Introduction to Django

Chander Ganesan
OSCON 2010

One Copley Parkway, Suite 210 Morrisville, NC 27560 Phone: 919.463.0999 Fax: 866-229-3386
www.opentechnologygroup.com

Table of Contents

Introduction

About Django

- High level web framework
  - Basic modules, classes, and tools to quickly develop and deploy web apps
  - Contains an ORM (Object-Relational Mapper) that allows for the use of standard Python language syntax when interfacing with a back-end database.
    - Developer need not learn SQL, DDL, etc!
  - Provides a template framework that allows HTML, XML, and other "documents" to be converted into "templates", which can then be converted to "output" via a wide range of substitution techniques.
  - Elegant URL support (fancy, beautiful URL's, no "?blah=blah", etc.)
  - Multi-lingual
- Fast and easy to use, robust, flexible, and lots of contributed components available!
  - Built in administrative interface to manage data models.
  - Built-in authentication/access control components
  - Contributed geospatial support components (GeoDjango)
  - Extensible!

Installing & Configuring Django Components

Django Pre-Requisites

- Python 2.3 or higher
  - No Python 3.0 support (yet)
- Database drivers for the database you wish to use
  - PostgreSQL, Oracle, SQLite, MySQL
- Web Server
  - Django has a built-in web server for development
    - Handles a single connection at a time.
    - Not security-tested
    - By default listens only for "local" connections.
  - Apache, IIS, etc. for deployment

One Copley Parkway, Suite 210 Morrisville, NC 27560 Phone: 919.463.0999 Fax: 866-229-3386
www.opentechnologygroup.com

Downloading & Installing Django

- Django can be downloaded from http://www.djangoproject.com/download
- Unpack and install Django
  - Unpack the sources and run the command below from the Django source directory as the root user:
    `python setup.py install`
    - This command works the same on Windows as it does in the Linux environment, you just need to make sure you call the correct interpreter.

```
import django
print django.VERSION
```

Sample code to verify that Django is installed correctly.

Choosing a Database

- Current version of Django provide support for three databases:
  - PostgreSQL using the psycopg and psycopg2 drivers .
    - Unix versions can be downloaded from http://initd.org/pub/software/psycopg/
    - Windows version are available at http://stickpeople.com/projects/python/win-psycopg/
  - MySQL using the MySQLdb package.
    - This package is available at http://sf.net/projects/mysql-python/ .
  - SQLite 3 using the pysqlite package.
    - This package is available at http://trac.edgewall.org/wiki/PySqlite .
      - SQLite support is included in Python version 2.5 and newer.
- Django supports all of these databases in the same manner, so which you choose to use here makes little difference, since everything is coded in the same way.
- When you move to your own development system, you'll need to choose a database based on your needs and the software deployed in your enterprise.

The Django ORM makes writing an application that works with multiple databases easy, since the ORM takes care of the database interaction and schema generation.

Creating a New Project

- The **`django-admin.py`** tool is used to create a directory and create "default" files for a new Django project.
  - A project is generally made up of one or more "apps"
  - Functional code is contained within the apps.
- Use the **`django-admin.py`** script to create the new project files.

  `django-admin.py startproject` ***`name`***
  - The startproject argument tells the command that we need it to initialize a new Django project.
  - The *name* argument allows us to specify the name of the project to be created.
  - When the command runs, it will create a new directory called *name* in the current directory, and populate it with a few files:
    - **`__init__.py`** : A file that causes Python to treat the directory as a package.
    - **`manage.py`**: A command line utility to communicate with Django.
    - **`settings.py`**: Configuration settings for the project.
    - **`urls.py`**: URL declarations for the project. The URL's tell Django what to do when certain URL's are put into the browser.

On Unix (and Linux) systems, the **`django-admin.py`** command will be an executable under your system PATH. As a result, you can just type in the command to run it.
On Windows systems, this script normally gets installed in the 'Scripts' directory under your Python installation, so you'll need to either add that directory to your search path, or run the command by specifying the full path to the file.

Starting the Test Server

- Once you've created your project, you can test it by starting up the Django development server.
  - You can start the server using the command:
    `python manage.py runserver`
    - This command will start up the server and listen on port 8000 of your system.
    - You can use your web browser to connect to the server using the URL the command returns.

```
/> python.exe manage.py runserver
Validating models...
0 errors found

Django version 1.0.2 final, using settings 'demo.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

Running the Django development web server.

Creating a New Application

- Once you've created a project, you'll need to create an application.
  - Django models and views are handled through the application itself – not the project, which consists of primarily the controller and base settings (such as data store information)
- You can begin a new app by opening a shell window in your project directory and issuing the 'startapp' command.

```
manage.py startapp html_exercises
```

  - Like projects, you should avoid using reserved words for your app name, otherwise you might run into problems later.
- Once you've created a new app (or installed an app from a third party), you'll want to add it to the list of INSTALLED_APPS in your projects 'settings.py' file.
  - Apps are not actually used until they are installed, so you'll want to make sure you install your app, or it won't be usable in your project.

Django Contributed Apps

- Django provides some contributed apps for a wide range of different tasks.
- Using these apps is simply a matter of including the app you wish to use in INSTALLED_APPS in your projects 'settings.py' file.

Building Django Applications

One Copley Parkway, Suite 210 Morrisville, NC 27560 Phone: 919.463.0999 Fax: 866-229-3386
www.opentechnologygroup.com

Overview of A Typical Django Request

- When a request comes in for a Django application via the web, it typically follows a specific path:
  1. Django uses a set of regular expression matching patterns based on the URL.
     - Each expression is tied to a function (called a "view" in Django parlance), when a match is found the view is executed.
  2. Views are passed a HTTPRequest object (and possibly other information) - which represents the "state" of the inbound request (meta-parameters, GET, POST, COOKIE, etc. are all contained in the request object)
  3. The view may make a request for data using the ORM, and uses the information obtained to populate a dictionary with parameters that can be used to compose the response.
  4. A template is chosen, and used along with the information in (3) above to substitute data and generate some output data.
  5. The output data (4) is made part of an HTTPResponse object, which also contains HTTP headers that should be sent to the client.
  6. The HTTPResponse object is returned by the view function, and Django uses it to generate the response back to the client.

Setting up URL Patterns

- When requests come in to Django via the web, they come in via a request for a particular URL.
- That URL (the portion after the '/' after the host name) is matched against a set of regular expressions to determine the view to call.
- The URL patterns are processed in the order that they are entered, so "specific" expressions should appear before generic expressions.
- Regular expressions can also be used to pass named parameters to view functions (the first argument to the view will always be the HttpRequest object, the others can be passed based on the URL, such as in the example here)

```
from django.conf.urls.defaults import *
from project.appname.views import welcome, date_view
urlpatterns = patterns('',
    (r'^$', 'welcome'),
    (r'^(?P<mon>\w+)/(?P<year>\d{4})', 'date_view')
)
```

A sample URL pattern. In this case when the URL is empty, the project.appname.views.welcome function is called and passed a HTTPRequest object.

The second URL pattern grabs the month and year from the URL and calls the date view with named arguments of 'mon' and 'year'

```
http://localhost:8000/jan/2007
date_view(HttpRequest,month='jan',year='2007')
```

The URL shown here would (when processed by the URL patterns shown above) cause the function call shown below it.

- The first argument to the patterns() function can be the name of the module that the patterns are contained in (thus eliminating the need to import the module)

```
from django.conf.urls.defaults import *

urlpatterns = patterns('django.contrib.auth.views',
    (r'^accounts/login/$','login' ),
    (r'^password_reset/$', 'password_reset'),
    (r'^password_reset/done/$', 'password_reset_done'),
    (r'^reset/(?P<uidb36>[0-9A-Za-z]+)-(?P<token>.+)/$', 'password_reset_confirm'),
    (r'^reset/done/$', 'password_reset_complete'),
    (r'^password_change/$', 'password_change'),
    (r'^password_change/done/$', 'password_change_done'),
)
urlpatterns += patterns('project.appname.views',
    (r'^$','welcome'),
    (r'^(?P<mon>\w+)/(?P<year>\d{4})', 'date_view')
)
```

Setting up URL patterns for multiple applications in a single project.

- Note that we can add multiple patterns together (as shown here)
- There are a variety of other ways to split URL patterns into individual applications.

Django Views

- View functions, as we already learned are passed in some arguments:
  - HttpRequest objects
  - Parameters from URLs
- View functions should return HttpResponse objects, or a subclass of one, such as:

Object	Description
`HttpResponse`	A "normal" response (the argument is the content to send to the user)
`HttpResponseRedirect`	A redirect (302) response (the argument is the URL to redirect to)
`HttpResponseForbidden`	Just like HttpResponse but returns a 403 status code
`HttpResponseServerError`	Just like HttpResponse but returns a 500 status code
`HttpResponseNotFound`	Just like HttpResponse but returns a 404 status code

- The constructor to HttpResponse accepts an argument that is the data to be returned in the response, and optionally accepts a mimetype argument (the MIME type of the data being returned)
- There are more HttpResponse subclasses - check out the documentation!

More About HttpResponse

- The HttpResponse object works much like a file - you can use the 'write' method of the object to add information to the output.
  - Alternately, you can pass it an iterator that returns strings and it will use the iterator to generate the output.
- If you treat a HttpResponse object like a dictionary, you add/remove headers.
- The HttpResponse object has set_cookie and delete_cookie methods that may be used to set and remove cookies (refer to the documentation for more information.)

```
from django.http import  HttpResponse
def getDate(request, year, mon):
    r=HttpResponse('''<html>
                   <head><title>Time Example</title></head>
                   <body><p>The date you asked for was %s, %s</p>
                   </body></html>''' % (mon,year))
    r['Pragma'] = 'no-cache'
    return r
```

  - The example above sets the output content, as well as a single header.

Django Templates

One Copley Parkway, Suite 210 Morrisville, NC 27560 Phone: 919.463.0999 Fax: 866-229-3386
www.opentechnologygroup.com

Template Basics

- In Django, a template is a file that contains substitution parameters.
- Parameters are assigned within the view, and the template is loaded, parsed, and substitution takes place.
- Templates allow for the separation of display and business logic, and make it easy for designers to create pages, and programmers to populate them with dynamic data.
- Using a template typically involves a few steps:
  1. Assign objects into variables used for template substitution
  2. Call a function to perform substitution
- Although there are a number of ways to work with templates (and render them), here we'll focus on the 'render_to_response' method.

Storing Template Files

- The location if template files is specified using the settings.py file.
  - The TEMPLATE_DIRS setting is a list that contains the directories that Django should search when loading new Templates.
    - These must always be **absolute** directory references, no relative directories allowed!
  - This allows us to locate our template files in a separate directory and location from our other files and project components.
  - By doing this, we can gain better organizational control over our template files/locations.

```
TEMPLATE_DIRS = (
    "/var/www/project1/templates"
)
```

    - The example above shows a modified TEMPLATE_DIRS based on the location of project templates for this project.

Easy Rendering of Templates

- Django provides an easy way to accomplish template processing/substitution - it's called 'render_to_response'.
  - In order to use render_to_response, you must first import it from django.shortcuts.
- The function loads a template (the first argument) and uses a dictionary (the second argument) for substitution.
- The function returns an HttpResponse object that contains the rendered template.

```
from django.shortcuts import render_to_response
from datetime import datetime

def template_ex1(request):
    c= {'month' : 'Jan',
        'day'   : 'Tuesday',
        'people': ['JOE','Mark','Alan'],
        'time'  : datetime.today(),
        'names' : {'Jones': 'Alan',
                   'Anderson': 'Jim',
                   'Smith': 'Jonas' } }
    return render_to_response('template_ex1.html', c)
```

Using render to response, here we simply pass a template name and dictionary into the render_to_response function and it will generate our HttpResponse object.

Media Files

- Django focuses on the dynamic content generated for your pages and/or applications, but really isn't designed to serve up your graphics/media files (with the obvious exception being dynamically generated media).
- In fact, the general recommendation is to have your media files served from a totally separate web server, thus reducing the load on the server running Django.
  - In order to do this, there are several configuration steps required to make sure that Django knows the location of media files - and to allow our applications to use the appropriate URL's for media.
  - These settings are applied in your settings.py when you deploy in a production environment, but with the test server, Django is unable to serve up such media.

We can also make the test server serve up media files as well (by using the built-in 'django.views.static.serve' view), but this is only for development.

```
urlpatterns += patterns('',
    (r'^/?site_media/(?P<path>.*)$', 'django.views.static.serve',
                    {'document_root': '/var/www/media'}),
)
```

Setting up Django to serve up media using a built-in view.

Media Settings

- Later, when we populate templates with data, we'll use the {{MEDIA_URL}} tag to indicate the base URL to be used for media.
  - This location is noted in settings.py, and allows you to move/relocate the media and make the base URL change in a single location.
- In addition to MEDIA_URL, Django also has a MEDIA_PATH setting that indicates the full path to your media file.
  - This may be used, for example, with file uploads, etc. where we want to add new media to the site via the Django interface.
  - The directory is also used by Django's development server when we use the 'django.view.static.serve' view (allowing us to have a development server that serves the media as well).

Basic Template Substitutions

- When processing templates, Django looks for keywords appearing enclosed in a set of double braces.

```
    c={'month' : 'Jan',
       'day'   : 'Tuesday',
       'people': ['Joe','Mark','Alan'],
       'time'  : datetime.today(),
       'names' : {'Jones': 'Alan',
                  'Anderson': 'Jim',
                  'Smith': 'Jonas' } })
```

- It replaces the variable named inside the braces with the template variable corresponding to the variable.
- Following the variable by a '.' and then another names causes that method or attribute to be called or retrieved.

```
<html><head><title>Sample template</title></head><body>
The current month is {{ month }} and today is a {{ day }}<br>
<!-- Calling the now() method of the time object -->
Oops! I meant that the current date is {{ time.now }}<br>
<!-- Retrieving the Jones key of the names dictionary -->
The person whose last name is Jones has a first name of
{{ names.Jones }}<br>
<!-- Retrieving index 0 of the people list -->
The next person in line is {{ people.0 }}
</body></html>
```

A sample HTML template that goes with the previously defined Context object.

Template Filters

- In some cases, we might have data that we wish to set a single time in a context, but perhaps needs to be used differently in different places in the template itself.
- In order to accomplish such tasks, we can use a *template filter* .
  - Template filters are specified by specifying the variable, a vertical pipe (|), and then the name of the filter .
  - The filter itself may be followed by a ":", and then arguments (if allowed) for the template.
  - Template filters can be chained to one another using additional pipes.
- Your Django cheat sheet contains a list of Template filters, along with a basic description of what they do and how they work.

```
The next person in line is {{ people.0|lower|capfirst }}
```

A sample template filter that changes a name from any case to capfirst case.

One Copley Parkway, Suite 210 Morrisville, NC 27560 Phone: 919.463.0999 Fax: 866-229-3386
www.opentechnologygroup.com

Template Tags

- Template tags allow us to have Templates do a bit more than generate simple output.
- Tags are designed to allow us to do looping, decision making, or other special actions within a template, and are most commonly used for performing loop type operations (such as as generating a drop down from a list).
- The most commonly used Django tags include:

Tag/Example	Description

`{% if %}`
 	
`{% endif %}`
`{% if person.0 %}`
`  The person's name is {% person.0 %}`
`{% else %}`
`  Sorry, no person exists!`
`{% endif %}`
	This tag evaluates a variable, and if the variable is true, the system will display everything between it and the corresponding endif.
	An optional {% else %} tag exists as well.
	This tag can also use words like 'not', 'and', 'or', and grouping.
	There is no {% elseif %} tag, but this can be accomplished with a nested {% if %} tag.

`{% for %}`
`{% endfor %}`
`{% for person in people %}`
	This tag works like a Python for statement, (for X in Y). This ends with a corresponding {% endfor %}

{{ person }} {% endfor %}	For tags may be nested, and will work for both dictionaries and lists, as well as an other iterable object.

- Your Django cheat sheet contains a list of all Django template tags.
- We'll discuss the most important tag - the {for} loop, next!

More on For Loops

- For loops are fairly common when iterating over lists of things to generate tables, lists, or other common HTML page components.
- In many cases, however, we end up needing to know a bit more:
  - The template should be able to determine when it is just entering the loop (so it can, for example, generate the starting table tags.
  - In order to make these determinations, there are several special template variables defined inside of for loops:

Variable Name	Description
`{{ forloop.counter }}`	An integer representing the number of times the loop has been entered (i.e., the first time through it is set to 1)
`{{ forloop.revcounter }}`	An integer representing the number of items remaining in the loop (i.e., the last time through this is set to 1)
`{{ forloop.first }}`	A boolean that is set to True the first time through a loop.
`{{ forloop.last }}`	A boolean set to True the last time through a

	loop.

`{{ forloop.parentloop.counter }}`
`{{ forloop.parentloop.revcounter }}`
`{{ forloop.parentloop.first }}`
`{{ forloop.parentloop.last }}`
	The parentloop variables are used to determine the status of the parent loop (in the case of a nested for loop).

RequestContext Processors

```
from django.template import RequestContext
from django.shortcuts import render_to_response

def site_context(request):
    return {'company': 'Open Technology Group, Inc.',
            'user': request.user,
            'remote_ip': request.META['REMOTE_ADDR'],
            'logo': 'http://otg-nc.com/images/OTGlogo.gif'}

def IntroView(request):
    return render_to_response('intro.html',
           { 'title': 'Welcome to my Site'},
           context_instance=RequestContext(request,
                            processors=[site_context]))
```

Using a RequestContext processor to provide some default settings. This RequestContext might be used for every view in this site.

- When rendering a template we've been building Context objects to contain the substitution parameters for the template.
  - Django provides a separate type of context known as django.template.RequestContext

- RequestContext processors work mostly the same as regular Context's, with the exception that they add a bunch of "default" variables to the template context.
  - They automatically populate the context using the context processors found in the settings.py TEMPLATE_CONTEXT_PROCESSORS setting.

```
from django.template import RequestContext
from django.shortcuts import render_to_response

def IntroView(request):
    return render_to_response('intro.html',
           { 'title': 'Welcome to my Site'},
           context_instance=RequestContext(request))
```

Using only Django's default RequestContext processors to provide default Context data.

Global Context Processors

- In order to make things even easier, Django provides us with the ability to have a set of global context processors.
- These are processors that should always be applied to RequestContext (so you no longer need to apply the processors manually)

You should notice that TEMPLATE_CONTEXT_PROCESSORS does not exist in settings.py. This is because it is set by default. To change the setting (add new context processors to the list – even your own), you'll need to add a TEMPLATE_CONTEXT_PROCESSORS line to your settings.py and set it as desired.

```
TEMPLATE_CONTEXT_PROCESSORS = ("django.core.context_processors.auth",
                               "django.core.context_processors.debug",
                               "django.core.context_processors.i18n",
                               "django.core.context_processors.media")
```

Djangos default TEMPLATE_CONTEXT_PROCESSORS. You can add your own to this as well.

- The setting for the site-wide context processors needs to be added to settings.py. If it isn't there, it defaults to the setting shown above.
- More information about default context processors is in the Django documentation.

Database Models with Django

About Database Models

- Since modern web applications typically use a back-end data store to manage and manipulate data, it makes sense that Django incorporates a mechanism to automatically handle data inside the database.
- Django does this by providing on Object-Relational-Mapper
- A Django model is a method used to describe how data is represented in the database.
- Relationships between models are analogous to relationships between tables in a database.

Configuring Django for Database Access

- In order for Django to manage data within the database, we first need to configure the database access parameters.
  - These parameters are in the settings.py file for your project, and are described in the table below:

Parameter	Description
`DATABASE_ENGINE`	The database engine that Django will be using. Valid values are (all in lower case): `postgresql` - PostgreSQL using psycopg **`postgresql_psycopg2`** - PostgreSQL using psycopg2 (used here) `mysql` - MySQL `sqlite3` - Sqlite `oracle` - Oracle Be sure the driver is installed for the database you wish to use prior to trying to use it!
`DATABASE_NAME`	The name of the database (or, in the case of SQLite, the path to the database file)
`DATABASE_USER`	The username to use when connecting to the database.

DATABASE_PASSWORD	The password to use when connecting to the database.
DATABASE_HOST	The hostname on which the database may be accessed.
DATABASE_PORT	The port number to use to connect to the database.

- Once you've set your parameters, you can test that your database is accessible using the following command line commands (from within your project directory):

```
python manage.py shell
from django.db import connection
cursor=connection.cursor()
```

Defining Django Models

- If you look at the models.py file that is contained in a new app, you'll notice that it (by default) contains a line such as:
  ```
  from django.db import models
  ```
  - Django uses a subclass of django.db.models.Model to represent each model.
- In order to create a new model (the analog of a table in the database), we'll need to create a class that inherits from django.db.models.Model
  - The class will contain a number of class attributes, and each attribute will usually correspond with a column in the table.
  - Each attribute name will correspond to the name of the column in the table.
  - Each attribute type will correspond to the database column type.
    - Each model field type also may have one or more named parameters passed into it, indicating restrictions or other requirements for the field.

If you examine your Django cheat sheet, you'll see a "Model fields" section that describes each of the model fields.
These fields are created by executing the field function, followed by the optional named parameters.

- Django (by default) will automatically add an "id" column to each table (unless you define a primary key for the table), as the primary key.

Understanding Model Fields & Options

- Django model fields fall into a few categories:
  - Model fields that specify a data type
  - Model fields that specify a relational model.
    - These (generally) are model fields that specify a field in this model references a field in another model.
    - These include "ForeignKey", "ManyToManyField", "OneToOneField", and "GenericForeignKey"
      - The GenericForeignKey is used to implement certain types of relationships that tie fields in one app model to another model (and do a range of other non-standard model things).
- Each model field is also customizable to some degree using a set of model field options.
  - There are several options that are common to all model fields, and others that are specific to particular field types, the common options are:

Option Name	Description
`null`	Defaults to False, when set to True, indicates that the field may not be null.
`blank`	If True, the field is allowed to be blank. Default is False.
`db_column`	The name of the database column to use for this field. If this isn't given, Django will use the field's name.
`db_index`	If True, this will be an indexed field. Default is False

Option Name	Description
`primary_key`	If True, this field is the primary key for the model.
`unique`	If True, this field must be unique throughout the table.

Table Naming Conventions

- When a model is converted to a table, Django will always use a specific naming convention (unless you've overridden that).
  - Django will always use 'app_*classname*' for the name of a table, where *classname* is the name of the class you specified.
  - Case is always folded to lower case, so table names are in all lower case.
- When using Relational model fields (for example, ManyToManyField), Django will append the field name to the end of the table to create a new table that contains the relationship specified in the model.

One Copley Parkway, Suite 210 Morrisville, NC 27560 Phone: 919.463.0999 Fax: 866-229-3386
www.opentechnologygroup.com

Creating A Django Model

- Suppose you were creating a web site to contain a registration system for events.
  - In such a case, you would probably need a few tables:
    - A table to contain events, with event names, start dates, end dates, start times, end times, prices, currency, and an event description.
    - A table to contain registered people, with an email address, name, phone
    - A table to contain event registrations, which links the registered people with the events.
      - This table will be implemented using a many-many relationship, so the model will create it on its own.
  - Sample code to implement this model might be:

```
from django.db import models

class Events(models.Model):
    name = models.CharField(max_length=128,
                            help_text='Name of Event')
    start_date = models.DateField(help_text='Start date of Event')
    end_date = models.DateField(help_text='End date of Event')
    start_time = models.TimeField(help_text='Start time of Event')
    end_time = models.TimeField(help_text='End Time of Event')
    description = models.TextField(help_text='Description of Event')

class People(models.Model):
```

```
    fname = models.CharField(max_length=64,
                             help_text='First Name')
    lname = models.CharField(max_length=64,
                             help_text='Last Name')
    email = models.EmailField(max_length=128,
                              help_text='Email Address')
    phone = models.CharField(max_length=16,
                             help_text='Phone Number')
    enrollments= models.ManyToManyField(Events)
```

- Once converted to a database structure in Django, this is the SQL code that would be produced:

```
BEGIN;
CREATE TABLE "demoapp_events" (
    "id" integer NOT NULL PRIMARY KEY,
    "name" varchar(128) NOT NULL,
    "start_date" date NOT NULL,
    "end_date" date NOT NULL,
    "start_time" time NOT NULL,
    "end_time" time NOT NULL,
    "description" text NOT NULL
)
;
CREATE TABLE "demoapp_people" (
    "id" integer NOT NULL PRIMARY KEY,
```

```
    "fname" varchar(64) NOT NULL,
    "lname" varchar(64) NOT NULL,
    "email" varchar(128) NOT NULL,
    "phone" varchar(16) NOT NULL
)
;
CREATE TABLE "demoapp_people_enrollments" (
    "id" integer NOT NULL PRIMARY KEY,
    "people_id" integer NOT NULL REFERENCES "demoapp_people" ("id"),
    "events_id" integer NOT NULL REFERENCES "demoapp_events" ("id"),
    UNIQUE ("people_id", "events_id")
)
;
COMMIT;
```

- Notice how Django automatically creates the additional table to implement the many to many relationship.

Validating the Model

- You can validate your models to ensure they use the correct syntax and packages using the 'validate' command with manage.py.
  - This command will validate all models in the project, not just a single app. As a result, it's generally advisable to add an app and validate it, as opposed to adding and validating a whole bunch of apps at once.
  - Correct any errors and keep validating until the command returns 0 errors.

```
python manage.py validate
```

Issue this command in your project folder to validate all apps that are part of the project.

One Copley Parkway, Suite 210 Morrisville, NC 27560 Phone: 919.463.0999 Fax: 866-229-3386
www.opentechnologygroup.com

Generating & Running the SQL

- Once you've created the app, you'll want to review the SQL that your model will generate.
- The command to create the SQL code for review is 'sqlall'. Once again, use 'mange.py sqlall' followed by the app name to generate the SQL code for review.

```
python manage.py sqlall demoapp
```

Create the SQL code for the app 'demoapp' for review.

- When you are ready to create the tables in the database itself, you can issue the command: `python manage.py syncdb`
  - This command will connect to the database and create tables for all apps that are connected to this project.
  - If a table already exists, Django will verify that it contains at least the columns defined with the app.
    - If the table contains extra columns (or extra tables exist in the database), Django won't complain.
    - If the table lacks the required columns or types differ, Django will complain.

Adding Data to the Model

- Once you've created the model, adding data to it is fairly easy:
  - First, create a new object for the model you wish to add data to.
    - The constructor should be passed all the required field data in name/value pairs.
  - Once you've created the object, you can save it to the database using the .save() method.
- This technique works for all models, adding data is simply a matter of creating an object for the model and saving it.

```
from demoapp.models import Events
from datetime import date

event = Events(name='Introduction to Python',
               start_date=date(2007,01,01),
               end_date=date(2007,01,04),
               start_time='09:00:00',
               end_time='17:00:00',
               description='This is a Python training event')
event.save()
```

An example of adding some data to the existing data model. This could be executed by a view function, or it could be executed from the `'python manage.py shell'` *environment.*

Simple Data Retrieval Using a Model

- Now that we know how to put data into the model and update it, let's talk a little bit about retrieving data from a model.
  - There are a few different methods that we may use to retrieve data from a simple model such as the one we already created:
    - To retrieve all rows from a particular table, you can use the all() method of the Manager.
    - The all() method (like most methods that return data) will return a QuerySet.
      - If iterated over, the QuerySet will return instances of the model object, with each instance containing the data from that record.
      - Each object may be modified and saved, in order to update the database.

Every model in Django has an associated *Manager* (the Manager is referenced by the .object attribute of the model.). A Manager is an interface through which query operations are provided to Django models. The manager itself provides us with a set of methods to interface with the database.

```
data = Events.objects.all()
for e in data:
    print e.name, 'starts on', e.start_date
```

Retrieving all of the rows from the Events table using the Django Model.

Understanding QuerySets

- A QuerySet in Django represents a collection of objects from the database.
  - Whenever you use the manager for a model to retrieve data, you get back a QuerySet
  - So far, we've only learned how to use a manager to get all the rows back from a database, but we'll soon learn how we can pass the manager criteria to limit the size of the QuerySet returned)
- Each QuerySet may have zero or more *filters* associated with it (a filter limits the number of elements in the collection, given a set of filtering criteria
  - Filters applied to a QuerySet are basically converted by Django into 'WHERE', 'LIMIT' and other SQL clauses.
- QuerySets are evaluated when any of the following occur with/on them: iteration, slicing, Pickling/Caching, repr(), len(), or list().

You might think that a query is executed with the statement:

```
data = Events.objects.all()
```

But actually, that's not the case. A QuerySet is generated from the statement, but the query isn't actually executed until you try to retrieve results from the QuerySet (usually, by iterating over it).

This feature allows you to create a base query set, and then perform a wide range of operations to it (such as applying filters), all before the query executes on the server.

This means that Django tries to wait until the last minute to execute the query, and it tries to maximize the work done on the server itself.

Applying Filters

- The first thing to understand about filtering results, is that there are a number of operations that can be performed on QuerySets to reduce the rows that they return.
- Every time a filter (or other method/operation) is applied to a QuerySet, a new QuerySet is returned (i.e., the existing QuerySet isn't modified, unless you overwrite it).
- The following methods are the most common ones applied to a QuerySet:

Method	Description
`.filter(`*`Field lookup`*`)`	The filter() method returns a new QuerySet that has a filter applied to it. This is analogous to a WHERE clause for a query. The argument passed into this method is a Field lookup (which we'll cover next).
`.get(`*`Field lookup`*`)`	Returns the object matching the given lookup parameters. The argument passed into this method is a Field lookup (which we'll cover next).
`.exclude(`*`Field lookup`*`)`	Returns a new QuerySet containing objects that do *not* match the given lookup parameters. This is the opposite of .filter(), and also accepts as an input argument a Field lookup.

Specifying Field Lookups/Filters

- Field Lookups are the method that we use to specify the "WHERE" clause of the query to be executed.
  - Field lookups are passed as arguments into the .filter(), .get() or .exclude() method of a QuerySet, and return a new QuerySet that contains the earlier QuerySet with the appropriate filter applied.
- Field Lookups begin with the name of the field that the lookup pertains to (the name of the field, of course, is specified in the model.
  - In our examples, we'll use the Models shown here.
- The attribute of the model is then followed by two underscores "__" and then a *lookup type.*

```
class Events(models.Model):
    name = models.CharField(max_length=128)
    start_date = models.DateField()
    end_date = models.DateField()
    start_time = models.TimeField()
    end_time = models.TimeField()
    description = models.TextField()

class Schedule(models.Model):
    start_date = models.DateField()
    end_date = models.DateField()
    name = models.CharField(max_length=128)

class People(models.Model):
    fname = models.CharField(max_length=64)
    lname = models.CharField(max_length=64)
    email = models.EmailField(max_length=128)
    phone = models.CharField(max_length=16)
    enrollments= models.ManyToManyField(Events)
```

The model that we'll base our examples upon.

- The lookup type indicates the type of match that needs to be performed for this particular field.
- Lastly, an '=' is specified followed by the value that the match applies to.

Lookup Types

- The table below describes the Lookup Types that are most commonly used, along with a Description of each (examples are shown in shaded text):

```
e=Events.objects.all()
p=People.objects.all()
s=Schedule.objects.all()
```

The examples shown in the table are based on the variables e and p shown here.

Lookup Type	Description

`__exact`
`e=e.filter(start_date__exact='2007-01-01')`
`e=e.filter(start_date='2007-01-01')`
	Exact match. The value provided for comparison should exactly match the field specified. In the value provided is `None`, then the field should contain null. Note that when you provide no type, __exact is implied.

`__iexact` `p=p.filter(fname__iexact='joe')`	Case insensitive exact match. The same as exact, but in a case-insensitive fashion.
`__contains` `e=e.filter(name__contains='Python')`	Case-sensitive contains. This is analogous to an SQL : WHERE name LIKE '%Python%'

Lookup Type	Description
__icontains `e=e.filter(name__contains='python')`	Case-insensitive contains. This is analogous to an SQL: WHERE name ILIKE '%python%'

__in
`e=e.filter(start_date__in=['2007-01-01','2008-01-01'])`
`# This example uses a query string to`
`# generate a subquery expression for in`
`q=s.filter(name__contains='Python').values('start_date').query`
`e=e.filter(start_date__in=q)`
	Selects items where the column matches one of the values in the list. This is like in IN statement in SQL.
	It is also important to note that we can pass in another QuerySet in certain cases to have Django build a subquery.

__gt `e=e.filter(start_date__gt=datetime.date())`	Greater than. Filter should contain items that are greater than the value specified.
_lt `e=e.filter(start_date__lt=datetime.date())`	Less than. Filter should contain items that are less than the value specified.
__lte __gte	Less than or equal to. Greater than or equal to.

Lookup Type	Description
`__startswith` `__istartswith`	Starts with (or its case insensitive version). This is a LIKE clause with the anchor at the start of the string.
`__endswith` `__iendswith`	Ends with (or its case insensitive version). This is a LIKE clause with the anchor at the end of the string.
`__range` `e=e.filter(start_date__range=('2007-01-01','2007-12-31'))`	Values are within the range specified by the tuple. This is inclusive of both high and low values, and translates to a BETWEEN clause.
`__year` `e=e.filter(start_date__year=2005)`	For date and datetime fields, this is an exact year match. Using this (in some databases) might omit the use of indexes.
`__month` `__day`	For date and datetime fields, this is a month or day match.
`__isnull`	Filters records where the field specified

Lookup Type	Description
	contains a null.
`__search`	Utilizes a boolean full text search mechanism to locate matching records. This may not be supported in all database engines.
`__regex` `__iregex`	Perform a regular expression or case insensitive regular expression match.

Slicing QuerySets

- In order to apply/provide LIMIT functionality, Django querysets may be sliced.
  - Slicing a queryset produces a new QuerySet that adds a LIMIT clause to the query being processed.

```
e=Events.objects.all()
e=e.exclude(description__isnull=True)
e=e.filter(start_date__gte=date.today())[10:20]
```

Using a slice of a QuerySet to return records 10-20

```
e=Events.objects.all()
e=e.exclude(description__isnull=True)
e=e.filter(start_date__gte=date.today())[:20]
```

Using a slice of a QuerySet to return the first 20 records

```
e=Events.objects.all()
e=e.exclude(description__isnull=True)
e=e.filter(start_date__gte=date.today())[20:]
```

Using a slice of a QuerySet to skip the first 20 records.

Specifying Ordering in QuerySets

- You can provide ordering in values in a QuerySet using the order_by method of the QuerySet
  - Here, you will need to specify a comma separated list of columns to order by.
- You can pass in a "?" to the order_by method to have Django order the results randomly (this could be expensive and time consuming - depending on the number of records in the table)

```
e=Events.objects.all()
e=e.exclude(description__isnull=True)
e=e.order_by('start_date','start_time')
```

Using the order_by method to specify that results should be ordered by start_date and then start_time.

Common QuerySet Methods

- The QuerySet has numerous other methods, some of the commonly used ones are listed in the table below (all return a new QuerySet):

Method	Description
.reverse()	Reverse the order in which query set results are returned. This can be combined with .order_by() to reverse order results.
.distinct()	Uses SELECT DISTINCT to eliminate duplicate values in the query result. The argument is a comma separated list of fields that would need to be distinct as a group.
.none()	Returns an empty QuerySet.
.select_related()	Returns a QuerySet that automatically expands foreign key relationships (automatically selects related data in child tables). This method will follow FK relationships as far as possible, so it might end up returning lots more data than you expect! It accepts an optional 'depth=' argument that specifies the maximum depth. You can optionally also pass in a comma separated list of FK's that it should expand.
.extra()	Allows you to specify a manually generated SQL query portion. Review documentation for more details.

.count()	Returns the number of records that are returned as a result of QuerySet execution.

- More QuerySet methods are available in the Django documentation online.

One Copley Parkway, Suite 210 Morrisville, NC 27560 Phone: 919.463.0999 Fax: 866-229-3386
www.opentechnologygroup.com

Deleting Records

- The .delete() method of your Model object is used to delete records from the data model.
- The .delete() method may also be executed on a QuerySet to remove all records that the QuerySet references.

```
data = Events.objects.all()
for e in data:
    e.delete()
```

Removing all the records from the Event model.

```
data = Events.objects.all().delete()
```

Removing all the records from the Event model.

```
data = Events.objects.all()
data = data.filter(start_date__lte=date.today())
data.delete()
```

Removing all past events from the Event model.

Managing Related Records

- Although it is possible for us to create records in a child table using standard query methods, Django provides us with some simpler methods of doing so.
  - These methods will vary depending on the type of relationship.
- In order to create related records, you need only to pass into a child record a reference to the parent (or related) record.

Even though when we define a model we only define a relationship in one of the two models that are related, Django will automatically map the reverse relationship as well. This makes it easier to locate related records without having to duplicate model definition information

```
class People(models.Model):
    fname = models.CharField(max_length=64)
    lname = models.CharField(max_length=64)
    email = models.EmailField(max_length=128)
    phone = models.CharField(max_length=16)
    enrollments= models.ManyToManyField(Events)

class Address(models.Model):
    addr1 = models.CharField(max_length=64)
    addr2 = models.CharField(max_length=64)
    city = models.CharField(max_length=64)
    state = models.CharField(max_length=2)
    zip = models.CharField(max_length=10)
    create_date = models.DateTimeField(auto_now=True)
    peeps = models.ForeignKey(People)
```

Example model with a PK/FK relationship. Notice that the Address table uses the name 'peeps' for the relation (people would have been better). This results in a special Address.peeps attribute.

- Given the models shown here, we could create related records using code such as those shown in the examples here.
  - First, if we examine the Address model definition, you can see a special attribute has been created called 'people' for the FK relationship.
    - This automatically creates a special attribute called 'Address.people'.
    - The attribute contains a reference to a Manager that can be used to create related records in the People object.

```
people = People(fname='Ralph',
                lname='Edwards',
                email='rewdwards@example.com',
                phone='(111)123-4567');
people.save()
a = people.address_set.create(addr1='1 Nowhere Lane',
                              city='Anytown',
                              state='NM',
                              zip='12345')
```

Using the relationship to create a related record in another table. Since the FK is defined in the Address model, the People model has a special property called 'address_set' (the _set is there because the FK is defined in another model) that may be used to retrieve a manager for the other mdoel.

```
e = people.enrollments.create(name='Advanced Python',
                              start_date=date(2007,01,01),
                              end_date=date(2007,01,04),
                              start_time='09:00:00',
                              end_time='17:00:00',
                              description='This an event'
                              )
```

Create a new event tied to a person record using the ManyToMany relationship defined in the People Model

- Since a FK relationship exists in Address, Django automatically creates a special attribute in the parent model (People) called 'address_set'.
  - The name is always the child object name in lower case (in this case, the child is 'address') followed by '_set'.
  - This attribute returns a Manager that may be used to create records in the child table.

```
p = e.people_set.create(fname='Joe',
                        lname='Smith',
                        email='jsmith@example.com',
                        phone='(000)123-4567')
```

Create a new record in the People model that is associated with the Event table. people_set is used here since the relationship is defined in the People model and not the Event model.

Retrieving Related Records

- When retrieving data from the database, we often find that we wish to exploit a FK relationship to retrieve information from a related table.
  - There are several ways that we can do this.
- Whenever working with a model that has a child (is the Parent in the FK relationship), a special attribute is defined for a QuerySet object that returns a Manager for the related records.
  - This manager can then be used to retrieve records associated with that child:
    - The .select_related() method will return a QuerySet that contains all of the related records. It returns a result set that exploits all relationships and returns all the associated results in a single result set.
      - It essentially populates all of the child/parent attributes at query execution time, using a single query to retrieve all the related results.

```
p= People.objects.get(pk=2)
e = p.enrollments.filter(start_date__gte=date.today())
for enrollment in e:
  print enrollment.id

e = Events.objects.get(pk=1)
persons = e.people_set.all()
for p in persons:
    print p.fname, p.lname, "is enrolled in", e.name
```

Two examples to leverage relationships between records. In the first example, we only return future events for that person using a filter.

- The .all() and .filter() methods work the same as we discussed earlier, and return a QuerySet that contains related records.
  - Of course, here you must specify the relation that you wish to retrieve related data for.

In general, using the .select_related() method (and optionally specifying a depth) will be faster and more efficient than retrieving all the related records individually. This is because Django uses a single query to return results in .select_related(), as opposed to lots of smaller individual queries by generating and executing a QuerySet.

- On other method that we can use is to use filter expressions that "follow" relationships in loopkups.
  - In such cases, we can specify the field name of related fields across models, using the __ seperator.
  - Here, we would use *model__field__criteria* to perform a match.
    - Before we were using *field__criteria,* the addition of a model name allows us to leverage relationships between models.

```
from demoapp.models import Events

qs=Events.objects.filter(people_set__fname__exact='Joe')
```

Retrieving only those events who have someone with a first name of Joe registered in them. The use of people_set allows this query to span to the people model, which has a relationship with this one.

Using Q Objects

- Q objects are objects that are used to append more complex criteria to queries.
- A Q object is an object that encapsulates a number of different keyword arguments.
  - These arguments use field lookups (_lte, _startswith, _ge, _contains, etc.).
- To get complex queries, we can combine two Q objects together with a "|" or an "&" .
  - Joining 2 Q objects this ways results in a new Q object.
  - The resulting Q object essentially joins the two objects together like a OR query (for a "|") or an AND query (for an "&")
- Q objects, once created, may be passed into QuerySet filter methods (which would then generate new QuerySet's).

```
from demoapp.models import Events
from django.db.models import Q

qs=Events.objects.all()
q = Q(id__lte=3) | Q(id__gte=14)
qs=qs.filter(q)
```

Using Q objects to eliminate a range of records from a query.

Using the Django Admin Interface

One Copley Parkway, Suite 210 Morrisville, NC 27560 Phone: 919.463.0999 Fax: 866-229-3386
www.opentechnologygroup.com

Enabling the Admin Interface

- The Administrative interface is designed to provide an easy way to manage and manipulate data stored in the Model.
  - For a particular Model to appear in the admin interface, you must first create an admin.py in your application.
  - The admin.py file contains representations of your models in the admin interface (ModelAdmin class).
  - Once you create the representation, you'll need to register it with the Admin site.
- Once you've updated your models, perform a syncdb to ensure that the Model is in sync with the database schema.
- To enable the admin interface, you'll need to add a URL for it to urls.py:

```
from django.contrib import admin
admin.autodiscover()
urlpatterns += patterns ('',(r'^admin/', include(admin.site.urls)))
```

- You'll also have to add 'django.contrib.admin' to INSTALLED_APPS in settings.py

```
from django.contrib import admin
from demo.demoapp.models import *

class EventsAdmin(admin.ModelAdmin):
    list_display=('name','start_date')

admin.site.register(Events, EventsAdmin)
class PeopleAdmin(admin.ModelAdmin):
    list_display=('lname','fname')
admin.site.register(People, PeopleAdmin)
```

admin.py: This file contains a representation of the People and Events models in the admin interface. Once this has been added, these objects will appear in that interface.

- Finally, execute syncdb to create any admin components required.
- Lastly, the ModelAdmin has a number of attributes that may be set to control how the admin site appears.
  - For example, setting the 'list_display' property (a tuple) allows us to list the columns to display on the change list page of the admin.

Access Control with Sessions and Users

Cookies & Django

- Every request object in Django has a special dictionary defined called COOKIES.
  - Like GET and POST, COOKIES contains key/value pairs that refer to COOKIE data.
- In order to set cookies, the HTTP Response object has a .set_cookie() method that needs to be called.
  - The method requires the first two arguments to be:
    - The name of the cookie to be set (first argument).
    - The value to be stored in the cookie (second argument).
  - In addition, the following named arguments may also be passed in.

Parameter	Description
`max_age`	The age (in seconds) the cookie should last. The default is for the cookie to stay until the browser closes.
`expires`	The date/time when the cookie should expire. This needs to be in the format: 'Wdy, DD-Mth-YY HH:MM:SS GMT' This parameter overrides max_age.
`path`	The path prefix the cookie is valid for (default is /)
`domain`	The domain this cookie is valid for. A leading period indicates that the cookie is valid for all sub-domains of the domain specified.

Parameter	Description
`secure`	If set to True, this is a cookie that may only be passed via HTTPS.

- Cookies are fundamentally flawed in that the data stored in them in susceptible to being read and modified by the browser – that's why Django provides something better in the form of sessions!

The Django Session Framework

- The session framework is a set of methods and objects that allow you to store an identifier on the browser via a cookie, and then arbitrary data in a server-side data store.
  - Since the data is stored on the server, you can store lots of it (it doesn't incur network overhead), and its secure.
- In order to enable the session framework, you must first edit your settings.py. Add to your `MIDDLEWARE_CLASSES` the '`django.contrib.sessions.middleware.SessionMiddleWare`' class.
- Next, ensure that '`django.contrib.sessions`' is one of your `INSTALLED_APPS`.
  - Note: These are already set by default when you create a new project using the startproject command.

```
MIDDLEWARE_CLASSES = (
    'django.contrib.sessions.middleware.SessionMiddleware',
)
INSTALLED_APPS = (
    'django.contrib.sessions',
)
```

These lines are required in settings.py to enable sessions.

Sessions in Views

- Once you have the session components enabled, you can use the .session property of the request object to save and retrieve values for the session.
  - You can treat the .session property just like a dictionary.
- Try not to use session keys with a leading '_', they are reserved for Django's internal use.

```
def Session(request):
    if request.session['counter'] >= 10:
        del request.session['counter']
    if request.session.get('counter',False) is False:
        request.session['counter']=0
    request.session['counter']+=1
    return render_to_response('session.html',
                              { 'pagetitle': "Session Counter",
                                'visits': request.session['counter'] })
```

Using a session to store a basic page visit counter. The use of del here is to illustrate how to remove session values.

The Django auth Application

- One of the most useful Django contributed apps is the auth app.
  - This is a set of functions/views that may be used to manage various user-related tasks, such as password reset and recovery and access control.

```
import django.contrib.auth.views

urlpatterns = patterns('django.contrib.auth.views',
    (r'^accounts/login/$','login' ),
    (r'^password_reset/$', 'password_reset'),
    (r'^password_reset/done/$', 'password_reset_done'),
    (r'^reset/(?P<uidb36>[0-9A-Za-z]+)-(?P<token>.+)/$', 'password_reset_confirm'),
    (r'^reset/done/$', 'password_reset_complete'),
    (r'^password_change/$', 'password_change'),
    (r'^password_change/done/$', 'password_change_done'),
)
```

Using the django.contrib.auth application to ease the creation of password/account management functions within an application.

Using Authentication in Views

- Once you've got the auth components installed, you'll find that your HttpRequest object has a new attribute: the .user attribute.
  - The .user attribute is created by the auth system, and is an object that represents the currently logged-in user.
    - If no user is logged in, it will contain an AnonymousUser object instead.
- The user object contains lots of useful members and methods, that are easily accessible. Some are shown here:

Member/Method	Description
`.is_authenticated()`	Returns true if the user is authenticated. This is the most important (and basic) check to perform.
`.is_anonymous()`	Returns true for the AnonymousUser object, and false otherwise.
`.get_full_name()`	Returns the full name of the user (first_name last_name)

Login and Logout

- Django provides two built in view functions to handle log in and log out (these are provided in django.contrib.auth.views).
  - The login view looks for a default template called 'registration/login.html'
  - The logout view looks for a default template called 'registration/logout.html'. However, if you pass in a 'next_page' parameter, it will redirect the user to the specified page on logout, instead of using the template.

```
from django.conf.urls.defaults import *
from django.contrib.auth.views import login, logout

urlpatterns = patterns('',
    (r'login', login, {'template_name': 'login.html'}),
    (r'logout', logout, {'template_name': 'logout.html'}),
```

An example of using the built-in Django views for handling login and logout. In this case, Django will look for a login.html and logout.html template for the view.

The Login Required Decorator

- You can use the login_required decorator to verify that a user is logged in.
  - If the user is not logged in, the user will be redirected to '/accounts/login/', and the current URL will be passed into the URL as the query string next.
    - That may then be used to redirect the user (after they authenticate) back to the page they tried to view prior to logging in.

```
from demoapp.models import Events, People, Address
from django.shortcuts import render_to_response
from django.forms.models import ModelForm
from django.forms.util import ErrorList
from django.contrib.auth.decorators import login_required
import forms

@login_required
def addEvent(request):
    if (request.method == 'POST'):
        form = forms.EventAddForm(request.POST)
    else:
        form = forms.EventAddForm()
    if form.is_valid():
        name = form.cleaned_data['name']
        obj=form.save()  # Save the data into the model
        return render_to_response('EventAddForm.html',
                                  {'message': """Your event "%s" has been
added with an id of %d""" % (name, obj.id),
                                   'form': forms.EventAddForm()})
    else:
        return render_to_response('EventAddForm.html', {'form': form })
```

Using the login_required decorator to verify the user is logged in.

Django Contributed Apps

- Django provides some contributed apps for a wide range of different tasks – one of the most useful is the auth app.
  - This is a set of functions/views that may be used to manage various user-related tasks, such as password reset and recovery.

```
import django.contrib.auth.views

urlpatterns += patterns('django.contrib.auth.views',
    (r'^accounts/login/$','login' ),
    (r'^password_reset/$', 'password_reset'),
    (r'^password_reset/done/$', 'password_reset_done'),
    (r'^reset/(?P<uidb36>[0-9A-Za-z]+)-(?P<token>.+)/$', 'password_reset_confirm'),
    (r'^reset/done/$', 'password_reset_complete'),
    (r'^password_change/$', 'password_change'),
    (r'^password_change/done/$', 'password_change_done'),
)
```

Using the django.contrib.auth application to ease the creation of password/account management functions within an application.

Generic Views

Simple Generic Views

- In the previous examples, we wrote view functions to handle the output of data into a view.
- Generic views extend this concept by providing us a simple mechanism to generate template content without having to go through all the work of creating a view function.
- The django.views.generic.simple module contains some simple views to handle some common cases.
  - The two cases it handles are issuing redirects, and rendering a template when no view logic is needed.
- The first of these is django.view.generic.simple.direct_to_template
  - It renders a template (passed in via the template argument) and passes it a {{ params }} template variable.
    - The {{ params }} variable is a dictionary of parameters captured in the URL.
  - Additionally, it may be passed a dictionary as a named parameter called 'extra_context'.

```
from django.views.generic.simple import direct_to_template

urlpatterns = patterns('',
    (r'info/(?P<title>[^/]+)', direct_to_template,
        {'template':'generic.html',
         'extra_context':
          {'message': 'Generic Message' } } ),
```

Using direct_to_template to generate a generic view.

- This is a set of dictionary values to add to the template context.
- Lastly, a 'mimetype' parameter may be used to sent the MIME type for the resulting document.

- You can even use direct_to_template to build views that are more dynamic, but still using a generic view.

```
from django.contrib.auth.decorators import user_passes_test
from django.views.generic.simple import direct_to_template
from django.template import TemplateDoesNotExist
from django.http import Http404

def checkAuth(user):
    return user.is_authenticated() and
           user.is_active

@user_passes_test(checkAuth)
def InfoView(request, title):
    try:
        return direct_to_template(request,
               template='generic-%s.html' % title)
    except TemplateDoesNotExist:
        raise Http404()
```

Using direct_to_template to generate output using a template based on the URL (this uses a URLConf similar to the one on the previous page). If the TemplateDoesNotExist exception is raised, we'll return a 404 error message. The user_passes_test decorator ensures that only valid, logged in, users are able to see this page.