Pro Django

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
0.5	Dec 2012		SF

Contents

1 About This Course **1**
1.1 Course Description . . . 1
1.1.1 Target Student . . . 1
1.1.2 Course Prerequisites . . . 1
1.2 Course Objectives . . . 1
1.3 Set Up . . . 2

2 Environment and Installation **3**
2.1 Django Developer Environment Setup . . . 3
2.1.1 virtualenv . . . 3
2.2 PRACTICE ACTIVITY . . . 4
2.2.1 pip . . . 4
2.3 PRACTICE ACTIVITY . . . 5
2.3.1 virtualenvwrapper . . . 5
2.4 REVIEW QUESTIONS . . . 5

3 Reviewing the Basics **6**
3.1 Lesson Objectives . . . 6
3.2 Introduction . . . 6
3.3 A Django Project . . . 6
3.3.1 __init__.py . . . 7
3.3.2 settings.py . . . 7
3.3.3 templates . . . 7
3.3.4 urls.py . . . 7
3.4 Django Applications . . . 7
3.4.1 admin.py . . . 7
3.4.2 models.py . . . 7
3.4.3 views.py . . . 8
3.5 Best Practices and the Django tutorial . . . 8
3.6 PRACTICE ACTIVITY . . . 8
3.7 *Reviewing the Basics* Follow-up . . . 10

4 More Tools **11**
4.1 Lesson Objectives . . . 11
4.2 debug_toolbar . . . 11
4.3 PRACTICE ACTIVITY . . . 11
4.4 django_extensions . . . 12
4.5 PRACTICE ACTIVITY . . . 12
4.6 *More Tools* Follow-up . . . 13

5 Data Migrations **14**
5.1 Lesson Objectives . . . 14
5.2 Why migrations? . . . 14
5.3 PRACTICE ACTIVITY . . . 14
5.4 Django database creation support . . . 15
5.4.1 syncdb . . . 15
5.5 Using south . . . 16
5.6 PRACTICE ACTIVITY . . . 16
5.7 *Migrations* Follow-up . . . 17

6 Adding Media **18**
6.1 Lesson Objectives . . . 18
6.2 Static media . . . 18
6.2.1 `STATIC_ROOT` . . . 18
6.2.2 `STATIC_URL` . . . 18
6.2.3 `STATICFILES_DIRS` . . . 19
6.2.4 `STATICFILES_FINDERS` . . . 19
6.3 PRACTICE ACTIVITY . . . 19
6.4 *Adding Media* Follow-up . . . 19

7 Users and Authentication **20**
7.1 Lesson Objectives . . . 20
7.2 Users and Permissions via contrib.auth . . . 20
7.3 Users in Requests . . . 20
7.4 Users in the admin . . . 21
7.5 Login/Logout . . . 22
7.6 PRACTICE ACTIVITY . . . 23
7.7 *Users and Authentication* Follow-up . . . 23

8 Intro to Forms **24**

8.1 Lesson Objectives . . . 24
8.2 Forms . . . 24
8.3 Working with forms in views . . . 24
8.4 Displaying forms in templates . . . 25
8.5 Validation . . . 25
8.6 PRACTICE ACTIVITY . . . 26
8.7 *Intro To Forms* Follow-up . . . 26

9 More Forms **27**

9.1 Lesson Objectives . . . 27
9.2 Field and Widgets . . . 27
9.3 Modifying Form Output . . . 28
9.4 PRACTICE ACTIVITY . . . 28
9.5 Dynamic Choices . . . 29
9.6 PRACTICE ACTIVITY . . . 29
9.7 *More Forms* Follow-up . . . 29

10 Writing Tests with Django **30**

10.1 Lesson Objectives . . . 30
10.2 Running Tests . . . 30
10.3 Writing Tests . . . 31
10.4 PRACTICE ACTIVITY . . . 32
10.5 *Writing Tests* Follow-up . . . 32

11 Customizing Templates **33**

11.1 Lesson Objectives . . . 33
11.2 Template Loading . . . 33
11.3 PRACTICE ACTIVITY . . . 33
11.4 Builtin Tags and Filters . . . 34
11.5 PRACTICE ACTIVITY . . . 34
11.6 Custom template filters . . . 34
11.7 PRACTICE ACTIVITY . . . 35
11.8 Custom template tags . . . 35
11.9 *Customizing Templates* Follow-up . . . 35

12 Even more (Model)Forms **36**
12.1 Lesson Objectives . . . 36
12.2 ModelForms . . . 36
12.2.1 Customizing fields and widgets . . . 36
12.2.2 Saving ModelForms . . . 37
12.3 User Profiles . . . 37
12.4 PRACTICE ACTIVITY . . . 38
12.5 Nicer Forms . . . 38
12.5.1 django-floppyforms . . . 38
12.5.2 django-crispy-forms . . . 38
12.6 PRACTICE ACTIVITY . . . 39
12.7 *Even More (Model)Forms* Follow-up . . . 39

13 Tying it all together: Lab 1 **40**
13.1 Lesson Objectives . . . 40
13.1.1 What We Want . . . 40
13.1.2 How to Get There . . . 40
13.1.3 Real World Data . . . 41
13.2 PRACTICE ACTIVITY . . . 41

14 QuerySets **42**
14.1 Lesson Objectives . . . 42
14.2 QuerySet Basics . . . 42
14.3 Spanning Relationships . . . 42
14.4 Query Operators . . . 43
14.5 PRACTICE ACTIVITY . . . 43
14.6 Aggregation and Annotation . . . 44
14.7 PRACTICE ACTIVITY . . . 44
14.8 *QuerySets* Follow-up . . . 45

15 Performance **46**
15.1 Lesson Objectives . . . 46
15.2 QuerySet Performance Basics . . . 46
15.3 PRACTICE ACTIVITY . . . 46
15.4 Denormalisation . . . 47
15.5 Signals . . . 47
15.6 PRACTICE ACTIVITY . . . 47
15.7 Caching . . . 47
15.8 Practice Activity . . . 48
15.9 *Performance* Follow-up . . . 48

16 Deploying Django Web Apps **49**
16.1 Lesson Objectives . . . 49
16.2 Execution Model . . . 49
16.2.1 Where should my application live? . . . 49
16.2.2 Tradeoffs - and an alternative. . . . 49
16.3 Scripting Deployment with Fabric . . . 50
16.4 Practice Activity . . . 50
16.5 *Deployment* Follow-up . . . 51

17 Using Celery **52**
17.1 Lesson Objectives . . . 52
17.2 Celery and Asynchronous Jobs . . . 52
17.3 Configuring celery for development . . . 52
17.4 PRACTICE ACTIVITY . . . 53
17.5 *Celery* Follow-up . . . 53

18 REST with Tastypie **54**
18.1 Lesson Objectives . . . 54
18.2 REST API's . . . 54
18.3 Consuming our API . . . 55
18.4 PRACTICE ACTIVITY . . . 56
18.5 *REST with Tastypie* Follow-up . . . 56

Chapter 1

About This Course

1.1 Course Description

This course is designed to get new users up to speed on advanced usages and best practices for the Django Framework. Continuing where the Django Tutorial leaves off we will explore advanced usage of Models and Managers, the Form library including ModelForms and FormSets, and many of the built in applications that support web development with Django. We will also explore standard third party tools and applications that enable us to change our database structure over time, debug and profile our apps, and ease our interactions with the built in management shell.

1.1.1 Target Student

The target student has basic levels of experience with Python 2.5 or above and Django 1.4.

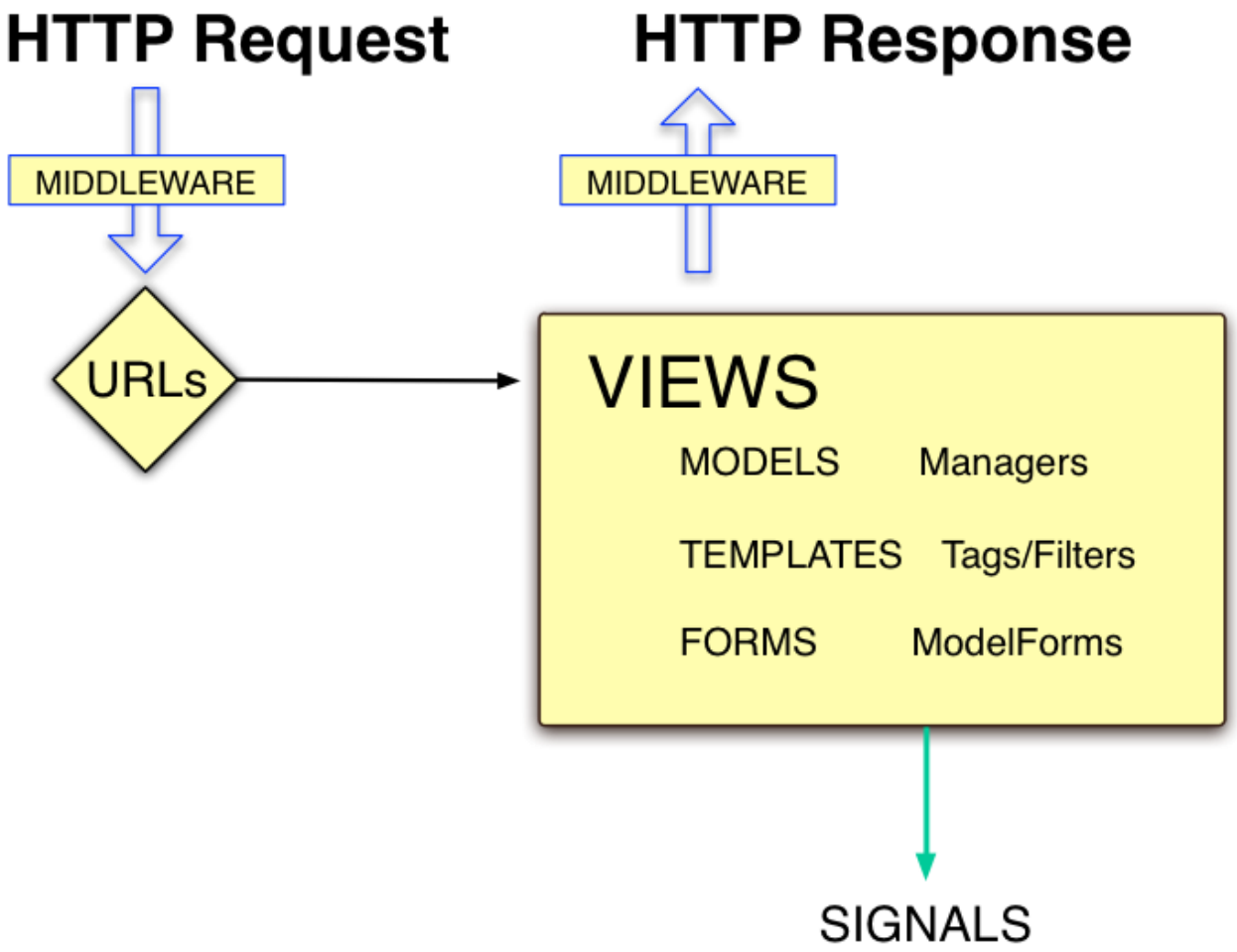
1.1.2 Course Prerequisites

Suggested prerequisites include the Python Fundamentals class at Marakana for basic Python knowledge and the Django Tutorial found at https://docs.djangoproject.com/en/1.4/intro/tutorial01/

1.2 Course Objectives

In this course you will move beyond the basics of Django and gain hands on experience with advanced usage of the components that make up any Django app. You will also gain familiarity with the tools, 3rd party apps, and techniques common to advanced Django projects.

You should be very familiar with all the basic components typically used in web applications:



1.3 Set Up

Install the newest version of Python 2 (2.5 or greater) - Linux and Mac should already have Python installed while Windows users should see http://goo.gl/Y3N9T. We'll install Django together as part of the class.

Be sure to bookmark the documentation at https://docs.djangoproject.com/en/1.4/

The single best way to increase best way to swiftly increase your Django proficiency is to read all of the excellent and comprehensive documentation.

Chapter 2

Environment and Installation

2.1 Django Developer Environment Setup

The Django tutorial and documentation are excellent and every Django programmer should study them thorougly. We'll set up our development environment, however, with tools that let us easily install and manage different versions of Django and any third party apps you need.

2.1.1 virtualenv

The official documentation (https://docs.djangoproject.com/en/1.4/topics/install/#installing-official-release) suggests relying on the included `setup.py` file to install Django:

```
$ tar xzvf Django-1.4.tar.gz
$ cd Django-1.4
$ sudo python setup.py install
```

The disadvantage of this approach is that Django ends up installed globally. The built-in installer uses setuptools and places the django directory containing importable code in your shared system packages directory. On Windows, for example, this might be in `C:\Python2.7\Lib\site-packages`.

Using a single global location to install Python modules means that it is impossible to to have more than one version of Django installed. It also means that you have to have administrative permissions to install Python modules. Fortunately there's a better way to manage installation of Python modules.

virtualenv

```
'virtualenv' is a tool to create isolated Python environments.
```

Ian Bicking's tool virtualenv allows us to create Python environments that are isolated from one another. It supports all three major OS platforms (Windows/Mac/Linux) and has become a defacto standard with integration in other tools such as mod_wsgi and the pip installer.

To install virtualenv we can use easy_install, the installer provided by setuptools. This should be automatically installed with Python on Mac and Linux but Windows users may have to download and run the setuptools installer for your version of Python - you can find it at http://pypi.python.org/pypi/setuptools

Installing virtualenv

```
$ easy_install virtualenv
```

`virtualenv` works by creating a Python environment, copying or symnlinking your Python executable and creating a local directory in which to install Python modules. This is best demonstrated by example:

Let's make a virtualenv

```
$ cd ~
$ mkdir pro_django
$ cd pro_django/
$ virtualenv PROJ1
New python executable in PROJ1/bin/python
Installing setuptools.............done.
$ which python              # The virtual env is not yet activated
/usr/bin/python
$ source PROJ1/bin/activate # Activate using the source command
(PROJ1)$ which python       # Notice the prompt indicates the active env
/home/simeon/pro_django/PROJ1/bin/python
```

The `virtualenv` command is used to create a virtualenvironment in the directory specified. Virtualenvironments are just directories with a Python environment - they can be activated by running the activate script using the `source` command on the activate script on Mac/Linux or by running the `activate.bat` on windows.

The prompt is modified to show the currently active virtualenvironment and now running commands like `python` or `easy_install` runs a local copy in the `bin` directory of the virtualenvironment instead of the global shared executable.

Install Django in a virtualenv

```
(PROJ1)$ easy_install django==1.4 Searching for django==1.4 Reading
http://pypi.python.org/simple/django/ Reading
http://www.djangoproject.com/ Best match: Django 1.4 Downloading
http://media.djangoproject.com/releases/1.4/Django-1.4.tar.gz
Processing Django-1.4.tar.gz Running Django-1.4/setup.py -q bdist_egg
--dist-dir /tmp/easy_install-zp16WQ/Django-1.4/egg-dist-tmp-ZJdjNy
zip_safe flag not set; analyzing archive contents...  ... snip output
...  Installed
/home/simeon/pro_django/PROJ1/lib/python2.6/site-packages/Django-1.4-py2.6.egg
Processing dependencies for django==1.4 Finished processing
dependencies for django==1.4 $ which django-admin.py
/home/simeon/pro_django/PROJ1/bin/django-admin.py
```

Notice that easy_install was not run with administrative permissions and Django 1.4 was installed in the site-packages directory of the currently activated virtualenv. Not only is the Python code put in the right place but utilities like the django-admin.py site creation tool are put in a `bin` folder in the currently activated environment as well.

2.2 PRACTICE ACTIVITY

1. Lets make sure your environment is all set up. Go ahead and install setuptools if on Windows - other OS's should already have setuptools installed. You can check by running the `easy_install` command in your shell - if you've got it your already have setuptools installed.
2. Install virtualenv using easy_install. Create a folder for the class called `pro_django` in your Desktop, Documents, or other easily accessible location.
3. Create a virtualenv called PROJ1. Install Django 1.4 in it - either by using easy_install or by adding a symlink that places the `django` folder in the site-packages directory of your virtualenv.

2.2.1 pip

Using easy_install works for installing packages but other installers provide additional features. Another Ian Bicking project named `pip` is widely used by Django developers. We'll look at its features in greater detail later on but for now note that pip is a drop in replacement for easy_install that also supports

- installing directly from source control (like github)

- uninstalling packages
- listing installed packages and reinstalling from a list

and of course `pip` also plays well with virtualenv. To use pip go ahead and easy_install it, then just use it to install python packages:

```
(PROJ1)$ pip install django_debug_toolbar
Downloading/unpacking django-debug-toolbar
  Downloading django-debug-toolbar-0.9.4.tar.gz (150Kb): 150Kb downloaded
  Running setup.py egg_info for package django-debug-toolbar
    no previously-included directories found matching 'example'
Installing collected packages: django-debug-toolbar
  Running setup.py install for django-debug-toolbar
    no previously-included directories found matching 'example'
Successfully installed django-debug-toolbar
Cleaning up...
(PROJ1)$ python
Python 2.6.5 (r265:79063, Apr 16 2010, 13:09:56)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import debug_toolbar
>>> debug_toolbar.__file__
```

2.3 PRACTICE ACTIVITY

1. Go ahead and get the first set of lab files that we'll need. You can download a sample completion of the Django tutorial by visiting the author's github account at https://github.com/simeonf/django-project/archive/tutorial.zip
2. Extract the .zip file to the pro_django folder you made. Enter the `mysite` directory. Activate the `PROJ1` virtualenv and run the syncdb and shell commands via manage.py to make sure the sample Django app is working.
3. Use pip to install the django-debug-toolbar in your virtualenv. Import the module `debug_toolbar` and look at it's __file__ attribute to see where the importable Python module lives.

2.3.1 virtualenvwrapper

Doug Hellman's virtualenvwrapper is a set of helper commands for managing virtualenvs. It is written in bash so unfortunately is not available for Windows - although Windows users might check out https://github.com/davidmarble/virtualenvwrapper-win for an implementation in batch scripts.

The idea behind virtualenvwrapper is to store all virtualenvs in a specific place (by default `~/.virtualenv` but configurable) and add utility commands for creating, activating, manually adding locations, and listing the contents of a particular virtualenv. We won't be covering installation and usage of virtualenvwrapper in this course as it doesn't modify the functionality of the underlying `virtualenv` tool, but many Python hackers find virtualenvwrapper an indispensable part of their toolkit to manage their development environment.

2.4 REVIEW QUESTIONS

1. What is virtualenv? What problems does it solve.
2. What is pip? What problems does it solve?

Chapter 3

Reviewing the Basics

3.1 Lesson Objectives

This lesson will review the basic components of Django by reviewing the code required to complete the tutorial. We'll also look at a few places where the tutorial's advice could be improved with best practices common to many Django projects.

3.2 Introduction

In this course you will move beyond the basics of Django and gain hands on experience with advanced usage of the components that make up any Django app. You will also gain familiarity with the tools, 3rd party apps, and techniques common to advanced Django projects.

Let's start with what is hopefully familiar ground - the Django tutorial. Many Django programmers get introduced to Django by walking through the four part tutorial before diving deeper into the excellent documentation. Let's review the basic compenents of the Django framework by looking at the code required to complete the Django tutorial for Django v1.4. (See https://docs.djangoproject.com/en/1.4/intro/tutorial01/)

3.3 A Django Project

If you haven't ever completed the Django tutorial now would be the time to start. Your instructor will step you through the 4 part tutorial - go ahead and start at https://docs.djangoproject.com/en/1.4/intro/tutorial01/

Alternatively you may already have downloaded a sample completion from the author's github account at https://github.com/-simeonf/django-project/archive/tutorial.zip - if you've previously completed the tutorial your instructor may just have you start from this sample completion. Go ahead and save the .zip file to your `pro_django` directory and extract it. The file listing should look something like:

```
(PROJ1)$ ls -l
total 12
-rwxr--r-- 1 simeon simeon  249 2012-12-10 15:19 manage.py
drwxr-xr-x 2 simeon simeon 4096 2012-12-10 15:36 mysite
drwxr-xr-x 2 simeon simeon 4096 2012-12-10 15:36 polls
```

This is the root of your Django project. Let's look at the project first:

```
(PROJ1)$ cd mysite
(PROJ1)$ ls -l
total 72
-rw-r--r-- 1 simeon simeon     0 2012-03-03 15:39 __init__.py
-rw-r--r-- 1 simeon simeon  4956 2012-03-03 15:53 settings.py
```

```
drwxr-xr-x 3 simeon simeon  4096 2012-03-03 15:54 templates
-rw-r--r-- 1 simeon simeon   476 2012-03-03 15:59 urls.py
-rw-r--r-- 1 simeon simeon  1134 2012-03-03 15:59 wsgi.py
```

3.3.1 __init__.py

This file tells Python that this directory is a module that can be imported. Modules in Python are a file or a directory that contains a file called `__init__.py`

3.3.2 settings.py

The settings.py contains the configuration information for our Django project - database settings, installed applications, middleware, email settings, etc.

3.3.3 templates

This is a directory to hold our custom templates. By default this directory is usually called `templates` because Django looks for a directory called `templates` in our app directories when we have the app_directories template loader enabled. This templates directory is our main template directory for our project and the absolute path must be specified in our `settings.py`.

3.3.4 urls.py

The top level urls.py contains mapping for path prefixes to views or to other url files.

3.4 Django Applications

Let's also take a peek inside our app. We'll look only at the most significant files for now:

```
(PROJ1)$ ls -l polls
total 24
-rw-r--r-- 1 simeon simeon  587 2012-03-03 15:49 admin.py
-rw-r--r-- 1 simeon simeon    0 2012-03-03 15:41 __init__.py
-rw-r--r-- 1 simeon simeon  529 2012-03-03 15:50 models.py
drwxr-xr-x 3 simeon simeon 4096 2012-03-03 16:55 templates
-rw-r--r-- 1 simeon simeon  383 2012-03-03 15:41 tests.py
-rw-r--r-- 1 simeon simeon  984 2012-03-03 20:05 urls.py
-rw-r--r-- 1 simeon simeon 1623 2012-03-03 20:07 views.py
```

3.4.1 admin.py

`admin.py` is the file that determines how our poll shows up in the built-in Django admin application.

3.4.2 models.py

`models.py` contains our models. Models use a declarative syntax to represent database tables and allow us to perform queries on our data using Django's ORM. Models are the "mandatory" part of a Django application but an app can have an empty models.py.

3.4.3 views.py

Django follows a variant of the Model-View-Controller (MVC) pattern - preferring the acronym MTV for Model, Template, and View. Views are simply python callables that accept an HTTP request object and return an HTTP response object. From the tutorial's point of view most of our programming happens in views and views are always python functions. As we'll see this isn't always the best practice.

3.5 Best Practices and the Django tutorial

Let's start looking at some places we might want to diverge from the tutorial a little bit to make our life easier as we tackle real-world development tasks.

The default settings.py that the startproject generates for us has a sample template_dirs configuration in a comment

```
TEMPLATE_DIRS = (
    # Put strings here, like "/home/html/django_templates" or "C:/www/django/templates".
    # Always use forward slashes, even on Windows.
    # Don't forget to use absolute paths, not relative paths.
)
```

TEMPLATE_DIRS paths need to be absolute paths. But in practice we may not want to hard code our template directory and require that other users of our code have the exact same path structure we do. **We want to build relocatable projects with reusable applications whenever possible.** One common way to solve this problem is to include the os module and use the builtin variable __file__ to find the parent directory of our settings file. Our settings file might look like this:

```
from os.path import abspath, dirname, join

DIR = dirname(abspath(dirname(__file__)))

... snip ...

TEMPLATE_DIRS = (
    join(DIR, "templates"),
)
```

The same trick also works to relatively specify the MEDIA_ROOT that will hold uploaded files. Later on we'll discuss the best ways of maintaining multiple settings files for different environments.

It's also worth noting that the tutorial initially has you specify all your urls in the main `urls.py` file. In part three we move all the urls that start with a common prefix ("/poll/") into the app level url.py and use include in the main urls.py. This is good but the comments in the generated urls.py may mislead us at this point:

```
urlpatterns = patterns('',
    # Examples:
    # url(r'^$', 'mysite.views.home', name='home'),
    # url(r'^mysite/', include('mysite.foo.urls')),
    )
```

If we followed the advice in the comment we'd end up including the project name "mysite" in the include string. This may work but far better is to simply refer to the application without the project name. Again - we want our projects to be movable (from development to production servers, for instance) and we want our applications to be re-usable. **Never use the project name in an import statement.**

3.6 PRACTICE ACTIVITY

The tutorial didn't teach about Django's template inheritance patterns and our poll pages currently aren't complete html pages. Let's go ahead and make our templates a little more realistic.

1. Make sure your polls application is working: add a poll via the admin and visit http://localhost:8000/polls/ to make sure it shows up.
2. Take a few minutes to read the Django documentation on template inheritance at https://docs.djangoproject.com/en/1.3/-topics/templates/#id1
3. Let's add a base template that represents the design of our site. We're not too concerned with design but let's make a simple html page with a Django block called content where we want individual apps to put their output. This should go in mysite/templates/base.html. This template might look like:

```
<html>
  <head>
  <title>Pro Django Class - {% block title %}{% endblock %}</title>
  </head>
  <body>
    <h1>Header</h1>
  {% block content %}
  {% endblock %}
    <h3>Footer</h3>
  </body>
</html>
```

4. Let's also add a application base template. An extremely common design pattern is that we want to share a template between all the pages of our app - perhaps to show things like breadcrumb navigation between our different pages. Also - our application may not know that our project base template defines a block called content - so the application base template is where we can configure how our application templates will connect to our project templates. This template might look like:

```
{% extends 'base.html' %}
{% block content %}
 {% block poll %}
 {% endblock %}
{% endblock %}
```

5. Edit the three existing templates in `mysite/polls/templates/polls` to extend the application base template. You ought to end up with something like this:

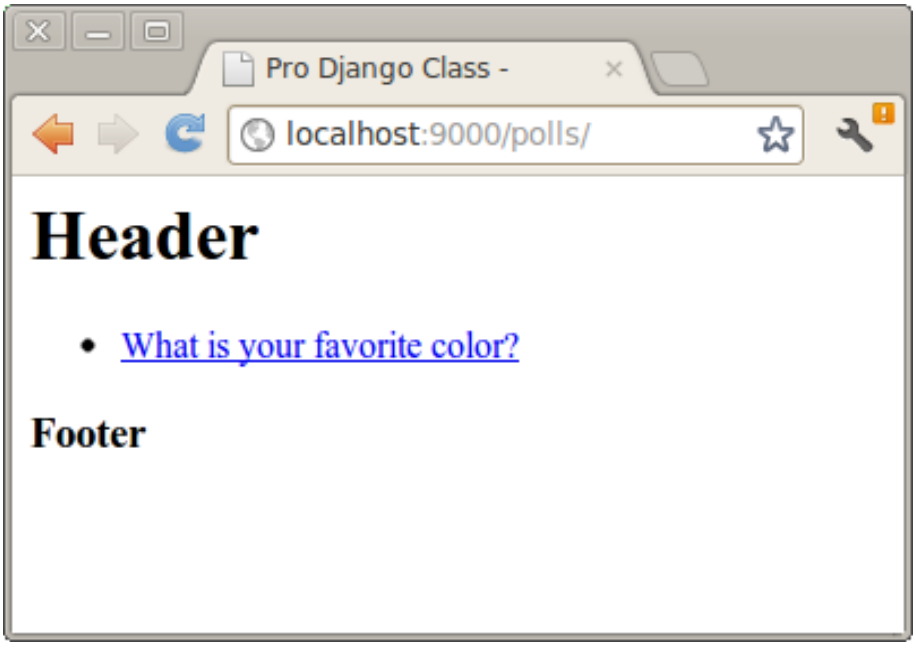


Figure 3.1: Base Template

3.7 *Reviewing the Basics* Follow-up

You should be fairly comfortable and familiar with all the material we've covered so far - mapping urls to views and using models are basic Django concepts. Building on the tutorial, we should understand Django's template inheritance model. We're now ready to begin building a more complex Django application.

Chapter 4

More Tools

4.1 Lesson Objectives

This lesson will review some commonly used 3rd party Django applications. We'll roughly profile our execution speed, list our sql queries, view the templates used to inspect a page and the contents of the template contexts they rendered.

We'll also look at a few tricks to ease writing views and interacting with the shell.

4.2 debug_toolbar

We've already installed the django-debug-toolbar package (see the section on pip in *Basic Tools*). Let's edit our settings to enable the debug_toolbar and use it to measure our page build time, count our SQL queries, and exampine the templates used to render a page.

4.3 PRACTICE ACTIVITY

1. Add *debug_toolbar* to the list of installed apps in your settings.py file.
2. add the *debug_toolbar.middleware.DebugToolbarMiddleware* middleware to the list of middleware classes in your settings.py file.
3. Add a new configuration directive to your settings.py file. It should look like

   ```
   INTERNAL_IPS = ('127.0.0.1',)
   ```

4. Start the built-in Django testing server with the runserver command. Open a browser and visit http://localhost:8000/polls/

   The debug toolbar is now activated and you should see something like:

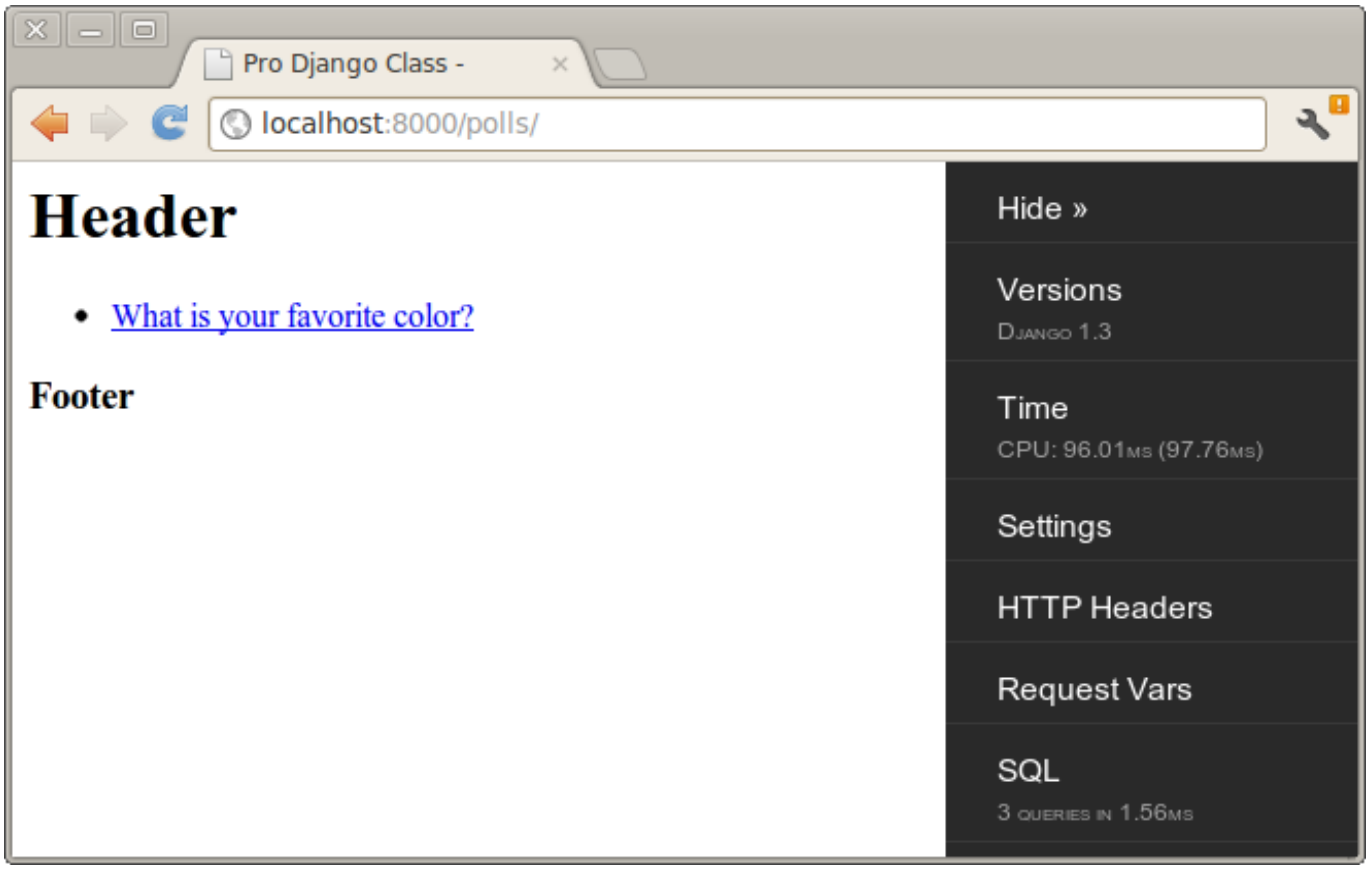


Figure 4.1: Base Template

For more configuration options visit https://github.com/django-debug-toolbar/django-debug-toolbar

5. Click on the button image on the right side of the screen and use the toolbar to answer the following questions: how long did it take to build the page? How many sql queries were executed? What tables did they touch? And how many templates were used?

4.4 django_extensions

Another third party app that I frequently use is django_extensions. This is an app that provides many additional management commands, some additional model field types, and a faciity to create cron-style jobs, among other things. For the moment we'll just explore one of the new management commands it offers.

4.5 PRACTICE ACTIVITY

1. Install the django-extensions package using pip. Add *django_extensions* to the list of installed apps in your settings.py file.
2. Run the `shell_plus` command provided by django_extensions with

```
python manage.py shell_plus
```

This command works just like the built-in `shell` command but autoloads the models for all your installed apps. You should see the list of autoloaded models when your shell starts up.

```
(PROJ1)$ ./manage.py shell_plus
From 'auth' autoload: Permission, Group, User, Message
From 'contenttypes' autoload: ContentType
From 'sessions' autoload: Session
From 'sites' autoload: Site
From 'admin' autoload: LogEntry
From 'polls' autoload: Poll, Choice
Python 2.6.5 (r265:79063, Apr 16 2010, 13:09:56)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>>
```

3. Use the User model in the interactive shell to find the admin user you created and print out the associated email address (run `python manage.py createsuperuser` if you don't have a superuser created yet.) With shell_plus you don't have to remember `from django.contrib.auth.model import User` in order to use the `User` model.

4.6 *More Tools* Follow-up

We've only scratched the surface of these apps, in particular django_extensions can also help us write management commands more simply, generate graphs of our data model and much much more. Django has a huge ecosystem of 3rd party apps so part of your development as a skilled Django developer means getting acquainted with what's out there. Remember - the best code is code you didn't have to write yourself. Always check for existing Django apps before writing one yourself.

Chapter 5

Data Migrations

5.1 Lesson Objectives

The single most necessary feature that is not baked into Django is support for evolving your database tables as you make changes to your models. We'll learn to use the popular django application `south` to provide migration support for our models.

5.2 Why migrations?

It's a fact of life that our data model will change over time. Many projects are exploratory in nature: we start coding, establish a working prototype, and then refine it. But what happens when we change our models file?

Let's find out:

5.3 PRACTICE ACTIVITY

1. Let's make our `polls` application more useful. First let's make owners for each `Poll` object. We can do this by adding a ForeignKey field named *user* to the `Poll` model that points to the built in django.contrib.models.User model. See the Related Objects documentation (https://docs.djangoproject.com/en/1.3/ref/models/relations/) if you haven't done this before.
2. Use the `runserver` command to start the development webserver and navigate to http://localhost:8000/polls/ in your browser. What do you see?

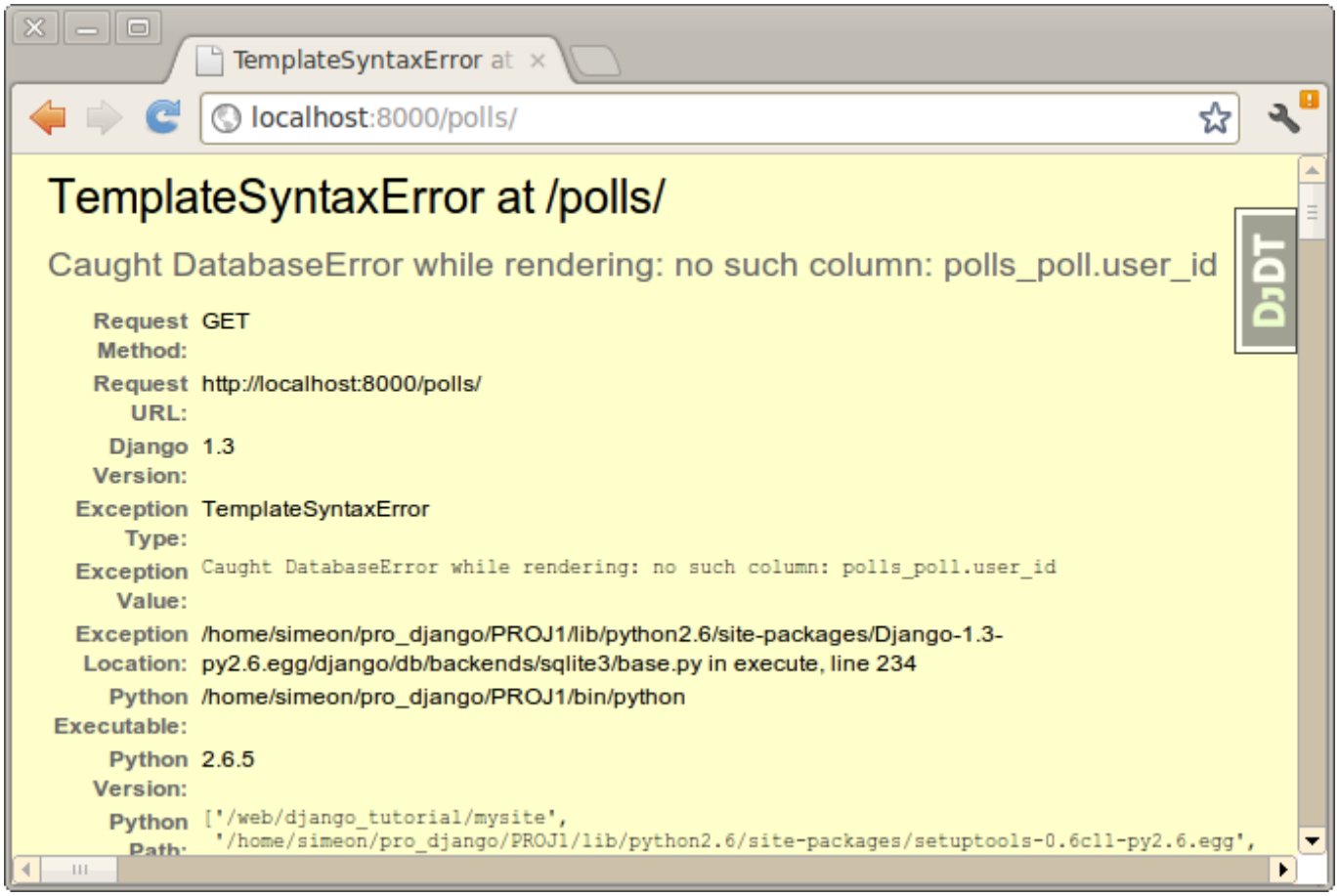


Figure 5.1: Base Template

5.4 Django database creation support

How can we solve this problem? Let's review the support Django provides for creating our database tables.

5.4.1 syncdb

The builtin management command `syncdb` does not handle *changes* to existing tables, only *creating* tables for newly added models. To quote the Django documentation

> syncdb will only create tables for models which have not yet been installed. It will never issue ALTER TABLE statements to match changes made to a model class after installation. Changes to model classes and database schemas often involve some form of ambiguity and, in those cases, Django would have to guess at the correct changes to make. There is a risk that critical data would be lost in the process.
>
> If you have made changes to a model and wish to alter the database tables to match, use the sql command to display the new SQL structure and compare that to your existing table schema to work out the changes.
>
> https://docs.djangoproject.com/en/1.3/ref/django-admin/#syncdb

One way that suggests itself is just to delete the tables (the `sqlclear` command will give us the necessary sql to run on our database) and then use `syncdb` to recreate brand new tables.

This might work early on - but later we'll want to make changes without losing any data.

5.5 Using south

`south` to the rescue! We'll be using `south` to create database migrations. South works by keeping adding a table to keep track of the current migration for your app. When you make changes to your model you'll write a migration that south can run to change the database to keep it in sync with your model. Migrations are sequentially named files stored in the `migrations` directory of your app. The new command `migrate` can then see what what migrations exist for your app and what migrations have already been run and apply any migrations that are needed.

If writing database migrations every time you make a model change sounds like a lot of work, don't despair! South is very smart and can write most migrations for us.

5.6 PRACTICE ACTIVITY

1. Install south using pip. Add *south* to your `INSTALLED_APPS` tuple in settings.py and run syncdb to create the necessary tables for south to store migration history. Notice that the syncdb command has been altered - now the output shows the apps that have no migrations and the tables south will manage.

```
(PROJ1)$ ./manage.py syncdb
Syncing...
Creating tables ...
Creating table south_migrationhistory
Installing custom SQL ...
Installing indexes ...
No fixtures found.

Synced:
 > django.contrib.auth
 > django.contrib.contenttypes
 > django.contrib.sessions
 > django.contrib.sites
 > django.contrib.messages
 > django.contrib.staticfiles
 > django.contrib.admin
 > django.contrib.admindocs
 > polls
 > debug_toolbar
 > django_extensions
 > south

Not synced (use migrations):
 -
(use ./manage.py migrate to migrate these)
```

2. Comment out the new field *user* in `mysite/polls/models.py` that you added in the previous practice activity - we need to get our app set up with south before we make any changes. Be sure to at least read the south tutorial - but you should be able to simply run

```
(PROJ1)$ python manage.py convert_to_south polls
```

3. Uncomment your *user* field on the model for Poll. We're now ready to let south generate a migration for this change. But think about our database table - we haven't specified a default value and the column type is a foreign key linking to the User table. How will the migration know what data to provide?

   In fact, south will note that this is a problem and provide us with the opportunity to either add a default to our field definition or provide a one-off value that the migration will use for the currently existing rows that are getting a new column.

   Enter the number *1* at the prompt - assuming you created an admin user and the admin user has an id of 1 this will assign the existing poll to the admin user. To create the migration run

```
(PROJ1)$ python manage.py schemamigration polls --auto
```

4. Look at the migration that south created - it should be the migration starting with 0002 in your mysite/polls/migrations directory. You can now run the migrate command (try it first with the --list option) to apply your new migration.
5. Did it work? Start the dev server and use the admin to look at your existing Poll object. Does it have an assigned user? Hint:

```
(PROJ1)$ cat polls/admin.py

... snip ...

class PollAdmin(admin.ModelAdmin):
    fieldsets = [
        (None, {'fields':['question', 'user']}),
        ('Date Information', {'fields':['pub_date'], 'classes': ['collapse']})
        ]
    inlines = [ChoiceInline]
    list_display = ('question', 'pub_date', 'was_published_today')
    lists_filter = ['pub_date']
    search_fields = ['question']
    date_hierarchy = 'pub_date'

admin.site.register(Poll, PollAdmin)
```

5.7 *Migrations* Follow-up

While Django has no built in support for making changes to our models and keeping our database tables in sync, the 3rd party application south provides all the support we need to pursue an evolutionary strategy with our data models.

Be sure to read parts 2 (http://south.aeracode.org/docs/tutorial/part2.html) and 3 (http://south.aeracode.org/docs/tutorial/part3.html) of the south tutorial. Not only can we add simple columns - south can be used to add complex field types and create data-only migrations.

Chapter 6

Adding Media

6.1 Lesson Objectives

This lesson will demonstrate Django's support for static media and demonstrate best practices for including resources like css, images and Javascript. In the process we'll use Twitter Bootstrap to make our sample site look a little nicer.

6.2 Static media

Web applications typically rely on a variety of static resources - css, images, Javascript and a variety of rich media files like audio and video clips.

Django doesn't have much involvement with your static files when you have successfully finished and deployed your application. On the way though it provides a lot of functionality for configuring, collecting, and managing your static files.

For example: you can specify a static files url value so that you can refer to the location of static files in your templates without hardcoding a path. You can specify a file-based location so that user uploaded files will be put in the right place. Via the included `django.contrib.staticfiles` app you can collect collect static files from locations you specify and put them in a location you specify - this feature is very useful for deployment when static files may be served directly from the webroot (eg: /var/www/html) but the Python source of your Django project files should not be in the webroot and browsable.

There is also some suport for configuring how the media for the contrib admin application is served.

Finally there is built in support for serving static files when running the development server and an application for collecting static files from individual applications and placing them in the appropriate directory. This is a lot of functionality and configuring it can be confusing so we'll start simply with the `STATIC_*` configuration variables..

The settings that start with `STATIC` all have to do with collecting your static files, putting them someplace, and telling your application what url to use to refer to them.

6.2.1 `STATIC_ROOT`

STATIC_ROOT` is the place that static files will end up if they need to be collected from individual application's */static/* subdirectory. We don't have any static files that are specific to an application yet so we're going to leave this blank.

6.2.2 `STATIC_URL`

`STATIC_URL` is the url that will be used to refer to static media in HTTP requests. We'll set this to "/static/".

6.2.3 STATICFILES_DIRS

STATICFILES_DIRS takes a tuple of paths that represent the location of additional static files that are not associated with any particular app. Files in the STATICFILES_DIRS are automatically served by runserver if the django.contrib.staticfiles is enabled. We'll be using this functionality

6.2.4 STATICFILES_FINDERS

This is a list of "finder" classes. The built in FileSystemFinder and AppDirectoriesFinder handle looking in the /static subdirectories of our applications and in the paths specified in STATICFILES_DIRS. We don't need to make any changes here.

6.3 PRACTICE ACTIVITY

1. Create a directory called media in your pro_django folder. We'll try to keep just python code and Django templates in our mysite folder. A directory listing should look like:

```
(PROJ1)$ ls -l
-rwxr--r-- 1 simeon simeon   249 2012-12-10 15:19 manage.py
drwxr-xr-x 2 simeon simeon  4096 2012-12-10 21:58 media
drwxr-xr-x 2 simeon simeon  4096 2012-12-10 21:21 mysite
drwxr-xr-x 3 simeon simeon  4096 2012-12-10 21:53 polls
-rw-r--r-- 1 simeon simeon    78 2012-12-10 21:20 requirements.txt
drwxr-xr-x 3 simeon simeon  4096 2012-12-10 21:05 templates
drwxr-xr-x 5 simeon simeon  4096 2012-03-10 10:41 PROJ1
```

2. We won't worry about uploaded media files yet and will only modify two of the settings. Find the STATIC_ROOT and STATIC_URL configuration variables in your settings.py and adjust them to look like:

```
STATIC_URL = '/static/'
STATICFILES_DIRS = (
   join(DIR, "media"),
)
```

   Notice that the path to the media directory is not hardcoded - os.path.dirname gives the parent directory of our current location.

   Add *django.contrib.staticfiles* to the list of installed applications.

3. Let's go ahead and add a css file. Use a text editor to create a file in the media directory called test.css. Add a css comment like

```
/* this is a comment */
```

   and try to load the file by visiting http://localhost:8000/static/test.css

   Can you see the file contents?

4. If this was successful, lets add some more static resources and make our templates a little nicer. Download Twitter Bootstrap from http://twitter.github.com/bootstrap/. Unzip the file in our media directory and pull the updated mysite/templates/base.html template from https://github.com/simeonf/django-tutorial/tree/users

   Reload the site - the addition of some .css styles makes our site look a lot better!

6.4 *Adding Media* Follow-up

Django's configuration settings around serving and setting up static files can be a little confusing. We're starting out simply, however, by specifying only the STATIC_URL and the STATICFILES_DIRS configuration variables and letting the staticfiles app automatically serve our media when running the development server.

Chapter 7

Users and Authentication

7.1 Lesson Objectives

This lesson will introduce Django's user and permission model. We will learn to check the current user and the user's permissions and how to limit data in our views and in the built-in admin app based on user.

We'll make our tutorial app a little more useful by improving our templates, views and the admin.

7.2 Users and Permissions via contrib.auth

Authentication support is included with Django in the `django.contrib.auth` application - see https://docs.djangoproject.com/en/1.3/topics/auth/ for details. Included are models to store users, groups and permissions and methods for authenticating users. Django provides the ability to add custom authentication backends but by default the authentication mechanism checks the session that accompanies an HTTP request to see if it includes a logged-in user. If not the user must enter a username and password that matches an enty in the User model.

This is appropriate for most web applications - including ours. Let's modify our sample application to support the following features:

- only logged in users with appropriate permission can vote on polls
- everyone can see poll results
- only the owner of a poll can edit it in the admin
- our application should provide login/logout functionality

7.3 Users in Requests

Django includes two `middlewares` that add information about the currently logged in user to the request object that is passed to all views. By default the `SessionMiddleware` and `AuthenticationMiddleware` are enabled and allow us to access the `request.user` object. This object will return either a User object or an instance of `AnonymousUser`. You can check with the `request.user.is_authenticated()` method in view code or refer to user variable in templates that have been supplied a RequestContext.

As we saw in the tutorial we can render our templates with a RequestContext explicitly using the render_to_response shortcut:

```
    return render_to_response('polls/detail.html', {
        'poll': poll,
        'error_message': "You didn't select a choice.",
    }, context_instance=RequestContext(request))
```

My experience has been that we might as well always use a RequestContext: the extra overhead of adding additional context variables via your installed `TEMPLATE_CONTEXT_PROCESSORS` is neglible and it is nice not to have to think about whether or not the user variable will exist in your templates. Django 1.3 includes a new shortcut `render` that assumes you want a RequestContext without making you specify it. Get into the habit of using it instead of render_to_response:

```
    return render(request, 'polls/detail.html', {
        'poll': poll,
        'error_message': "You didn't select a choice.",
        })
```

Once we have a user variable in a template we can check that the user is logged

```
{% if user.is_authenticated %}
 <p>Logged in user: {{ user }}
{% else %}
 <p>You are not logged in.
{% endif %}
```

7.4 Users in the admin

The included `contrib.admin` app is a large swiss-army-chainsaw of CRUD functionality for your apps. Be sure to look at the docs at https://docs.djangoproject.com/en/1.3/ref/contrib/admin/ - in general the admin's approach is to limit access to logged in users with the `is_staff` attribute set to True. You should have been prompted to create a `superuser` when you created your database (if not run the `createsuperuser` command via `manage.py`) and superusers by default have both the *.is_staff* and *.is_superuser* attributes set to `True`.

```
>>> User.objects.all()
[<User: admin>]
>>> admin = User.objects.all()[0]
>>> admin.is_staff
True
>>> admin.is_superuser
True
```

Django allows us to set permissions for specific users but users with `is_superuser` set to `True` will always have every permission.

Django doesn't provide explicit support for "row-level" permissions but it does allow our ModelAdmin objects to define a queryset function that takes a request object and returns the queryset that will be used by the admin. This allows us to check the logged in user in the admin and filter the data the current user can see:

```
class PollAdmin(admin.ModelAdmin):
    fieldsets = [
        (None, {'fields':['user', 'question']}),
        ('Date Information', {'fields':['pub_date'], 'classes': ['collapse']})
        ]
    inlines = [ChoiceInline]
    list_display = ('question', 'pub_date', 'was_published_today')
    lists_filter = ['pub_date']
    search_fields = ['question']
    date_hierarchy = 'pub_date'

    def queryset(self, request):
        qs = super(PollAdmin, self).queryset(request)
        if request.user.is_superuser:
            return qs
        return qs.filter(user=request.user)

admin.site.register(Poll, PollAdmin)
```

To check this functionality you'll have to add a new user while logged in to the admin site (http://localhost:8000/admin/) as the superuser. You may notice when you create a user you can also set permissions. Django by default creates three permissions (add, change, and delete) for each model in your installed apps. Be sure to give your new user permission to edit polls and choices.

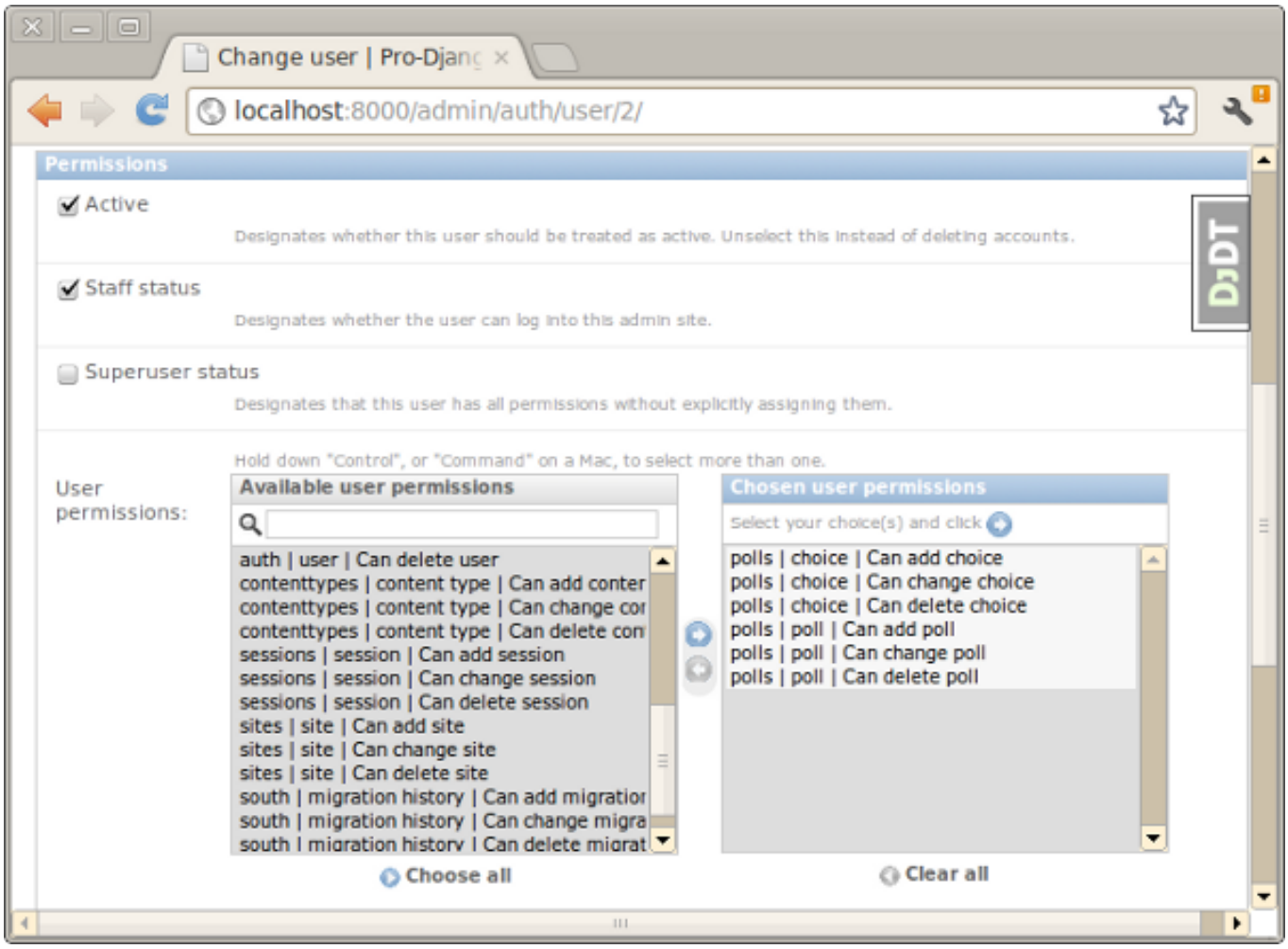


Figure 7.1: Permissions

If our new user account is_active, is_staff, and has Poll and Choice permissions then it will be able to edit Poll objects. But it should only be able to see Poll objects that are explicitly assigned to it. Of course if it assigns a poll to some other user it will no longer be able to see it. We can get around this limitation by adding another method to our ModelAdmin class:

```
    def formfield_for_foreignkey(self, db_field, request, **kwargs):
        if db_field.name == "user" and not request.user.is_superuser:
            kwargs['queryset'] = User.objects.filter(id=request.user.id)
        return super(PollAdmin, self)\
                .formfield_for_foreignkey(db_field, request, **kwargs)
```

7.5 Login/Logout

So far you've probably been logging in and logging out by visiting the admin site which lets you login. But we'd like to provide our own customized loging and logout screens to our users. We can take advantage of the pre-built views that come with the django.contrib.auth view simply by adding them to our urls.py and passing a few configuration parameters.

```
urlpatterns = patterns('',
    # ... snip ...
    url(r'^login/$', 'django.contrib.auth.views.login'),
    url(r'^logout/$', 'django.contrib.auth.views.logout'),
)
```

Visiting http://localhost:8000/login/ will cause a template error however as the views depend on template (*registration/login.html*) that doesn't exist. The Django documentation supplies a sample login template to use and I've added it to my github `users` branch at https://github.com/simeonf/django-tutorial/tree/users. Adding a directory `registration` under your templates directory and putting the login.html template in the `registration` directory makes the login form display when you reload the login url. Be sure to look at the contents of the template - it uses Django's form processing library which we'll be discussing shortly.

We still have one piece of configuration to do. The `login` view accepts an argument `next` that it will redirect the browser to when a user logs in sucessfully. We could link to our login page with "/login/?next=/polls/" if we want users who login to end up on the front page of our poll application. If the `next` variable isn't present in the GET or POST, the `login` view redirects to `settings.LOGIN_REDIRECT_URL` which defaults to "/accounts/profile/". We can set this settings variable to */polls/* to make the default behavior do what we want.

Be sure to visit http://localhost:8000/logout/ - by default it renders a "Goodbye" message using *registration/logged_out.html*. Since we haven't provided this template it uses the one that is provided by the `contrib.admin` app. Let's supply our own template so that we don't see the `admin` application's styles:

```
{% extends "base.html" %}

{% block content %}
<p>You have been logged out.</p>
{% endblock %}
```

7.6 PRACTICE ACTIVITY

We've covered enough of the authentication system that you should be able to upgrade our tutorial app to meet the following criteria:

1. Add a login and logout link to the base.html template and hook them up the contrib.auth login/logout views.
2. Only let logged in users vote on polls.
3. Restrict editing of polls in the admin to the user who owns a poll or the superuser. Be sure to create another non-superuser to test this functionality.

7.7 *Users and Authentication* Follow-up

Authentication is an important part of most webapps. We've covered enough to let users log in and out and to restrict access to content based on the user. Be sure to review the User Authentication documentation at https://docs.djangoproject.com/en/1.3/-topics/auth/

Chapter 8

Intro to Forms

8.1 Lesson Objectives

This lesson will introduce Django's form library. We will learn to validate form data and handle forms in our views. We'll also begin to discuss ways to customize the html output from our forms.

8.2 Forms

Form objects are a collection of data fields and validation rules that define acceptable data for a form. Forms are declarative in style, much like Django's Model syntax and provide a large number of built in field types with a rich collection semantic meanings attached to them.

Most field types share common attributes like `required` and `label` that allow you to impose restrictions on the data that the field can accept and customize its display. Many field types have their own unique atttributes - `max_value` and `min_value` for numeric fields - for instance that provide simple validation. We can also specify the `widget` for a form which maps to an HTML input control and allows us to customize its attributes and add media that may be needed to render it.

Be sure to read the forms documentation starting at https://docs.djangoproject.com/en/1.3/topics/forms/

The best place to put forms is in their own module. Django doesn't create a blank `forms.py` in your application when you use the `startapp` command so you'll have to do that yourself.

Consider a sample form in `polls/forms.py`

```
from django import forms

class RegistrationForm(forms.Form):
    username = forms.CharField(max_length=30)
    email = forms.EmailField(label="E-mail")
    password1 = forms.CharField(label="Password")
    password2 = forms.CharField(label="Password Again")
```

This is a simple form with 4 fields - each of which will output an html <input type="text"> control. Let's see how we might use form processing in a view and form rendering in a template.

8.3 Working with forms in views

Django distinguishes between *bound* forms and *unbound* forms. Basically this is the difference between forms that are handling data input (and therefore doing validation) and forms that have no data yet. Django makes this distinction because we don't want to complain about validation errors if we haven't entered any data in the form yet.

The django documentation suggests the following method for handling forms in a view:

```
def contact(request):
    if request.method == 'POST': # If the form has been submitted...
        form = ContactForm(request.POST) # A form bound to the POST data
        if form.is_valid(): # All validation rules pass
            # Process the data in form.cleaned_data
            # ...
            return HttpResponseRedirect('/thanks/') # Redirect after POST
    else:
        form = ContactForm() # An unbound form

    return render_to_response('contact.html', {
        'form': form,
    })
```

Notice that the code creates one of two different form objects, one bound, the other unbound, depending on whether the view is handling an HTTP POST or not. Additionally there is a nested if - the bound form may or may not be valid and form processing should only occur if it is valid.

We can actually get the same effect with simpler code. I first saw the following pattern demonstrated by Daniel Greenfeld:

```
def register(request):
    form = RegistrationForm(request.POST or None)
    if form.is_valid():
        # Do processing
        return HttpResponseRedirect(reverse('polls.views.index'))
    return render(request, 'polls/register.html', {'form': form})
```

This is more than just a clever trick based on the shortcutting behavior of Python's `or` operator - it removes duplication of code and a level of nesting in our if.

Note that the form api is fairly simple from the vantage point of our views: forms are created with request.POST or with no data and bound forms can be checked for validity. If a form `is_valid()` you can access the submitted data in the `.cleaned_data` member variable. Notice also that the pattern remains to issue a redirect when a form is sucessfully processed. This prevents the user from accidentally resubmitting the form by pressing refresh in their browser.

8.4 Displaying forms in templates

In the view above the form is passed to a template as the context variable `form`. In the template we can simply evaluate the form variable and the default action is to render all the form fields as html input controls inside a table structure. Django doesn't provide the <form> or <table> tags for us so we supply them ourselves:

```
<form action="" method="POST">
{% csrf_token %}
<h1>Enter your information to sign up</h1>
<table>
    {{ form }}
    <tr><th></th><td><input type="submit" value="Submit"></td></tr>
</table>
</form>
{% endblock %}
```

Notice the template tag `csrf_token` - that should always be the first thing in your form and emits the token Django uses to prevent CSRF attacks - see https://docs.djangoproject.com/en/1.3/ref/contrib/csrf/ for details.

8.5 Validation

Django provides many hooks for validating our form fields and raising validation errors. See https://docs.djangoproject.com/en/-1.3/ref/forms/validation/ for details - among our options are providing validator functions to our fields, writing custom fields that

do their own validation, providing `clean_fieldname` methods, and overriding the built in `clean` method. We'll look at the last two options for improving our form:

```
class RegistrationForm(forms.Form):
    username = forms.CharField(max_length=30)
    email = forms.EmailField(label="E-mail")
    password1 = forms.CharField(label="Password")
    password2 = forms.CharField(label="Password Again")

    def clean_username(self):
        username = self.cleaned_data['username']
        if not username.isalpha():
            raise forms.ValidationError("Usernames must contain only letters!")
        return username

    def clean(self):
        data = self.cleaned_data
        if not data.get('password1') == data.get('password2'):
            raise forms.ValidationError("Passwords must match!")
        return data
```

Our `clean_username` method will automatically be called by our form after all other individual validations have run. It must return the cleaned data (modified if appropriate) and raise a `forms.ValidationError` if there was an error - the result will be an html error message attached to the control.

The `clean` method is called last and allows us to do validations that depend on several field values. We don't have any guarantee that fields exist in the `cleaned_data` - but we can check the values if any and either raise validation errors or return the cleaned data. before

8.6 PRACTICE ACTIVITY

Let's take what we've learned about forms so far and create a registration page for our poll application. This will mean adding a registration form to handle the data, a view which creates a new user when the form is valid, and a template to display the form. Go ahead and add a "Register" link to the base template as well.

8.7 *Intro To Forms* Follow-up

Form handling is a huge area in most data heavy Django applications and getting a handle on Forms, Fields, Widgets, and Validators will take some time and experience. As always - be sure to read all of the excellent Django documentation in the forms section.

Chapter 9

More Forms

9.1 Lesson Objectives

In *Intro to Forms* we learned to add custom validation to forms and how to instantiate and process our forms in our views. This lesson will focus on customizing the html output of our forms by using widgets and exploring alternative methods for rendering and layout.

9.2 Field and Widgets

We added validation to our RegistrationForm with custom validation methods - but it still isn't very sophisticated in either presentation or validation. We can use specific field types to automatically provide some forms of validation and we can use `widgets` to further customize the html display of our form components.

Be sure to check out Django's extensive documentation at https://docs.djangoproject.com/en/1.3/ref/forms/fields/ of all the built in field types. One field type that looks promising is the RegexField which is basically a Charfield that validates submitted data against a regular expression. Using a Regexfield for our username would automatically provide the validation we supplied with the `clean_username` method. This might look like:

```
username = forms.RegexField(regex=r'^[\w]+$',
                            max_length=30,
                            error_messages={'invalid':
                                            "This value must contain only letters, "
                                            "numbers and underscores."}
                            )
```

We might also notice that our password fields are showing their contents when we type. We're currently using a CharField for our password entry and looking at the list of fields shows no "PasswordField" to use. We can however choose a different widget when rendering our CharField:

```
password1 = forms.CharField(widget=forms.PasswordInput(attrs={'class': 'required'},
                                                       render_value=False),
                            label="Password")
```

Django distinguishes between form fields which have data implications and form widgets which only effect the html display of the form field. Note that if we supply the widget we can also pass the attrs argument which is a dict whose keys and values will be used as the attributes of our html tag. This allows us, for example, to specify classes that can be used to style the html our form will generate.

The pattern of picking a widget to customize the output of a particular form field is a common one in Django forms. For instance consider that there is only one `ChoiceField` for selecting an item or items from a list. The `ChoiceField` renders with a `Select` widget by default - but we could use the same control and a `RadioSelect` widget to also allow the user to pick one

item from a list. If we wanted to let the user make multiple selections we might use a `SelectMultiple` widget to display a multi-select combobox or a `CheckboxSelectMultiple` widget to show a series of checkboxes. In each case the Field provides validation while the Widget controls display output.

9.3 Modifying Form Output

It's also worth noting that we can change the way we output our forms as a whole. So far we've simply evaluated the template variable containing the form. This by default outputs a series of table rows containing the label and input controls for each form field like:

```
<tr>
  <th><label for="id_username">Username:</label></th>
  <td><input id="id_username" type="text" class="required" name="username" maxlength="30" / ↩
      ></td>
</tr>
```

Django allows us to customize the row output by providing a class or classes that will be applied to rows with required inputs and rows with and error message:

```
class RegistrationForm(Form):
    error_css_class = 'control-group error' # Works with twitter bootstrap
    required_css_class = 'required'
```

We can also choose to call the `as_ul` or `as_p` methods which emit <p> and <li> tags instead of table rows. And if none of the predefined methods of layout meet our criteria we can always manually specify the layout of the form by accessing the fields directly.

```
 <div>
  <label for="username">Username</label> {{ form.username }} {{ form.username.errors }}
 </div>
```

This style is to be avoided if possible since changes to the form will require changes to the template as well. With strong css skills the layout of your forms is not very dependent on the html structure - but you may find times when you have to manually specify your form layout to precisely match a design.

9.4 PRACTICE ACTIVITY

Can we improve our registration form? Try the following:

- add the *control-group* and *error* classes to form rows with errors - this will apply styles to the input control from our twitter-bootstrap styles.
- make sure the form validates usernames to limit to letters and numbers. Usernames should also be checked against the database to make sure they won't conflict with an existing user.
- The password fields should be displayed in a "password" input control that conceals typing from the user. The two passwords should match.
- Play around with using css to customise the layout of the generated form. Do error messages display nicely? Is there enough space for the labels? Do we have good alignment for our labels, form inputs, and error messages?

9.5 Dynamic Choices

One other common usage pattern is to write a form that selects a particular object from the database. Django provides a `ModelChoiceField` for this situation but it requires a queryset as part of its construction. Frequently, however, we can't write the queryset when the form is defined because the queryset will differ depending on the view. For example - we might want to write a form that displays a `ModelChoiceField` that allows the user to select a Choice for a particular Poll - but when a different Poll is selected we need to change the list of Choices as well.

The most common way of addressing this pattern is provide a blank queryset of the appropriate Model and let the view set a queryset directly on the field after the form is created. For example:

```
class ChoiceForm(forms.Form):
    choice = forms.ModelChoiceField(queryset=Choice.objects.none())

def detail(request, poll_id):
    poll = get_object_or_404(Poll, pk=poll_id)
    form = ChoiceForm()
    form.fields['choice'].queryset = poll.choice_set.all()
    return render(request, 'polls/detail.html', {'poll': poll, 'form': form})
```

9.6 PRACTICE ACTIVITY

Our detail page for an individual poll still has manually constructed radio inputs and the detail view is manually checking POST to validate the data. Update the `detail` and `vote` views and the detail.html template to use a form instead.

9.7 *More Forms* Follow-up

We still haven't explored all the features of the form library - but we do have the ability to customize the html the form classes produce and have seena few more field and widget types. Be sure to look at the list of fields https://docs.djangoproject.com/en/-1.3/ref/forms/fields/ and the list of widgets https://docs.djangoproject.com/en/1.3/ref/forms/widgets/ in the official documentation.

Chapter 10

Writing Tests with Django

10.1 Lesson Objectives

This lesson will introduce Django's test integration. We will learn to write unit tests that test our Python code as well as use the Django test client to simulate browser requests. We'll also learn how to run our tests and which kinds of tests to write.

10.2 Running Tests

Django integrates the Python stdlib UnitTest module by providing a default test runner and a management command to run it. Try it by running:

```
$ ./manage.py test polls
Creating test database for alias 'default'...
.

Ran 1 test in 0.000s

OK
Destroying test database for alias 'default'...
```

What test? What just happened?

It turns out in an effort to encourage testing the startapp command created a single (trivially nonsensical) unit test when we created our polls sample app.

```
$ cat polls/test.test
"""
This file demonstrates writing tests using the unittest module. These will pass
when you run "manage.py test".

Replace this with more appropriate tests for your application.
"""

from django.test import TestCase


class SimpleTest(TestCase):
    def test_basic_addition(self):
        """
        Tests that 1 + 1 always equals 2.
        """
        self.assertEqual(1 + 1, 2)
```

If you're familiar with UnitTest this should appear straightforward. For those of you new to UnitTest - using UnitTest means subclassing UnitTest and writing methods that start with the string "test". Each method should use a `self.assert*` as the test - methods are available to check equality, inequality, containment, exceptions and so forth. See http://docs.python.org/-library/unittest.html#unittest.TestCase.assertEqual for the complete list of UnitTest `assert` methods. Django provides us with `django.test.TestCase` which subclasses UnitTest and adds some methods for integrating with Django's urls and ORM.

For your more complicated tests you may need to define a `setUp` method on your test class. The `setUp` (note the case) method will be called befor any `test*` methods and can be used to create initial data. Don't worry about messing up your database - Django actually creates a new blank database when you run your tests so any test that relies on data accessed via the ORM will need to create that data first. Of course our existing test doesn't touch the database - it just verifies that the Python addition operator works. This isn't really the kind of thing we'll need to test.

10.3 Writing Tests

So what sort of tests might we write? And what sort of things should we test? Clearly we don't need to test that + works in Python code. Similarly there's no point in testing core Django functionality - if Django itself isn't working we have problems bigger than we can fix. What we want to test is our application functionality. Django provides us with a request simulator that we can use to simulate loading a particular url but I'd like to strongly discourage its use.

Tests written checking if a view works rely on the url mapping being correct, the view code working, any included templates rendering correctly and correct behavior of any model of form code we might write. In short - they aren't unit tests because they aren't testing a particular `unit` in isolation. They're also likely to be much slower than genuine unit tests.

Let me show you an example:

```
class RealUnitTests(TestCase):
    def test_username_clean(self):
        data = dict(username='bogus#', email="test@gmail.com",
                    password1="test", password2="test")
        form = RegistrationForm(data)
        form.is_valid() # Must call to set up form.cleaned_data
        # Did we get an error for username?
        self.assertIn('username', form.errors)

class IntegrationTests(TestCase):
    def setUp(self):
        self.user = User.objects.create_user(username="bob",
                                             email="bob@gmail.com",
                                             password="test")
        self.poll = Poll(user=self.user,
                         question="Test Poll #1",
                         pub_date=datetime.now())
        self.poll.save()

    def test_index_view(self):
        # Assumes named url poll_index
        resp = self.client.get(reverse('poll_index'))
        self.assertIn("Test Poll #1", resp.content)
```

Notice the difference between these two tests - one creates a form with sample data and tests a the custom clean_username method. No requests are made, no database access occurs.

The second example requires sample data and then simulates a load of a particular url. This sort of test is useful - but it isn't really a unit test as it tests too many different pieces at once.

Write lots of unit tests - be sure to see the testing documentation at https://docs.djangoproject.com/en/1.3/topics/testing/ to explore the different places you can put tests and the different tests you can write. Also write integration tests that use the built-in request.client or more sophisticated tools like Twill or Selenium to test the pages of your web application - but store those in their own app and run those test separately.

10.4 PRACTICE ACTIVITY

Write your own tests for the poll app. Test at least:

1. Write integration tests for each of your pages. Can you test posting to the vote url?
2. Write unit tests for any custom Model methods or Form methods you've added.

10.5 *Writing Tests* Follow-up

Testing is a vast field, particularly in web apps. Best practices in the area include test automation via a continuous integration server like Jenkins and the use of sophisticated browser emulation tools like Selenium. It's also important that all the tests pass (so failures are noticed) and that all the tests run swiftly so that actually running them is encouraged. Simply starting to write a few Unit Tests, however, has the benefit of catching your code errors and also forcing you to write more testable code. Make an effort - every commit should have a test along with it that runs fast and passes!

Chapter 11

Customizing Templates

11.1 Lesson Objectives

We've been using templates to develop our app without much explanation. That's because Django's template library emphasizes ease of authoring - presuming that the least technical task of creating a web application ought to be working with the html output. There are a few tricks and conventions we should know, however, and we can also explore that ability to write our own templatetags and filters to provide complex display logic to our templates.

11.2 Template Loading

So far we've been storing our templates in a project wide template directory. We tell Django where to find our templates by adding our location to the `TEMPLATE_DIRS` tuple.

You may have thought that application templates really should be stored with the applications - and you would be right. Django allows us to customize how templates are discovered by providing `TEMPLATE_LOADERS`. Two provided loaders are enabled by default - the `filesystem` loader and the `app_directories` loader. The `filesystem` loader looks in the directories specified in the `TEMPLATE_DIRS` setting while the `app_directories` loader looks in the templates subdirectory of each installed app. Since the default behavior is to have the `filesystem` loader first, this means the project wide template directories location can serve as an override to application level templates. Django convention is:

- applications store their templates in a self-named directory in the application. Our poll application should store all its templates in `mysite/polls/templates/polls`.
- If the user of an application needs to edit an application template first copy it to the appropriate supdirectory in the shared project template directory. This means if you update the app you don't have to worry about losing template customizations.

For example: if someone else installed our polls app and wanted to tweak the display page they would make a subdirectory `polls` in their `project/templates` directory and copy `details.html` there, making whatever edits were necessary. When Django runs our views and we try to render the template *polls/detail.html* Django will first look in the `TEMPLATE_DIRS` directory and if it finds a *polls/detail.html* it will use it. If not it would continue to each of the applications in turn - and yes this means one application can ship customisations of another applications templates.

11.3 PRACTICE ACTIVITY

Create a templates directory in our `polls` directory and move the `polls` specific templates there. Have we made any templates that should stay at the project level?

11.4 Builtin Tags and Filters

Django templates really only include three bits of syntax. In a template, double braces indicate variable evaluation. The template engine will attempt to convert variables to a string in intelligent fashion - calling them if they're callable. So given a Poll object `poll` we can display it with

```
<p>
{{ poll }}, {{ poll.published_today }}
</p>
```

The `published_today` method will be called and its return value displayed even though we can't supply the parentheses inside the templates.

A brace and a percent sign are used to indicate template tags. Django uses template tags to manage template logic and supplies built in template tags to do looping, logical evaluation and template operations like extending a parent template or including a child template. Templatetags are frequently paired with an end delimiter like

```
{% for p in polls %}
    <tr class="{% cycle 'row1' 'row2' %}">
        <td>{{ poll }}</td><td>{{ poll.pub_date|timesince }}</td>
    </tr>
{% endfor %}
```

Note the {% for %} tag has a matching {% endfor %} tag to indicate the body of the loop but the {% cycle %} tag does not. Can you guess what {% cycle %} does?

Also note the use of the pipe ("|") character. Pipes are used to call template filters. Templatefilters are typically simple functions that provide output based on input - the built in `timesince` filter will return a string representing the time elapsed since a date.

11.5 PRACTICE ACTIVITY

Lets update the index page for our polls. We're showing only the most recent polls but we've added users to our polls and we aren't displaying this information. Instead of a simple list, adjust the index.html template to show a table of recent polls with columns for the user and a column showing the timesince the poll was created.

11.6 Custom template filters

We can write our own custom tags and filters. For instance - it wouldn't be easy to use the template language to count the total number of votes cast for a poll. Since we authored the application we could just add a utility method to the model, but if we were using a 3rd party application we might not want to edit it and might prefer to write a templatefilter instead. To create custom tags and filters we need to create some more files in our application directory:

```
polls/
    templatetags/
        __init__.py
        poll_extras.py
```

Now if we registered a template tag or filter in poll_extras it can be loaded and accessed in any template. For example the nonsensical filter "question" could be defined:

```
from django import template

register = template.Library()

@register.filter
def question(poll):
    return poll.question
```

and used like

```
{% load poll_extras %}
{{ poll|question }}
```

As you can see prefixing our templatetags files with the application name is important to prevent name clashes.

11.7 PRACTICE ACTIVITY

Create a custom filter that accepts a poll object and returns the total number of votes cast. Update the index page for our polls and add a column with the total vote count.

11.8 Custom template tags

Writing filters is simply a matter of writing a Python function. Writing a full-blown template tag is a little more difficult - Django tags can have arbitrary syntax so Django splits the work of writing tags into a parse step and a render step. The render portion works by writing a class that subclasses `template.Node`.

This has always seemed like a lot of work to me - fortunately we have some shortcuts for writing the most common types of tags. We can use the `simple_tag` decorator to create a tag with a simple function:

```
@register.simple_tag
def count_votes(poll):
    return sum([c.votes for c in poll.choice_set.all()]
```

and then (assuming we've loaded its tag library appropriately) use it in a template:

```
{% for p in polls %}
  {{ p }}: {% count_votes p %}<br>
{% endfor %}
```

Be sure to read the Django documentation at https://docs.djangoproject.com/en/1.3/howto/custom-template-tags/ on creating custom tags and filters. Note that we can write simple tags that are passed the current template context (to add additional variables to the context) and also simple tags that are automatically rendered with a template. These patterns are powerful enough to keep us from having to delve into writing full blow template tags in most cases.

11.9 *Customizing Templates* Follow-up

By understanding template loading we can overwrite application templates on a case-by-case basis. This is true for all applications - including the builtin admin app. We also can see how to use the simple language with tags and filters to write more complex rendering logic - and when we can't achieve what we want with the built in tags and filters Django lets us write our own.

Chapter 12

Even more (Model)Forms

12.1 Lesson Objectives

In *Intro to Forms* and *More Forms* we learned about validating data and customising form output. Frequently we use forms to process data that closely resembles an instance of our model. In this lesson we'll learn automatically generate a form that knows how to represent the data in a model and knows how to save its data if it passes validation. We'll use this to save additional data attached to the `django.contrib.auth.models.User` model. We will also learn about some popular 3rd party applications that provide more sophisticated layouts, controls, and client side behaviors for forms.

12.2 ModelForms

Django provides an alternative base class `ModelForm` that can generate a form instance when given a model. For instance to create a form based on our `Poll` model we only need to add to our `polls/forms.py`:

```
from django.forms import ModelForm
from models import Poll

class PollForm(ModelForm):
    class Meta:
        model = Poll
```

Inspecting this form at the interactive console reveals the expected three fields:

```
>>> f = PollForm()
>>> f.fields
{'user': <django.forms.models.ModelChoiceField object at 0x9e147ac>,
'question': <django.forms.fields.CharField object at 0x9e1480c>,
'pub_date': <django.forms.fields.DateTimeField object at 0x9e1486c>}
```

12.2.1 Customizing fields and widgets

ModelForms are already useful and can be used just like a regular form - instanciating bound and unbound versions and checking for data validation. We might sometimes also want to customize the fields from our model that appear on our form. Be sure to read the docs at https://docs.djangoproject.com/en/1.3/topics/forms/modelforms/ but if you are familiar with configuring ModelAdmin instances you can probably guess what the `fields` and `exclude` attributes of a ModelForm's Meta class are meant to accomplish.

You might also want to override the widgets for the automatically created form. You can do this most simply by adding Widget classes or instances to the widgets attribute of the Meta class. For example:

```
class PollForm(ModelForm):
    class Meta:
        model = Poll
        fields = ('user', 'question')
        widgets = {'user': forms.RadioSelect}
```

would create a form with only the user and question fields and the user field would use a RadioSelect widget.

If you want to customise the field itself you can manually specify fields on your ModelForm class (just like a regular `forms.Form`) and the explicity definition will override the default generated form field of the same name.

12.2.2 Saving ModelForms

ModelForms also differ from regular forms in that they have a `.save()` method. The save method handles updating the database if the form passes data validation. By default the `.save()` method returns a brand new model instance but we also instanciate ModelForms with an initial object to be updated. In this case it will copy the form data to the appropriate fields of the existing object and call save on the model instance.

The save method also accepts an optional parameter `commit` that defaults to True. If we pass `commit=False` to `.save()` it will return either the new instance of our model or the updated instance but in either case won't call `.save()` on the model object. This can be useful if we have a partial form that doesn't provide enough data to pass model validation - we can get a partially built model instance, supply the additional missing parts ourself and call `.save()` on the model instance directly.

Tip
Obviously this raises problems with ManyToMany relationships. Django can't save the relation until it has a primary key of the main object, which won't exist until it is saved. In this case ModelForms saved with `commit=False` have a `save_m2m()` method you can call manually once the parent object has been saved.

12.3 User Profiles

A commonly requested feature in Django applications is the ability to extend the built in `User` model that Django provides. We can do this by building our own custom model that includes the extra data fields we want (address, twitter handle, cell phone #, etc) and link our model to Django's User model with a one-to-one field. Traditionally our model is called the UserProfile.

```
class UserProfile(models.Model):
    user = models.OneToOneField(User)
    twitter = models.CharField(max_length=100)
```

Immediately we can access the UserProfile model from the User objects. For ForeignKey relationships Django builds a reverse manager so we might expect to access the `userprofile_set` attribute of the `User` model but for OneToOne fields we won't have a set, we'll have just one object! So we can simply look at the `userprofile` attribute of the `User` model instances. Django doesn't automatically create a UserProfile object there isn't one that links back to the instance of User so we'll need to figure out a way (signals!) to make sure that newly created User objects have a corresponding UserProfile object.

Django includes one piece of configuration to support the idea of extending the User model with a OneToOne relationship. If we set `settings.AUTH_PROFILE_MODULE` to a special string of the form `lowercaseappname.ModelName` (note the absence of .models! - this is not the actual import string for your class). In our case

```
AUTH_PROFILE_MODULE = 'polls.UserProfile'
```

All this gets us is a generic link to our user profile class. 3rd party code can't know what your User Profile model name will be but if the AUTH_PROFILE_MODULE setting is filled properly the `.get_profile()` method of User objects will return your User Profile instance. You can see additional details about adding data to the User model at https://docs.djangoproject.com/en/1.3/topics/auth/#storing-additional-information-about-users

12.4 PRACTICE ACTIVITY

Let's enhance the registration form for our polls app. In addition to collecting the data for the User object go ahead and develop a model to hold additional "user" information (address, city, state, phone#, etc) and link it to the User model as a described above.

The registration view should display and process two forms - the RegistrationForm you already wrote and an additional ModelForm based on your user profile class.

12.5 Nicer Forms

The automatic form layout methods (`.as_p`, `.as_ul`, `.as_li`) are typically too limited to allow detailed styling or conform to specific layouts recommended by CSS frameworks such as Twitter Bootstrap. On the other hand it's typically too much work to manually layout every control in a lengthy form (don't forget the labels. And the errors. And the hidden fields. Did you remember the non_field_errors?)

Additionally Django doesn't provide any out-of-the-box support for client validation. While we can't depend on client-side validation for security, it is nice to help the user accurately fill out a form as quickly as possible.

12.5.1 django-floppyforms

django-floppyforms is an application that, among other features, provides replacement widgets based on HTML5 input types. HTML 5 provides new input types like date, color, number, email, etc. HTML 5 also adds new attributes to form inputs like `required`. In modern browsers new input controls may have additional UI (color pickers, date-time pickers, etc) that help the user fill out the form. Browsers which do not know what `<input type="date">` means will simply display standard text input controls.

We can take advantage of django-floppyforms new widget types (and our browser's UI and automatic client side validation) by simply replacing the baseclass for our form (`django.forms.Form`) with `floppyforms.Form`. Be sure to read the documentation at http://django-floppyforms.readthedocs.org/en/latest/usage.html#forms

Our registration form won't have changed much

```
import floppyforms as forms

class RegistrationForm(forms.Form):
    username = forms.CharField(max_length=30)
    email = forms.EmailField(label="E-mail")
    password1 = forms.CharField(label="Password", widget=forms.PasswordInput)
    password2 = forms.CharField(label="Password Again", widget=forms.PasswordInput)
```

but modern browsers will now enforce client-side validation for each field that is required, check the email field to verify that it looks like an email, and so on.

12.5.2 django-crispy-forms

While django-floppyforms includes facilities to customize form layout based on templates we'll actually use another popular package, django-crispy-forms, to handle our form layout.

django-crispy-forms allows us to programatically specify the way our form should be laid out directly on our form class. It also provides some default form layouts. One of them is twitter bootstrap compatible and will suit our needs nicely.

We can use crispy-forms in addition to floppyforms. Crispy forms works by building form helper objects. If we attach the helper to our form and name it "helper" crispy-forms provides a templatetag that will automatically use the helper to layout our form. Be sure to read the docs at http://django-crispy-forms.readthedocs.org/en/d-0/tags.html but I'll provide a basic example.

```
from crispy_forms.helper import FormHelper
from crispy_forms.layout import Submit

class RegistrationForm(forms.Form):
    username = forms.CharField(max_length=30)
    email = forms.EmailField(label="E-mail")
    password1 = forms.CharField(label="Password", widget=forms.PasswordInput)
    password2 = forms.CharField(label="Password Again", widget=forms.PasswordInput)

    def __init__(self, *args, **kwargs):
        self.helper = FormHelper()
        self.helper.form_id = 'id-myform'
        self.helper.form_class = "form-horizontal"
        self.helper.form_method = 'post'
        self.helper.add_input(Submit('submit', 'Submit'))
        super(RegistrationForm, self).__init__(*args, **kwargs)
```

Clearly all magic happens in the `__init__` method. I'm instantiating a `FormHelper` object and assigning it to forms `helper` attribute. I can specify some properties on the form tag this will produce and also am adding a submit button to the layout. Now my template code which previously had form tags, a csrf templatetag, enclosing table tags, and a manually constructed submit button has been reduced to

```
{% load crispy_forms_tags %}
{% crispy form %}
```

and the templatetag will produce a bootstratp compatible layout including all the details I had to manage manually before.

12.6 PRACTICE ACTIVITY

Update your registration view to use floppy/crispy forms. Play around with the client side validation. What happens when the form is submitted but isn't completely filled out? Do you have a form that takes advantage of Twitter bootstrap's styles?

12.7 *Even More (Model)Forms* Follow-up

ModelForms can save both form code that creates a form and view code that processes the data from a form. Django crispy_forms and floppyforms allow us to use HTML5 widgets and bootstrap compatible layouts, reducing the template code we have to write by hand. Taken together you should feel confident in your ability to quickly produce forms that are attractive and easy to use and know how to get their data into the database.

Chapter 13

Tying it all together: Lab 1

13.1 Lesson Objectives

We've discussed forms, templates, urls, views, and models. We've also explored using South to manage changes to our database. It's time to define a larger project that will use everything we've learned so far.

13.1.1 What We Want

Our Poll application can evolve into a social app by tying everything to users. We would like to have the following features as part of our application:

1. Allow Registrations. Registered users can use the admin to create polls. Users can only see Polls that belong to themselves.
2. Allow voting on Polls. Voting can only be done by logged in members and votes should be tracked by user so can answer questions like "How many times has Bob voted? On which polls?"
3. Allow Polls to be "favorited". This is in addition to voting.
4. Make the index page show the most active Polls - all time and recently, with the total number of votes, and also show the most favorited polls.
5. Allow users to be favorited.
6. Add a user index page. The user page should show the favorited and voted on polls for a particular user. It should also show the list of favorited users.

13.1.2 How to Get There

You know you're writing a social app when every model has a `user` field! We need a separate model for Votes(user, poll, choice), and either a model for FavoritePolls(poll, user) and FavoriteUsers(user, user) or a way to store two different kinds of favorites in the same table. See https://docs.djangoproject.com/en/dev/ref/contrib/contenttypes/ if this sounds interesting.

We'll want to add one model at a time and generate South migrations to add them to the database. We'll also need to add views to handle "favoriting" polls and users, update our index view, and add a user homepage view. It would be nice if our favorite views could be called via AJAX so that we can add "favorite" buttons many different places in our templates but not have to navigate in order to favorite a poll or user. Finally, we might start by writing integration tests that load each url we expect to function and start implementing our views, one by one until our tests pass.

13.1.3 Real World Data

It would also be nice to see our site with realistic amounts of data. One classic performance mistake often made is to write and test our sites as developers with hand entered data - one or two records of each type. In production, however, the same app might have 100, or 1000, or 10,000 records and frequently performance and UI choices do not scale.

Consider a dropdown box used to select a user on a form from a list - this UI choice works fine for 5 or even 50 users but becomes totally unnavigable if we have 1000 records from which to select.

Similarly template code which creates one query per related record might work fine for a handful of records but take precious seconds of rendering time given a few hundred or thousand records.

To avoid these sorts of errors we ought to populate our database with realistic amounts of sample data. Django includes the concept of data fixtures to enable us to load initial or sample data - but I've generally found that fixtures don't scale very well with large amounts of data or frequent model changes. Instead I typically use the python `random` module and some nonsense text like `lorem ipsum` and write a script which inserts many records into my database programmatically.

A simple example for creating many users might be:

```
from django.contrib.auth.models import User
import random

lorem = [word.strip(".,").lower() for word in lorem]
existing_names = [u.username for u in User.objects.all()]
names = []
while len(names) < 100: #in case we create dupes
    length = random.randrange(6, 12)
    username = ''.join(random.sample("aaabcdeeeefghiiiijklmnooopqrrrsssttuuvwy", length))
    if username not in names and username not in existing_names:
        names.append(username)
for name in names:
    User.objects.create_user(name, name + "@gmail.com", "test")
```

13.2 PRACTICE ACTIVITY

1. Start with model changes - create a `Vote` model and a `Favorite` model and evolve `Choice` as necessary. Be sure to work step by step, generating and running migrations for each change.
2. Create your sample data. It would be nice to have a management command that we could run to populate the database like:

   ```
   $ ./manage.py build_sample_data
   ```

   Adding management commands to Django apps requires a slightly arcane addition of directories and imports. I can never remember it but fortunately django_extensions ships with the `create_command` sub command that will get us started. Run `./manage.py create_command polls` and go looking in `polls/management/commands` to get started. Replace the `NotImplementedError` with a script to populate your database with Users, Polls, Choices, Votes, and Favorites - as many as you think realistic.
3. With our new models and our database populated - lets fill out the index page, rework the detail/vote page, sprinkle favorite/like/+1 buttons everywhere and build the user page. See instructor and classmates work for ideas...

Chapter 14

QuerySets

14.1 Lesson Objectives

We've used the ORM Django provides for our models to create simple queries. Queries can be much more complicated - the ORM has a rich set of operators, a range of methods for querying, and facilities for extending our conditions across relationship between Models.

14.2 QuerySet Basics

We've already seen the use of `.get()` to produce a single Model object and `.all()` and `.filter()` to produce QuerySets.

The QuerySet class is Django's way of representing an SQL query. QuerySets are constructed by accessing a Model's `manager`, by default called `objects`. QuerySets are lazily evaluted:

```
polls = Poll.objects.all()
```

At this point the database query may have been constructed, but it hasn't actually been run! QuerySet's aren't actually run against the database until they have been evaluated in some way.

QuerySet's may also be chained which makes it easy to gradually accumulate conditions. Consider the following code:

```
def get_qs(user=None):
    polls = Poll.objects.filter(is_active=True)
    if user:
        polls = polls.filter(user=user)
    return polls
```

You may not be aware there are other methods that also produce QuerySets. For example the `.exclude()` method is the opposite of the `.filter()` method - excluding all records that match its criteria.

14.3 Spanning Relationships

Consider the following code:

```
>>> p = Poll.objects.all().exclude(user__username="admin")
>>> str(p.query)
'SELECT "polls_poll"."id", "polls_poll"."user_id", "polls_poll"."question",
        "polls_poll"."pub_date"
        FROM "polls_poll"
            INNER JOIN "auth_user"
              ON ("polls_poll"."user_id" = "auth_user"."id")
              WHERE NOT ("auth_user"."username" = admin )'
```

You might notice that the generated SQL query spans a relationship between two models. Django uses a double underscore "__" as an operator to reach from a field that represents a relationship to a field in the related table. When we reach across a relationship the ORM automatically constructs the `JOIN` clause to relate our tables.

Interestingly this behavior is supported for reverse lookups as well - you specify the lowercase name of a model that has a relationship to the parent model and the ORM will figure out the relationship. For instance the Choice model has a ForeignKey field pointing to Poll - but we can refer to the Poll models "choice" field to construct a relationship.

```
>>> p = Poll.objects.filter(choice__id=1)
>>> str(p.query)
'SELECT "polls_poll"."id", "polls_poll"."user_id",
        "polls_poll"."question", "polls_poll"."pub_date"
   FROM "polls_poll"
   INNER JOIN "polls_choice"
      ON ("polls_poll"."id"="polls_choice"."poll_id")
   WHERE "polls_choice"."id" = 1 '
```

Django may process these forward and reverse lookups differently for ManyToMany relationships so be sure to read the documentation at https://docs.djangoproject.com/en/1.3/topics/db/queries/

14.4 Query Operators

The ORM also uses the __ in QuerySet argument names to indicate particular operators. To find all Polls with an id greater than 5 we could say

```
>>> str(Poll.objects.filter(id__gt=5).query)
'SELECT "polls_poll"."id", "polls_poll"."user_id",
     "polls_poll"."question", "polls_poll"."pub_date"
     FROM "polls_poll"
     WHERE "polls_poll"."id" > 5 '
```

Notice that in this case the "__" didn't span a relationship - it altered the sql operator from a equality check to a > operator. Other common operators include `contains` (string searching), `in` (inclusiveness, pass this operator a list to compare against), `gt`, `gte`, `lt`, `lte` (great than, greater than or equal to, etc) and date operators like `year` and `month`. Support for particular operators may vary across database backends so it is worthwhile reading the documentation at https://docs.djangoproject.com/en/1.3/ref/models/querysets/#field-lookups

14.5 PRACTICE ACTIVITY

At this point we should have models that look roughly like:

```
class Poll(models.Model):
    user = models.ForeignKey(User)
    question = models.CharField(max_length=200)

class Choice(models.Model):
    poll = models.ForeignKey(Poll)
    choice = models.CharField(max_length=200)

class Vote(models.Model):
    choice = models.ForeignKey(Choice)
    user = models.ForeignKey(user)
```

Given these relationships construct the following queries in the interactive shell (construct data to meet each criterion as necessary):

- What polls has the admin user voted on?

- Assuming we allow multiple votes per user, how many unique users have voted for a particular poll?
- Can you find votes where the voter also created the poll?

14.6 Aggregation and Annotation

Django's ORM also has QuerySet methods that don't return QuerySets. We've see `.get()` and `.delete()` - but there are also methods aimed at counting and grouping. For instance

```
>>> Poll.objects.all().count()
12
```

Slightly more complicated is the concept of Aggregation. Aggregates give us a single fact about a population. So to find the newest Poll we could say:

```
>>> from django.db.models import Max
>>> Poll.objects.all().aggregate(Max('pub_date'))
{'pub_date__max': datetime.datetime(2012, 3, 12, 11, 28, 20)}
```

Essentially what we've said is to select the max value of the `pub_date` column from the table - and we could have filtered the records returned first if we wanted to.

Also common is the need to return some kind of cumulative operation for each record in a table. This is called *annotation* in Django's ORM and generates a `GROUP BY` query in SQL. So for instance - to count how many choices each Poll object has we could say:

```
>>> from django.db.models import Count
>>> Poll.objects.all().annotate(Count('choice'))
>>> polls = Poll.objects.all().annotate(Count('choice'))
>>> p = polls[0]
>>> p.choice__count
3
```

We call this annotation because it annotates (adds a value) to each resulting Poll object. Notice that where aggregation returns values instead of QuerySets, annotation just adds a value onto the Model objects in the resulting QuerySet. The annotated fields exist in the SQL query, however, so they can then be used in further query operations like `order_by` or even aggregation.

It's also worth noting that the heavy use of the double underscore may sometimes make our annotated field names clash with existing related fields or lookups making further usage ambiguous. We can use named parameters to force the annotated field names (this works for aggregated key names as well).

```
Poll.objects.all().annotate(num_choices=Count('choice')).filter(num_choices__lt=3)
```

Notice that by specifying the name "num_choices" for our annotated field, it is now a valid field name for further querying.

14.7 PRACTICE ACTIVITY

Use the interactive shell to construct queries that answer the following:

- What is the largest number of Choices attached to a Poll?
- What is the largest number of Votes for a Poll?
- Create a query that counts the number of votes cast for each Poll in the database.

14.8 *QuerySets* Follow-up

We've reviewed lookups that span relationships and some of the many operators available. We've also looked at annotating and aggregating values on our QuerySets. QuerySets support a lot more than simple retrieval queries - and there's still more ground we haven't covered. Be sure to review the documentation to see methods for bulk updating, creating OR queries with Q objects, and much more.

Chapter 15

Performance

15.1 Lesson Objectives

We've used the ORM Django provides for our models to create complicated queries - and possibly are beginning to think about the performance of our web application. Performance in Django is mostly a matter of optimizing the data access layer - but we'll also look at using caching to speed up a variety of tasks.

15.2 QuerySet Performance Basics

We should be aware of how our QuerySets are constructed. Consider the main polls page where we are now showing a table with poll question, the owner, the date, and the total number of votes. Our template might look something like:

```
<table>
{% for p in polls %}
    <tr><td>{{ poll }}</td><td>{{ poll.user }}</td>...</tr>
{% endfor %}
```

If we use the debug toolbar to look at our SQL queries we can see that we have 1 query that does a SELECT from the polls_poll table and 1 query per user that does a SELECT from the user table. If we show the top 20 records, we'll require 21 queries. This sort of linear scaling of queries to output lines is a bad performance pattern.

By default simple queries like `Poll.objects.all()` only run against one table. But Django lets us control this behavior by using `select_related()` on our QuerySets. This forces JOINS against all of our ForeignKey related tables. The optional depth parameter lets us limit the number of relationships that are followed and we can also optionally specify specify the specific fields we want followed. See the documentation at https://docs.djangoproject.com/en/dev/ref/models/querysets/#select-related for full details.

As an example - if we added to our view:

```
polls = Poll.objects.all().select_related("user").order_by('-pub_date')[:5]
```

the same template would have only one query no matter how many users were evaluated. *HOWEVER*: `select_related` is not a performance pancea. The resulting JOIN query is more expensive for the database server to construct then the simpler single table version so in some cases multiple simple queries will be faster than a single more expensive query. The golden rule of optimization is "measure"! Don't assume that a change made things faster - measure, find slow spots, and fix them.

15.3 PRACTICE ACTIVITY

Use the debug shell to count the number of queries on your index page. Does select related bring the number down? Is the total execution time less than the combined execution time of the multiple queries before?

15.4 Denormalisation

Another technique we might use to improve performance is database denormalisation. Denormalisation means introducing redundancy in our data with an eye towards improving performance. Consider the Vote totals for a Poll. We can annotate a Poll query to provide a count of total Votes objects - but this causes a relatively expensive JOIN and GROUP BY query. What if we added a `Poll.votes` column directly on our Poll model? Now displaying the number of votes would have no additional performance impact - but of course we would need to update the `.votes` value every time a Vote was added to the Poll or our number would be inaccurate.

Of course - this is the sort of number that *could* be inaccurate. Do we really care if our displayed vote totals lags slightly behind the actual state of our data? Probably not!

15.5 Signals

We can trigger code to update denormalized data in a variety of ways. We could overwrite the 'Vote` objects' `.save()` method and update the `Poll` the `Vote` belongs to. This would actually keep our denormalized field perfectly in sync. It might be easier to use a signal handler in our `models.py` to do the same thing:

```
from django.db.models.signals import post_save
from django.dispatch import receiver

@receiver(post_save, sender=Vote)
def update_votetotal(sender, **kwargs):
    vote = kwargs['instance']
    poll = vote.choice.poll
    poll.num_votes = Vote.objects.filter(choice__poll=poll).count()
    poll.save()
```

Of course this adds an additional write every time a vote is cast. That might be OK - we probably have less writes than reads. Of course if the particular records we are concerned about is the high traffic Polls on the front page - we might be able to get away with only updating the `.num_votes` *sometimes* when we catch the signal. Flip a coin (virtually speaking of course: the random module would be helpful here) and if it's heads we could update our record.

Even further - updating denormalized fields is exactly the sort of thing that can be handled separately from the actual process of rendering an HTTP response. An extremely common performance pattern on CPU intensive or high traffic apps is to use a jobs queue to tackle tasks that can occur "later" but don't have to be accomplished before a response is returned.

15.6 PRACTICE ACTIVITY

Update the front page of your polls app to show "total votes" and "total favorites" using denormalized fields updated by a signal. Take out any annotation queries that were calculating Vote totals and be sure to compare the before/after SQL execution time.

15.7 Caching

Of course denormalisation is just one special application of the general technique called caching. Caching simply means that we should make an effort to remember the answer to an intensive operation and return the cached version instead of re-calculating the next time the same operation is invoked.

Django has support for a variety of cache backends as we'll discuss in class. For now you can test caching by adding

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.locmem.LocMemCache',
     }
}
```

to your `settings.py` file. In a production environment memcached should always be your first choice but memory constrained servers may still get performance benefits by using a file-based cache. As always - measure, change, and measure again!

Django's cache api supports caching via templatetags, decorators on views, functions applied to views in `urls.py` and even just turning on a whole-site cache via settings. You can also use the Caching API manually to save (almost) any arbitrary value to the cache and retrieve it with a key.

15.8 Practice Activity

For now let's just add a caching decorator to our index view to see how the cache works. Given a view named `index` we can add the following code:

```
from django.views.decorators.cache import cache_page

@cache_page(60) #After 60 seconds the cached record will expire
def index(request):
    # existing view code should be unmodified
```

Use the debug toolbar to record the total CPU time and the SQL execution time on cache misses and cache hits.

15.9 *Performance* Follow-up

Web application performance has many pieces - but you'll get a good start by profiling your database queries, applying denormalisations where appropriate, and adding a caching layer to heavily viewed dynamic pages. For much more about site speed - including client side rendering speed - see the Performance best practices document at http://code.google.com/speed/page-speed/docs/rules_intro.html

Chapter 16

Deploying Django Web Apps

16.1 Lesson Objectives

Deployment is a common source of confusion for newcomers to Django. In this lesson we'll discuss the execution model and the two primary deployment styles.

16.2 Execution Model

If you're coming from another web development technology like PHP you might be used to a different execution model for applications. PHP applications are usually run anew upon every request - when the webserver receives a request it passes your entire application to an already running PHP process (perhaps using FastCGI) or even starts a new PHP process to handle a request and then terminates the process at when the application terminates.

Django expects to start an application process or thread and then have the application serve multiple requests. You can actually deploy Django using CGI without any long-lived process, but the architecture doesn't support that pattern well and performance is very poor.

16.2.1 Where should my application live?

There are basically two approaches to repeatedly handing requests to a long-running application. We could have the webserver start our application and manage process or thread lifetime. Alternatively we could start our application using some other method and tell the webserver how to talk to it via a socket or proxy.

The first style is typically implemented with the WSGI protocol. WSGI defines a generic standard for starting up applications and handing them requests and any program in any language that can provide WSGI support can be embedded in a webserver that speaks WSGI. Most commonly used web servers can use wsgi - either natively or through an add-on like mod_wsgi.

Alternatively we could configure our webserver simply to route certain requests to a remote program it didn't start. We could use the FastCGI protocol for this (essentially - agree on a shared socket to use for input and output) or even have our application start its own webserver for handling HTTP requests and use HTTP level proxying to pass certain requests that arrive at our production webserver on to our application HTTP server.

16.2.2 Tradeoffs - and an alternative.

There are a variety of implications to choosing, say, mod_wsgi in Apache over Apache and FastCGI. On the one hand you don't have to manually manage your application execution - Apache can be configured to start your application automatically. On the other hand this means your application is embedded in each Apache worker and if you have a large number of Apache workers you might have high memory usage because of the multiple copies of your application.

On the other hand running your application separately may provide you finer granularity over your performance (no embedded application, web server restarts don't restart your application) but now you are responsible for starting and monitoring your application process - perhaps using a tool like Supervisord.

There isn't a simple and consistent answer to the complicated issue of deployment. But it might be worth checking out the many emerging Cloud-based application hosts - companies like Gondor and Heroku promise to take care of deployment issues for you transparently while allowing more complicated performance patterns like adding additional application servers dynamically.

16.3 Scripting Deployment with Fabric

Ideally we'd like to be able to script our deployment to remote servers. Django itself provides no facility for writing deployment scripts but a common tool used to script deployment is Fabric.

Fabric presents an api to let us do the kinds of things we typically need to do to deploy to remote Linux servers: run shell commands, transfer files via SSH, and run commands remotely. Install fabric and look at the following sample `fabfile` to see how it works.

```
from fabric.api import run, local, cd, put

def build():
    # could check out from svn, run tests, build zip, etc
    local("pip freeze > requirements.txt")

def transfer():
    run("mkdir ~/newsite")
    with cd("~/newsite"):
        # could transfer tarball, symlink new version of code, touch
        # .wsgi file, update db etc
        put("requirements.txt")
        run('virtualenv ENV')
        run('pip install -E ENV -r requirements.txt')


def deploy():
    build()
    transfer()
```

Notice that we're taking advantage of PIP's ability to list each package it has installed. Be sure to look at the format of `requirements.txt` - it includes precise version numbers of our packages to make the new environment as much like the old one as possible.

Installing Fabric not only gives you the API above - simple functions like `run`, `local`, `put` and so on - it also gives you a new command `fab`. The fab command looks in the current directory for a script called `fabfile.py` by default, takes arguments to specify what hosts to run against, and exposes each function in your fabfile as an argument to fab. To run our fabfile we could save it in the parent directory of our django project as `fabfile.py` and run

```
fab -u myusername -H localhost deploy
```

Try it out! Note that remotely running commands is only supported over SSH so target machines must be SSH servers.

16.4 Practice Activity

Our sample fab script builds a Django environment, but we've forgotten our code! Obviously we will probably want to talk to source control tools when building real deploy scripts, but for now can you use `tar` or `zip` to package up the source for your django project and replicate the django project you have created in a new directory. Make sure you have ssh enabled on your own box and create a `fabfile` that replicates your project in a new directory on "remote host" `localhost`.

Be sure to check out the fabric docs at http://docs.fabfile.org/en/1.4.1/index.html

16.5 *Deployment* Follow-up

Deployment can be a complicated topic with Django but automating our deployment is not such a difficult task. For medium sized projects my `fabfile` scripts are usually less than 100 lines - and that includes checking out source code, installing packages, running migrations, and restarting apache. Advanced users with very complicated multi-server setups might also want to check out tools like `chef` and `puppet`.

Chapter 17

Using Celery

17.1 Lesson Objectives

In this lesson we'll discuss the advantages of using a Job Queue to execute tasks asynchronously. We'll learn to use a *development-only* configuration of Celery to learn to write asynchronously executing tasks.

17.2 Celery and Asynchronous Jobs

A common pattern used by web applications to increase both speed and scalability is to outsource units of work to a Job Queue to be done asynchronously. This increases our speed by allowing us to immediately return a response instead of waiting for the work to be finished before returning a response. It also increases our scalability because many job queues support remote worker clients that perform the tasks on the queue but live on other machines.

Celery is a popular Job Queue written in Python. It has support for several storage backends, such as RabbitMQ or Redis, and has good integration with Django via the django-celery project.

Celery calls itself a "Distributed Task Queue" but can be used as a simple Job Queue. The Job Queue model dates all the way back to early mainframes and the basic idea remains the same. The server accepts jobs, puts them in storage of some kind, and client workers request jobs from the server and execute them upon reception.

Getting Celery set up properly means installing and configuring a storage backend to store jobs. The preferred storage backend is RabbitMQ but installing and configuring RabbitMQ is beyond the scope of this class. Fortunately Celery also has support for using a database via Django's ORM.

Real world usage of Celery also requires running Celery as a service automatically and possibly monitoring it to make sure it stays up, doesn't get too full, etc. For our example, however, we'll be running the `celeryd` management command manually so we can see what's going on and get started without additional configuration.

I should hasten to point out that this is not the recommended way of running Celery. Think about using Celery with the Django ORM and manually running `celeryd` command as a sort of "runserver" for distributed tasks - great for development but not suitable for production.

17.3 Configuring celery for development

We'll need to install `celery` and `django-celery` and provide some configuration.

1. pip install celery and django-celery.
2. add `djcelery` and also `kombu.transport.django` to your installed apps

3. Add the following settings to your `settings.py`:

```
import djcelery
djcelery.setup_loader()
BROKER_URL = "django://"
```

4. Add a `tasks.py` to your `polls` app. `tasks.py` should contain

```
from celery.task import task

@task()
def add(x, y):
    return x + y
```

5. Start the celery server up with

```
$ ./manage.py celeryd -l INFO
```

6. Run your task from the interactive Python shell

```
>>> from polls.tasks import add
>>> add.delay(4,5)
```

Be sure to look at the `celeryd` server to see the task received and run...

17.4 PRACTICE ACTIVITY

Obviously our task doesn't actually do anything particularly useful. Go ahead and write a more useful function - one that counts the number of votes for a Poll and stores the result on the Poll.num_votes field. Can you call this task asynchronously from a view? Be sure to watch the output of `celeryd` to see your tasks executing.

Tip

Since celery clients can be separate threads or even processes on another machine and run at some later time be sure to keep your tasks as simple and stateless as possible. Don't pass Django ORM objects to Celery tasks. Instead pass in the primary key (object.id) and have the task get the object from the database with a query.

17.5 *Celery* Follow-up

Be sure to check out the extensive docs at http://docs.celeryproject.org/en/latest/index.html and http://django-celery.readthedocs.org/-en/latest/index.html for more details. Real world usage of Celery requires a bit more configuration to Celery and a storage backend running as services - but we know enough to write our web application taking advantage of asynchronous capabilities.

Chapter 18

REST with Tastypie

18.1 Lesson Objectives

In this lesson we'll learn to use Tastypie to generate a REST API for our application. We'll learn to limit the information Tastypie shares and how to manually add reverse relationships. We'll also see how we might consume the data Tastypie exposes with a Javascript client.

18.2 REST API's

Web applications frequently also provide an API - a web interface designed for computers rather than for humans. REST API's tend to follow design principles that work with the nature of the web - using hierarchical URI's to specify the location of resources and accepting HTTP verbs (not just GET and POST but others like PUT and DELETE) on URI's to perform actions.

We'll provide a read-only API for now and save the configuration of authentication in Tastypie for another chapter.

Tastypie has a familiar feel - we need to define resources as classes with fields that look much like Models or Fields. There is no requirement but convention is to add an `api.py` file to our application and define any resources there.

```
from tastypie.resources import ModelResource
from tastypie import fields
from django.contrib.auth.models import User
from polls.models import Poll, Choice

class UserResource(ModelResource):
    class Meta:
        queryset = Users.objects.all()
        resource_name = "user"

class PollResource(ModelResource):
    user = fields.ForeignKey(UserResource, 'user')
    class Meta:
        queryset = Poll.objects.all()
        resource_name = "poll"

class ChoiceResource(ModelResource):
    poll = fields.ForeignKey(PollResource, 'poll')
    class Meta:
        queryset = Choice.objects.all()
        resource_name = "choice"
```

Now to see our resources we need to configure our `urls.py`. Tastypie has several different ways of handling resource urls but the best way if you'll be exposing multiple resources is to use the built-in support for grouping resources into an api. Adding to our `polls/urls.py`

```
from polls.api import PollResource, ChoiceResource, UserResource
from tastypie.api import Api

v1_api = Api(api_name="v1")
v1_api.register(ChoiceResource())
v1_api.register(PollResource())
v1_api.register(UserResource())
```

and the single url

```
    url(r"^api/", include(v1_api.urls)),
```

allows us to visit http://localhost:8000/polls/api/v1/poll/1/?format=json and see JSON for the poll whose primary key is 1. This data looks like:

```
{"id": "1",
 "is_active": false,
 "pub_date": "2011-01-25T00:00:00",
 "question": "Laboris enim  id ?",
 "resource_uri": "/polls/api/v1/poll/1/",
 "slug": "",
 "user": "/polls/api/v1/user/22/",
 "vote_total": 0}
```

Tastypie by default exposes all the fields that belong to our model. Note also that following REST principles the URI's for related objects are in this document - `user` is a resource so its value is a URI which we could follow to see the resource representing a User.

We probably don't want to expose all the fields for our objects. The UserResource object probably should show nothing more than usernames while the PollResource might want to hide the calculated vote_total field. Tastypie lets us specify either `excludes` - a blacklist of fields not to include - or specify the list of fields to include in `fields`. We also would like to include reverse relationships - the Poll resources should have a list of Choice resources as well. We have to build a `ToManyField` field on the PollResource that points to the resource that will be returned and the member of the Poll class that provides a manager. Our PollResource now looks like

```
class PollResource(ModelResource):
    user = fields.ForeignKey(UserResource, "user")
    choices = fields.ToManyField("polls.api.ChoiceResource", "choice_set")
    class Meta:
        queryset = Poll.objects.all()
        resource_name = "poll"
        excludes = ["vote_total"]
```

Tastypie is extremely configurable - even just looking at the read side of our API we can allow filters and orderings so that constructed urls can return only the objects we want in the order we want. Looking at the root url for our API also reveals that we can do limit queries - look at http://localhost:8000/polls/api/v1/poll/?format=json and see if you can figure out how to do a limit query in the url.

18.3 Consuming our API

REST API's can be used for all sorts of things but for now we'll just use our Read-only API to implement a carousel widget that displays the most recently created polls. This will require adding a little bit of Javascript to load our API urls and insert the resulting data into our html page. The following will do roughly what we want and should be straightforward if you are familiar with JQuery:

```
<script type="text/javascript"
   src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js"></script>
<script type="text/javascript">
```

```
  var polls = [];
  var current = 0;
  function show_next(){
      var poll = polls[current];
      var link = "<a href=\"" + poll.resource_uri + "polls\">";
      link += poll.question + "</a>";
      $("#scroller").html(link);
      current = (current + 1) % polls.length;
  }
  $(function(){
      $.getJSON("/polls/api/v1/poll/?format=json&limit=5",
             function(data){
                 $.each(data.objects, function(k, v){polls.push(v);});
                 show_next();
             });
      $("#next").click(show_next);
   });
</script>
<div>
 <a href="#" style="float:left;">&lt; </a>
   <div id="scroller" style="height:20px;width:400px;float:left;"></div>
 <a id="next" href="#" style="float:left;"> &gt;</a>
</div>
```

This sort of usage is not the only way to consume our API - but it does demonstrate the ease with which we can manipulate the resources being exposed and parse their representations.

18.4 PRACTICE ACTIVITY

Install Tastypie (`pip install django-tastypie`, plus add to your `INSTALLED_APPS`). Read the tutorial at http://django-tastypie.readthedocs.org/en/latest/tutorial.html and configure your Resources so that you can view at least Users, Polls, and Choices. Our carousel currently fetches the first 5 items - enable ordering by `pub_date` for the `PollResource` and verify that the carousel widget loads the 5 most recently created polls.

18.5 *REST with Tastypie* Follow-up

To fully explore supporting a REST API we would also have to discuss throttling, Authentication, Caching and Tastypie's hooks for serializing and deserializing our models. Even at this level, however, we can quickly get data exposed that our website's UI might want to consume and have the foundation in place to expose more sophisticated functionality as needed.