Design your URLs

A clean, elegant URL scheme is an important detail in a high-quality Web application. Django encourages beautiful URL design and doesn't put any cruft in URLs, like **.php** or **.asp**.

To design URLs for an app, you create a Python module called a URLconf. A table of contents for your app, it contains a simple mapping between URL patterns and Python callback functions. URLconfs also serve to decouple URLs from Python code.

Here's what a URLconf might look like for the **Reporter/Article** example above:

mysite/news/urls.py

```
from django.conf.urls import url

from . import views

urlpatterns = [
    url(r'^articles/([0-9]{4})/$', views.year_archive),
    url(r'^articles/([0-9]{4})/([0-9]{2})/$', views.month_archive),
    url(r'^articles/([0-9]{4})/([0-9]{2})/([0-9]+)/$', views.article_detail),
]
```

The code above maps URLs, as simple regular expressions, to the location of Python callback functions ("views"). The regular expressions use parenthesis to "capture" values from the URLs. When a user requests a page, Django runs through each pattern, in order, and stops at the first one that matches the requested URL. (If none of them matches, Django calls a special-case 404 view.) This is blazingly fast, because the regular expressions are compiled at load time.

Once one of the regexes matches, Django imports and calls the given view, which is a simple Python function. Each view gets passed a request object – which contains request metadata – and the values captured in the regex.

For example, if a user requested the URL "/articles/2005/05/39323/", Django would call the function **news.views.article_detail(request, '2005', '05', '39323')**.

Write your views

Each view is responsible for doing one of two things: Returning an **HttpResponse** object containing the content for the requested page, or raising an exception such as **Http404**. The rest is up to you.

Generally, a view retrieves data according to the parameters, loads a template and renders the template with the retrieved data. Here's an example view for **year_archive** from above:

mysite/news/views.py

```
from django.shortcuts import render
```

Database setup

Now, open up **mysite/settings.py**. It's a normal Python module with module-level variables representing Django settings.

By default, the configuration uses SQLite. If you're new to databases, or you're just interested in trying Django, this is the easiest choice. SQLite is included in Python, so you won't need to install anything else to support your database. When starting your first real project, however, you may want to use a more robust database like PostgreSQL, to avoid database-switching headaches down the road.

If you wish to use another database, install the appropriate database bindings, and change the following keys in the **DATABASES** **'default'** item to match your database connection settings:

- **ENGINE** – Either **'django.db.backends.sqlite3'**, **'django.db.backends.postgresql_psycopg2'**, **'django.db.backends.mysql'**, or **'django.db.backends.oracle'**. Other backends are also available.
- **NAME** – The name of your database. If you're using SQLite, the database will be a file on your computer; in that case, **NAME** should be the full absolute path, including filename, of that file. The default value, **os.path.join(BASE_DIR, 'db.sqlite3')**, will store the file in your project directory.

If you are not using SQLite as your database, additional settings such as **USER**, **PASSWORD**, **HOST** must be added. For more details, see the reference documentation for **DATABASES**.

Note

If you're using PostgreSQL or MySQL, make sure you've created a database by this point. Do that with "**CREATE DATABASE database_name;**" within your database's interactive prompt.

If you're using SQLite, you don't need to create anything beforehand - the database file will be created automatically when it is needed.

While you're editing **mysite/settings.py**, set **TIME_ZONE** to your time zone.

Also, note the **INSTALLED_APPS** setting at the top of the file. That holds the names of all Django applications that are activated in this Django instance. Apps can be used in multiple projects, and you can package and distribute them for use by others in their projects.

By default, **INSTALLED_APPS** contains the following apps, all of which come with Django:

- **django.contrib.admin** – The admin site. You'll use it in part 2 of this tutorial.
- **django.contrib.auth** – An authentication system.
- **django.contrib.contenttypes** – A framework for content types.
- **django.contrib.sessions** – A session framework.
- **django.contrib.messages** – A messaging framework.
- **django.contrib.staticfiles** – A framework for managing static files.

These applications are included by default as a convenience for the common case.

Some of these applications make use of at least one database table, though, so we need to create the tables in the database before we can

Write views that actually do something

Each view is responsible for doing one of two things: returning an **HttpResponse** object containing the content for the requested page, or raising an exception such as **Http404**. The rest is up to you.

Your view can read records from a database, or not. It can use a template system such as Django's – or a third-party Python template system – or not. It can generate a PDF file, output XML, create a ZIP file on the fly, anything you want, using whatever Python libraries you want.

All Django wants is that **HttpResponse**. Or an exception.

Because it's convenient, let's use Django's own database API, which we covered in Tutorial 1. Here's one stab at a new **index()** view, which displays the latest 5 poll questions in the system, separated by commas, according to publication date:

polls/views.py

```
from django.http import HttpResponse

from .models import Question


def index(request):
    latest_question_list = Question.objects.order_by('-pub_date')[:5]
    output = ', '.join([p.question_text for p in latest_question_list])
    return HttpResponse(output)

# Leave the rest of the views (detail, results, vote) unchanged
```

There's a problem here, though: the page's design is hard-coded in the view. If you want to change the way the page looks, you'll have to edit this Python code. So let's use Django's template system to separate the design from Python by creating a template that the view can use.

First, create a directory called **templates** in your **polls** directory. Django will look for templates in there.

Your project's **TEMPLATES** setting describes how Django will load and render templates. The default settings file configures a **DjangoTemplates** backend whose **APP_DIRS** option is set to **True**. By convention **DjangoTemplates** looks for a "templates" subdirectory in each of the **INSTALLED_APPS**. This is how Django knows to find the polls templates even though we didn't modify the **DIRS** option, as we did in Tutorial 2.

Organizing templates

We *could* have all our templates together, in one big templates directory, and it would work perfectly well. However, this template belongs to the polls application, so unlike the admin template we created in the previous tutorial, we'll put this one in the application's template directory (**polls/templates**) rather than the project's (**templates**). We'll discuss in more detail in the reusable apps tutorial *why* we do this.

Why you need to create tests

So why create tests, and why now?

You may feel that you have quite enough on your plate just learning Python/Django, and having yet another thing to learn and do may seem overwhelming and perhaps unnecessary. After all, our polls application is working quite happily now; going through the trouble of creating automated tests is not going to make it work any better. If creating the polls application is the last bit of Django programming you will ever do, then true, you don't need to know how to create automated tests. But, if that's not the case, now is an excellent time to learn.

Tests will save you time

Up to a certain point, 'checking that it seems to work' will be a satisfactory test. In a more sophisticated application, you might have dozens of complex interactions between components.

A change in any of those components could have unexpected consequences on the application's behavior. Checking that it still 'seems to work' could mean running through your code's functionality with twenty different variations of your test data just to make sure you haven't broken something - not a good use of your time.

That's especially true when automated tests could do this for you in seconds. If something's gone wrong, tests will also assist in identifying the code that's causing the unexpected behavior.

Sometimes it may seem a chore to tear yourself away from your productive, creative programming work to face the unglamorous and unexciting business of writing tests, particularly when you know your code is working properly.

However, the task of writing tests is a lot more fulfilling than spending hours testing your application manually or trying to identify the cause of a newly-introduced problem.

Tests don't just identify problems, they prevent them

It's a mistake to think of tests merely as a negative aspect of development.

Without tests, the purpose or intended behavior of an application might be rather opaque. Even when it's your own code, you will sometimes find yourself poking around in it trying to find out what exactly it's doing.

Tests change that; they light up your code from the inside, and when something goes wrong, they focus light on the part that has gone wrong - *even if you hadn't even realized it had gone wrong*.

Tests make your code more attractive

You might have created a brilliant piece of software, but you will find that many other developers will simply refuse to look at it because it lacks tests; without tests, they won't trust it. Jacob Kaplan-Moss, one of Django's original developers, says "Code without tests is broken by design."

That other developers want to see tests in your software before they take it seriously is yet another reason for you to start writing tests.
