COMP1745 Lab Exercises

Release 0.1

Nick Efford

28 January 2015

CONTENTS

CHAPTER

ONE

HTTP, CGI & FORM HANDLING

Author Nick Efford

Contact N.D.Efford@leeds.ac.uk

Status Final

Revised 2015-01-14

1.1 Probing HTTP Requests & Responses in Chrome

1. Start the Chrome browser. Click the menu button at the top right of the browser window and choose *More Tools → Developer Tools*, or press `Ctrl+Shift+I`. The Dev Tools panel should appear at the bottom of the browser window.
2. Click on the *Network* heading on the Dev Tools menu bar. Then use the browser to visit the web site of the University of Leeds, http://www.leeds.ac.uk. The Dev Tools panel will now contain a list of all the requests that were made in order to display the page.
3. Click on one of the requests, then select the *Headers* tab on the panel that appears. You should see the headers for the HTTP request made by the browser and for the corresponding response from the server. Click on the 'view source' link to see the raw header text and then on the 'view parsed' link to return to the default format.

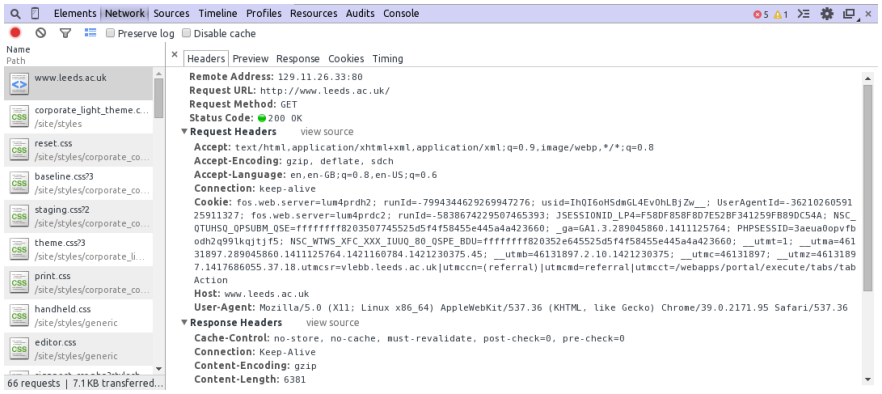

Figure 1.1: Chrome's Dev Tools panel showing request & response headers for HTTP traffic

Note: For some requests, you might not see any request or response headers. Typically, this occurs because the request can be satisfied from the cache. Dev Tools will display '(from cache)' to indicate this.

1.2 Probing HTTP Requests & Responses in Firefox

1. Start the Firefox browser. Click the menu button at the top-right of the browser window and choose *Developer → Web Console*, or simply press `Ctrl+Shift+K`. A panel should appear at the bottom of the browser window, similar to the one you saw in Chrome.
2. Click on the *Network* button in the Web Console (top row, far right). Visit the University of Leeds web site, just as you did in Chrome, and you will see that a list of the requests made by the browser appears in the panel.
3. Click on one of the listed requests. A new panel will appear at the the bottom-right of the window, within which you can see the request and response headers. Repeat this for some of the other requests.

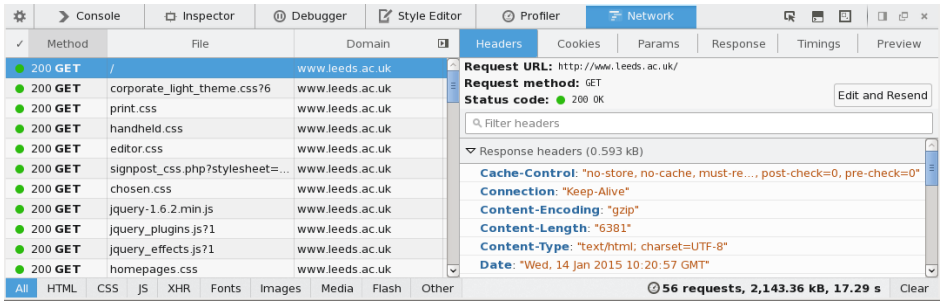

Figure 1.2: HTTP requests displayed in Firefox

Note: These instructions refer to the version of Firefox installed on SoC Linux machines. The user interface for the developer tools in a more recent version of Firefox (e.g., on your own PC) might be different.

1.3 Constructing HTTP Requests Manually

Hurl.it is a nice online tool for constructing HTTP requests by hand and then inspecting the responses.

1. In your browser, visit http://hurl.it. You will see a form that allows you to specify an HTTP method and a URL. The form also allows you to specify authentication options and add headers or parameters to the request.

   For Destination, specify GET and enter `http://www.comp.leeds.ac.uk`. Then click on the *Launch Request* button.

   Examine the response that is displayed. You should see a response of 302 Found - a typical (though not entirely correct) way of indicating that a redirect is necessary.

   To see the redirect happen, click on the 'Off' link next to 'Follow redirects' to enable redirects, then click the *Launch Request* button again. You should see a 200 OK and the full HTML of the School of Computing's home page.

2. Now try encoding a parameter into a GET request. For this, we will use a geoPlugin web service that takes an IP address supplied with the request and returns information about the geographical location of that IP address.

   You will need to find out the IP address of your machine first. On a SoC Linux machine, you can do this from within the terminal window using the command

   ```
   hostname -i
   ```

   Enter `http://www.geoplugin.net/json.gp` into the Destination field, then click the *+ Add Parameter(s)* button. Enter `ip` as the parameter name and the IP address of your machine as the value. Then click the *Launch Request* button.

Above the response, you should see the full GET request displayed. Notice how the parameter and its value have been assembled into a query string that is then appended to the web service URL (after a question mark).

Now examine the response. Notice that it isn't HTML! Instead, you should be seeing JSON-formatted geographical data pertaining to your IP address.

1.4 A Simple CGI Program

1. Open a terminal window and create a directory to hold files for this set of exercises. In that directory, create a subdirectory called `cgi-bin` and, in that subdirectory, create a file called `clock.cgi`, containing the following Python code:

```
#!/usr/bin/env python3

import cgitb
cgitb.enable()

from datetime import datetime

current_time = datetime.now()

print("Content-type: text/plain\n")
print(current_time)
```

   Take care not to make any typing errors here.

2. In the terminal window, set execute permission for this program like so:

```
chmod u+x clock.cgi
```

3. Still in the terminal window, move up a level so that you are in the parent directory of the `cgi-bin` directory. Also, if you are doing this exercise on a SoC Linux machine, activate Python 3 by entering `p3`.

   Now enter the following command:

```
python -m http.server --cgi 8000
```

   This runs a very simple CGI-enabled web server, listening on port 8000. (Use a different number if port 8000 is unavailable.)

4. Start a web browser if necessary. In that browser, visit `http://localhost:8000/cgi-bin/clock.cgi`. You should see the current date and time displayed. Wait a few seconds, then click the reload button. Repeat this a few times to satisfy yourself that fresh content is being generated by the CGI script each time that the page is reloaded.

5. Leave the web server running in your terminal window and use a text editor to edit `clock.cgi`. Introduce a small error into the program - e.g., changing the first `datetime` to `dtetime`. Save the file.

   Now return to your web browser and reload the page. The server will attempt to run the CGI program again, but this time execution will fail.

   Notice how the details of the error appear in the browser window. This happens because we have imported the `cgitb` module and enabled CGI tracebacks; without this code, we would see a blank screen in the browser and an error message in the server log. Try it now by editing the program, commenting out the two lines relating to the `cgitb` module and then reloading the page.

6. Go back to the terminal window and shut down the web server by pressing `Ctrl+C`.

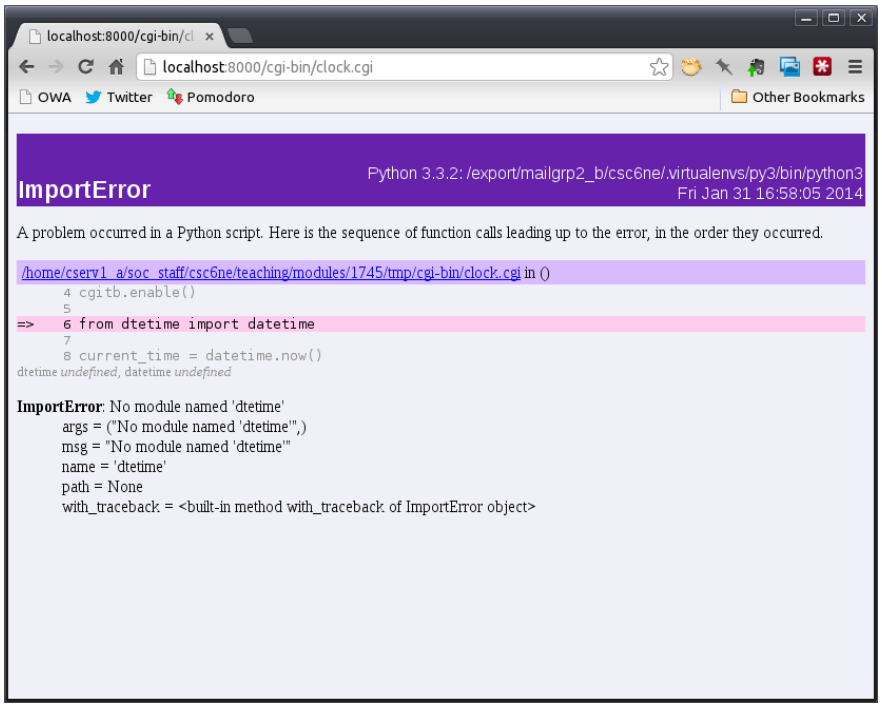

Figure 1.3: Display of CGI errors as a traceback in the browser

1.5 Handling Form Data With a CGI Program

1. In the parent directory of your `cgi-bin` directory, create a small HTML document called `add.html`. Give your document a title and an `h1` heading of 'Number Addition' and add a `form` element to it. Use the following as the value of your form's `action` attribute:

   `/cgi-bin/add.cgi`

   Give your form two small text input fields with name attributes of `a` and `b`, respectively. Also, give it a submit button with the equals sign as the button text. Put a '+' in between the two text fields and enclose all of the form elements inside a paragraph element. The page should look something like this when viewed in a browser:

2. Move down into your `cgi-bin` directory and create a new program called `add.cgi`. Lines 1-4 of this new program should be identical to those of the `clock.cgi` program used earlier.

   After these lines, add some code to import the `cgi` module and create a `FieldStorage` object called `form`. Then add some code that calls the `getvalue` method of this object, in order to retrieve values for form parameters `a` and `b`. Use the lecture example as your guide here.

Next, add some code that converts the strings returned by `getvalue` to a pair of numbers, which you can add together. Finally, write some code that prints the result of the calculation using HTML markup. Remember that you will need to specify `Content-type` in the header of the response, telling the browser that you are sending it HTML. This can be achieved with a simple `print` statement:

```
print("Content-type: text/html\n")
```

After this, you can put further `print` statements to output the HTML.

3. Use `chmod` to set the execute permission for `add.cgi`, just as you did for `clock.cgi`. Then move up to the parent directory and run the web server again. Visit `http://localhost:8000/add.html` in your browser, enter a pair of numbers in the form and click the '=' button. Hopefully, you will see a response that shows the sum of the numbers. If you see a traceback instead, use the information in the traceback to locate and fix the problem in `add.cgi`.

4. Try reloading the page a few times. If you've specified GET in the `method` attribute of your form, or if you've not used the `method` attribute at all, then reloading will simply resubmit the form data to the server, causing the `add.cgi` program to run again.

   Edit `add.html` and change the opening tag of your `form` element so that it looks like this:

   ```
   <form action="/cgi-bin/add.cgi" method="POST">
   ```

   Terminate the server by pressing `Ctrl+C`, then run it again. Visit `http://localhost:8000/add.html` again in the browser. Submit a pair of numbers and click '=' to get a result, then try reloading the page. This time, the browser should warn you about resubmission of form data.

5. It is very easy to break the program as it stands. Try it now by returning to the form and leaving one of the fields empty, or try entering non-numeric data in one of the fields. The resulting traceback isn't very user-friendly, so see if you can improve on it.

   The trick here is to use Python's `try` and `except` to catch any exceptions that might occur when the strings retrieved from the form are converted to numbers. In the `except` block, you can put some code that outputs an HTML page containing an error message of some kind.

CHAPTER

TWO

GETTING STARTED WITH DJANGO

Author Nick Efford

Contact N.D.Efford@leeds.ac.uk

Status Final

Revised 2015-01-14

These exercises provides a concise introduction to setting up Django on SoC machines and to creating and configuring a project. Further information can be found in Part 1 of the official Django tutorial.

2.1 Accessing Django on SoC Machines

Note: If you are doing this exercise on your own PC and have already installed Django, you can skip the instructions below and begin with Starting a Project.

Django is a large framework, requiring over 40 MB of disk space. Therefore it is best to use the version we provide, rather than installing your own copy of it in your SoC filestore. The following instructions, which need to be carried out once only, will set up access for you.

1. Open a terminal window and enter the following command:

   ```
   gedit $HOME/.bashrc
   ```

   Be sure to type the '.' in the filename! After any existing content in this file, add the following line:

   ```
   export PYTHONPATH=/home/csunix/scpython/lib
   ```

   Save the file and close *gedit*.

2. Now close the old terminal window and open a new one. In that new terminal window, enter `p3` to activate Python 3, then enter `python` to run the Python interpreter. Check that the interpreter is reporting the Python version as 3.3.2.

   At the >>> prompt, enter the following commands:

   ```
   >>> import django
   >>> django.get_version()
   ```

   You should see `'1.7.3'` displayed; if you don't see this, ask for help.

3. The final step is to install the `django-admin.py` program.

   Check your home directory and make sure it contains a subdirectory named `bin`. If it does not, create this subdirectory now. Then download the file `django-admin.py` into that `bin` subdirectory.

In your terminal window, enter the following command to give the program execute permissions:

```
chmod u+x $HOME/bin/django-admin.py
```

Check that installation has succeeded by entering the following command in the terminal window:

```
django-admin.py
```

You should see a lengthy program usage message. If you get 'command not found' or some other error, ask for help.

2.2 Starting a Project

1. Create a directory in which to hold all the files for this and subsequent Django exercises. In your terminal window, `cd` into this newly-created directory, then enter the following command:

   ```
   django-admin.py startproject football
   ```

   This creates a `football` directory. Use `ls` in the terminal window or the file browsing tools available on the desktop to explore what has been created under this directory. You should see a structure like this:

   ```
   football/
       manage.py
       football/
           __init__.py
           settings.py
           urls.py
           wsgi.py
   ```

   The most important files here are: `manage.py`, which is used to perform tasks such as synchronising with a database, running a development web server, etc; `settings.py`, which contains configuration details for the project; and `urls.py`, which specifies how the URLs of HTTP requests map onto Python code that will service those requests.

2. In the terminal window, move into the `football` directory with `cd football`, then enter the command

   ```
   python manage.py
   ```

   This will show you all of the tasks that can be performed using the program.

2.3 Configuring a Project

1. Open `football/settings.py` in a text editor such as *gedit*. Study the settings therein. The Django documentation provides full details of these settings, should you need it. We will return to some of them in later exercises.

   Find the `DATABASES` setting and check that the `sqlite3` database engine is specified. Then change the name of the database file from `db.sqlite3` to `football.db`.

2. Change the `LANGUAGE_CODE` setting to `en-gb`. If you would prefer times to be localised, change the `TIME_ZONE` setting to `Europe/London`. Then save your changes.

2.4 Database Creation

Django has some built-in apps that perform useful tasks such as handling authentication of users, managing sessions and managing static files (CSS, images, etc). The default settings for a Django project assume that you will be using these apps; you can see them listed in the `INSTALLED_APPS` setting in `settings.py`.

These apps make use of database tables, so the next step in setting up a project is to create these tables.

1. Run the following command to create the database tables:

   ```
   python manage.py migrate
   ```

   List the contents of the project directory with the `ls` command. You should see a new file called `football.db` [1].

2. Django allows you to examine the contents of the database directly. Try this now by entering the following command:

   ```
   python manage.py dbshell
   ```

   At the `sqlite>` prompt, enter the command `.tables` (be sure to include the '.' at the start). This will show you all of the tables that Django has created. To see more detailed information such as the names and type of the columns in each table, enter the command `.schema`.

3. Now try querying the database for details of system users. Enter the following **SQL query** at the `sqlite>` prompt, making sure that you end it with a semicolon:

   ```
   select * from auth_user;
   ```

   This should generate no output, because no users have been set up yet.

4. Leave the database shell by entering `.quit` (again, be sure to include the '.' at the start). Then create an administrator account for the project by entering the following command:

   ```
   python manage.py createsuperuser
   ```

   Supply a username, email address and password of your choosing.

   Note: Take care to remember the username and password, as you will need these credentials *in a later worksheet*.

   Run `manage.py` with the `dbshell` command again and repeat the SQL query shown above. This time, you should see a single database row that includes your username, encrypted password and email address.

2.5 Running The Development Server

Django includes a simple web server that can be used to test your application whilst it is in development [2].

1. Run the development server in a terminal window like so:

   ```
   python manage.py runserver
   ```

   Django will check your project code for errors and then start the server, listening on port 8000 by default. If this port is in use, you can specify a different port number as an additional command line argument.

[1] This file is how SQLite represents a database. You won't see it if you configure Django to use a database server such as MySQL or PostgreSQL.

[2] To keep things simple, we will use the development server throughout COMP1745, but do keep in mind that it isn't really suitable for production use.

2. In a browser, visit `http://localhost:8000/`. You should see a page like this:

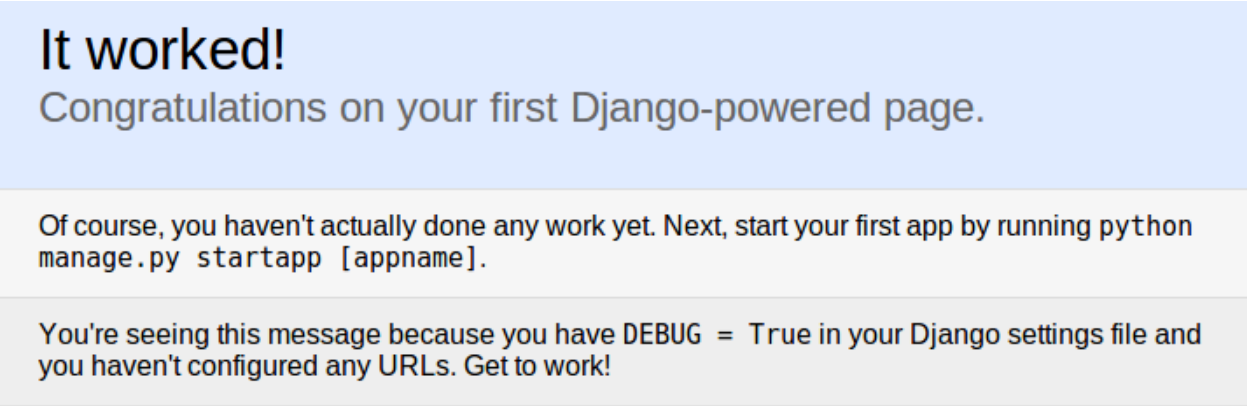

Figure 2.1: Default home page of a Django project

Go back to the terminal window from which you ran the development server and shut it down by pressing `Ctrl+C`.

CHAPTER

THREE

DJANGO MODELS

Author Nick Efford

Contact N.D.Efford@leeds.ac.uk

Status Final

Revised 2015-01-14

Django models are hosted within **apps**, so this worksheet first covers how to create a Django app and then goes on to explore how to create and configure models within that app. Note that we require you to have completed *Getting Started With Django* before you begin these exercises.

You may find it useful to have the Django documentation on models open in a browser while you work.

3.1 Creating & Installing an App

1. In a terminal window, enter `p3` to activate Python 3, then `cd` to the `football` project directory that you created in the *previous exercise* and enter the following command to create an app called `club`:

   ```
   python manage.py startapp club
   ```

   Enter the command `ls` to see the directory that has been created. Enter `ls club` to see the contents of this directory.

2. Creating an app doesn't make your project aware of its existence! For that, you need to *install* the app. To do this, edit `settings.py` in the `football` subdirectory of the project and modify the `INSTALLED_APPS` tuple, adding the name of the app to it so that it resembles the code below. (The added line is highlighted.)

   ```
   INSTALLED_APPS = (
       'django.contrib.admin',
       'django.contrib.auth',
       'django.contrib.contenttypes',
       'django.contrib.sessions',
       'django.contrib.messages',
       'django.contrib.staticfiles',
       'club',
   )
   ```

3.2 Creating a Simple Model

1. Edit `models.py` in the `club` app and replace the comment line with the class definition below.

   Don't copy and paste this code! The act of typing it in will help you become more familiar with Django.

```
class Club(models.Model):
    """Model for a football club."""

    name = models.CharField(max_length=30)
    year = models.IntegerField("year established")
    ground = models.CharField(max_length=30)
    capacity = models.IntegerField(null=True, blank=True)
    website = models.URLField(max_length=50, blank=True)
```

Note the use of `max_length`. This is required for text-based fields. The `year` field demonstrates another feature: the provision of a verbose name for a field. Django will use this in certain contexts, e.g. the admin interface. Note also the `null=True` and `blank=True` options. The meaning of these is explained fully in the field options section of the model field reference.

Make sure that have saved your changes to `models.py` before proceeding.

2. When a model is created or it changes, you need to generate a **database migration** that can make the required changes to the underlying database. Do this now with the following command:

```
python manage.py makemigrations club
```

You should see output like this:

```
Migrations for 'club':
  0001_initial.py:
    - Create model Club
```

You can check the SQL commands that will be run in the database to perform this migration by using the following command:

```
python manage.py sqlmigrate club 0001
```

Note that you are not modifying the database here; modifications only take place when the migration is run.

3. Now perform the migration with the following command:

```
python manage.py migrate
```

To see the effect that this has had in the database, use the `dbshell` management command to run the SQLite command line interface, then use `.tables` to see the database tables. Note the existence of a new table called `club_club` (the name is derived from the app name and the model name).

Use `.schema club_club` to see the SQL used to create the table, then do a query with

```
select * from club_club;
```

(Don't forget the semicolon at the end of this command!)

This should return nothing, as the table currently contains no data. Use `.quit` to quit the SQLite command shell.

4. Django provides other field types that may be more appropriate than some of the choices made in `Club`. For example, `PositiveSmallIntegerField` is more appropriate for the year than `IntegerField`; similarly, `PositiveIntegerField` is a more appropriate choice for the capacity of a football ground. (You can find out more about these and other field types by consulting the model field reference.)

Make these changes now to the `Club` model. You should end up with the code below.

```
class Club(models.Model):
    """Model for a football club."""

    name = models.CharField(max_length=30)
```

```
    year = models.PositiveSmallIntegerField("year established")
    ground = models.CharField(max_length=30)
    capacity = models.PositiveIntegerField(null=True, blank=True)
    website = models.URLField(max_length=50, blank=True)
```

The model has changed, so you also need to generate a new migration for it. Do this with:

```
python manage.py makemigrations club
```

Then apply the migration:

```
python manage.py migrate
```

Note: **Remember**: whenever you add a model, remove a model or change a model's fields, you need to generate migrations for the apps that have changed, using the `makemigrations` management command, and then apply these migrations, using the `migrate` management command.

3.3 Creating & Deleting Model Instances

1. Use `manage.py` to run a Python interpreter like so:

   ```
   python manage.py shell
   ```

   Then, at the >>> prompt, enter the following commands:

   ```
   >>> from club.models import Club
   >>> c = Club(name="Everton", year=1878, ground="Goodison Park")
   ```

   Variable `c` references a `Club` object. You can access the fields with `c.name`, `c.year`, etc. Try it now.

   What about `c.id`? You will find that it doesn't have a value, because the data for this `Club` object are not in the database yet. To save the data, simply call the `save` method like so:

   ```
   >>> c.save()
   ```

   Check `c.id` again. You should find that it now has a value.

2. Quit the Python interpreter by pressing `Ctrl+D`. Use `manage.py` to run the `dbshell` command again and do the SQL query from the previous exercise. You should now find that the `club_club` table contains a single row. Use the `.quit` command to quit and return to the Linux command prompt.

3. Use `manage.py` to run the Python interpreter again. Then, at the >>> prompt, enter the following commands:

   ```
   >>> from club.models import Club
   >>> c = Club.objects.get(id=1)
   >>> c.delete()
   ```

   Repeat the commands used earlier to query the `club_club` database table. You should find that it is empty once more.

4. Django provides a shortcut method that will create a model instance and then immediately save it to the database. To try it out, use `manage.py` to run the Python interpreter again. Then, at the >>> prompt, enter the following commands:

   ```
   >>> from club.models import Club
   >>> Club.objects.create(name="Everton", year=1878, ground="Goodison Park")
   >>> Club.objects.create(name="Chelsea", year=1905, ground="Stamford Bridge")
   ```

Keep the Python interpreter running for the next exercise.

3.4 Adding a Method to a Model

1. Enter the following at the >>> prompt:

   ```
   >>> Club.objects.all()
   ```

   This retrieves details of all the clubs currently in the database, returning these details as a sequence of model instances. You should see the following output displayed:

   ```
   [<Club: Club object>, <Club: Club object>]
   ```

   A default string representation of `Club` objects is being used here, and clearly it isn't very informative.

2. We can improve matters by adding a custom `__str__` method to the model. This method will be called whenever a string representation of a `Club` object is needed.

   Quit the interpreter with `Ctrl+D`. Edit `models.py` and add the following method to the `Club` model, taking care to use the correct indentation level:

   ```
   def __str__(self):
       return self.name
   ```

   Save `models.py`. Note that it isn't necessary to generate a migration here, because this change is unrelated to the underlying database. (If you try using the `makemigrations` management command, Django will tell you that it hasn't detected any database-relevant changes.)

3. Restart the Python interpreter using the `shell` management command and enter the following commands at the ``>>> prompt:

   ```
   >>> from club.models import Club
   >>> Club.objects.all()
   ```

   The output should now look like this:

   ```
   [<Club: Everton>, <Club: Chelsea>]
   ```

   Notice how the clubs are displayed in the order in which they were added to the database, not in alphabetical order of name.

3.5 Working With Model Metadata

1. To fix the ordering issue noted at the end of the last exercise, we need to add some **metadata** to the model.

   Quit the Python interpreter with `Ctrl+D` and edit `models.py` once more. Add to it the following code, again taking care to use the correct indentation level:

   ```
   class Meta:
       ordering = [ "name" ]
   ```

   This specifies that the default ordering of `Club` objects should be based on the `name` field.

   This change could affect the database, so you will need to generate and apply a migration for it after saving `models.py`.

2. Restart the Python interpreter using the `shell` management command and enter the following commands at the >>> prompt:

```
>>> from club.models import Club
>>> Club.objects.all()
```

   You should now see this:

```
[<Club: Chelsea>, <Club: Everton>]
```

See the Model Meta options page of the Django documentation for further information on how to specify metadata.

CHAPTER

FOUR

THE ADMIN INTERFACE

Author Nick Efford

Contact N.D.Efford@leeds.ac.uk

Status Final

Revised 2015-01-14

This worksheet covers some of the features of Django's administrative web interface, which Django generates automatically from your models. Note that completion of *Django Models* is a prerequisite for these exercises.

4.1 Viewing The Admin Interface

1. Open a terminal window and enter `p3` to activate Python 3. Then `cd` to the `football` project directory and run Django's development web server with the following command:

   ```
   python manage.py runserver
   ```

2. In a browser, visit `http://localhost:8000/admin/`. You will see a login screen:

   Log in with the username and password that you specified when you first synchronised with the database. (You did remember it, didn't you? :-) This will take you to a Site Administration page displaying links to admin pages for any models that have been registered with the admin interface. The links are grouped by the app to which the models belong. Initially, you should be able to see links for the `Group` and `User` models of Django's built-in authentication app.

3. Click on the 'Users' link. This will take you to a page listing all registered users of the site. You can add new users and edit or delete existing users from this page.

   Spend a few minutes exploring the features of the Users admin page, then log out of the admin interface and shut down the development server by moving into the terminal window in which it is running and pressing `Ctrl+C`.

4.2 Enabling Admin For a Model

1. Edit the file `admin.py` in the `club` directory. This module is where you must register models of the `club` app with the admin interface. Modify the file so that it looks like this:

   ```
   from django.contrib import admin
   from club.models import Club

   admin.site.register(Club)
   ```

   Save your changes.

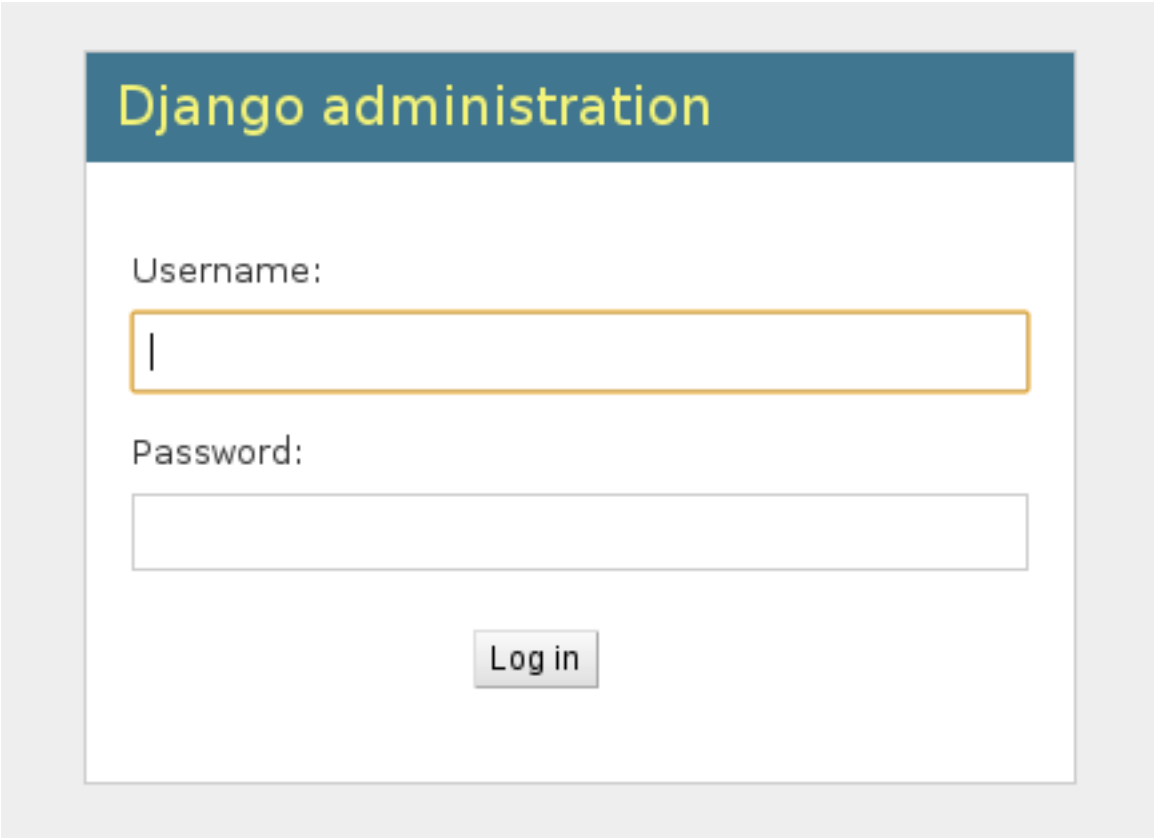

Figure 4.1: Django's admin login screen

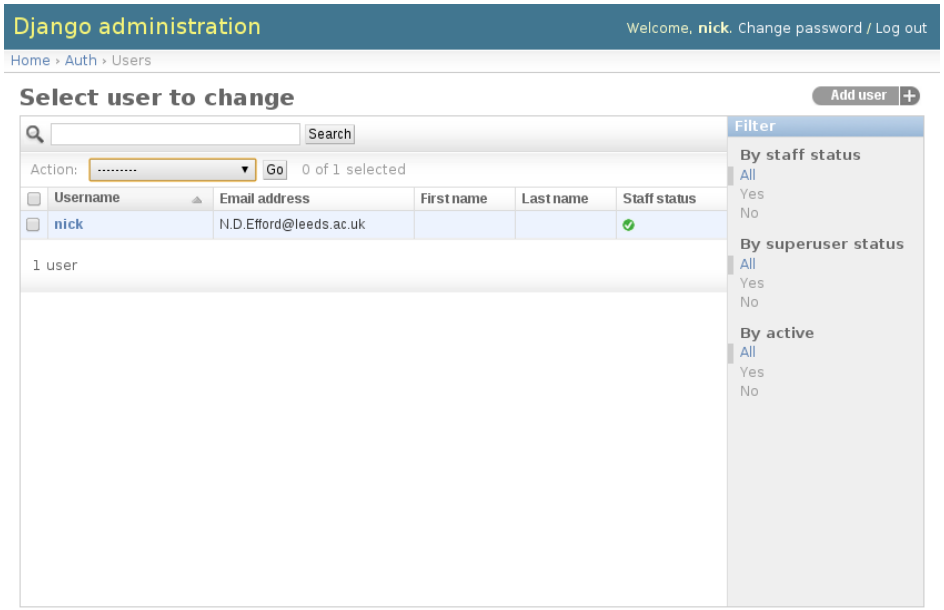

Figure 4.2: Django's user admin page

2. Run the development server again. Visit the admin URL and log in. The Site Administration page should now show the `club` app and a link to the admin page for the `Club` model. Click on this link and you will see a list of clubs currently stored in the database. There should be just two: Chelsea and Everton. Notice how they are ordered - reflecting the metadata options defined inside the `Club` model.

3. Click on the 'Chelsea' link. This will take you to an edit page for this particular club:

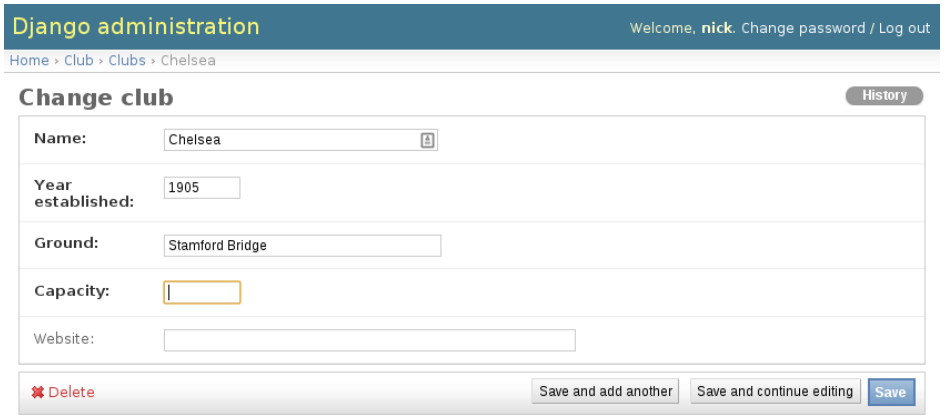

Figure 4.3: Admin page used to edit club details

   Enter `41798` for capacity and `http://www.chelseafc.com` for the website, then click the *Save* button.

4. Go back to the admin page for the `Club` model. Tick the checkboxes next to both clubs and choose 'Delete selected clubs' from the drop-down box of available actions. Then click the *Go* button. Confirm the deletion by clicking *Yes I'm sure*. (Don't worry - you'll be repopulating the database in the next exercise!)

5. Go back to the main Site Administration page. Notice the box on the right. Django's admin interface keeps track of the actions you perform using it and lists them here as a handy 'audit trail'.

   Log out of the admin interface and shut down the development server, just as you did at the end of the previous exercise.

4.3 Admin Customisation

We will investigate admin customisation by introducing a new app and new model into the project.

1. Create a new app called `match` like so:

   ```
   python manage.py startapp match
   ```

   Edit `settings.py` and add `'match'` to the `INSTALLED_APPS` list.

2. Download the file `models.py` into the `match` directory, overwriting the empty version of this file that Django creates for you. Take a moment to examine the `Match` model defined in this file. Then generate and apply a migration for this new model with the following pair of commands:

   ```
   python manage.py makemigrations match
   python manage.py migrate
   ```

3. Download the file `data.json` into the `football` project directory (the directory containing `manage.py`). This is a JSON-formatted file containing data for twenty clubs and a full season of matches played by those clubs [1].

[1] The data in this case are for the 2012-13 season of the English Premier League. The original data source was http://www.football-data.co.uk/.

To import the data enter the following command in the terminal window:

```
python manage.py loaddata data.json
```

If all is well, you should see this message displayed:

```
Installed 400 object(s) from 1 fixture(s)
```

4. Check the database by running the `dbshell` management command and then trying the following pair of SQL queries:

```
select * from club_club;
select * from match_match;
```

Enter `.quit` to leave the SQLite command shell.

5. Edit `admin.py` **in the match app**. Alter the code so that it looks like this:

```
from django.contrib import admin
from match.models import Match

admin.site.register(Match)
```

Save your changes, then run the development server and access the admin site for the project. You should see a new box on the home page, for the `match` app. Click on the 'Matches' link and you will see details of all 380 matches listed. Leave the development server running, ready for the next step.

6. Edit `admin.py` in the `match` app again. Alter the code so that it looks like this:

```
from django.contrib import admin
from match.models import Match

class MatchAdmin(admin.ModelAdmin):
    list_per_page = 20

admin.site.register(Match, MatchAdmin)
```

Save your changes. Check the terminal window in which the development server is running and you should see that it has detected the change and, as a result, has restarted automatically.

Back in the browser, reload the 'change list' for the matches. It should now be paginated, with details of twenty matches on each page.

Note: If nothing seems to have changed, you may need to shut down and restart the development server manually before reloading the page.

---

7. Add the following customisations to the `MatchAdmin` class in `admin.py`. Make sure that the indentation matches the existing contents of the class. As you add each of these settings, look up an explanation of what it does in the Django admin site documentation.

```
list_filter = ["home_goals", "away_goals"]
search_fields = ["home_club__name", "away_club__name"]
date_hierarchy = "date"
```

Save your changes, wait for the server to restart, then reload the page in your browser. You should see something like this:

Experiment with filtering by goals scored and with the search option (use football club names for the latter). Use the links at the top of the screen to drill down by date.

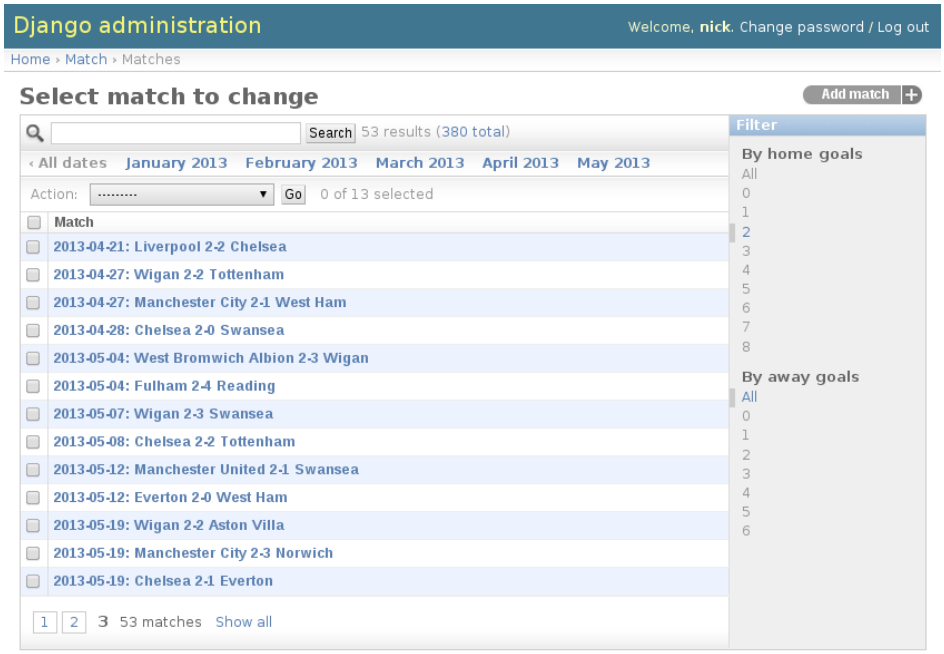

Figure 4.4: Customised change list for matches

8. Try customising the admin for `Club`. For example, try using the `list_display` setting to display club name, year established and ground on the 'change list' page for clubs.

   Log out of the admin interface and shut down the server with `Ctrl+C` when you are done.

CHAPTER

FIVE

QUERYSETS & THE MODEL API

Author Nick Efford

Contact N.D.Efford@leeds.ac.uk

Status Final

Revised 2015-01-14

This worksheet deals with how you query the database of a web application using Django models. Note that completion of *The Admin Interface* is a prerequisite for these exercises.

You may wish to refer to the material from Lecture 2-05. You may also find the official Django documentation on making queries and the QuerySet API useful here.

5.1 Retrieving Single Instances

1. Open a terminal window and `cd` into the `football` project directory used for previous exercises. Run a Python shell like so:

   ```
   python manage.py shell
   ```

2. Import the `Club` model and then retrieve the details for Reading FC by entering the following commands:

   ```
   >>> from club.models import Club
   >>> c = Club.objects.get(name="Reading")
   ```

   Enter `c.id` to examine the object's `id` attribute. Do the same for `ground`, `capacity` and `website`.

3. Try using the same technique to retrieve the details for Bolton. What happens?

4. Now try the following:

   ```
   >>> c = Club.objects.get(name__startswith="M")
   ```

   What happens?

   The key thing to remember here is that `get` expects there to be exactly one object that matches the query; if no object or more that one object matches, you will get an exception.

   Note: Tip

   When you are writing views for your app, use Python's `try` and `except` blocks to enclose uses of `get`. This will allow your app to deal gracefully with the consequences of a failed query.

5. The `get_or_create` method is useful for cases where you want to look up an object and create that object if it isn't found. Try this now by entering the following commands:

```
>>> Club.objects.get_or_create(name="Arsenal")
>>> Club.objects.get_or_create(name="Leeds", year=1919, ground="Elland Road")
```

   Notice how `False` is returned with the `Club` object in the first example - indicating that Arsenal's details are already in the database. The second example returns `True` alongside the `Club` object - indicating that Leeds' details were *not* already in the database but have now been added.

5.2 Working With All Instances

1. Call the `all` method on a model's manager to retrieve all model instances. Try this now:

```
>>> Club.objects.all()
```

2. What if you don't need to access the fields of any objects and simply need to know how many of them there are? For that, you should use `count` instead of `all`. Try it now.

   Note: Tip

   Using `count` is more efficient than using a combination of `all` and Python's built-in `len` function.

3. The `order_by` method called on the manager will also give you all model instances, but with an ordering that you specify rather than the default ordering or the ordering specified by model metadata.

   Try these out in the Python shell:

```
>>> Club.objects.order_by("year")
>>> Club.objects.order_by("-year")
```

   What is the significance of the "-" in the second example?

5.3 Using `filter` & `exclude`

1. Use `filter` on the manager of `Club` to find all the clubs whose name begins with 'S'.
2. Use `filter` on the manager of `Club` to find all the clubs whose name contains the letter sequence "ham".
3. Use `filter` to find all the clubs established after 1900.
4. Modify the previous query so that it finds clubs established between 1901 and 1910.

   (Note: there are at least two different ways of doing this!)
5. Count all the clubs who do *not* play at a ground whose name contains the word "Stadium".

   (Hint: `count` can be called on a `QuerySet` as well as on a model's manager.)
6. Find all the clubs with names that do *not* begin with 'S' that were established in 1880 or earlier.

   (Hint: you can chain calls to `filter` or `exclude`.)

5.4 Navigating Model Relationships

Before trying any of the following queries, import the `Match` model like so:

```
>>> from match.models import Match
```

1. Without querying the `Club` model directly, find all the matches that Swansea played at home.

   (Hint: take a look at `match/models.py` if you need to remind yourself of the fields in the `Match` model. Remember that we can navigate the relationship from one model to another by using __ in a query parameter.)

2. Modify the previous query so that it finds all the matches that Swansea played at home in which they scored more than two goals.

3. Count how many matches were played between clubs that were both established after 1900.

4. Modify the previous query so that it limits the count to matches played in 2013.

5. Use `get` on the `Club` model to retrieve the object representing Stoke City FC:

   ```
   >>> stoke = Club.objects.get(name="Stoke")
   ```

   Then use this `stoke` object to find the matches that Stoke played at home that ended up as goalless draws.

   (Hint: Django creates a way of navigating many-to-one foreign key relationships in the reverse direction, from the 'one' side to the 'many' side. It generates a 'related name' field in the model on the 'one' side - `Club`, in this case. You can find the name of this field by looking for the `related_name` parameter in the `Match` model. You can call `all`, `filter`, etc, on this 'related name' field.)

5.5 Aggregation

Django supports the generation of **aggregate values** from queries. An example of an aggregate would be the average value for a numeric field in a Django model.

Study the documentation on aggregation, then, in your Python shell, execute queries against the `Match` model that will tell you the average number of goals scored at home and the average number of goals scored away from home.

Do your results tend to support or undermine the idea that teams tend to win their home matches more often than their away matches?

CHAPTER

SIX

URLCONFS, VIEWS & TEMPLATES

Author Nick Efford

Contact N.D.Efford@leeds.ac.uk

Status Final

Revised 2015-01-14

This worksheet provides a basic introduction to the front-end of a Django web application, comprising view functions, HTML templates and the URL configurations that map URLs onto views. Note that completion of *QuerySets & The Model API* is a prerequisite for these exercises.

You may wish to refer to the material from Lectures 2-06 & 2-07. You may also find the official Django documentation on URLconfs, template syntax and standard tags and filters useful here.

6.1 URL Configuration

1. In a terminal window, `cd` into the `football` project directory that you have been used for the earlier worksheets. Move into the `football` subdirectory, edit `urls.py` and alter it so that it looks like this:

   ```
   from django.conf.urls import patterns, include, url
   from django.contrib import admin

   urlpatterns = patterns('',
       url(r'^club/', include('club.urls')),
       url(r'^admin/', include(admin.site.urls)),
   )
   ```

   (The highlighted line is the one that needs to be added.)

   This URLconf specifies that URLs whose post-domain part begins with `club/` should be handled by the URLconf in the `club` app. This 'two-level' approach is common in Django projects. The idea is that the project-level URLconf passes each request on to the URLconf of the relevant app. This leads to a cleaner project structure and also makes the reuse of apps across multiple projects much easier.

2. Now move into the `club` app directory and create a file `urls.py` that looks like this:

   ```
   from django.conf.urls import patterns, url
   from club.views import club_list

   urlpatterns = patterns("",
     url(
       regex=r"^$",
       view=club_list,
       name="club_list"
   ```

```
  ),
)
```

This URLconf specifies that a URL in which nothing follows `club/` should be handled by a view called `club_list`.

6.2 A Simple View & Template

1. Edit `views.py` in the `club` app and implement the `club_list` view specified in the URLconf. The code you need for this can be found on Slide 12 of Lecture 2-07.

   Note: type in the code - do not attempt to copy and paste it!

2. Create a subdirectory for templates inside the `club` app, using the following command:

   ```
   mkdir -p templates/club
   ```

   Then create the template `club_list.html` inside this newly-created directory. The code that you need for the template can be found on Slide 13 of Lecture 2-07.

3. Move back up to the project directory and run the development server:

   ```
   python manage.py runserver
   ```

   In a browser, visit `http://localhost:8000/club/`. You should see a list of clubs displayed.

6.3 Adding a View & Template For Club Details

1. Edit `views.py` in the `club` app and create a view called `club_detail`. Your function should take two parameters, called `request` and `id`, respectively. The latter represents the unique numeric identifier of a `Club` object. Django generates this field for you automatically and uses it as the primary key of the database table that sits behind `Club`.

   The body of your function should

   - Look up the `Club` object with the given `id` value, using `Club.objects.get`
   - Store that object in a context dictionary
   - Call the `render` function on template `club/club_detail.html` and the context

   Use the code for `club_list` as a guide to the overall structure of the view.

2. In the `templates/club` subdirectory, create a new HTML template called `club_detail.html`. Write the template so that it will look something like this when rendered and viewed in a browser:

   Note the display of "41,798" rather than 41798 in the example above. If you want the comma in your rendered page, you will need to use the `intcomma` filter - see the documentation on humanization for more details.

   If you are stuck on how to proceed, Slide 10 of Lecture 2-07 gives a big hint on how filters are used!

3. Finally, edit `urls.py` and add a new URL mapping to the `urlpatterns` list. Follow the format used for the `club_list` view. The regular expression you will need is this:

   ```
   r"^(?P<id>\d+)/$"
   ```

   Let us deconstruct this, from the inside out:

   - The `\d` matches a digit and the `+` means 'one or more of', so `\d+` matches one or more digits.

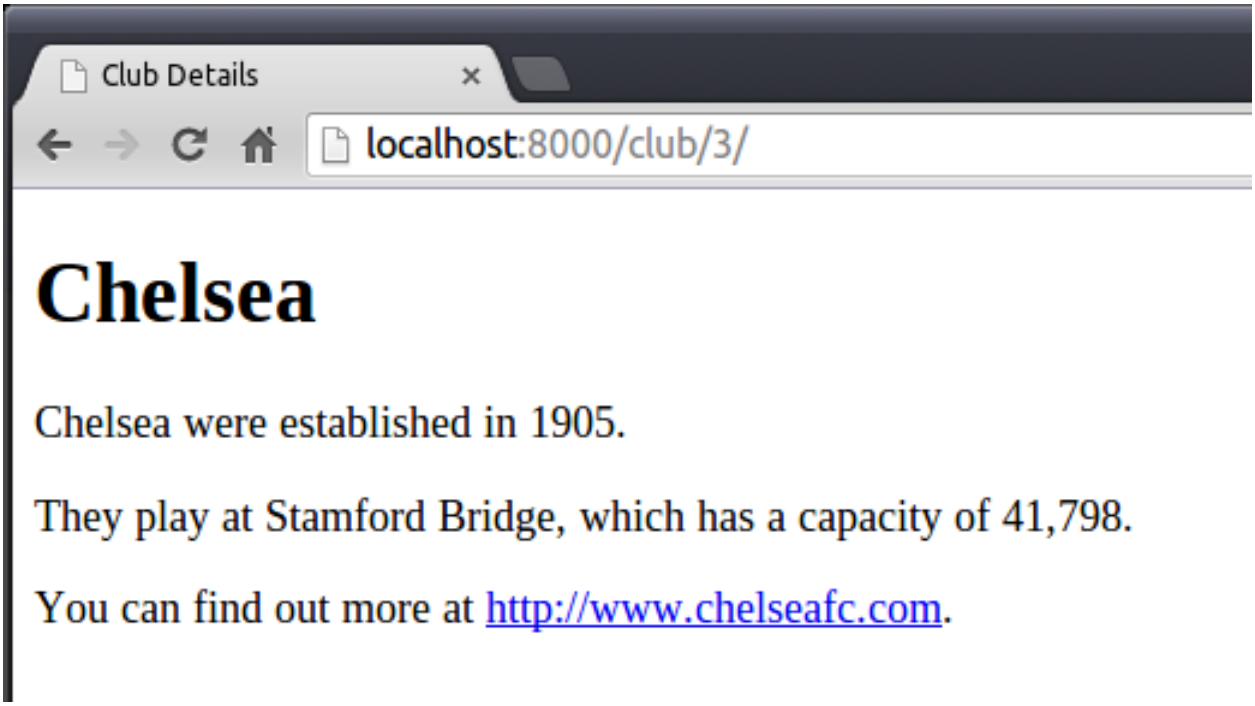

Figure 6.1: Page returned by the 'club detail' view

- The enclosing `()` ensure that this part of the match is captured for future use.
- The `?P<id>` part inside the `()` ensures that the matching text can be referenced as `id`. (Strictly speaking, you don't need to include this here.)
- The `^` and `/$` enclosing the matched digits ensure that these digits must be immediately preceded by the `club/` part of the URL and that they must be immediately followed by a `/`, with nothing after it.

Save your changes and then run the development server. Switch to a web browser and visit `http://localhost:8000/club/1/`. This should give you details of the club with an ID of 1. Try several other numbers in the range 1-21. Try 22 or a larger number to see the error that is produced, then try something non-numeric.

4. The `DoesNotExist` exception produced for non-existent clubs is not helpful in a real web application. It would be preferable to either intercept the exception and display a custom error page or issue a 404 Page Not Found response. Django makes doing the latter straightforward.

   Edit `views.py` again and modify the `django.shortcuts` import so that it looks like this:

   ```
   from django.shortcuts import render, get_object_or_404
   ```

   Then modify the query so that it looks something like this:

   ```
   context = {
     "club": get_object_or_404(Club, pk=id)
   }
   ```

   Rerun the development server and try a club ID greater than 21 again. This time, you should see a 'Page Not Found' error displayed.

6.4 Adding Hyperlinks

It would be useful if the club detail page had a hyperlink to take you back to the club list page. It would be even more useful if each club name on the club list page was a hyperlink taking you to the corresponding club detail page. It is easy to provide these.

1. Edit the `club_detail.html` template and add the following to the bottom of the document body:

   ```
   <p>[ <a href="{% url 'club_list' %}">Club List</a> ]</p>
   ```

   Here, the `url` template tag will generate the URL for URL pattern with the given name. This is a much better and more portable way of doing it than hardcoding a URL in the template.

   Save your changes, restart the server and visit any club detail page. click on the 'Club List' link to check that it performs as expected.

2. Now edit `club_list.html` and modify the list items so that they look like this:

   ```
   <li><a href="{% url 'club_detail' club.id %}">{{ club }}</a></li>
   ```

   We assume here that you have used `club_detail` as the name of the 'club detail' pattern in `urls.py`.

   Note the inclusion of the club's `id` field as an argument. This is necessary because the 'club detail' pattern expects to capture a club's `id` as part of the URL, so that it can be passed to the `club_detail` view. We therefore need to supply an `id` value when we generate the URL using the `url` template tag.

   Save your changes, restart the server and visit the club list page at `http://localhost:8000/club/`. It should now be possible to navigate to club detail pages and back again via hyperlinks.

6.5 Further Work

Here are some other things you can try:

- Modify the `club_detail` so that it deals with the problem of a club not having ground capacity specified - as is the case for Leeds. (Hint: you will need to use an `if` statement in the template.)
- Modify the templates for the `club_list` and `club_detail` views so that they extend a base template containing the code common to both.
- Add a view and template for a page that lists clubs in order of the year they were established.
- Add some views and templates to the `match` app; for example, you could implement a page that lists all the matches where the home team scored more than 5 goals, or a page that lists all the matches for a given club.

CHAPTER

SEVEN

ADDING STATIC RESOURCES

Author Nick Efford

Contact N.D.Efford@leeds.ac.uk

Status Final

Revised 2015-01-14

This short worksheet shows you how static resources (style sheets, images, etc) can be added to a Django project. Note that completion of *URLconfs, Views & Templates* is a prerequisite for these exercises.

For further information on this topic, consult the Django documentation page on managing static files.

Note: We focus here on simple techniques that will allow you to serve static resources from the Django development server, for the purposes of doing the coursework. Note that this approach is not recommended for a real web site; see the Django documentation page on serving static files in production for more on this.

7.1 App-Specific Resources

1. Move into the directory for the `club` app. Then create a subdirectory for app-specific resources with the following command:

   ```
   mkdir -p static/club
   ```

   Inside the `static/club` subdirectory, create a style sheet called `style.css`. In this style sheet, add rules to

   - Set the background colour of the page (note: don't make it black)
   - Set the font family to a suitable sans serif font
   - Set the font size and line height properties
   - Add space at the left margin of the page
   - Change the colour of level 1 headings
   - Change the colour of visited and unvisited hyperlinks

   Add other rules if you wish.

2. Edit your `club_list.html` and `club_detail.html` templates. Add the following tag to both templates, immediately after the DOCTYPE declaration:

   ```
   {% load staticfiles %}
   ```

Then add the following to the `head` element:

```
<link rel="stylesheet" href="{% static 'club/style.css' %}">
```

Run the development server and visit the pages for the `club` app. You should see your style applied to these pages.

7.2 Project-Wide Resources

As with templates, static resources can also be made available across all apps within a project.

1. Move up to the top-level directory of your project (the one containing the app directories, `manage.py`, etc) and create a subdirectory called `static`. Download the image file `football.png` into it.
2. Edit `settings.py` for the project and add the following new setting to the bottom of the file:

   ```
   STATICFILES_DIRS = [
     os.path.join(BASE_DIR, "static")
   ]
   ```

3. Now edit one of the templates in the `club` app and add an `img` element to it - e.g. using code like this:

   ```
   <p><img src="{% static 'football.png' %}" alt=""></p>
   ```

   Restart the development server and visit the page where you added the `img` element. The football image should now be visible on the page.