Django introduction

Created by 이 윤 준 (yoonjoon.lee@gmail.com)

May, 2019

- "What is Django?"
- An overview of what makes this web framework special
- Outline of the main features, including some of the advanced functionality
- Show some of the main building blocks of a Django application

What is Django?

Django is a high-level Python web framework that enables rapid development of secure and maintainable websites.

Django takes care of much of the hassle of web development, so we can focus on writing our app without needing to reinvent the wheel.

It is free and open source, has a thriving and active community, great documentation, and many options for free and paid-for support.

Django helps us write software that is:

Complete: Django follows the "Batteries included" philosophy and provides almost everything developers might want to do "out of the box".

Versatile: Django can be used to build almost any type of website — from content management systems and wikis, through to social networks and news sites. It can work with any client-side framework, and can deliver content in almost any format.

Secure: Django helps developers avoid many common security mistakes by providing a framework that has been engineered to "do the right things" to protect the website automatically.

Scalable: Django uses a component-based “shared-nothing” architecture. Having a clear separation between the different parts means that it can scale for increased traffic by adding hardware at any level: caching servers, database servers, or application servers.

Maintainable: Django code is written using design principles and patterns that encourage the creation of maintainable and reusable code. In particular, it makes use of the Don't Repeat Yourself (DRY) principle so there is no unnecessary duplication, reducing the amount of code. Django also promotes the grouping of related functionality into reusable "applications" and, at a lower level, groups related code into modules

Portable: Django is written in Python, which runs on many platforms. That means that you are not tied to any particular server platform.

Where did it come from?

Django was initially developed between 2003 and 2005 by a web team who were responsible for creating and maintaining newspaper websites.

After creating a number of sites, the team began to factor out and reuse lots of common code and design patterns.

This common code evolved into a generic web development framework, which was open-sourced as the "Django" project in July 2005 and recently released 2.0 in 2017.

How popular is Django?

A better question is whether Django is "popular enough" to avoid the problems of unpopular platforms.

- Is it continuing to evolve?
- Can you get help if we need it?
- Is there an opportunity for us to get paid work if we learn Django?

High-profile sites that use Django include: Disqus, Instagram, Knight Foundation, MacArthur Foundation, Mozilla, National Geographic, Open Knowledge Foundation, Pinterest, and Open Stack.

Is Django opinionated?

Opinionated frameworks are those with opinions about the "right way" to handle any particular task.

They often support rapid development *in a particular domain* because the right way to do anything is usually well-understood and well-documented.

However they can be less flexible at solving problems outside their main domain, and tend to offer fewer choices for what components and approaches they can use.

Unopinionated frameworks, have far fewer restrictions on the best way to glue components together to achieve a goal, or even what components should be used.

They make it easier for developers to use the most suitable tools to complete a particular task, albeit at the cost that we need to find those components ourself.

Django is "**somewhat opinionated**", and hence delivers the "best of both worlds".

It provides a set of components to handle most web development tasks and one (or two) preferred ways to use them.

However, Django's decoupled architecture means that we can usually pick and choose from a number of different options, or add support for completely new ones if desired.

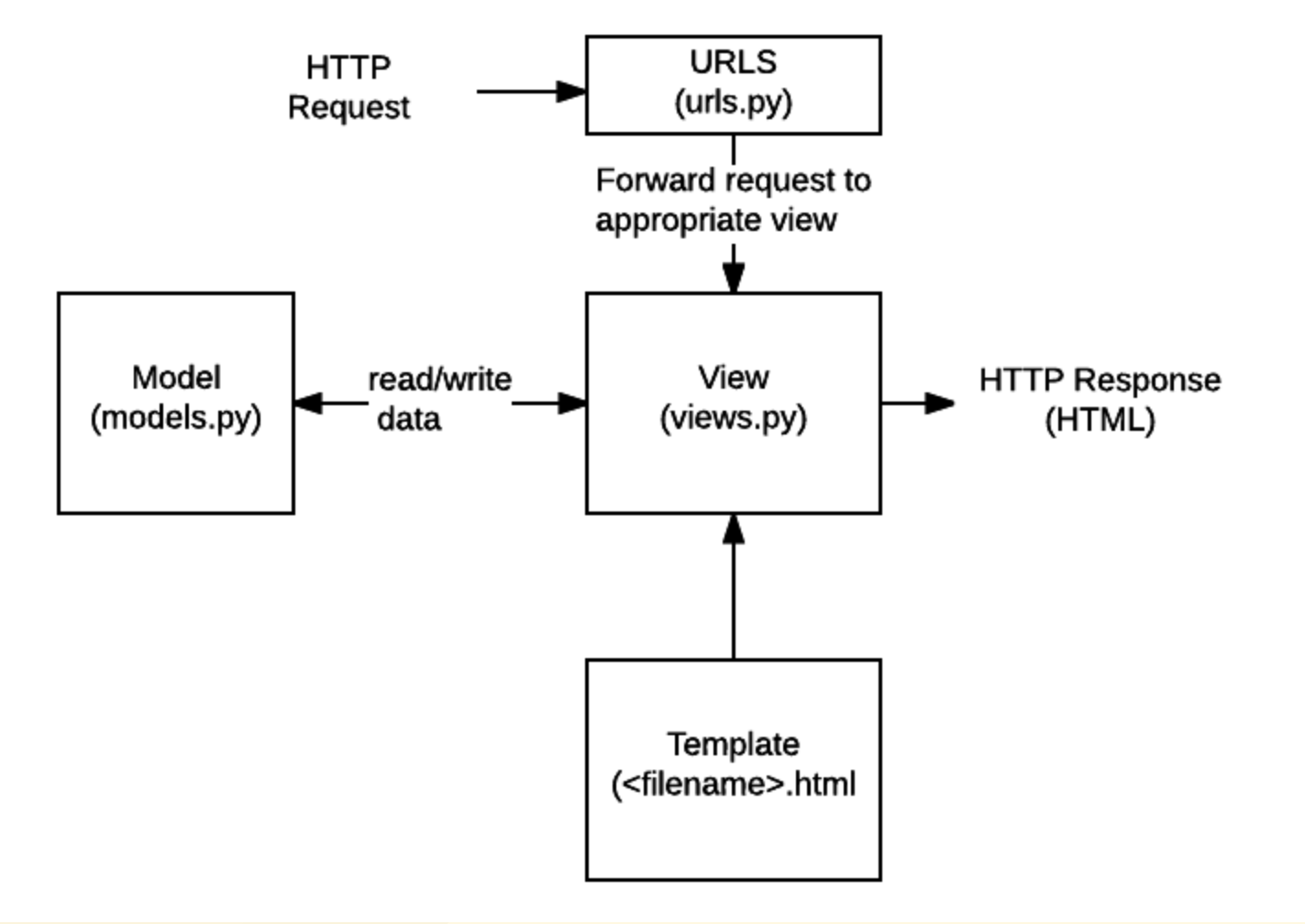
What does Django code look like?

In a traditional data-driven website, a web application waits for HTTP requests from the web browser.

When a request is received the application works out what is needed based on the URL and possibly information in `POST` data or `GET` data.

Depending on what is required it may then read or write information from a database or perform other tasks required to satisfy the request.

The application will then return a response to the web browser, often dynamically creating an HTML page


HTTP
Request
URLS
(urls.py)
Forward request to
appropriate view
Model
(models.py)
read/write
data
View
(views.py)
HTTP Response
(HTML)
Template
(<filename>.html

- **URLs**: While it is possible to process requests from every single URL via a single function, it is much more maintainable to write a separate view function to handle each resource.
- **View**: A view is a request handler function, which receives HTTP requests and returns HTTP responses. Views access the data needed to satisfy requests via *models*, and delegate the formatting of the response to *templates*.
- **Models**: Models are Python objects that define the structure of an application's data, and provide mechanisms to manage (add, modify, delete) and query records in the database.
- **Templates**: A template is a text file defining the structure or layout of a file (such as an HTML page), with placeholders used to represent actual content. A *view* can dynamically create an HTML page using an HTML template, populating it with data from a *model*. A template can be used to define the structure of any type of file.

Sending the request to the right view (urls.py)

A URL mapper is typically stored in a file named **urls.py**. The mapper (`urlpatterns`) defines a list of mappings between routes (specific URL *patterns*) and corresponding view functions.

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('book/<int:id>/', views.book_detail,
         name='book_detail'),
    path('catalog/', include('catalog.urls')),
    re_path(r'^([0-9]+)/$', views.best),
]
```

The `urlpatterns` object is a list of `path()` and/or `re_path()` functions.

Handling the request (views.py)

Views are the heart of the web application, receiving HTTP requests from web clients and returning HTTP responses. In between, they marshall the other resources of the framework to access databases, render templates, etc.

```
# filename: views.py (Django view functions)

from django.http import HttpResponse

def index(request):
    # Get an HttpRequest - the request parameter
    # perform operations using information from the request.
    # Return HttpResponse
    return HttpResponse('Hello from Django!')
```

Defining data models (models.py)

Django web applications manage and query data through Python objects referred to as models.

Models define the structure of stored data, including

- the field *types*,
- their maximum size,
- default values,
- selection list options,
- help text for documentation,
- label text for forms, etc

The definition of the model is independent of the underlying database.

Once we've chosen what database want to use, we don't need to talk to it directly at all — we just write our model structure and other code, and Django handles all the dirty work of communicating with the database.

The code snippet shows a very simple Django model for a `Team` object.

```
# filename: models.py

from django.db import models

class Team(models.Model):
    team_name = models.CharField(max_length=40)

    TEAM_LEVELS = (
        ('U09', 'Under 09s'),
        ('U10', 'Under 10s'),
        ('U11', 'Under 11s'),
        ...  #list other team levels
    )
    team_level = models.CharField(max_length=3,
                                  choices=TEAM_LEVELS,
                                  default='U11')
```

Querying data (views.py)

The Django model provides a simple query API for searching the database. This can match against a number of fields at a time using different criteria (e.g. exact, case-insensitive, greater than, etc.), and can support complex statements.

The code snippet shows a view function (resource handler) for displaying all of our U09 teams.

```
# filename: views.py

from django.shortcuts import render
from .models import Team

def index(request):
  list_teams = Team.objects.filter(team_level__exact="U09")
  context = {'youngest_teams': list_teams}
  return render(request, '/best/index.html', context)
```

Rendering data (HTML templates)

Template systems allow us to specify the structure of an output document, using placeholders for data that will be filled in when a page is generated.

Templates are often used to create HTML, but can also create other types of document.

Django supports both its native templating system and another popular Python library called Jinja2 out of the box.

The code snippet shows what the HTML template called by the `render()` function.

```
# filename: best/templates/best/index.html

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Home page</title>
</head>
<body>
  {% if youngest_teams %}
    <ul>
      {% for team in youngest_teams %}
        <li>{{ team.team_name }}</li>
      {% endfor %}
    </ul>
  {% else %}
    <p>No teams are available.</p>
  {% endif %}
</body>
</html>
```

What else can you do?

A few of the other things:

- **Forms**: HTML Forms are used to collect user data for processing on the server. Django simplifies form creation, validation, and processing.
- **User authentication and permissions**: Django includes a robust user authentication and permission system that has been built with security in mind.
- **Caching**: Django provides flexible caching so that we can store all or part of a rendered page so that it doesn't get re-rendered except when necessary.
- **Administration site**: The Django administration site is included by default. It makes it trivially easy to provide an admin page for site administrators to create, edit, and view any data models in our site.
- **Serialising data**: Django makes it easy to serialise and serve our data as XML or JSON. This can be useful when creating a web service or