What is Django?

Django is a high-level Python web framework that enables rapid development of secure and maintainable websites. Built by experienced developers, Django takes care of much of the hassle of web development, so you can focus on writing your 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 you write software that is:

Complete

Django follows the "Batteries included" philosophy and provides almost everything developers might want to do "out of the box". Because everything you need is part of the one "product", it all works seamlessly together, follows consistent design principles, and has extensive and up-to-date documentation.

Versatile

Django can be (and has been) 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 (including HTML, RSS feeds, JSON, XML, etc). The site you are currently reading is built with Django!

Internally, while it provides choices for almost any functionality you might want (e.g. several popular databases, templating engines, etc.), it can also be extended to use other components if needed.

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. For example, Django provides a secure way to manage user accounts and passwords, avoiding common mistakes like putting session information in cookies where it is vulnerable (instead cookies just contain a key, and the actual data is stored in the database) or directly storing passwords rather than a password hash.

A password hash is a fixed-length value created by sending the password through a cryptographic hash function. Django can check if an entered password is correct by running it through the hash function and comparing the output to the stored hash value. However due to the "one-way" nature of the function, even if a stored hash value is compromised it is hard for an attacker to work out the original password.

Django enables protection against many vulnerabilities by default, including SQL injection, cross-site scripting, cross-site request forgery and clickjacking (see Website security for more details of such attacks).

Scalable

Django uses a component-based "shared-nothing" architecture (each part of the architecture is independent of the others, and can hence be replaced or changed if needed). 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. Some of the busiest sites have successfully scaled Django to meet their demands (e.g. Instagram and Disqus, to name just two).

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 (along the lines of the Model View Controller (MVC) pattern).

Portable

Django is written in Python, which runs on many platforms. That means that you are not tied to any particular server platform, and can run your applications on many flavours of Linux, Windows, and Mac OS X. Furthermore, Django is well-supported by many web hosting providers, who often provide specific infrastructure and documentation for hosting Django sites.

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.

Django has continued to grow and improve, from its first milestone release (1.0) in September 2008 through to the recently-released version 3.1 (2020). Each release has added new functionality and bug fixes, ranging from support for new types of databases, template engines, and caching, through to the addition of "generic" view functions and classes (which reduce the amount of code that developers have to write for a number of programming tasks).

Django is now a thriving, collaborative open source project, with many thousands of users and contributors. While it does still have some features that reflect its origin, Django has evolved into a versatile framework that is capable of developing any type of website.

How popular is Django?

There isn't any readily-available and definitive measurement of popularity of server-side frameworks (although you can estimate popularity using mechanisms like counting the number of GitHub projects and StackOverflow questions for each platform). 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 you need it? Is there an opportunity for you to get paid work if you learn Django?

Based on the number of high profile sites that use Django, the number of people contributing to the codebase, and the number of people providing both free and paid for support, then yes, Django is a popular framework!

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

Is Django opinionated?

Web frameworks often refer to themselves as "opinionated" or "unopinionated".

Opinionated frameworks are those with opinions about the "right way" to handle any particular task. They often support rapid development in a particular domain (solving problems of a particular type) 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.

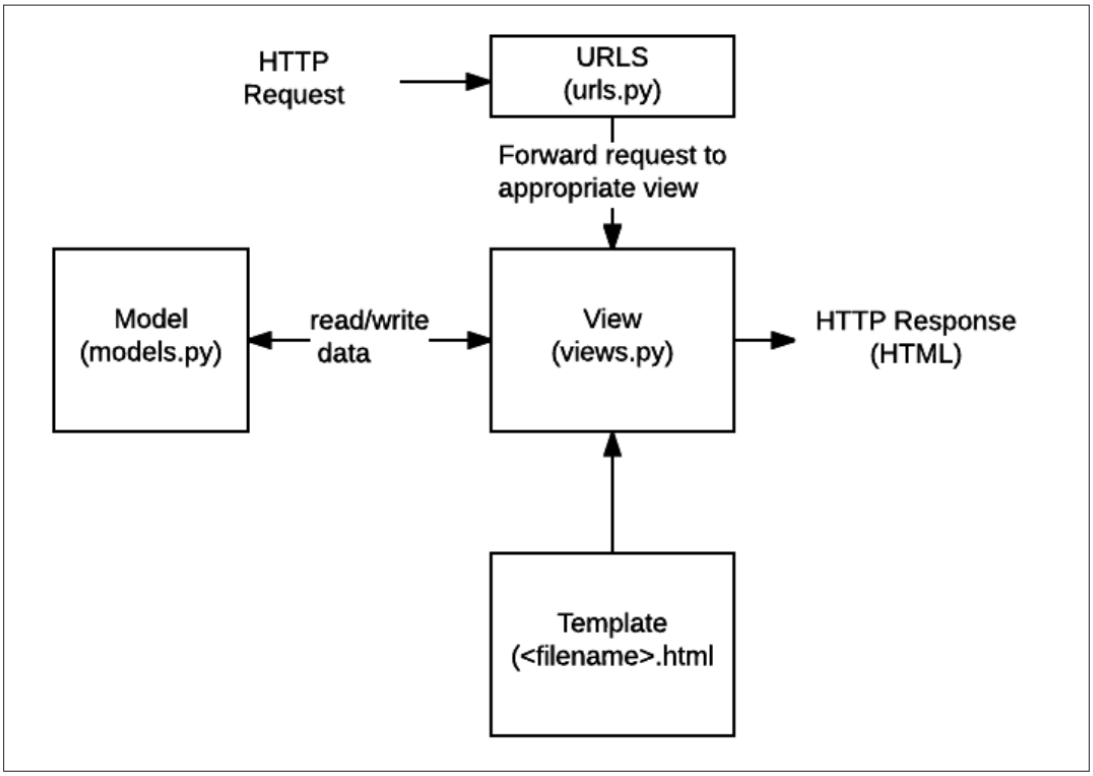
Unopinionated frameworks, by contrast, have far fewer restrictions on the best way to glue components together to achieve a goal, or even what components should be used. They make it easier for developers to use the most suitable tools to complete a particular task, albeit at the cost that you need to find those components yourself.

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 you 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 (or other client). 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 for the browser to display by inserting the retrieved data into placeholders in an HTML template.*

A project is a sum of many applications. Every application has an objective and can be reused into another project, like the contact form on a website can be an application, and can be reused for others. See it as a module of your project.



- ***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. A URL mapper is used to redirect HTTP requests to the appropriate view based on the request URL. The URL mapper can also match particular patterns of strings or digits that appear in a URL and pass these to a view function as data.*
- ***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; it doesn't have to be HTML!*

- *The sections below will give you an idea of what these main parts of a Django app look like (we'll go into more detail later on in the course, once we've set up a development environment).*

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

- *A URL mapper is typically stored in a file named **urls.py**. In the example below, the mapper (`urlpatterns`) defines a list of mappings between routes (specific URL patterns) and corresponding view functions. If an HTTP Request is received that has a URL matching a specified pattern, then the associated view function will be called and passed the request.*

```
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 (Python lists are defined using square brackets, where items are separated by commas and may have an optional trailing comma. For example: `[item1, item2, item3,]`).

The first argument to both methods is a route (pattern) that will be matched. The `path()` method uses angle brackets to define parts of a URL that will be captured and passed through to the view function as named arguments. The `re_path()` function uses a flexible pattern matching approach known as a regular expression. We'll talk about these in a later article!

The second argument is another function that will be called when the pattern is matched. The notation `views.book_detail` indicates that the function is

called `book_detail()` *and can be found in a module called* `views` *(i.e. inside a file named* `views.py`*)*

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.

The example below shows a minimal view function `index()`*, which could have been called by our URL mapper in the previous section. Like all view functions it receives an* `HttpRequest` *object as a parameter (*`request`*) and returns an* `HttpResponse` *object. In this case we don't do anything with the request, and our response returns a hard-coded string. We'll show you a request that does something more interesting in a later section.*

```
# 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!')
```

Views are usually stored in a file called ***views.py****.*

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 and possibly also 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 — you can choose one of several as part of your project settings. Once you've chosen what database you want to use, you don't need to talk to it directly at all — you just write your model structure and other code, and Django handles all the "dirty work" of communicating with the database for you.

The code snippet below shows a very simple Django model for a `Team` object. The `Team` class is derived from the django class `models.Model`. It defines the team name and team level as character fields and specifies a maximum number of characters to be stored for each record. The `team_level` can be one of several values, so we define it as a choice field and provide a mapping between choices to be displayed and data to be stored, along with a default value.

```
# 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 associated 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 (for example, you can specify a search on U11 teams that have a team name that starts with "Fr" or ends with "al").

The code snippet shows a view function (resource handler) for displaying all of our U09 teams. The line in bold shows how we can use the model query API to filter for all records where the `team_level` field has exactly the text 'U09' (note how this criteria is passed to the `filter()` function as an argument, with the field name and match type separated by a double underscore: ***team_level__exact****).*

```
## 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)

This function uses the `render()` function to create the `HttpResponse` that is sent back to the browser. This function is a shortcut; it creates an HTML file by combining a specified HTML template and some data to insert in the template (provided in the variable named "`context`"). In the next section we show how the template has the data inserted in it to create the HTML.

Rendering data (HTML templates)

Template systems allow you 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 (it can also be made to support other systems if needed).

The code snippet shows what the HTML template called by the `render()` function in the previous section might look like. This template has been written under the assumption that it will have access to a list variable called `youngest_teams` when it is rendered (this is contained in the `context` variable inside the `render()` function above). Inside the HTML skeleton we have an expression that first checks if the `youngest_teams` variable exists, and then iterates it in a `for` loop. On each iteration the template displays each team's `team_name` value in an `<li>` element.

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?

The preceding sections show the main features that you'll use in almost every web application: URL mapping, views, models and templates. Just a few of the other things provided by Django include:

- ***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**: Creating content dynamically is much more computationally intensive (and slow) than serving static content. Django provides flexible caching so that you 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 when you create an app using the basic skeleton. It makes it trivially easy to provide an admin page for site administrators to create, edit, and view any data models in your site.*

- ***Serialising data****: Django makes it easy to serialise and serve your data as XML or JSON. This can be useful when creating a web service (a website that purely serves data to be consumed by other applications or sites, and doesn't display anything itself), or when creating a website in which the client-side code handles all the rendering of data.*