Django Overview

Mark G.

May 23, 2020

Figure 1: Jazz Guitarist and Composer Django Reinhardt[1]

How is Django pronounced?

Listen. https://www.red-bean.com/~adrian/django_pronunciation.mp3

[1]This work is from the William P. Gottlieb collection at the Library of Congress. In accordance with the wishes of William Gottlieb, the photographs in this collection entered into the public domain on February 16, 2010.

Contents

1 What is Django?

The best place to start learning about Django is the Django Project Web Site. The documentation for the project is superb. To quote the Django home page, Django is:

> The web framework for perfectionists with deadlines.

It is not your typical site building tool that has HTML interspersed with embedded, interpreted code like PHP (although there is some logic in the template system). Indeed, once your HTML base is decided on, you rarely work with HTML directly and spend most of your effort with code that fetches and updates database table values. This code will set a bunch of variables into a context, which is then displayed back to the user through the template rendering system.

Sounds complicated.

2 Why did I choose Django?

I had these requirements for developing web sites many years ago, which were met by Django and its underlying Python programming language.

- Bilingual support mechanisms.
- Unicode character set support with UTF-8 encoding.
- Database backed Object Reference Models (ORM).
- Efficient HTML generation and minimal HTML code duplication.
- Automated testing system support.
- Options for customization.

Other bonus considerations:

- Authentication and Authorization.
- Geographic Database Support.

I can't cover those bonus items, since time is constrained. Suffice it to say that django has great support for them.

3 Common Things I Won't Cover Today

Django is a large framework. There are quite a few things that I am going to skip, because there are many online tutorials that cover the material very well. They are:

- Forms - the example project I've developed doesn't use any forms at this time. Lots of good information about them in the official django tutorial.
- The built-in Django Admin - it's very handy and maybe I'll show it off a bit, but I won't cover much about it.
- Setting up a PostgreSQL database. This is supposed to be about django, not a tutorial on setting up a database.
- Wiring the django project into an actual web server. The development environment for django includes a simple server for looking at a django site. This is what we'll use.
- Collecting and processing of static media files for web server deployment.

The reason I am skipping these items is because almost all tutorials handle them well. I'd like to cover material that goes slightly beyond the basics, those aspects of django that crop up only later in a development cycle, but turn out to be exceedingly important in the long run.

4 Important Principles

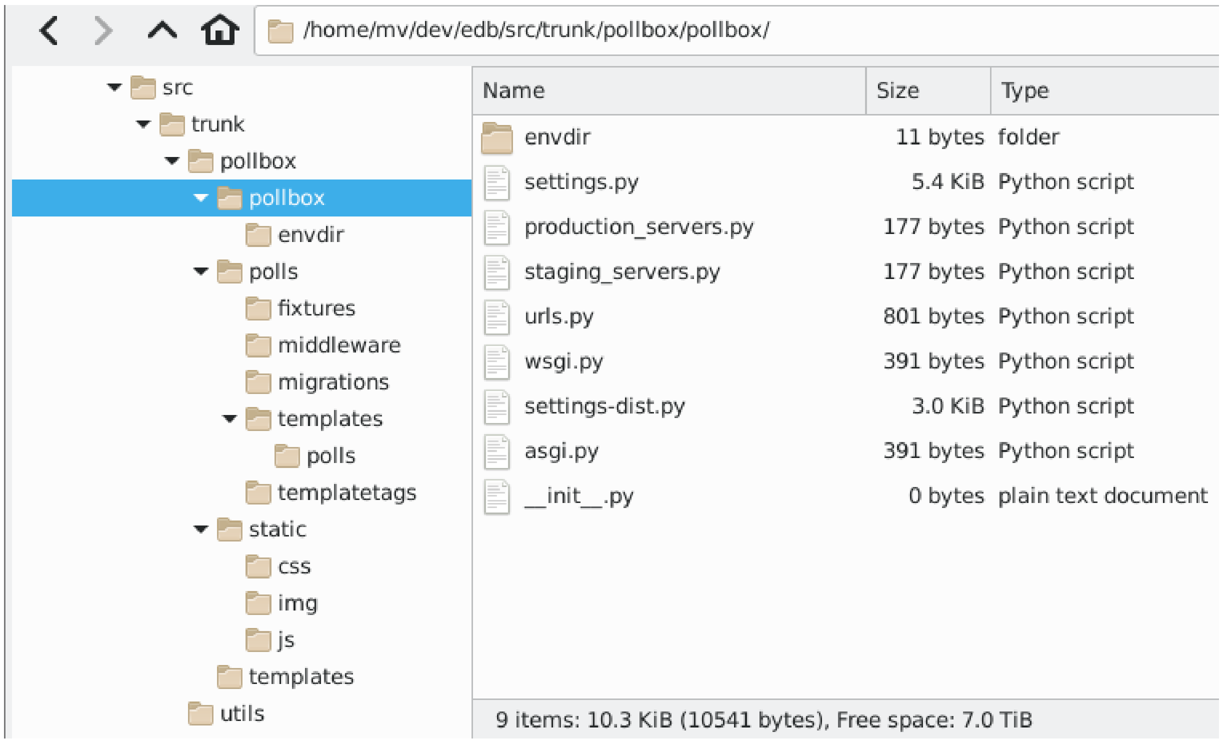
Software development is complicated, annoying and can be hard to do over an extended period of time. It can also be joyful, so don't despair.

In order to minimize the annoyance, it is critical that a programmer follows a set of principles in their day-to-day work. Any software that assists in the following of these principles helps in using best practices based on the principles.

1. Don't Repeat Yourself (DRY).
2. The Principle of Least Astonishment (PLA).
3. Just enough privilege to do your work (MINPRIV).
4. Setup for Reducing Future Maintenance (RFM).

5 My Development Checklist

- Setup a python virtual environment.
- Set all editors to Unicode with UTF-8 encoding.
- Start with multi-lingual code snippets from the beginning.
- Mock up your base template.
- Create the correct sets of folders for static files and media.
- Think about your site's URL layout.
- Sketch workflow pictures of your users' required actions and how they interact with the database models.
- Start a list of required session variables.



```
/home/mv/dev/edb/src/trunk/pollbox/pollbox/
```

```
src
  trunk
    pollbox
      pollbox
        envdir
      polls
        fixtures
        middleware
        migrations
        templates
          polls
        templatetags
      static
        css
        img
        js
      templates
    utils
```

Name	Size	Type
envdir	11 bytes	folder
settings.py	5.4 KiB	Python script
production_servers.py	177 bytes	Python script
staging_servers.py	177 bytes	Python script
urls.py	801 bytes	Python script
wsgi.py	391 bytes	Python script
settings-dist.py	3.0 KiB	Python script
asgi.py	391 bytes	Python script
__init__.py	0 bytes	plain text document

9 items: 10.3 KiB (10541 bytes), Free space: 7.0 TiB

Figure 2: **A directory tree of a django project named** `pollbox`

5.1 Virtual Environment - Self Containment

A website development project, that is based on a programming language, makes obvious the need for a containerized environment. There are a number of ways to do this, but the simplest is to use the Python virtual environment tool.

A python virtual environment insulates your normal operating system versions of python from the one you want to use for your project. I'm sure the system administrator doesn't want their python modules and libraries overwritten or updated by your website's python modules' needs. So keep them separate. There's a principle here, but it's not in our list, however it is based on separation of parts and is related to encapsulation.

See the official python venv documentation for more information.

We can skip over the details, they are here for reference.

Start by installing the python `venv` module:

```
mv@think:~/dev/edb$ sudo apt-get install python3-venv
```

Creating the environment is done using the following command. Note that the version of python you specify in the command will determine the python version within the virtual environment. We will get a python 3.7 environment with the command below, and the environment itself will live in the `.venv/` directory.

```
mv@think:~/dev/edb$ python3.7 -m venv .venv
```

Using a virtual environment requires activating it before carrying out any programming or django actions. We also need to activate it in order to install Django itself and generate the `requirements.txt` file.

There are three scripts to help us do this, one for each shell type:

```
mv@think:~/dev/edb $ ls .venv/bin/activate*
.venv/bin/activate  .venv/bin/activate.csh  .venv/bin/activate.fish
```

On this system, we'll use the `bash` based `.venv/bin/activate` script via the `source` command:

```
mv@think:~/dev/edb $ source .venv/bin/activate
(.venv) mv@think:~/dev/edb $
```

Notice how our prompt is now changed to show our active python environment. If you don't see the environment name in your prompt, you will likely get squirrely results trying to run django or database python programs.

A number of python programs/modules were installed into this virtual environment using the `pip` command. They are listed below. Important ones are `Django`, `envdir`, and `psycopg2`.

```
(.venv) mv@think:~/dev/edb/src/trunk/pollbox$ pip list
Package       Version
------------- -------
asgiref       3.2.3
Django        3.0
envdir        1.0.1
pip           18.1
pkg-resources 0.0.0
psycopg2      2.8.4
pytz          2019.3
setuptools    40.8.0
sqlparse      0.3.0
wheel         0.33.6
```

The `pip freeze` command will output the python software installed in the current virtual environment.

```
(.venv) mv@think:~/dev/edb/src/trunk/pollbox$ pip freeze
asgiref==3.2.3
Django==3.0
envdir==1.0.1
pkg-resources==0.0.0
psycopg2==2.8.4
pytz==2019.3
sqlparse==0.3.0
```

We can capture this output in a requirements file that can be used to recreate a new virtual environment that is exactly the same as the current one.

```
(.venv) $ pip freeze > pollbox_requirements.txt
```

The virtual environment must be activated whenever we want to work on our project. There are a couple of handy shell scripts found within the new virtual environment that will activate same.

I have created an alias for my project's virtual environment activation called `workedb`.

```
alias workedb='source ~/dev/edb/.venv/bin/activate; cd ~/dev/edb/src/trunk/pollbox'
```

When I type the alias and press return, the virtual environment is activated and my current directory is set to the base of the project.

```
mv@think:~ $ workedb
(.venv) mv@think:~/dev/edb/src/trunk/pollbox $
```

5.2 Git Repository

You should setup a git repository to store your django code. We won't cover the details of doing that, but it is a programming best practice. Having said that, a crucial step in setting up git for django involves making sure that at least the following patterns are present in your `.config/git/ignore` file (or `.gitignore`):

```
# Environments
.env
.venv
env/
venv/
ENV/
env.bak/
venv.bak/
envdir/

# Byte-compiled / optimized / DLL files
__pycache__/
*.py[cod]
*$py.class
```

There are many more entries in a programmer's git ignore file. I have a fairly comprehensive one that I can share.

6 The Example Project

Talking about django requires having an actual working project / website with a database. We have one and for some strange, unknown reason, I have implemented a rudimentary application for displaying Canadian federal election results. Specifically, to drill down to the level of a polling station, and be able to display a candidate's votes at that level.

A brief description of the data is required to get the most out of this overview. A picture is worth a thousand words:

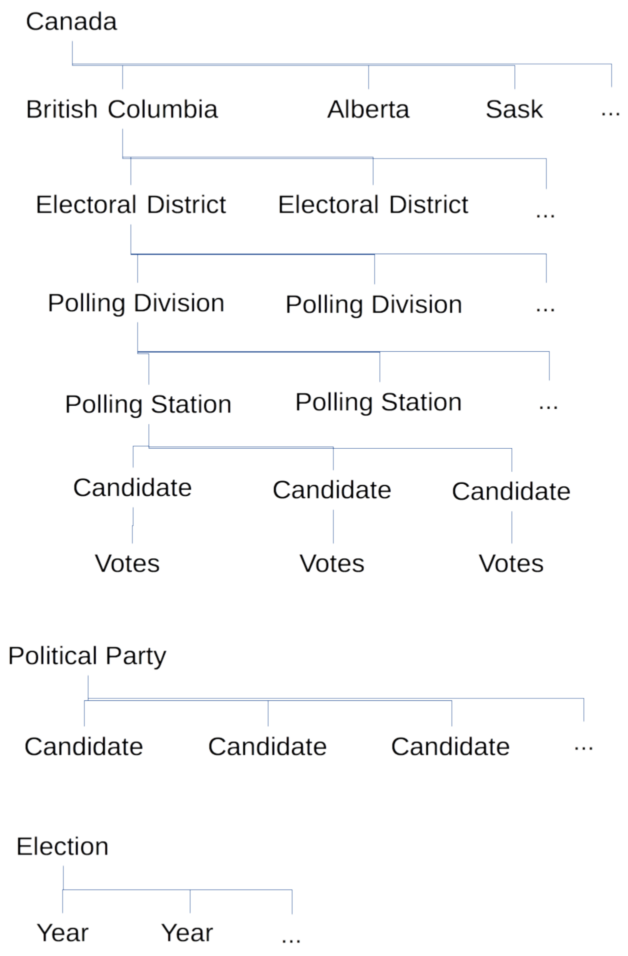


Figure 3: **A mostly accurate tree view of the relations among election data**

7 Django Projects and Django Apps

The word ‘project’ in django has a specific technical meaning, and corresponds largely to a single interactive web site. The web site is usually described as having a purpose, such as e-commerce, blogging or, as in this presentation’s case, displaying the contents of election polling station boxes.

Django ‘apps’ belong to a project and correspond to how the web site is carved into smaller, more manageable pieces. The are *not* like a phone app.

What do you mean by that?

Well, let’s say a web site represents a set of HTML pages displayed to a user, based on what URL the user sends from their browser.

Then a django project is like a container, which holds settings, media files and apps, and represents a whole website, with all its functionality. Apps are then modules for mapping user-sent URLs to views, that do specific things on, or within the website. This will become clearer.

A django project must have at least one app, and there are often many apps within one project. An app should be limited to include only the set of database tables, views and templates that logically belong together.

Splitting a large database’s models into several apps helps compartmentalize code, which reduces the impact of code changes. It can also reduce the maintenance burden by reducing inter-code dependencies.

8 Django Project

Our example django project is called `pollbox`. It represents a web site created to showcase the results of Canadian federal elections. To create it, we make sure our virtual environment is active and that we are in a directory in which we want our project to live.

```
(.venv) mv@think:~/dev/edb/src/trunk$ django-admin startproject pollbox
(.venv) mv@think:~/dev/edb/src/trunk$ ls
pollbox ...
```

Do a quick sanity check by changing to the project's directory (`pollbox`) and try and start the development server.

```
(.venv) mv@think:~/dev/edb/src/trunk$ cd pollbox/
(.venv) mv@think:~/dev/edb/src/trunk/pollbox$

(.venv) mv@think:~/dev/edb/src/trunk/pollbox$ python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 17 unapplied migration(s). Your project may not work
properly until you apply the migrations for app(s): admin, auth,
contenttypes, sessions.
Run 'python manage.py migrate' to apply them.

December 22, 2019 - 09:57:26
Django version 3.0, using settings 'pollbox.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

We quit the server using CTRL-C as directed. We have to setup a database server at this point, or use the default SQLite database that comes with django. We won't cover database setup here.

In fact, we already have a PostgreSQL server setup and running for our project.

8.1 Settings

All of the important settings for the project are in the `settings.py` file that is in the `pollbox/pollbox/` directory. It's slightly redundant to have the project directory contain another directory of the same name, but this was done to declutter the top level project directory and move project specific files into a subdirectory. It also helps with python package definitions.

We can have a quick look at the settings file to get an idea of what's inside.

8.2 Environment Directory - envdir

An environment directory is a method (and directory) for storing sensitive django settings. An example is the django `SECRET_KEY` and your database username and password. Why bother doing this? It's because when you store your settings file in git, those sensitive items can be viewed by anyone with access to the repository. Splitting these out of the settings file allows them to be kept out of the repository. It also allows you to dynamically select between development and production settings values.

Take a minute to look back at the file patterns in our git ignore setup and you'll see `envdir/` and friends.

How is it used? Follow these steps:

1. Install `envdir` using `pip install envdir` from within your active virtual environment.
2. Create an `pollbox/pollbox/envdir/` directory.
3. Place simple text files in the `envdir/` directory that will hold project specific secrets and settings.

   ```
   (.venv) mv@think:~/dev/edb/src/trunk/pollbox/pollbox$ ls -l envdir
   total 36
   -rw-r--r-- 1 mv mv 10 Jan  1 21:09 DEV_DBHOST
   -rw-r--r-- 1 mv mv  4 Dec 28 23:47 DEV_DBNAME
   -rw-r--r-- 1 mv mv  5 Feb  1  2018 DEV_DBPORT
   -rw-r--r-- 1 mv mv 26 Jan  1 21:12 DEV_DBPW
   -rw-r--r-- 1 mv mv  4 Jan  1 21:13 DEV_DBUSER
   -rw-r----- 1 mv mv 51 Jan  1 21:13 DEV_SECRET_KEY
   -rw-r--r-- 1 mv mv 43 Jan  1 21:17 DEV_STATIC_ROOT
   -rw-r----- 1 mv mv 51 Jan 21  2018 PROD_SECRET_KEY
   -rw-r----- 1 mv mv 51 Jan 21  2018 STAGING_SECRET_KEY
   ```

4. Import the `envdir` module and the `os` module into your settings file:

   ```
   import envdir
   import os
   ```

5. Use the `envdir` module and open the `envdir/` directory.

   ```
   ...
   ```

```
else:  # dev.
    DEBUG = True
    # Open our envdir/ and read it.
    envdir.open()
    secret_key_env_key = 'DEV_SECRET_KEY'
    settings_key_prefix = 'DEV_'
```

6. Use the `os.environ` function to read the values that the `envdir.open()` function call loaded into the process' envronment.

```
...
try:
    SECRET_KEY = os.environ[secret_key_env_key]

except KeyError:
    raise SuspiciousOperation("No secret key set!")
```

7. Use the mechanism for database information.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': os.environ[''.join([settings_key_prefix, 'DBNAME'])],
        'USER': os.environ[''.join([settings_key_prefix, 'DBUSER'])],
        'PASSWORD': os.environ[''.join([settings_key_prefix, 'DBPW'])],
        'HOST': os.environ[''.join([settings_key_prefix, 'DBHOST'])],
        'PORT': os.environ[''.join([settings_key_prefix, 'DBPORT'])],
    }
}
```

Environment directories are a neat way to keep sensitive information out of your repositories and can also be used to switch between development, staging and production configuration values.

8.3 URL Dispatching

An important project level file is the `pollbox/urls.py` file. It is the first step in figuring out what URL the user has sent to us for processing, and how to route the request to create a response. Our simple project has the following patterns:

```
urlpatterns = i18n_patterns(
    path('polls/', include('polls.urls')),
    path('admin/', admin.site.urls),
)
```

This project level dispatcher will do the following:

1. Discard/ignore the host and port part of the URL.
2. Try and match the start of the URL path with one of the patterns listed as paths.
3. When matched, the app's dispatching will then be processed. Note though, that the portion matched is removed before the patterns in the app's `urls.py` are checked.

We'll pause here and talk about the HTTP request, then return to an app's handling of an URL and its request.

8.4 The Request Abstraction

When a website passes a request to the django application framework via a the common gateway interface (CGI) methodology, django creates an HttpRequest object that contains all the meta data about the browser's request. This object is named `request` in all of the subsequent processing.

This is a fairly complex object and if we look at the django reference page for it, we can see the data that it holds.

It is the main vehicle for providing POST data (a.k.a. FORM data) to a view for processing. It also contains these useful values (not exclusive):

- request.method - whether the web page used GET or POST or another HTTP action.
- request.META - a dictionary holding all of the header values, such as user agent, referrer, user name, host and many more.
- request.POST or request.GET - dictionaries containing the user passed URL or post data.

The `request` object will also have more data attached to it via middleware, such as session data–how you keep track of a user's live session from page to page in your site–`request.session`. We will see an example of this in middleware, later in this presentation.

Suffice to say, whenever you see the `request` object in the various python files, you now know what it represents and contains.

9 Django App

A django app is how a user will interact with certain parts of the django project web site. An app can be thought of as a plugin (wordpress maybe), and ought to be written so that it can be moved from one django project to another without having to modify very much of it.

Create our first (and only) app under the `pollbox` project:

```
(.venv) mv@think:~/dev/edb/src/trunk/pollbox$ python manage.py startapp polls

(.venv) mv@think:~/dev/edb/src/trunk/pollbox$ ls
db.sqlite3  manage.py  pollbox    polls

(.venv) mv@think:~/dev/edb/src/trunk/pollbox$ ls polls
admin.py  apps.py  __init__.py    migrations  models.py  tests.py  views.py
```

An app has to be activated in the settings file:

```
INSTALLED_APPS = [
...
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'polls.apps.PollsConfig',
]
```

The `polls` app will contain the database models for a federal election.

9.1 Models and QuerySets

A model in django is a class that represents a database table. It is the most important part of the Object Reference Model (ORM) that hides all of the nasty SQL needed to use a database. There is far too much involved in django's model system to go into great depth in this presentation.

I'll simply show you some models that are used in our election database and explain a few bits and pieces. Also, in the views portion of an app, there is the concept of QuerySets, which is also huge and broad in scope. I will only show a few ways they are used, but they are fundamental and you will need to devote a considerable amount of time to both to master them.

This `polls` app contains quite a few database models and could be split into a few more apps. For example, it has all these models:

- class Election
- class Province
- class ProvinceElectionInfo
- class ElectoralDistrict
- class ElectoralDistrictElection
- class PoliticalParty
- class Candidate
- class CandidateElection
- class PollingDivision
- class PollingStation
- class PollingStationElection
- class PollingStationCandidateVotes
- class PollingStationCandidateResults
- class PartyProvinceResults

Each of those is a table in the database. It might make sense to place the geography related tables: Province, ElectoralDistrict, and PollingDivision into their own `places` app; place people related tables: Candidate and PoliticalParty into their own `people` app; as well as placing election related tables: Election, ProvinceElectionInfo, ElectoralDistrictElection, CandidateElection, and so on into their own `election` app.

Some of these decisions are tricky, because there is definite coupling between apps, as some of the model names suggest. For example, the `election` app, if we were to split it out, would have references to models in the `places` and `people` apps. This is what coupling refers to.

Where would you put the PartyProvinceResults model? It ties together three tables from possibly three different apps.

The concept of an app should be thought of as somewhat of a stand-alone set of models and views for portability and reuse reasons. Here, the various tables aren't likely to be used in other web sites, so portable apps are not really needed, and one large app is used.

If the scope of the project was to be increased and we started creating more models, then it would be prudent to refactor the `polls` app and split it into smaller apps.

9.2 URL Patterns

This section logically continues from the `pollbox` project's URL dispatcher section. Remember the main URL dispatching in the project level `urls.py` loads the app's URL patterns that matched the browser's URL. The next level of matching happens to the remaining part of the URL path, less the portion removed at the previous level. Thus, an app's `urls.py` will contain patterns to match based on the remaining browser's URL path, resulting in a call to the specified view function. This is easier to look at then it is to explain in words.

Our patterns are as follows:

```
from . import views
app_name = 'polls'

urlpatterns = [
    path('', views.index, name='index'),
    path('election/<int:election_id>/', views.election, name='election'),

    path('district/<int:district_id>/', views.district, name='district'),
    path('district/', views.districts, name='districts'),

    path('prov/<int:province_id>/', views.province, name='province'),
    path('prov/', views.provinces, name='provinces'),

    path('party/<int:political_party_id>/', views.party, name='party'),
    path('party/', views.parties, name='parties'),
...
]
```

Suppose the URL path was `polls/prov/`. The project level dispatch would have matched the `polls/` part, stripped it from the path, and loaded the `polls` app's patterns for processing. The app patterns would now match the `prov/` portion.

So, if the URL has `prov/` with no integer number following it, then the view function `views.provinces` will be called and be expected to return a full HTTP response (status, headers, and HTML). The `provinces` function will receive the `request` object as its first argument, and so will have access to all the information contained in the request sent by the browser.

Note that each path has a name argument. Naming url paths is an important part of the DRY principle. The names are then used in templates (via the url template tag) to populate `href` links. This allows changing the route to a view, via the url path, in one place, while automatically keeping links in templates accurate. That is, you won't have to edit a template when you change an url path. This also follows principle 4 (reduce future maintenance).

9.3 Views

This is the second major part of URL routing, where the actual request is received and processed to generate a response for the user. It is called via the URL dispatcher, as described above.

Most views are python functions and are found in an app's `views.py` file. Here is an example view for showing a province's election details. Note the `request` object that is passed as the first argument to the function.

There is also a second argument passed to the view, called `province_id` and it is an integer. It is given to us by the URL dispatcher based on the fact that `path('prov/<int:province_id>/', ...` was the matching pattern.

```
def province(request, province_id):

    # First get the electoral districts in the province.
    electoral_districts = ElectoralDistrict.objects.filter(
          province__pk=province_id
          ).filter(
          electoraldistrictelection__election__pk=request.session['election_id']
          )
    # Get all the candidates in the province, for this election
    candidates_in_province = CandidateElection.objects.filter(
         election__pk=request.session['election_id']
         ).filter(electoral_district__province__pk=province_id)

    # Get the number of elected candidates
    number_elected = candidates_in_province.filter(elected_indicator=True).count()

    province = get_object_or_404(Province, pk=province_id)
    context = {
        'province': province,
        'number_elected': number_elected,
        'electoral_districts': electoral_districts,
        'candidates_in_province': candidates_in_province,
    }
    return render(request, 'polls/province.html', context)
```

In the above code, there is quite a bit of ORM machinery. There are models, object managers that produce querysets, aggregating methods (`.count()`) and other interesting concepts.

There is also a dictionary created called `context`. This is an import part of creating an HTTP response object, and is the main vehicle in passing information to django's template rendering system. Note that all the variables created in the view–`province`, `number_elected`, and so on–get a spot in the context dictionary. Those are the names they'll be referred to as in the templates we'll see later.

9.4 Context and Context Processors

As noted above, when django starts a template rendering process, there is usually a dictionary called `context` involved. This context will contain all of the data (queries, variables and such) that a template needs to fully display a web page.

This context can be created within the view, and will be augmented by any custom context processors that we create. An example of a custom context processor is shown below. Its function is to lookup the active election using the election id that exists in the `request` session. It will then return a dictionary with both a list of elections as a queryset, and the active one as an `election` object.

These variables become available within a template when the view uses the common django `render` function. The `election_list` is used to populate the left sidebar of elections for every view.

```
(.venv) mv@think:~/dev/edb/src/trunk/pollbox/polls$ more context_processors.py
"""
Set of functions to return dictionaries to most templates
for commonly displayed data.
"""

from .models import Election

def get_elections(request):
    election_list = Election.objects.order_by('-election_date')

    # Use some session variables
    election_id = request.session['election_id']
    election = Election.objects.get(pk=election_id)

    return { 'election_list': election_list,
             'election': election }
```

The above context processor is added in the `settings.py` file to the `TEMPLATES` dictionary:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [ 'templates', ],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
                'polls.context_processors.get_elections',
            ],
        },
    },
]
```

Context processors are great for adding content used in sidebars, or menus that are visible in all or many of your views, and help reduce code duplication (DRY).

9.5 Returning A Response

So far we've seen URL dispatching to a view, and a view grabbing data, and placing it in a context. The next important part in this process is creating the actual HttpResponse object. The simplest and most sensible way to do this in a view is to import the django `render` shortcut, pass it a request, template name, and a context dictionary. This can be seen in our `province` view above.

```
    return render(request, 'polls/province.html', context)
```

At this point, we need to talk about the template system and just exactly where that dang HTML stuff lives.

10 Templates - Hypertext Markup Language (HTML) Files

Templates follow the DRY principle.

Templates are how you apply a theme and structure, in an HTML sense, to your project. A project can use one of a number of templating subsystems. There is the django default system, called the Django Template Language (DTL), and support for a popular alternative named Jinja2. We use the DTL.

There can be one or more base templates, although one is recommended so as to be able to stick to the don't repeat yourself principle. A base template is the most complete copy of the HTML structure that your site will have. It will contain the HTML declaration, the HEAD block, the BODY block and so on, along with replaceable text content block indicators.

Templating works by an extend-and-override methodology.

Let's study the `pollbox` project's base template, called `base-grid.html`.

10.1 Django Template Language

Django has a rich set of template tags and filters to go along with them. There is almost a mini programming language within the DTL. We'll look at some of our templates to get an idea of what can be done.

Let's look at the `polls/district.html` template.

10.2 Custom Filters

While trying to create the table for the electoral district results, I needed to use the modulus math function (determine the remainder of a division). Unfortunately, django didn't include a math function for modulus in their set of built-in filters. Fortunately, you can create your own template tags and filters.

To do so, we create a `templatetags/` directory within our app. In this file we can use django's template magic to register a function that we create, which will be available in our templates.

```
(.venv) mv@think:~/dev/edb/src/trunk/pollbox$ more polls/templatetags/poll_math.py

from django import template
register = template.Library()

@register.filter
def mod(value, arg):
    """Returns the remainder (modulus) of value divided by arg"""
    return value % arg
```

It can be used by loading the poll_math module at the top of the template, then by using it as a normal filter:

```
...
{% load i18n static poll_math %}
...
  {% if forloop.counter|mod:candidate_count == 0 %}
        </tr>
  {% endif %}
```

The above will close a table row whenever we've looped over a queryset (denoted by the `forloop.counter` variable–the `value` argument to our `mod` function) in the same number of times as we have candidates (denoted by the `candidate_count` variable–the `arg` argument to our `mod` function).

11 Middleware

Middleware is a name for a set of hooks for python code to run between URL dispatching and your view being called. It is a way of running code to help customize sessions, to handle language processing, and various other tasks. Mostly is just works without modification, but you may have to add your own at some point. I have added some middleware to use as an example. It simply sets a session variable inside the request object that our custom context processor can use to get the current election database object.

When you create your custom middleware, you add it to the list of already enabled middleware in the `settings.py` file:

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'polls.middleware.session.SessionMiddleware',
    'django.middleware.locale.LocaleMiddleware',
...
]
```

In the above list, our `polls` app adds a middleware component that will add a session variable to the request, which can be used by the subsequent view, via the `request` dictionary.

The actual python code for that middleware is located in `polls/middleware/session.py` and looks like this:

```
from polls.models import Election

class SessionMiddleware:
    def __init__(self, get_response):
        # One-time configuration and initialization.
        self.get_response = get_response
        # Uncommenting the next 'raise' line acts as an off
        # switch for the middleware.
        #raise MiddlewareNotUsed


    def __call__(self, request):
        # Code to be executed for each request before
        # the view (and later middleware) are called.

        current_election = Election.objects.get(election_date__year=2015)
        election_id = request.session.setdefault('election_id', current_election.pk)

        response = self.get_response(request)

        # Code to be executed for each request/response after
        # the view is called.

        return response
```

I realize the above contains a great deal of python and django code that is deep, strange magic, but the interesting bit is the part which has `election_id = request.session.setdefault('election_id', current_election.pk)`. That says to set a default election id for the user, since at this point there is only one election with data in our database.

Subsequently, since all our views are passed the `request` object, they will have access to the value stored in `request.session['election_id']`.

12 Media and Resource Files

I will briefly mention media and other resource files. The definitive guide is linked here. The set of files included in this are:

- Images - stored in the project's `static/img` directory.
- Cascading Style Sheets (CSS) - stored in the project's `static/css` directory.
- JavaScript - stored in the project's `static/js` directory.

Our simple site doesn't have any images and uses only minimal CSS and javascript:

```
(.venv) mv@think:~/dev/edb/src/trunk/pollbox$ tree static/

static/
 css
  grid.css
  site.css
 img
 js
     site.js

3 directories, 3 files
```

Within templates CSS and javascript are referenced like so (note the use of the `static` template tag):

```
...
{% load i18n static %}
...
        {% block extrastyle %}
        <link rel="stylesheet" type="text/css"
          href="{% static 'css/grid.css' %}" />
        {% endblock %}

...
<!-- Scripts -->
<script type="text/javascript"
  src="{% static 'js/site.js' %}"></script>
```

13 Fixtures

A wonderful feature of django's database driven framework is called fixtures. These are files of formatted data that contain the contents of your database from a particular point in time.

Fixtures are useful for storing a copy of your database's models and can be used for a few purposes such as:

- Per-app or full project data backups.
- Storing data that you wish to use in tests, so as to have a known state.

13.1 Dump Data

To create a fixture file to snapshot your database contents, use the `dumpdata` command:

```
python manage.py dumpdata -o fixtures/fixture_file
```

You can restrict your dumps to specific apps or models.

13.2 Load Data

To load a fixture file into your database contents, use the `loaddata` command:

```
python manage.py loaddata fixtures/fixture_file
```

You can restrict your loads to specific apps.

13.3 Fixtures for Testing

Testing (more on this below) often requires data to be present in a database to make the tests actually useful. Hence, you can create and use fixtures specifically for loading a test database. We do this with a fixture called `initial_data.json` that we created soon after we added an election and the 13 provinces/territories.

14 Database Migration Scripts

These are files that produce fully fledged SQL statements that can, and do, allow you to generate your database tables and structures. They are created for you by management commands. Changes to any database model will be noticed and a new migration script will be created to both store the actual change, and apply the changes to the database itself. This gives you a fantastic log of database evolution as well.

```
python manage.py makemigrations
python manage.py sqlmigrate
python manage.py migrate
```

When you add or change models within your project's apps, you will have to run the `makemigrations` command. This is very powerful and convenient, as it hides any database interactions that use the structured query language (SQL). Regardless, you can see the SQL commands that are involved, if you want, by using the `sqlmigrate` command.

Once you're satisfied with the updates, you then run the `migrate` command to send the changes to the database server.

15 Python Shell with Django

One of the more useful bits included with django is the ability to start a python shell using your django projects settings, which allows you to import and inspect your models.

```
(.venv) mv@think:~/dev/edb/src/trunk/pollbox$ python manage.py shell
Base dir:  /home/mv/dev/edb/src/trunk/pollbox
Static dirs:  ['/home/mv/dev/edb/src/trunk/pollbox/static', '/var/www/static/']
Python 3.7.5 (default, Apr 19 2020, 20:18:17)
[GCC 9.2.1 20191008] on linux
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> from polls.models import Election
>>> e = Election()
>>> e.description = "44th General Election"
>>> import datetime
>>> e.election_date = datetime.datetime.fromisoformat('2022-05-23')
>>> e.save()
>>> # If we look at our database, we will see a new election
>>> e.year
'2022'
>>> e.delete()
(1, {'polls.ProvinceElectionInfo': 0, 'polls.ElectoralDistrictElection': 0, 'polls.Po
>>> quit()
```

16 Internationalization and Localization

Internationalization (i18n) and Localization (l10n) are the mechanisms in which a software project can become multi-lingual. They are separate activities in a technical sense. Internationalization is the preparation and proper coding of a software project to be capable of supporting many languages. It includes marking all character strings as translatable. Localization is the subsequent gathering of these strings, organizing them into files and then having translators create the different language resources.

In our project we want to plan for English and French support.

16.1 Enable Translation

To specify the languages your project should support use `LANGUAGES`, along with a backup `LANGUAGE_CODE`. Add these lines to your `settings.py` file:

```
from django.utils.translation import gettext_lazy as _

LANGUAGES = [
    ('en', _('English')),
    ('fr', _('French')),
]

LANGUAGE_CODE = 'en-us'
```

Now our project will support english and french with the default being U.S. english.

To enable dynamic language detection, add the following to `MIDDLEWARE` in your `settings.py` file:

```
    'django.middleware.locale.LocaleMiddleware',
```

Note that its placement in the list of middleware items is important. See the django documentation about this.

Dynamic means that an algorithm is followed to locate the language the user is requesting:

- Look in the request URL
- Look in a cookie
- Look in the Accept-Language header from the browser.

Also ensure that these are set:

```
USE_I18N = True
USE_L10N = True
```

A further requirement is that `LOCALE_PATHS` be set. This is where the language resource files are stored.

```
LOCALE_PATHS = [
    os.path.join(BASE_DIR,'locale'),
    '/var/local/translations/locale',
]
```

For production, it is likely that your messages files are copied to a place (the second location in the list) to which your web server/django project has access.

16.2 Django GetText Support

Django allows for automatic translation of strings to aid in the use of multiple languages within your project. Behind the scenes, it uses the Gettext mechanism (see Gettext on Wikipedia).

Briefly, gettext is a function that accepts the text to be translated as a string parameter and returns the currently set language's translation of that string at runtime.

The gettext function is often aliased to just the underscore character to reduce typing.

The GNU Gettext software must be installed on the system. If you see this error:

```
(.venv) mv@think:~/dev/edb/src/trunk/pollbox$ python manage.py  makemessages
CommandError: Can’t find xgettext. Make sure you have GNU gettext tools 0.15 or newer
```

Then you need to install `gettext`.

```
mv@think:~$ sudo apt install gettext
...
```

Django comes with translation utilities based on gettext, and they should be imported at the top of most of your python files: `views.py`, `models.py`, and any others that will have text strings needing translation.

```
# For bilingual support
from django.utils.translation import gettext as _
```

Often, in your `models.py` files, you’ll have to use the deferred translation function, called `gettext_lazy`, to deal with database field definitions.

```
from django.utils.translation import gettext, gettext_lazy as _
```

In the above case, you’ll need to know when to use gettext itself, rather than the usual shortcut, which is now the lazy version. There are quite a few other gettext variants that are needed from time-to-time, for pluralization (`ngettext`), or other special case considerations (`gettext_noop`).

16.3 DTL Translation Directives

Within templates, you’ll load the `i18n` (internationalization) module and use the `trans` directive to mark strings for translation:

```
# For bilingual support
{% load i18n static %}

<title>{% trans ’Elections In Canada’ %} </title>
...
```

16.4 Extracting and Compiling Translation Strings

Once you've marked all your strings as translatable, they can be searched for and extracted using the django manage.py translation functions.

Extracting messages:

```
(.venv) mv@think:~/dev/edb/src/trunk/pollbox$ python manage.py help makemessages
usage: manage.py makemessages
...

Runs over the entire source tree of the current directory and pulls out all
strings marked for translation. It creates (or updates) a message file in the
conf/locale (in the django tree) or locale (for projects and applications)
directory. You must run this command with one of either the --locale,
--exclude, or --all options.
...
```

After your translators have provided all the translated strings and populated the *.po files, you'll run the message compiling command to generate the file format that the gettext software expects:

```
(.venv) mv@think:~/dev/edb/src/trunk/pollbox$ python manage.py help compilemessages
usage: manage.py compilemessages
...

Compiles .po files to .mo files for use with builtin gettext support.
...
```

The above command will generate new *.mo files and result in a set of files and directories as follows:

```
(.venv) mv@think:~/dev/edb/src/trunk/pollbox/locale$ pwd
/home/mv/dev/edb/src/trunk/pollbox/locale
(.venv) mv@think:~/dev/edb/src/trunk/pollbox/locale$ tree
.
 en
  LC_MESSAGES
      django.mo
      django.po
 fr
```

```
    LC_MESSAGES
        django.mo
        django.po

4 directories, 4 files
```

17 Testing Systems

The use of automated tests, for a reasonably sized website, is crucial to maintaining your sanity when you fix bugs or add features. A small website can be tested by clicking through most pages and satisfying yourself that there are no unintended side effects of the changes you made. Anything larger, though, is impossible to test this way, as the combinations of interactions in your website grow.

To address this testing need, we must create small, simple tests for each part of our apps.

17.1 Test Databases

Tests that use models, or other database information, do not mess with the production database. A new test database is created for the tests and destroyed when they are finished.

The test database can be loaded with pre-configured sample data by using fixtures.

17.2 DocTests

What is a doctest? When you create python code, such as classes, you usually provide some comments about the class and how to use it. Normally, these comments are multiline and called docstrings.

```
"""
This is a multiline
comment between
two sets of triple quotes.
```

```
It is also known as a docstring.
"""
```

Our app has one class with a docstring that is in the form of a doctest. Optimally, all classes will have docstrings in this form, so that all classes are tested properly.

Here is our Election class' doctest:

```
class Election(models.Model):
    """
    This is a class that contains information about a Canadian federal
    election.

    It contains a description, election date and
    a year property that is derived from the date.

    >>> import datetime
    >>> election = Election()
    >>> election.description = 'Doctest election description'
    >>> election.election_date = datetime.datetime.fromisoformat('2015-10-19')

    >>> election.save()
    >>> election  #doctest: +ELLIPSIS
    <Election: ...>

    >>> election.description
    'Doctest election description'
    >>> election.election_date
    datetime.datetime(2015, 10, 19, 0, 0)
    >>> election.election_date.isoformat()
    '2015-10-19T00:00:00'
    >>> election.year
    '2015'

    >>> epk = election.pk
    >>> election.delete()  #doctest: +ELLIPSIS
    (1, ...)

    >>> election = Election.objects.get(pk=epk)
    Traceback (most recent call last):
        ...
    polls.models.Election.DoesNotExist: Election matching query does not exist.

    >>> election.save()
    >>> epk = election.pk
    >>> election = Election.objects.get(pk=epk)
    >>> election  #doctest: +ELLIPSIS
    <Election: ...>

    # Clean up.
    >>> election.delete()  #doctest: +ELLIPSIS
    (1, ...)

    """
...
```

Doctests can be enabled in the app's `tests.py` file by adding this boilerplate:

```
import doctest
from polls import models
...

def load_tests(loader, tests, ignore):
    tests.addTests(doctest.DocTestSuite(models))
    return tests
```

When you run `manage.py test`, any models that have doctest strings will have their tests run.

17.3 Unit Tests

The second part of testing is to have unit tests. A unit test is designed to test a specific set of functionality for an app. I have created a simple unit test that simulates a browser client calling two of our project's URLs.

The unit tests also live in `tests.py` and are quite simple at this point:

```
from django.test import TestCase

# Create your tests here.
# load_tests clue gotten from StackOverflow:
# https://stackoverflow.com/questions/2380527/django-doctests-in-views-py

import unittest
import doctest
from polls import models

class SimpleTest(TestCase):
    # Our initial data has one election object and 13 provinces
    fixtures = ['initial_data']

    def test_provinces(self):
        response = self.client.get('/polls/prov/', follow=True)
        self.assertEqual(response.status_code, 200)

        # Check that the rendered context contains 13 provinces
        self.assertEqual(len(response.context['province_list']), 13)

    def test_index(self):
        response = self.client.get('/polls/', follow=True)
        self.assertEqual(response.status_code, 200)
```

Note the `fixtures` setting in our `SimpleTest` that denotes a set of test data to load into the test database. Finally, we can run the tests, with a little more detail of what's going on (`--verbosity 2`):

```
(.venv) mv@think:~/dev/edb/src/trunk/pollbox$ python manage.py test --verbosity 2
Base dir:  /home/mv/dev/edb/src/trunk/pollbox
Static dirs:  ['/home/mv/dev/edb/src/trunk/pollbox/static', '/var/www/static/']
Creating test database for alias 'default' ('test_edb')...
Operations to perform:
  Synchronize unmigrated apps: messages, staticfiles
  Apply all migrations: admin, auth, contenttypes, polls, sessions
Synchronizing apps without migrations:
  Creating tables...
    Running deferred SQL...
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying polls.0001_initial... OK
  Applying polls.0002_auto_20200107_0743... OK
  Applying polls.0003_auto_20200109_0604... OK
  Applying polls.0004_partyprovinceresults... OK
  Applying polls.0005_auto_20200110_0459... OK
  Applying polls.0006_auto_20200111_0455... OK
  Applying polls.0007_auto_20200210_0156... OK
  Applying sessions.0001_initial... OK
System check identified no issues (0 silenced).
test_index (polls.tests.SimpleTest) ... ok
test_provinces (polls.tests.SimpleTest) ... ok
Election (polls.models)
Doctest: polls.models.Election ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.192s

OK
Destroying test database for alias 'default' ('test_edb')...
```

All three of our tests ran and passed. One doctest, and two unit tests.