.com`) provides a convenient way to develop web applications, especially when linked to a database. As much as possible, pure Python code is used to process information supplied by the user, interact with the database and generate content to be sent back as a new web page. With a basic understanding of HTML, CSS and Javascript, it is possible to put together a usable and appealing webapp with relatively little effort.

This practical will guide you through the basics of setting up a new webapp with Django and using it to populate and query a database. It's fine if you don't complete everything today, but try to work through all of the main sections to get a feel for the basic capabilities of Django.

Setup

For this practical, you will work entirely on the virtual Linux machine, which will act as both the client and the server.

Unfortunately, the way in which your `homedir` is mounted means that you won't be able to work with Django inside this directory. Instead, make a new directory under your `home` (i.e. something like `/home/<username>/django/`) and remember to copy this back into your `homedir` at the end of the session to save your work.

Some example code is provided on Blackboard in `django_practical.zip`
Copy this into your directory and `unzip` it. We will use various files from here as needed later.

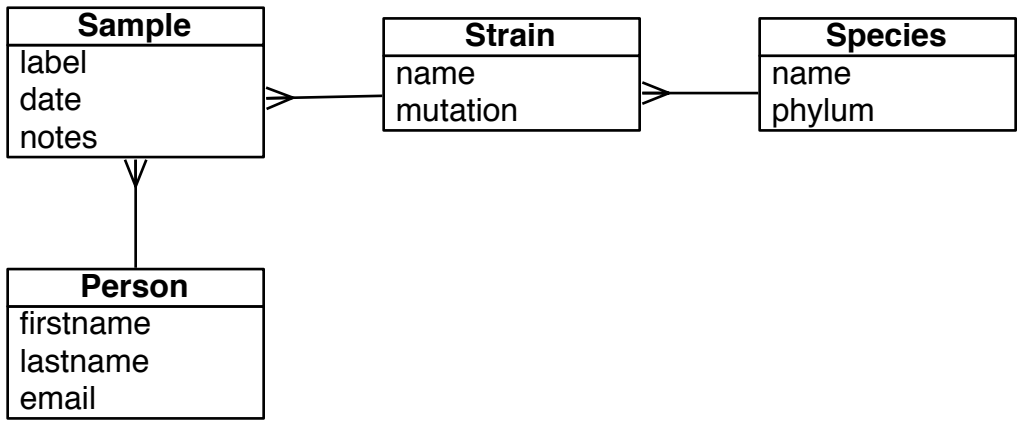
To install Django, move to your `django` directory and execute the following commands:

```
>wget https://bootstrap.pypa.io/get-pip.py
>python get-pip.py --user
>pip install --user django
```

You should see a message confirming that Django has been installed. You can use these commands on any Linux machine to make use of Django, even if you don't have root permissions.

The task

You will develop a simple database and web interface to help a biology lab keep track of the samples that are stored in their freezer. The database schema is shown below.



Although large databases often use system-level management software such as MySQL, in this practical we will use a simpler alternative called SQLite3. SQLite3 operates its databases from individual files, so it is a quick and convenient solution for smaller projects.

1 Setting up a new Django app

Within your `django` directory, type
```
>django-admin.py startproject mysite
>cd mysite
```

to start a new project. You will see that a new directory `mysite` has been generated, containing a subdirectory (also called `mysite`) and a script called `manage.py`. You use this script to manage the webapps that belong to your project.

To see the capabilities of manage.py, type
```
>python manage.py help
```

Now you can use `manage.py` to generate a new app for our freezer example:
```
>python manage.py startapp freezer
>cd freezer
```

Note the new files that have been generated for you.

2 Models and the database

A model is the single, definitive source of information about your database. It contains the essential fields and behaviours of the data you're storing. The goal is to define your data model in one place and automatically derive things from it.

Each model is represented by a class that subclasses `django.db.models.Model`.

Each model has a number of class variables, each of which represents a database field in the model. Each field is represented by an instance of a Field class – e.g., `CharField` for character fields and `DateTimeField` for datetimes. This tells Django what type of data each field holds. The name of each Field instance (e.g. `firstname` or `email`) is the field's name, in machine-friendly format. You'll use this value in your Python code, and your database will use it as the column name. A relationship between tables is defined using ForeignKey.

Define models corresponding to the database schema by editing `models.py`. In my examples I have provided models for `Species` and `Sample` in `example/mysite/freezer/models.py` to get you started. You will need to complete the models for `Person` and `Strain`.

Note that `Species` includes a pre-defined set of bacterial phyla with two-letter codes.

Now add the `freezer` app to the `INSTALLED_APPS` registry in `mysite/mysite/settings.py`. It should look something like this:

```
INSTALLED_APPS = [
'django.contrib.admin',
'django.contrib.auth',
'django.contrib.contenttypes',
'django.contrib.sessions',
'django.contrib.messages',
'django.contrib.staticfiles',
'freezer',
]
```

You are now ready to create your database. Django maintains version control of your database using 'migrations', which can help you later on if you need to modify the database. To make the migration corresponding to your initial schema, type
`>python manage.py makemigrations freezer`

You can see the SQL instructions that correspond to this migration by typing
`>python manage.py sqlmigrate freezer 0001_initial`

(This command just prints the SQL to the screen, it doesn't execute anything.)

To commit this migration and create the new tables, just type
`>python manage.py migrate freezer`

and you should see some encouraging messages. You will notice that the file `db.sqlite3` has been created - this is the SQLite3 database itself.

3 Adding data with SQL

We will need to add a bit of data to the database before using the webapp to display it. Now's your chance to practice some SQL!

`>sqlite3 db.sqlite3`
will start an interactive session with SQLite3.

Type
`>.help`
to see the allowed commands. What tables has Django made in the database?

Enter the following statements (remember the semicolons):

```
>insert into freezer_person (firstname,lastname,email) values
('Louis','Pasteur','l.pasteur@imperial.ac.uk');
>insert into freezer_species(name,phylum) values ('Escherichia coli','PR');
```

Have a look at the Species table:
```
>select * from freezer_species;
1|Escherichia coli|PR
```

Notice that an id (`1`) has been generated automatically.

Foreign keys are indicated using the `_id` suffix on a table column:
```
>insert into freezer_strain (name,mutation,species_id) values
('K12 MG1655','DEL(pspF),DEL(pspE)',1);
>insert into freezer_sample (label,date,person_id,strain_id,notes) values
('sample_1','2016-12-16','1','1','Preliminary experiment');
```

You should now have one row in each table, so we have something to start working with in Django. If you would like to practice some SQL, feel free to make a few more entries and try out some `select` statements if you like.

4 Views and URLs

A view is a kind of prototype for a web page within your Django application. A view is usually associated with a particular URL pattern. This makes it easier to keep the different functions of your webapp separate.

Configuration of URLs with Django projects can be a bit tricky, so I have provided a simple setup for you:

1. Copy the file
`example/mysite/mysite/urls.py`
into your own `mysite/mysite` directory. This will divert any request starting `/freezer` into your webapp. You don't need to do anything with this.

2. Copy the file
`example/mysite/freezer/freezer_urls.py`
into your own `mysite/freezer` directory. This tells Django which view to invoke when it receives a URL of a particular form. Take a look at this file, but don't change anything yet.

3. Copy the file
`example/mysite/freezer/views.py`
into your own `mysite/freezer` directory. This script contains basic definitions for the views declared in freezer_urls.py.

Now you can start up the Django server. The server logs requests and it can be useful to see its output for debugging purposes. The server is smart enough to reload most of your files as they change, but you may find that you need to restart it to refresh things like CSS templates (used later on).

Within your upper `mysite` directory, type
`>python manage.py runserver`

Now take a look at the contents of the freezer by pointing your browser to
`http://localhost:8000/freezer/contents`.

The message you receive comes from the definition of the view `contents` within `views.py`:

```
from django.http import HttpResponse
def contents(request):
  return HttpResponse('Here are the contents of the freezer...')
```

Try editing `views.py` to change this message.

Now let's make it do something slightly more useful:

```
from django.http import HttpResponse
from freezer.models import *

def contents(request):
  sample_list = Sample.objects.all().order_by('-date')
  output = ', '.join([s.label for s in sample_list])
  return HttpResponse(output)
```

Notice how you can retrieve the contents of the table (in this case `freezer_sample`) via `Sample.objects` and sort them by date. Other database queries are constructed in a similar way, using the models we defined earlier - no SQL required!

5 Templates

We would like to see more details about the samples in the freezer, so we will use an HTML table to organise the output. However, it is important to avoid hard-coding the design into the view itself. This is what templates are for.

Edit `views.py` like this:

```
from django.shortcuts import render
from freezer.models import *

def contents(request):
  sample_list = Sample.objects.all().order_by('-date')
  return render(request,'contents.html', {'sample_list':  sample_list})
```

This makes the same list of samples as before, but now sends it to the template `contents.html` to be rendered as HTML.

The template is just an HTML document with a bit of added Python. Copy the directory `templates` from my examples into your `freezer` directory and take a look at `templates/contents.html`.

Notice that you include Python code in the template by enclosing it in {% ... %} and you can echo the values of variables by enclosing them in {{ ... }}. It is easy to access the columns of other tables using the dot notation (e.g. `sample.strain.name`. What happens if you just put `sample.strain`?). Django already knows to look inside the `templates` directory for the templates it needs, so there is no need to specify the location of `contents.html`. Reload your page and you should see the output change accordingly.

6 Simple design using CSS

The table looks a bit boring but it is easy to liven it up with a bit of CSS. Copy my directory `examples/freezer/static` into your own `mysite/freezer` directory. Django searches this directory for 'static' (i.e. unchanging) content such as images, stylesheets and javascripts.

I have provided a basic stylesheet in `static/stylesheet.css`:

```
body {
  font-size:  1em;
  color:  black;
  background-color:  #eeeeee;
  font-family:  arial, helvetica;
  margin:  2em;
}

h1 {
  color:  #336633;
}

th {
  color:  #333388;
  ext-align:  left;
  width:  200px;
}

td {
  background-color:  #dddddd;
}

#pie {
  width:  250px;
  height:  250px;
}
```

See how this changes the look of your output. Feel free to edit the stylesheet as you like, but leave `#pie` alone for now.

7 Forms

Obviously the database is still looking a bit empty. The other people who work in the lab are eager to fill it with their samples, but they're not as handy with SQL as you are. We can use HTML forms to make it easier for them to input their data.

First, add the following to `views.py` to make a simple view of the Person table:

```
def person(request):
  person_list = Person.objects.all().order_by('lastname')
  return render(request, 'person.html', {'person_list':  person_list})
```

Make a copy of your `contents.html` named `person.html` and edit this so it will correctly render the Person table.

Now let's add a form to the foot of the template (before `</body>`) so we can add new people as needed:

```
<hr>
<h2>New Person</h2>
<form action="person" method="post">
  {% csrf_token %}
  <table>
    <tr>
      <th>firstname:  </th>
      <td><input type="text" name="firstname" /></td>
    </tr>
    <tr>
      <th>lastname:  </th>
      <td><input type="text" name="lastname" /></td>
    </tr>
    <tr>
      <th>email:  </th>
      <td><input type="text" name="email" /></td>
    </tr>
  </table>
  <p><input type="submit"></p>
  <p><input type="reset"</p>
</form>
```

When the form is submitted, it generates an HTTP POST request that directs back to the `person()` view you just created. It is good practice to use POST requests for any action that changes the database contents, and the default GET requests for everything else.

Notice the special field in the form generated by the code {% csrf_token %} (Use 'view source' on your webpage to see how this is rendered by Django). This is a security feature which makes use of cookies to ensure that the data you receive from POST really comes from the client to whom you sent the form.

We can catch the POST request by changing `person()` as follows:

```
def person(request):
  if request.method == 'POST':
    data = request.POST
    p = Person(
      firstname = data['firstname'],
      lastname=data['lastname'],
      email = data['email']
    )
    p.save()
  person_list = Person.objects.all().order_by('lastname')
  return render(request, 'person.html', {'person_list':  person_list})
```

Note that you extract the form data via the dictionary `request.POST`. A new `Person` object is created with the values submitted and the `Person.save()` method stores the data in the database. Easy!

The script `example/freezer/views2.py` contains similar views for Sample and Strain. Templates for these views are already in your `templates` directory. Take a look at these to see how to construct multiple-choice dropdown boxes for the form and how to retrieve an object, given its database ID.

Add a few more People, Strains and Samples to the database.

If you want to experiment further, make a new view and template to allow the user to add new Species to the database. Note that the list of bacterial phyla and their abbreviated codes can be accessed as `Species.PHYLA`.

8 Javascript for more advanced functionality

Javascript provides client-side functionality to your website and can be used to make the content more interactive. There are many free Javascript libraries available for all kinds of purposes. Here we will use a library called gRaphaël, which can be used to render charts.

Change your `contents()` view to use the alternative template `contents2.html`. This has the following additions in the HTML header:

```
<script src="/static/js/raphael-min.js"></script>
<script src="/static/js/g.raphael-min.js"></script>
<script src="/static/js/g.pie-min.js"></script>
<script>
  window.onload = function () {
    var r = Raphael("pie");
    r.piechart(100, 100, 100, [55, 5, 13, 32], { legend:  ["a", "b", "c", "d"]});
  };
</script>
```

This code creates a pie chart at with center at (100, 100) with respect to the top-left corner of the `<div id="pie">` element, radius 100 and the data vector [55, 5, 13, 32].

Instead of displaying this sample data, adapt your view and the new template to show the amount of space in the freezer that has been used by each person.

There is space in the freezer for 500 samples. Can you make the pie chart more useful by showing the amount of space remaining?

9 Further reading

9.1 Django

There is much more information available at the Django website about its more advanced features.

`www.djangoproject.com`

9.2 Flask

If you don't require a database, you will probably find Flask to be a more straightforward way to set up a webapp using Python.

`http://flask.pocoo.org/`

9.3 HTML and CSS

Some excellent tutorials for web developers are available at

`http://htmldog.com/`

9.4 Javascript

There are many Javascript libraries available online. Here are a few more that might come in useful:

Raphaël (on which gRaphaël is based) is a general system for making interactive vector graphics.

`http://dmitrybaranovskiy.github.io/raphael/`

jQuery is a very powerful library for developing dynamic content connect connected to your HTML document elements.

`http://jquery.com/`

Datatables (which uses jQuery) can be a great way to present large tables, allowing multiple pages and column sorting.

`http://datatables.net/`

Processing.js lets you use the simple and powerful Processing language within the browser to create animations and interactive graphics.

`http://processingjs.org/`
`http://processing.org/`