Introduction to Django

The full-featured web application framework for Python

Course setup

- Python 3.6 or higher installed (I'll be using 3.8)
  - Latest version: https://www.python.org/downloads/
- Course material downloaded and unzipped
  - https://github.com/ariannedee/intro-to-django
- Resources downloaded (PDF slides and course reference sheet)
- An IDE for Python
  - I'll be using PyCharm Community edition (free)
  - For work, I use the Professional edition (30-day free trial)
  - VS Code is okay

Today's Schedule

- First 2 hours:
  - Introductions
  - Overview of Django
  - Starting a new project from scratch
  - Q&A
- Last 2 hours:
  - Cover various topics based on class interest
  - I have slides for:
    - Settings, models, views, URLs, templates, Admin, and forms

Questions and breaks

- Use group chat throughout class
  - Only ask questions relevant to current discussion
  - If it's too specific or if I need to do research, put in the Q&A
  - Anyone can answer

- 3 Breaks (10 mins each)
  - I'll answer questions privately via the Q&A feature
  - Ask general or more in-depth questions

- Email more in-depth questions at arianne.dee.studios@gmail.com
- Post discussions on my course Discussion page on GitHub

Warning! There is a lot of info in these slides

- Since this is my first time trying to teach Django, I put more info here than is necessary
- I will skip to different slides depending on what questions attendees have
- Slides can be used as future reference, since we won't have time to discuss everything

Poll (multi-choice)

- How familiar are you with writing web applications?
  - Not at all
  - Some Flask or other Python framework experience
  - Done a Django tutorial
  - Worked on a hobby Django project
  - Worked on a professional Django project
  - Experience with another framework (e.g. Rails, .NET, etc)

Poll (multiple choice)

- What get out of this class?
  - Get a high-level understanding of Django
  - Understand why and when to use Django
  - Learn how to start a basic Django project
  - Get help getting through the tutorial
  - Get help moving beyond the tutorial
  - Get some specific questions answered
  - Get up to speed for work on a Django project

Poll (multiple choice)

- Which Python/programming skills are you comfortable with?
  - Functions, lists, and dictionaries
  - Creating classes
  - Inheriting classes
  - Decorators
  - Using external libraries
  - Working with databases
  - HTML and CSS
  - HTTP requests

Introduction to **Django**

The web framework for perfectionists with deadlines.

Meet Django

- Build features fast
- Security built-in
- Scales in size well
- Built in Python

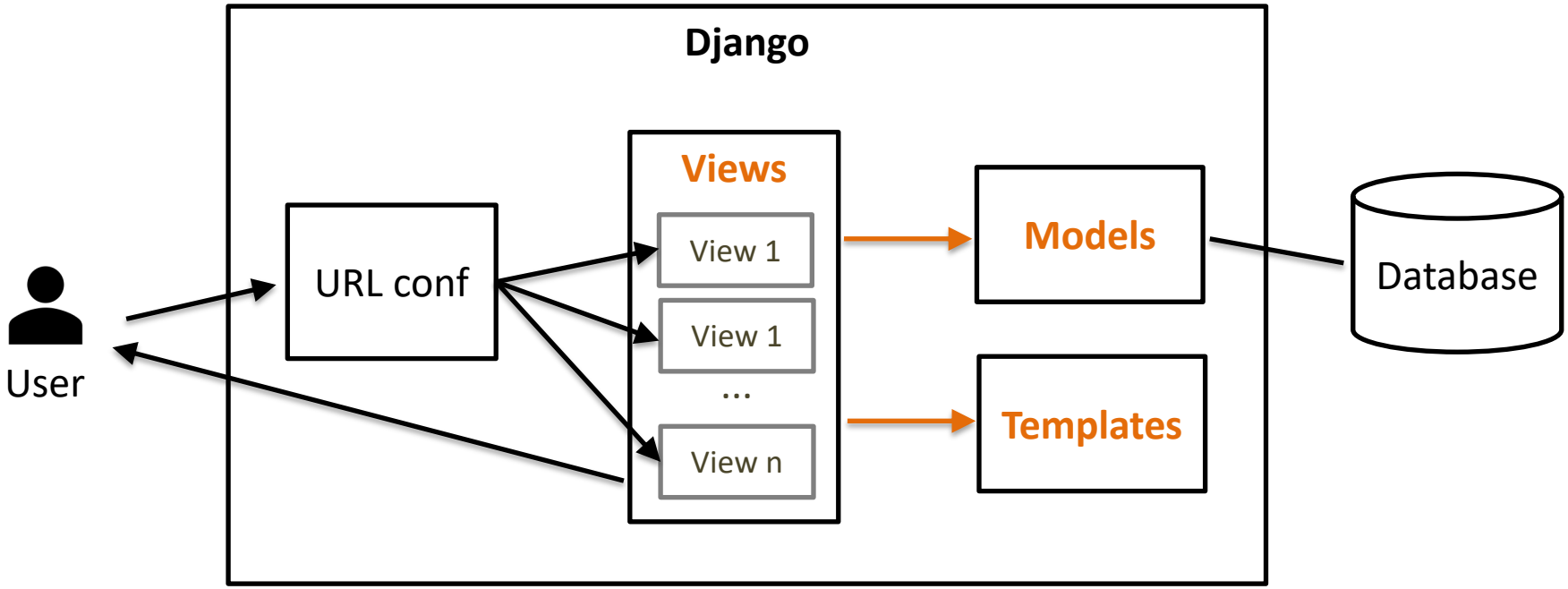
History of Django

- Started in 2003 by Adrian Holovaty and Simon Willison (and later, Jacob Kaplan Moss) at a newspaper company in Kansas
- Released as open-source in 2005, and named after the jazz guitarist, Django Reinhardt
- Built to create things fast, under deadlines, with many non-developers, like content creators
- Read more at Django Design Patterns – The Story of Django

Overall architecture

- Uses an MVT pattern:
  - **Models** – Interact with databases via an ORM
  - **Views** – Handle HTTP requests and return responses
  - **Templates** – Create dynamic HTML pages from Python data

Django architecture overview



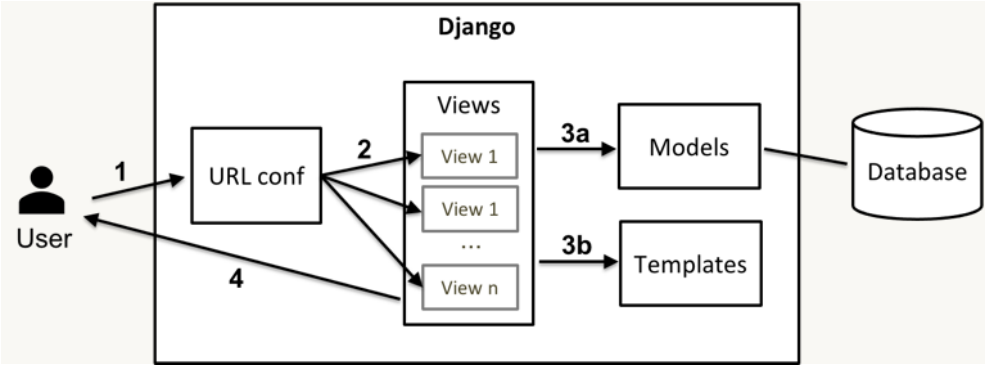
Django architecture

1. User sends an HTTP **request** to Django

2. **URL conf**iguration, contained in urls.py, selects a View to handle the request

3. The **View** gets the request and
   a) Talks to a database via the **Models**
   b) Renders an HTML **Template**

4. The View returns an **HttpResponse** which gets sent to the client to be rendered as a web page in the browser



Getting started with Django

Learning resources

Django resources

- Official Django site
  - https://www.djangoproject.com/
- Documentation
  - https://docs.djangoproject.com/
- Official tutorial
  - https://docs.djangoproject.com/en/3.2/intro/

Official tutorial

- Django at a glance
- Quick install guide
- Writing your first Django app, part 1
- Writing your first Django app, part 2
- Writing your first Django app, part 3
- Writing your first Django app, part 4
- Writing your first Django app, part 5
- Writing your first Django app, part 6
- Writing your first Django app, part 7
- Advanced tutorial: How to write reusable apps
- What to read next
- Writing your first patch for Django

- https://docs.djangoproject.com/en/3.2/intro/
- First steps to understanding Django
- Parts 1-3 are essential for beginners

Topic guides

- Managing files
  - Using files in models
  - The `File` object
  - File storage
- Testing in Django
  - Writing and running tests
  - Testing tools
  - Advanced testing topics
- User authentication in Django
  - Overview
  - Installation
  - Usage
- Django's cache framework
  - Setting up the cache
  - The per-site cache
  - The per-view cache

- https://docs.djangoproject.com/en/3.2/topics/
- If you've need help moving past the tutorial

“How-to” guides

- Authentication using `REMOTE_USER`
- Writing custom `django-admin` commands
- Writing custom model fields
- Custom Lookups
- Custom template backend
- Custom template tags and filters
- Writing a custom storage system
- Deploying Django
- Upgrading Django to a newer version
- Error reporting
- Providing initial data for models
- Integrating Django with a legacy database
- Outputting CSV with Django
- Outputting PDFs with Django
- Overriding templates
- Managing static files (e.g. images, JavaScript, CSS)
- Deploying static files
- How to install Django on Windows
- Writing database migrations

- https://docs.djangoproject.com/en/3.2/howto/

- If you need help accomplishing a task

API reference guides

- Databases
- `django-admin` and `manage.py`
- Running management commands from your code
- Django Exceptions
- File handling
- Forms
- Middleware
- Migration Operations
- Models
- Paginator
- Request and response objects

- https://docs.djangoproject.com/en/3.2/ref/

- If you want to know:
  - What a specific feature does
  - What the available features are
  - How to use a feature

Beginner Django resources

- **A Wedge of Django** (book, previously Django Crash Course)
  - By Daniel and Audrey Feldroy
  - feldroy.com/products/a-wedge-of-Django
- **Build a website with Django 3** (book)
  - By Nigel George
  - https://djangobook.com/build-a-website-with-django-3/
- YouTube, online courses, and website tutorials

Intermediate Django resources

- **Mastering Django** (book)
  - By Nigel George
  - https://djangobook.com/mastering-django-2-book/

- **Web Development in Python with Django** (video)
  - Building Backend Web Applications and APIs with Django
  - By Andrew Pinkham
  - https://learning.oreilly.com/videos/web-development-in/9780134659824/

Getting started with Django

Default project setup

Getting started with Django

Before you start

Before you start

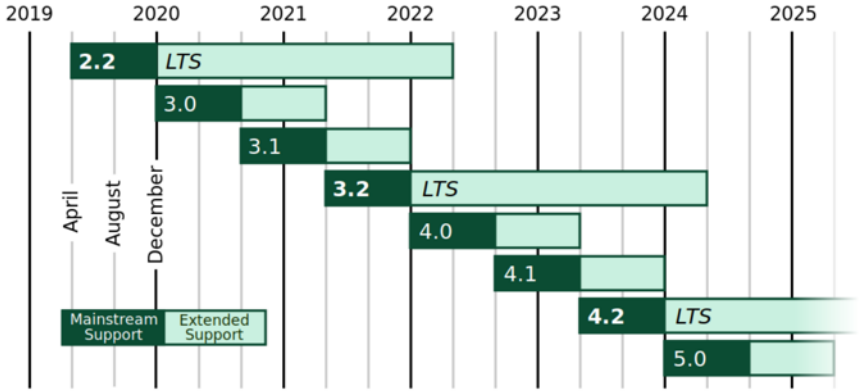
- Choose a Django version
- Choose a compatible Python version
- Create project folder
- Create a virtual environment with Python version

Before you start

- **Choose a Django version**
  - This class: Django 3.2
- Choose a compatible Python version
- Create project folder
- Create a virtual environment with Python version

Choose a Django version

- Use older version if you're running an existing project that hasn't been migrated yet
- 2 is very similar to 3
- 3 can run asynchronous with ASGI
- 3 supports MariaDB
- We're going to use the latest release (3.2)



Before you start

- Choose a Django version
- **Choose a compatible Python version**
  - This class: Python 3.8
- Create project folder
- Create a virtual environment with Python version

Choose a compatible Python version

- I'll be using 3.6 so that all code will be compatible with 3.6 – 3.9

What Python version can I use with Django?

Django version	Python versions
2.2	3.5, 3.6, 3.7, 3.8 (added in 2.2.8), 3.9 (added in 2.2.17)
3.0	3.6, 3.7, 3.8, 3.9 (added in 3.0.11)
3.1	3.6, 3.7, 3.8, 3.9 (added in 3.1.3)
3.2	3.6, 3.7, 3.8, 3.9

Version	Latest micro version	Release date	End of full support	End of security fixes
3.10		2021-10-04[62]	2023-05[62]	2026-10[62]
Italic is the latest micro version of currently supported versions as of 2021-05-11.				
3.9	*3.9.5*[60]	2020-10-05[60]	**2022-05**[61]	**2025-10**[60][61]
3.8	*3.8.10*[59]	2019-10-14[59]	2021-05-03[59]	2024-10[59]
3.7	*3.7.10*[58]	2018-06-27[58]	2020-06-27[b][58]	2023-06[58]
3.6	*3.6.13*[57]	2016-12-23[57]	2018-12-24[b][57]	2021-12[57]
3.5	3.5.10[55]	2015-09-13[55]	2017-08-08[56]	2020-09-30[55]
3.4	3.4.10[53]	2014-03-16[53]	2017-08-09[54]	2019-03-18[a][53]
3.3	3.3.7[52]	2012-09-29[52]	2014-03-08[b][52]	2017-09-29[52]
3.2	3.2.6[51]	2011-02-20[51]	2013-05-13[b][51]	2016-02-20[51]
2.7	2.7.18[31]	2010-07-03[31]	2020-01-01[c][31]	

Python version support timeline

Before you start

- Choose a Django version
- Choose a compatible Python version
- **Create project folder**
  - This class: intro-to-django folder that you got from GitHub
- Create a virtual environment with Python version

Before you start

- Choose a Django version
- Choose a compatible Python version
- Create project folder
- **Create a virtual environment with Python version**
  - Optional for this course

Create a virtual environment with Python

You should have already done this before class

Example for Linux/Mac:

```
$ cd Documents/intro-to-django
```

- Navigate into project folder

```
$ python3.8 -m venv venv
```

- Create a new virtual environment

```
$ source venv/bin/activate
```

- Activate virtual environment

Why create a virtual environment?

- You can have multiple Python versions on your computer
- Multiple projects that require different versions of the same external package

- Virtual environments create a "workspace" environment for a specific project
  - All installed apps are in one place, without conflicts
  - The "python" and "pip" commands point to the same place
  - Keeps your environment clean

Create a virtual environment with Python

Unfamiliar with *venv* or the command line?

Next Level Python LiveLessons

Lesson 3.3 - Create virtual environments using *venv*

https://learning.oreilly.com/videos/next-level-python/9780136904083/9780136904083-NLP1_01_03_03/

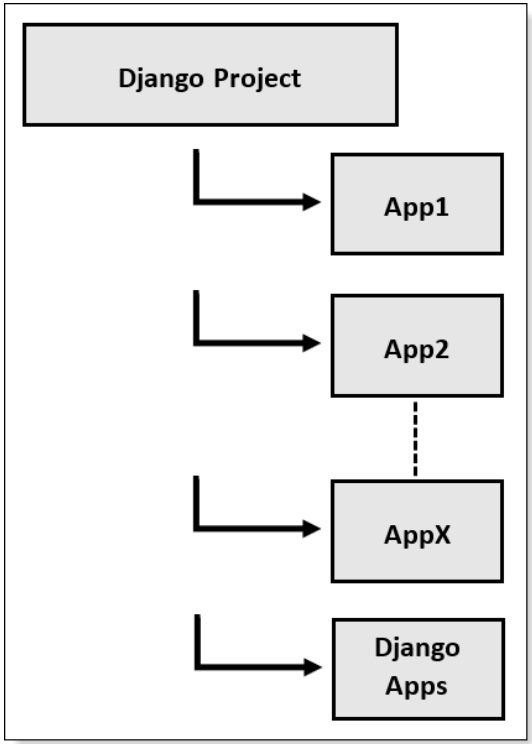
Lesson 3.1 - Work with the command line

https://learning.oreilly.com/videos/next-level-python/9780136904083/9780136904083-NLP1_01_03_01/

Getting started with Django

Create a project

Django project & apps



Create a Django project

1. Install Django using pip
2. Create a new project using **`django-admin startproject`**
3. Run the development server
4. Initialize the development database
5. Create a superuser
6. View the admin site

Create a Django project

1. **Install Django using pip**
2. Create a new project using **`django-admin startproject`**
3. Run the development server
4. Initialize the development database
5. Create a superuser
6. View the admin site

Install Django using pip

You should have already done this before class.

In your activated virtual environment, use one of the following:

```
$ pip install django
```
Latest version of Django (any version)

```
$ pip install "django>=3.0,<4”
```
Latest version of Django 3

```
$ pip install "django==3.2.4”
```
Specific version of Django

I also have a *requirements.txt* file you can install from

```
$ pip install –r requirements.txt
```

Install Django using pip

More instructions for installing packages using pip

Next Level Python LiveLessons

Lesson 3.2 - Install external libraries using pip

https://learning.oreilly.com/videos/next-level-python/9780136904083/9780136904083-NLP1_01_03_02/

Create a Django project

1. Install Django using pip
2. **Create a new project using `django-admin startproject`**
3. Run the development server
4. Initialize the development database
5. Create a superuser
6. View the admin site

Create a new Django project

```
$ django-admin startproject <project_name>
```

This will create a new folder with *project_name* and the following structure (if project name is **dj_test**)

- dj_test
  - manage.py
  - dj_test
    - __init__.py
    - settings.py
    - urls.py
    - wsgi.py
    - asgi.py (only in Django 3+)

Create a new Django project

- test_site
  - **manage.py** ← This is the file you'll be running for most tasks
  - test_site
    - __init__.py
    - settings.py
    - urls.py
    - wsgi.py
    - asgi.py (only in Django 3+)

Before we continue, navigate into the first **dj_test** folder in your terminal

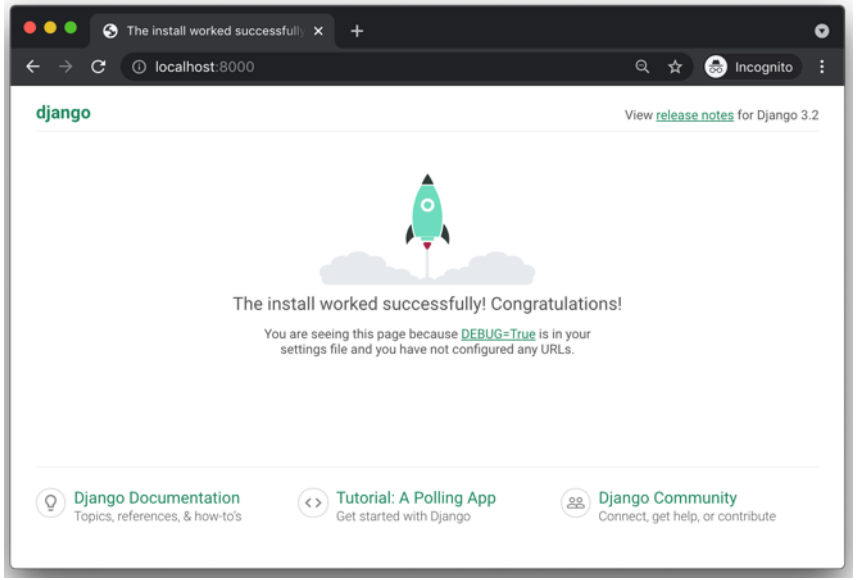
Create a Django project

1. Install Django using pip
2. Create a new project using `django-admin startproject`
3. **Run the development server**
4. Initialize the development database
5. Create a superuser
6. View the admin site

Run the development server

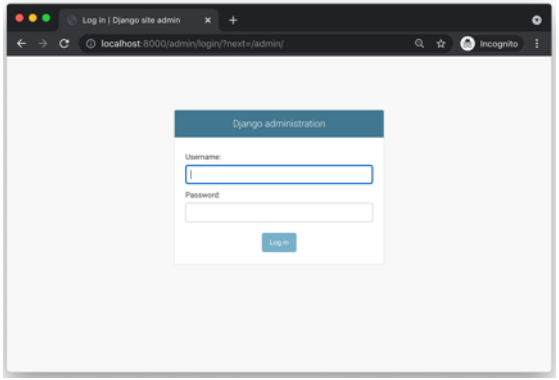
```
$ python manage.py runserver
```

Open http://127.0.0.1:8000/ or http://localhost:8000/ in your browser

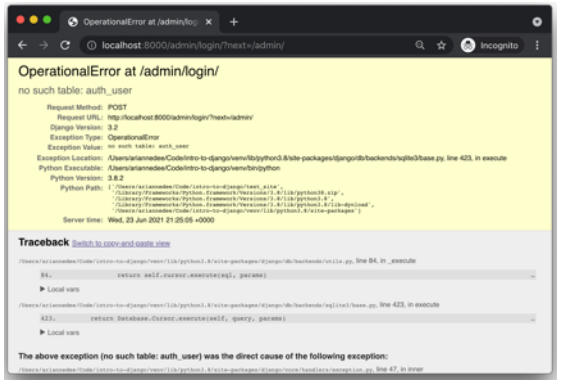


Run the development server

Now open http://127.0.0.1:8000/admin



If you try to log in with any username/password, you'll get a server error



Create a Django project

1. Install Django using pip
2. Create a new project using `django-admin startproject`
3. Run the development server
4. **Initialize the development database**
5. Create a superuser
6. View the admin site

Initialize the development database

```
$ python manage.py migrate
```

This will initialize a SQLite database (**db.sqlite3**) with the tables specified in the default project:

- authorization tables (user, permissions, groups)
- django_migrations
- django_admin_log
- django_sessions
- django_content_type

Initialize the development database

You can edit the database type and default tables by editing the **settings.py** file

- **django_migrations**
  - required for Django's ORM feature
- **authorization tables** (user, permissions, groups)
  - default user and permissions features
- **django_admin_log**
  - tracks changes made via the Django Admin
- **django_sessions**
  - used in the sessions app (e.g. cookies)
- **django_content_type**
  - used for generic model relations (like tags, comments, likes, etc)

View the data in the database

- **Command line interface**
- **An app**
  - e.g. DB Browser for SQLite or SQLiteStudio
- **A website**
  - e.g. https://inloop.github.io/sqlite-viewer/
- **PyCharm Professional** (built-in)
- **PyCharm Community**
  - plugin: Database Navigator
- **VS Code extension**
  - plugin: SQLite

Create a Django project

1. Install Django using pip
2. Create a new project using `django-admin startproject`
3. Run the development server
4. Initialize the development database
5. **Create a superuser**
6. View the admin site

Create a superuser

```
$ python manage.py createsuperuser
```

Creates your first user so that you can log into the admin site

You supply:

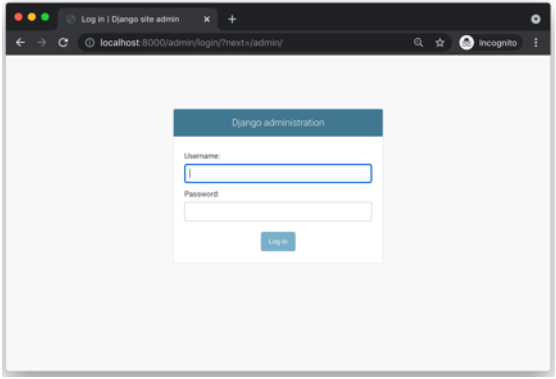
- username
- email address
- password
  - You can bypass the password requirements

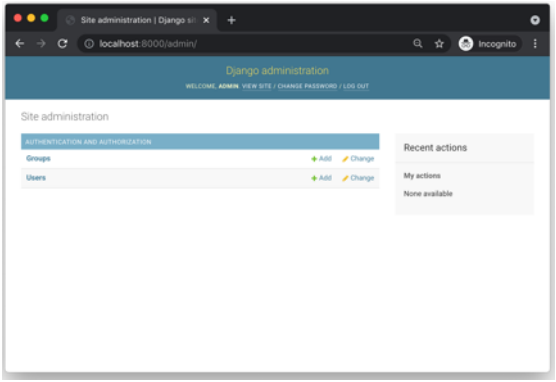
Create a Django project

1. Install Django using pip
2. Create a new project using `django-admin startproject`
3. Run the development server
4. Initialize the development database
5. Create a superuser
6. **View the admin site**

View the admin site

- Make sure your server is running
  - `$ python manage.py runserver`
- Open http://127.0.0.1:8000/admin
- Log in using your superuser credentials
- You can now browse, create, edit, and delete your Django models
  - only users and groups are configured initially
  - We'll add more later





Initial project structure

- dj_test
  - manage.py
  - dj_test
    - __init__.py
    - settings.py
    - urls.py
    - wsgi.py
    - asgi.py (only in Django 3+)

Let's look at each of these files in order

Root dj_test folder

- **dj_test** ← Root folder
  - manage.py
  - dj_test
    - __init__.py
    - …

manage.py

- Run this file to run the project
  - **runserver** – Run the development server
  - **migrate** – Apply the migrations to the db
  - **createsuperuser** – Create a super user in the db
  - **shell** – Run a python console (has access to Django project)
  - **makemigrations** – If models changed, create migration file

- Run the file without any arguments to see full list of commands

Docs - https://docs.djangoproject.com/en/3.2/ref/django-admin/

Custom manage.py commands

- You can create custom management commands

Docs

- https://docs.djangoproject.com/en/3.2/howto/custom-management-commands/

Tutorial

- https://simpleisbetterthancomplex.com/tutorial/2018/08/27/how-to-create-custom-django-management-commands.html

dj_test project folder

- dj_test
  - manage.py
  - **dj_test** ← Be careful if you want to rename this
    - __init__.py
    - ...

- This is where the overall configuration of your site goes
- Find all "dj_test" in project to see where it's being used

settings.py

- Configures Django settings for your site

- Some notable ones:
  - DEBUG mode (must turn off in production)
  - Database settings (don't use SQLite in production)
  - Installed apps
  - Timezone
  - Static and media folder locations
  - Password validators

- We'll discuss settings more in the next section

Docs - https://docs.djangoproject.com/en/3.2/topics/settings/

urls.py

Defines the URLs for your site

- For example, in our trivia app:
  - / # index
  - /trivia/question/5/view # view question with id=5
  - /admin/auth/user/1/change/ # change the user with id=1

- Map the URLs to templates, views or other URL sets (confs)

Docs - https://docs.djangoproject.com/en/3.2/topics/http/urls/

urls.py

- In the default urls.py

```
urlpatterns = [
    path('admin/', admin.site.urls),
]
```

- The only URL pattern defined is linking the "admin/" path to the set of admin urls

Try changing the "admin/" path. How does that change the URLs in our existing test project?

wsgi.py and asgi.py

- **WSGI** - Web Server Gateway Interface

- During development, you run your web server locally through
  - **`manage.py runserver`**

- In production, this is insufficient because it:
  - Loads static files slowly
  - Only handles one user at a time
  - Has not been tested for security

- Use WSGI when running in production

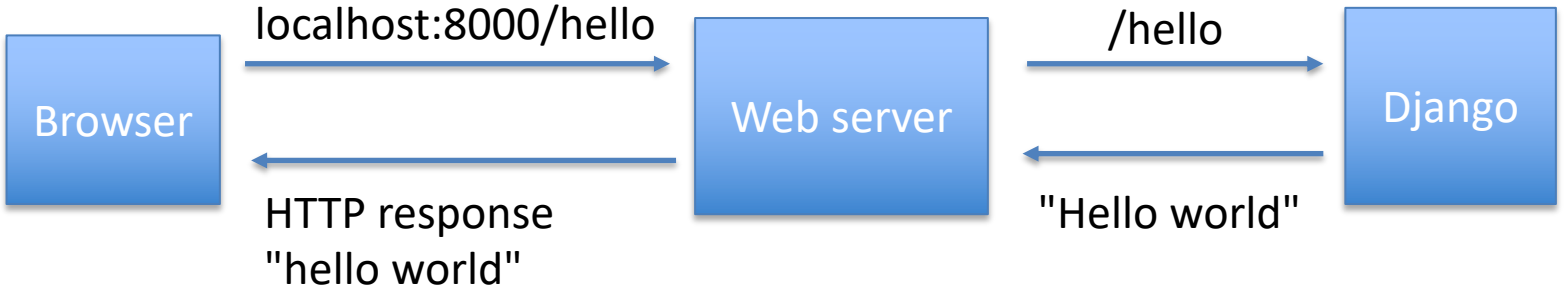
wsgi.py

- WSGI provides an interface for running your Django application in production with a dedicated web server (like nginx or Apache)
- You don't need to know much about web servers and application servers
- Just follow tutorials for setting up Django in production and you might end up needing this file
- Read more

asgi.py

- **ASGI** – Asynchronous Server Gateway Interface
- New in Django 3
- Similar to WSGI but for asynchronous web servers
  - Allows support for websockets, async and HTTP/2
- Overview of WSGI and ASGI in Django

Let's create our own view



Create a simple view

- Create a function that returns an HttpResponse
- Create a new path that returns that function

```
from django.http import HttpResponse

def my_func(request):
    return HttpResponse('Hello, world')

urlpatterns = [
    path('admin/', admin.site.urls),
    path('hello', my_func),
]
```

Create a simple view

- Create a function that returns an HttpResponse
- Create a new path that returns that function

```
from django.http import HttpResponse

def my_func(request):
    return HttpResponse('Hello, world')

urlpatterns = [
    path('admin/', admin.site.urls),
    path('hello', my_func),
]
```

Don't call the function!

Create a simple view

- Let's try returning some HTML in the HttpResponse

```
def my_func(request):
    return HttpResponse('<h1>Hello</h1><p>world<p>')
```

- We'll look at how to return HTML from templates later

Create a simple view

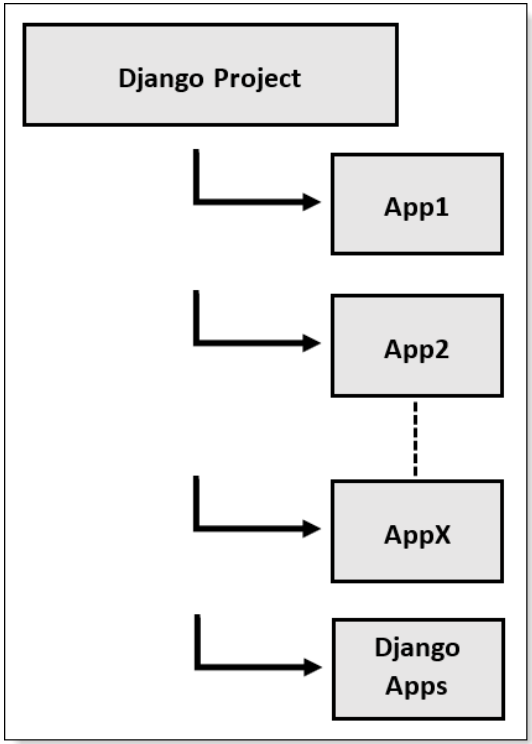
- If we put all of our logic in **urls.py**
  - Very big
  - Hard to separate functionality
  - Can't reuse features in other projects

- Next we'll look at how to do the same thing in a dedicated **app**

Getting started with Django

Create an app

Django project & apps



Possible apps

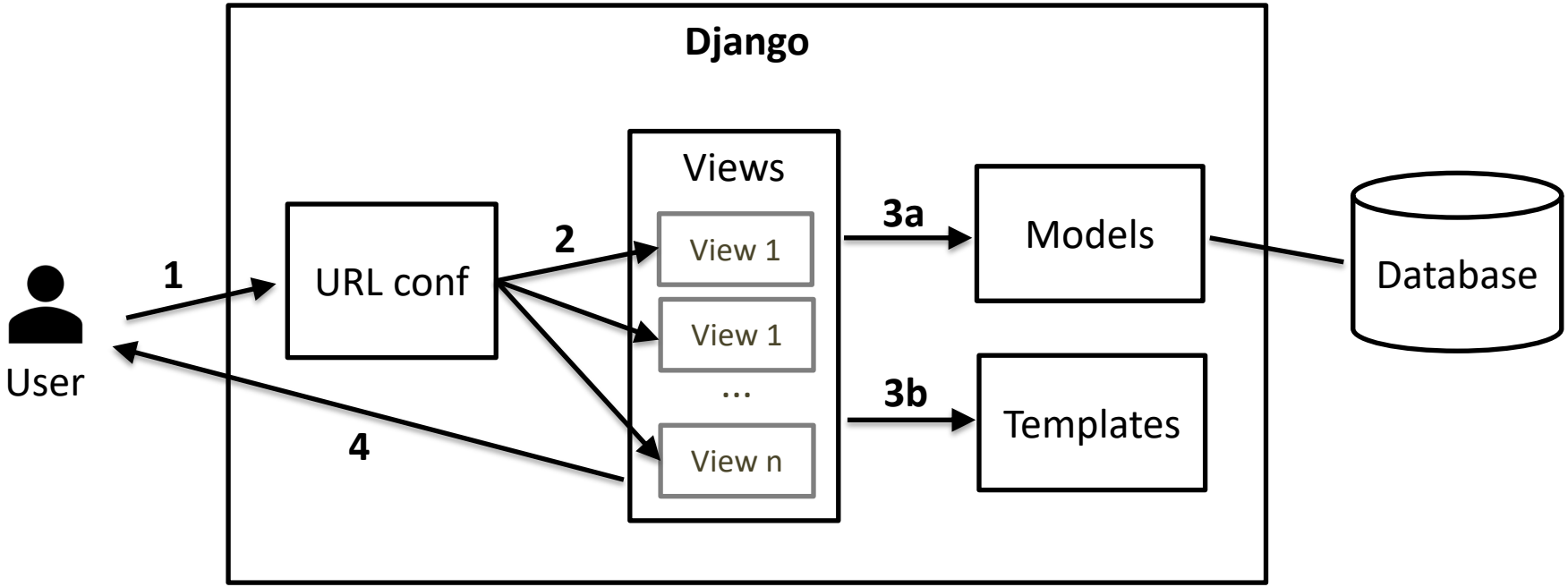
- General
  - Users
  - Blog
  - Events
  - News

- Trivia site specific
  - Questions
  - Categories
  - Leaderboard
  - Tournaments

Django apps

- Admin
- Authorization
- Sessions
- Messages
- + more

Django architecture



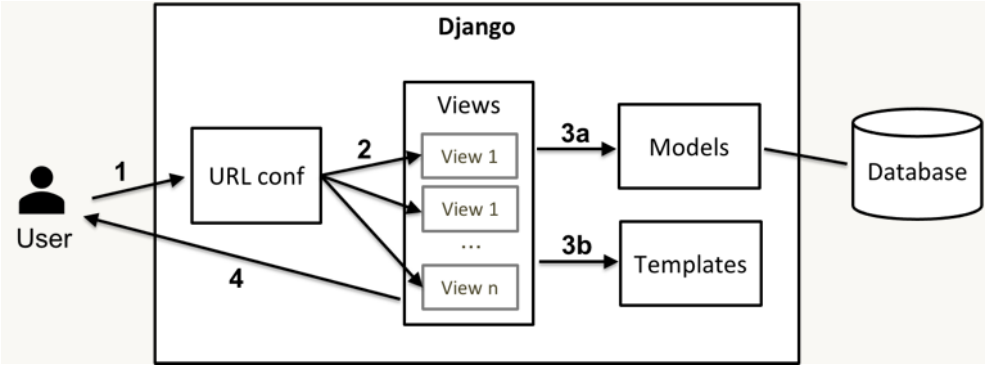
Django architecture

1. User sends an HTTP **request** to Django

2. **URL conf**iguration, contained in urls.py, selects a View to handle the request

3. The **View** gets the request and
   a) Talks to a database via the **Models**
   b) Renders an HTML **Template**

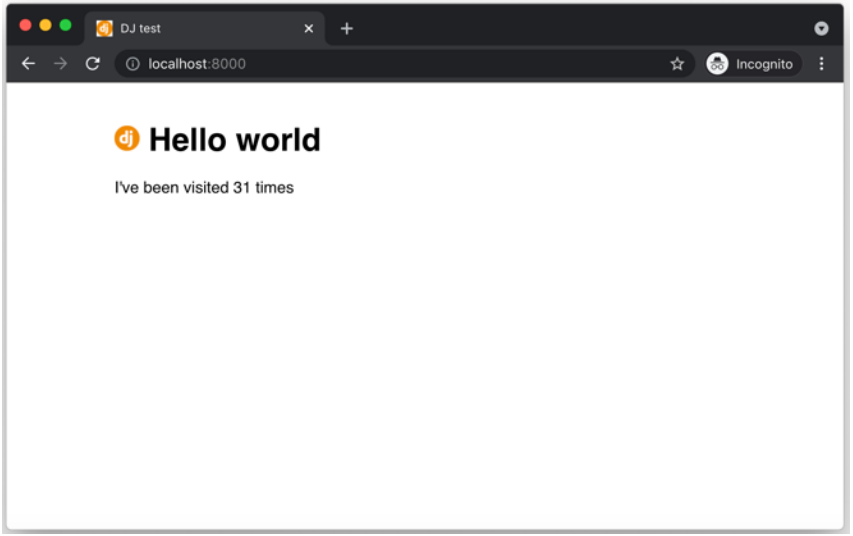
4. The View returns an **HttpResponse** which gets sent to the client to be rendered as a web page in the browser



Create a Django app

1. Create a new app using **`startapp`**
2. Add it to the **settings**
3. Create a **URL**
4. Link it to a **view**
5. Return a **template**
6. Make it dynamic with **context**
7. Use data from a **model**
8. Add some **static** files

Our simple app



- Visits app
- Tracks how many times your site has been visited

Create a Django app

1. **Create a new app using `startapp`**
2. Add it to the **settings**
3. Create a **URL**
4. Link it to a **view**
5. Return a **template**
6. Make it dynamic with **context**
7. Use data from a **model**
8. Add some **static** files

Create a new app using `startapp`

```
$ manage.py startapp visits
```

OR

```
$ django-admin startapp visits
```

- visits
  - __init__.py
  - admin.py
  - apps.py
  - models.py
  - tests.py
  - views.py
  - migrations
    - __init__.py

Create a Django app

1. Create a new app using **`startapp`**
2. **Add it to the settings**
3. Create a **URL**
4. Link it to a **view**
5. Return a **template**
6. Make it dynamic with **context**
7. Use data from a **model**
8. Add some **static** files

Add it to the settings

- In **settings.py:**

```
INSTALLED_APPS = [
    'visits.apps.VisitsConfig',
    'django.contrib.admin',
    ...
]
```

- Django uses **INSTALLED_APPS** as a list of places to look for models, management commands, tests, and other utilities

Create a Django app

1. Create a new app using **`startapp`**
2. Add it to the **settings**
3. **Create a URL**
4. Link it to a **view**
5. Return a **template**
6. Make it dynamic with **context**
7. Use data from a **model**
8. Add some **static** files

Create a URL – pt. 1

- Create a new file called "**urls.py**"

```
from django.urls import path
from . import views

app_name = 'visits'
urlpatterns = [
    path('', views.index, name='index'),
]
```

Create a URL – pt. 1

- Create a new file called "**urls.py**"

```
from django.urls import path
from . import views

app_name = 'visits'
urlpatterns = [
    path('', views.index, name='index'),
]
```

Relative import

Create a URL – pt. 2

- Connect it to **dj_test.urls.py**

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('visits.urls'))
]
```

Create a Django app

1. Create a new app using **`startapp`**
2. Add it to the **settings**
3. Create a **URL**
4. **Link it to a view**
5. Return a **template**
6. Make it dynamic with **context**
7. Use data from a **model**
8. Add some **static** files

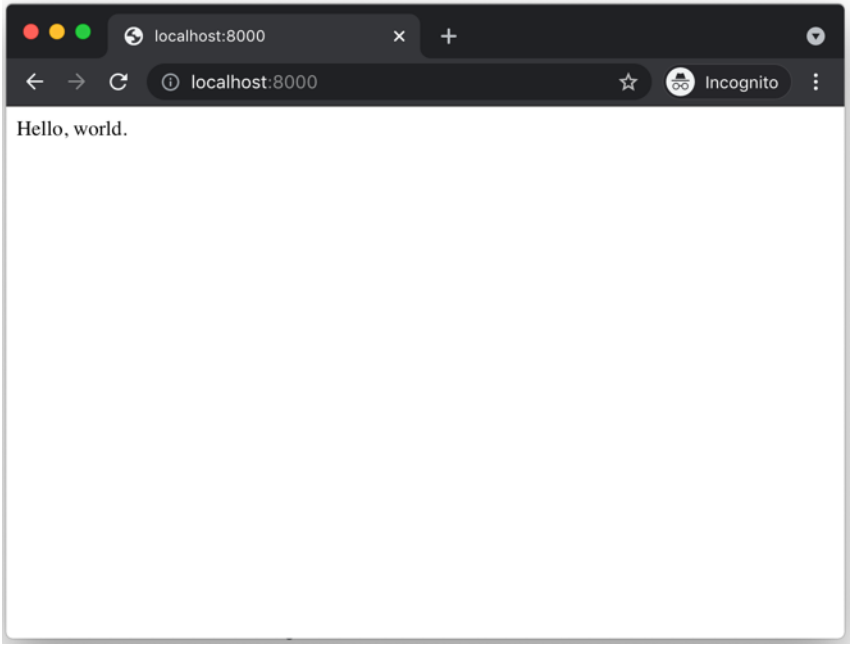
Link it to a view

- In **views.py**

```
from django.http import HttpResponse


def index(request):
    return HttpResponse("Hello, world.")
```

Link it to a view



- We now have a simple index page

Create a Django app

1. Create a new app using **`startapp`**
2. Add it to the **settings**
3. Create a **URL**
4. Link it to a **view**
5. **Return a template**
6. Make it dynamic with **context**
7. Use data from a **model**
8. Add some **static** files

Return a template – pt. 1

- Create a new folder called **templates** in your **visits** folder
- Create a new file in **templates** called **index.html**
- Add some html (or copy it from *sample_files/index1.html*)

```
<html>
  <head>
    <title>My Django Project</title>
  </head>
  <body>
    <h1>Hello world</h1>
    <p>I've been visited 1 time</p>
  </body>
</html>
```

Return a template – pt. 2

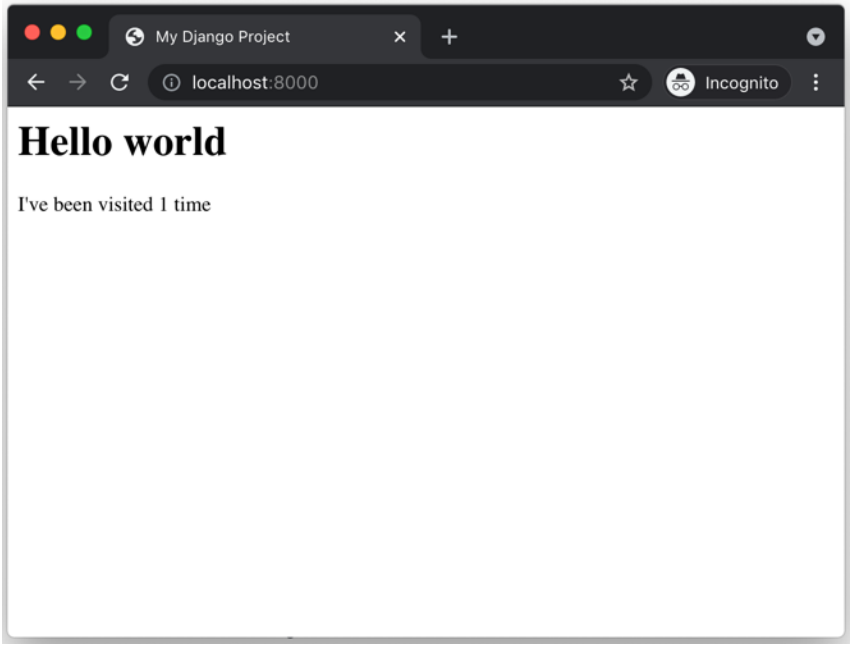
- Render the template from your view in **view.py**

```
from django.shortcuts import render

def index(request):
    return render(request, 'index.html')
```

- Render is a django "shortcut" that:
  1. Retrieves the template
  2. Renders it to an HTML file based on data that you pass in (none yet)
  3. Returns it as an HttpResponse

Return a template



- This is just an HTML page until we start adding some dynamic content

Create a Django app

1. Create a new app using **`startapp`**
2. Add it to the **settings**
3. Create a **URL**
4. Link it to a **view**
5. Return a **template**
6. **Make it dynamic with context**
7. Use data from a **model**
8. Add some **static** files

Make it dynamic – pt. 1

- Send some data to the template from **views.py**

```
from random import randint
from django.shortcuts import render

def index(request):
    context = {"num_visits": randint(1, 5)}
    return render(request, 'index.html', context=context)
```

- Context is a dictionary that gets passed to the template
- The template can then use that data when rendering to HTML

Make it dynamic – pt. 2

- Update your **index.html** template to use that data

```
<html>
  <head>
    <title>My Django Project</title>
  </head>
  <body>
    <h1>Hello world</h1>
    <p>
      I've been visited {{num_visits}} time{{num_visits|pluralize}}
    </p>
  </body>
</html>
```

Make it dynamic – pt. 2

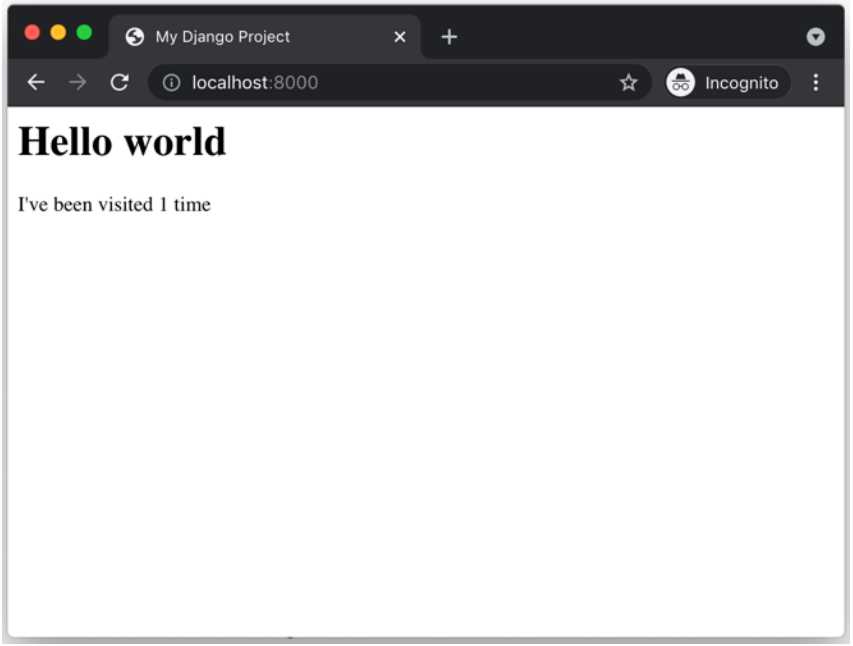
```
<body>
  <h1>Hello world</h1>
  <p>
    I've been visited {{num_visits}} time{{num_visits|pluralize}}
  </p>
</body>
```

The Django Template Language

- The **{{ var }}** syntax is a "**variable**" that gets evaluated (from the context) and is replaced when the template is rendered

- The **{{ var|filter }}** syntax is a "**filter**"
- The **pluralize** filter returns an "**s**" if the variable is greater than 1

Make it dynamic



- Every time you refresh the page, you will get a random number of visits
- "time" should be pluralized if num_visits is more than 1

Create a Django app

1. Create a new app using **`startapp`**
2. Add it to the **settings**
3. Create a **URL**
4. Link it to a **view**
5. Return a **template**
6. Make it dynamic with **context**
7. **Use data from a model**
8. Add some **static** files

Use data from a model – pt. 1

- In **models.py**, create a new model to store a visit to the site

```
from django.db import models


class Visits(models.Model):
    count = models.IntegerField(default=0)
```

Use data from a model – pt. 2

- Create a migration file
  - **`$ python manage.py makemigrations`**
  - This will create a new file 0001_initial.py in the migrations folder

- Migrate the database

  `$ python manage.py migrate`

Use data from a model – pt. 3

- Create your initial **Visits** object
  - Open up a Django console in your terminal
    - **$ python manage.py shell**
  - Import your Visits model
    - `>>> from visits.models import Visits`
  - Create a Visits object
    - `>>> v = Visits.objects.create()`
  - Check the current count value
    - `>>> v.count  # 0`

Use data from a model – pt. 4

- Update **views.py** so that
  - **visit.count** is updated every page view
  - it is sent to the context as **num_visits**

```
from django.shortcuts import render
from .models import Visits

def index(request):
    v = Visits.objects.first()
    v.count += 1
    v.save()
    context = {"num_visits": v.count}
    return render(request, 'index.html', context=context)
```

Use data from a model – pt. 4

- Update **views.py** so that
  - **visit.count** is updated every page view
  - it is sent to the context as **num_visits**

```
from django.shortcuts import render
from .models import Visits
```

Relative import

```
def index(request):
    v = Visits.objects.first()
    v.count += 1
    v.save()
    context = {"num_visits": v.count}
    return render(request, 'index.html', context=context)
```

More model options

1. Add a field to Visits that tracks the page that was viewed

2. Use a different Visit model that tracks individual page views
   - User who viewed (if logged in)
   - Date and time viewed
   - Page viewed

3. Keep track of user's IP address and location

Create a Django app

1. Create a new app using **`startapp`**
2. Add it to the **settings**
3. Create a **URL**
4. Link it to a **view**
5. Return a **template**
6. Make it dynamic with **context**
7. Use data from a **model**
8. **Add some static files**

Add some static files – pt. 1

- Create a new folder called **static** in your **visits** folder
- Copy **styles.css**, **logo.png**, and **favicon.ico** from ***sample_files*** to the new ***static*** folder

```
body {
  padding: 30px 100px;
  font-family: sans-serif;
}

img {
  width: 25px;
}
```

Add some static files – pt. 2

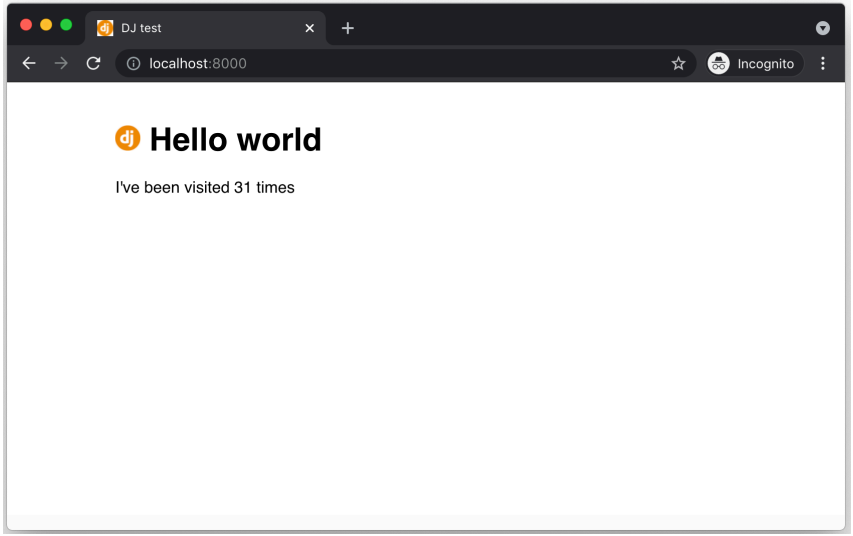
- Update your **index.html**, or copy it from **sample_files/index2.html**

```
{% load static %}
<html>
<head>
  <title>My Django Project</title>
  <link rel="icon" href="{% static 'favicon.ico' %}">
  <link rel="stylesheet" type="text/css" href="{% static 'styles.css' %}">
</head>
<body>
  <h1>
    <img src="{% static 'logo.png' %}"> Hello world
  </h1>
  <p>I've been visited {{ num_visits }} time{{ num_visits|pluralize }}</p>
</body>
</html>
```

Add some static files

- Restart the development server and refresh your browser

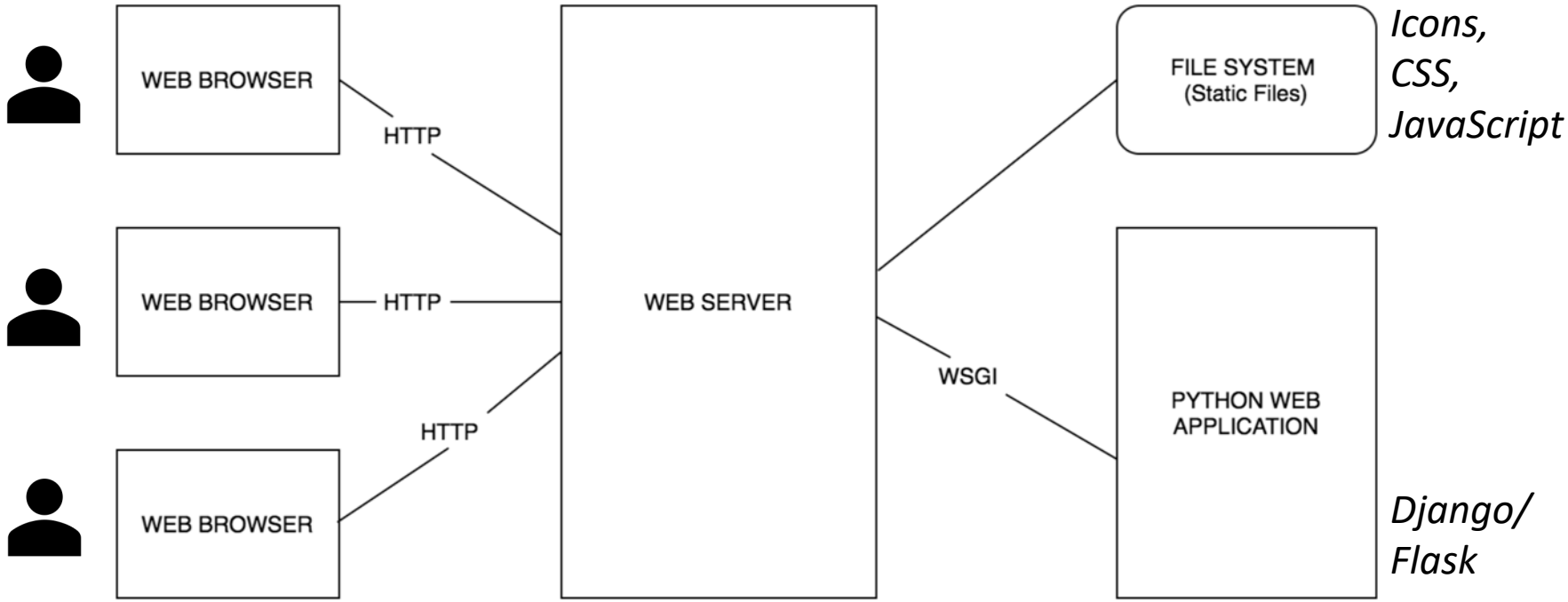
- You should now see slightly nicer style, a logo in the header, and an icon in the tab



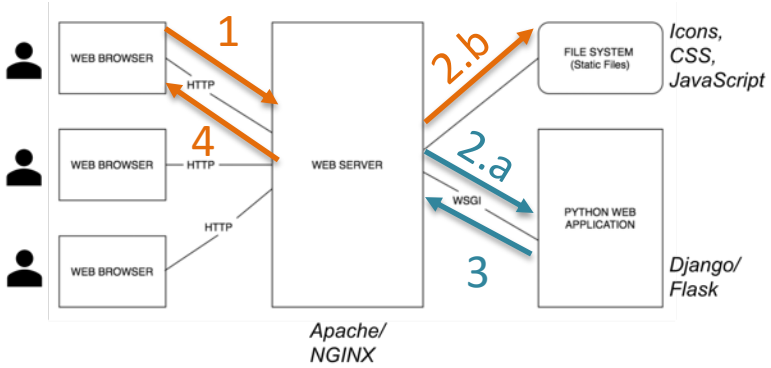
Web application architecture overview

How does Django work?

Web application overview



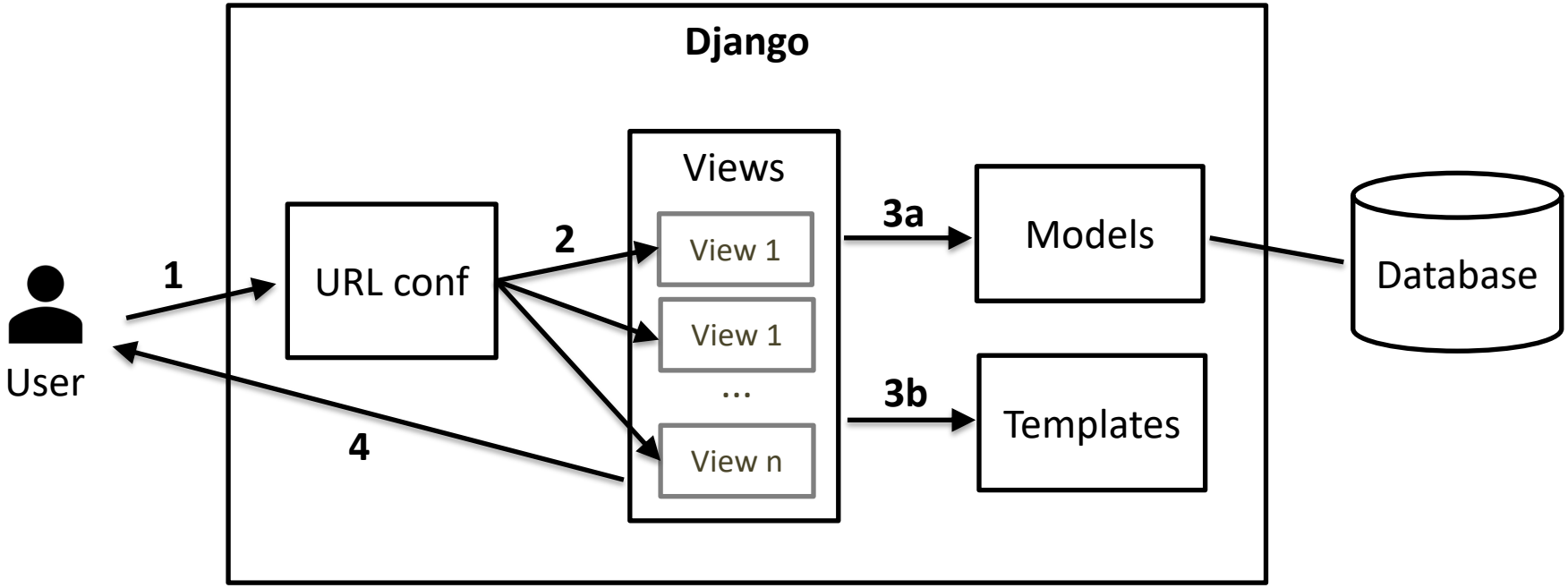
Web application overview



We mostly care about 2.a and 3

1. Browser sends an **HTTP** request to your web server
2. Web server sends the request to
   a) a Python app server via WSGI
   b) the file system to get a file
3. The application returns an HTTP response to the web server
4. A rendered web page is seen in the browser

Django architecture



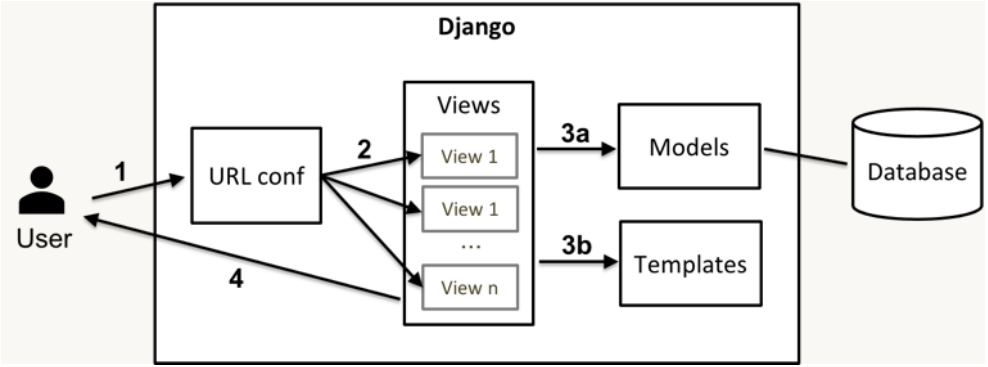
Django architecture

1. User sends an HTTP **request** to Django

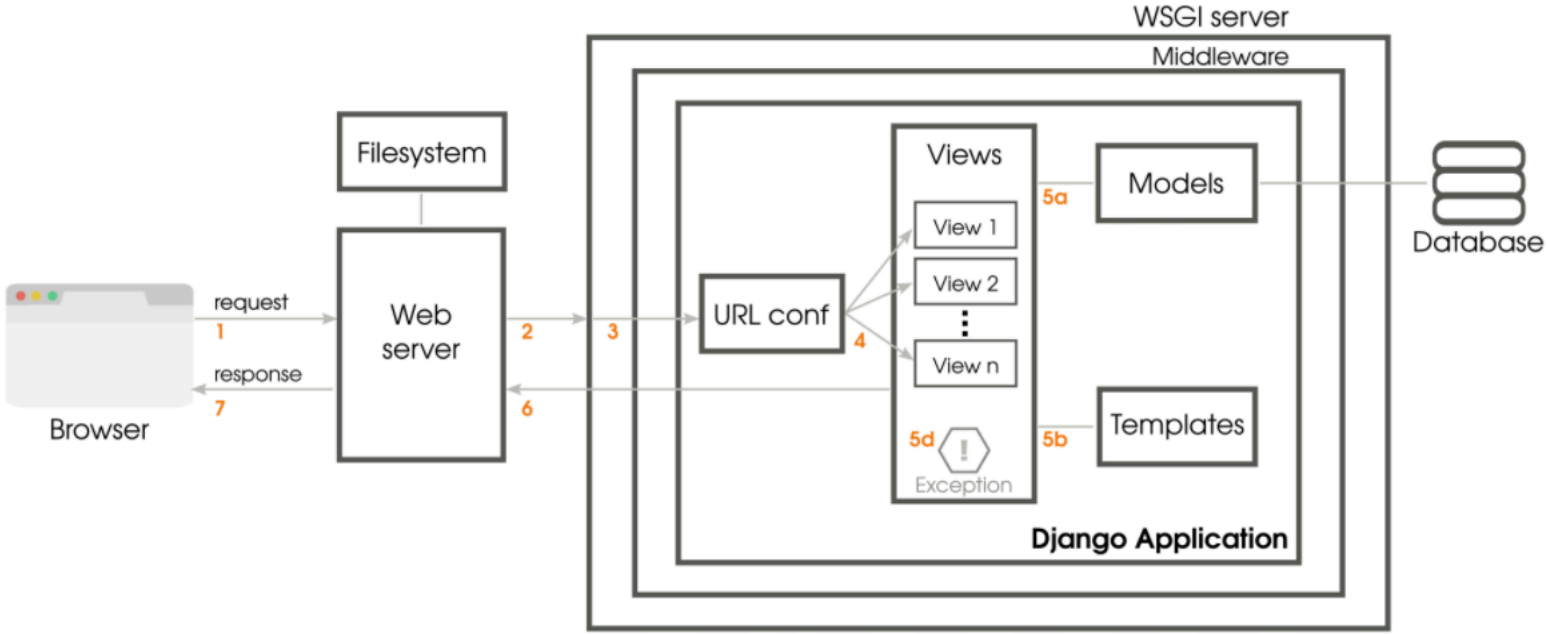
2. **URL conf**iguration, contained in urls.py, selects a View to handle the request

3. The **View** gets the request and
   a) Talks to a database via the **Models**
   b) Renders an HTML **Template**

4. The View returns an **HttpResponse** which gets sent to the client to be rendered as a web page in the browser



Overall architecture



Overall architecture

1. The browser sends the request (essentially, a string of bytes) to your web server.
2. Your web server (say, Nginx) hands over the request to a Web Server Gateway Interface (WSGI) server (say, uWSGI) or directly serves a file (say, a CSS file) from the filesystem.
3. Unlike a web server, WSGI servers can run Python applications. The request populates a Python dictionary called environ and, optionally, passes through several layers of middleware, ultimately reaching your Django application.
4. URLconf (URL configuration) module contained in the urls.py of your project selects a view to handle the request based on the requested URL. The request has turned into HttpRequest, a Python object.
5. The selected view typically does one or more of the following things:
   a) Talks to a database via the models
   b) Renders HTML or any other formatted response using templates
   c) Returns a plain text response (not shown)
   d) Raises an exception
6. The HttpResponse object gets rendered into a string, as it leaves the Django application.
7. A beautifully rendered web page is seen in your user's browser.

Excerpt from Django Design Patterns and Best Practices - Arun Ravindran

Main tasks

- Configure **URL** structure
- Create **models** of our data
- Use models to manage the database
- Create HTML **templates** (which can use models and other data)

- Create **views** that can retrieve data from **models** and dynamically create an HTML page from a **template**
- Specify which **URLs** go to which **views**

- Define **settings** for different environments (local, production, etc.)

Other Django features

- Built-in:
  - User authentication and permissions
  - Form creation
  - Admin interface
  - Many security features
  - Sessions/cookies

- Simple to set up:
  - Caching
  - Internationalization / localization
  - Debug toolbar

Security features

- Cross-site scripting (XSS) protection
- Cross-site request forgery (CSRF) protection
- SQL injection protection
- Clickjacking protection
- SSL/HTTPS
- Host header validation

When to use Django

- You want to work in Python
- You want to use its features, like the admin, ORM, authentication, sessions, etc
- You want to develop features quickly
- You need to worry about security and don't have the expertise/time to handle it from scratch
- You are okay working within the MVT architecture and don't need too much customization

Popular Django apps/libraries

- API creation
  - Django REST Framework (DRF) – REST APIs
  - Graphene Django – GraphQL APIs
- Debugging and performance testing
  - Django Debug Toolbar
- Scheduling and asynchronous tasks
  - Celery
- WebSocket support
  - Channels

Deployment options

- **Python Anywhere**
  - No WebSocket, ASGI, Celery queues, NoSQL, or Docker support
  - Postgres only on paid custom account
  - Needs virtualenvwrapper

- **Heroku**
  - Can't serve media files (use Amazon S3)
  - Most expensive to scale
  - Less direct server control

- **Digital Ocean, Amazon Web Services and Microsoft Azure**
  - More technical, better cost and performance if you need scale

More resources

- How to architect a Django website for the real world?

- Create new Django projects using CookieCutter
  - Docs: https://cookiecutter.readthedocs.io/
  - Templates: https://github.com/search?q=cookiecutter+django

Project structures

Don't just use the default

Alternative project structures

- Most people don't use the default project structure for their projects

- You can rename the root folder without worry
- You can put your apps in an **apps** folder
- You can have a single location for templates and static files

- You can start projects using cookie-cutter

Default project structure

- **dj_test**
  - manage.py
  - **dj_test**
    - templates
    - static
    - settings.py
    - urls.py
  - **users**
    - static
    - templates
  - **visits**
    - static
    - templates

Default project structure

- **dj_test** ← `django-admin startproject dj_test`
  - manage.py
  - **dj_test**
    - templates
    - static
    - settings.py
    - urls.py
  - **users** ← `python manage.py startapp visits`
    - static
    - templates
  - **visits** ← `python manage.py startapp users`
    - static
    - templates

Commonly used structure

- dj_test_root
  - manage.py
  - **dj_test_project**
    - settings.py
    - urls.py
  - apps
    - **users**
    - **visits**
  - static
  - templates
    - users
    - visits

Cookie-cutter structure

- dj_test
  - manage.py
  - config
    - urls.py
    - settings.py
  - dj_test
    - **users**
    - **visits**
    - static
    - templates
      - users
      - visits

Update settings for other structures

- In order to use other structures, you'll have to edit some settings and configurations
- Let **manage.py** know where the default setting file is
- In your settings file, add paths for Django to find your templates and static files
  - **TEMPLATES["DIRS"]**
  - **STATICFILES_DIRS**
- Your AppConfig's name in **apps.py** needs to use the full path
  - e.g. name = **'apps.users'**
- More details when we talk about settings

Poll (Single choice)

- Which area do you want to cover next?
  - Settings
  - Models
  - URLs
  - Views
  - Templates
  - Admin
  - Forms
  - Other (say in chat)

Settings

Settings

- Provides configuration for your Django project
- Default location is **settings.py**
- Apps with multiple environments usually split it into:
  - settings/
    - base.py
    - production.py
    - local.py
    - staging.py

Settings

- Python file, python syntax
- Settings are constants
  - Use all uppercase
  - Reload server if any settings change
- Specifying your settings file
  - In manage.py
    - `os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'dj_test.settings')`
  - Set environment variable **DJANGO_SETTINGS_MODULE**

Specifying paths

- Django >= 3.1

```
from pathlib import Path
BASE_DIR = Path(__file__).resolve(strict=True).parent.parent.parent
STATICFILES_DIRS = [BASE_DIR / "static"]
```

- Django <= 3.0

```
import os
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
STATICFILES_DIRS = [os.path.join(BASE_DIR, 'static')]
```

Must-know settings

- **DEBUG**
  - Show detailed error page vs 404.html or 500.html
  - Never set it to True in production
- **INSTALLED_APPS**
  - All Django, 3rd party and local apps used
- **DATABASES**
  - Can configure multiple databases
  - Should be different for local, production, staging

Project structure

- **BASE_DIR**
  - Set to project root (where manage.py is located)
- **APP_DIR**
  - Set to where apps are located
- **TEMPLATES**
  - **DIRS** - Where to look for templates
  - **APP_DIRS** – If True, look for a "templates" folder in each app
- **STATICFILES_DIRS**
  - Where to look for static files
- **MEDIA_ROOT**
  - Where to look for media files (user-uploaded)

Time zones

- **USE_TZ**
  - True if dates/times should be timezone-aware by default
  - Should set to True
- **TIME_ZONE** – Set it to your time zone

- **USE_I18N** – Enable text translation (set to False if not supported)
- **USE_L10N** – Format data based on locale

- Docs:
  - Time zones
  - Text translation
  - Date and time formatting

Security

- AUTH_PASSWORD_VALIDATORS
- SESSION_EXPIRE_AT_BROWSER_CLOSE

Configuring for deployment

- **DEBUG** = False
- **SECRET_KEY**
  - Set it to a unique, unpredictable value, over 50 characters
  - Don't commit to source control, use environment variables
- **ALLOWED_HOSTS**
  - A list of domains that this Django project can serve
- **LOGGING**

Static files in production

- **Static files** – JS, CSS and images used in your site
- **Media files** – User uploaded content (images, videos, files)

- In ***development***, your static files can be in many locations
  - E.g. in {installed_app}/static/ and any STATICFILES_DIRS

- In ***production***, you run **`manage.py collectstatic`** to collect them all to one location (STATIC_ROOT)

Static files in production

- Options for serving static and media files:
  1. On the same server as your app
  2. On a different server
  3. Using a cloud service or CDN

- Serving static files in production

Static files in production

1. On the same server as your app
   - Files are served from STATIC_ROOT and MEDIA_ROOT locations on the server
   - Ability to do this depends on your web host
   - May need to use the WhiteNoise middleware

2. On a different server

3. Using a cloud service or CDN

Static files in production

1. On the same server as your app

2. On a different server
   - Files are transferred from STATIC_ROOT to the other server after **collectstatic** is run (often using **rsync**)
   - Set the STATIC_URL and MEDIA_URL to the other server

3. Using a cloud service or CDN

Static files in production

1. On the same server as your app

2. On a different server

3. Using a cloud service or CDN
   - Set the STATIC_URL and MEDIA_URL to the other location
   - Set STATICFILES_STORAGE and DEFAULT_FILE_STORAGE to use the cloud service
   - For Amazon S3 storage, use Boto3 to automatically transfer files

Using environment variables

- You don't want to commit some settings to source control
  - SECRET_KEY
  - Any passwords, keys or tokens

- Some settings are different depending on the environment
  - DB type, location, user, password
  - DEBUG
  - ALLOWED_HOSTS

- Store these either in **environment variables** or a **.env** file that is not committed to source control

Using environment variables

- Replace hard-coded settings with values retrieved from environment variables

```
import os

SECRET_KEY = os.environ["DJANGO_SECRET_KEY"]
DEBUG = bool(os.getenv("DJANGO_DEBUG", default=False))
```

- Cons:
  - Default config hard to share with other developers
  - Everything is a string, need to convert types manually

Using .env files with **environ**

- Replace hard-coded settings with values form a .env file

```
import environ
env = environ.Env()                    # Get os environ
env.read_env(BASE_DIR / ".env")  # Read .env file

SECRET_KEY = env("DJANGO_SECRET_KEY")  # Required
DEBUG = env.bool("DJANGO_DEBUG", default=False)
```

- Commit a **.env-example** file for default config
- Variables can either be in environment variables or .env file

Splitting up settings.py

1. Have a settings.py and local_settings.py

2. Have a settings folder, containing:
   - base.py
   - production.py
   - local.py
   - etc

local_settings.py

- On your machine, create a **local_settings.py** with your local environment specific settings
  - Don't commit to source control
  - Commit a **local_settings.py.template** file to store default config

- At the bottom of **settings.py**

```
try:
    from .local_settings import *
except ImportError:
    pass
```

base.py

- Create a **base.py** file and a file for each environment

- At the top of each environment file

```
from .base import *
from .base import env

# Hard-code settings or read from env
```

- Update **manage.py** and **wsgi.py** to read from the correct settings file

More resources

- Official documentation
  - Overview
  - Full list of settings
  - Settings organized by topic

- Configuring Django Settings: Best Practices

Poll (Single choice)

- Which area do you want to cover next?
  - Settings
  - Models
  - URLs
  - Views
  - Templates
  - Admin
  - Forms
  - Other (say in chat)

Models and migrations

The Django ORM

- Manage your data and database with Python instead of SQL
- Lazy evaluation of queries (database only gets accessed when data is requested, not when forming queries)
- Easy things are easy, medium things are possible, hard things can be done in SQL

Creating models

```
from django.db import models

class Question(models.Model):
    text = models.CharField('question', max_length=255)
    details = models.TextField(blank=True)
    image = models.ImageField(blank=True)
```

- Create class that subclasses **models.Model**
- Add model fields
- Override __str__ and __repr__

Model id's

- When you create a model, an **id** field is automatically created
  - Unless you specify a primary key field
- Access via **object.id** or **object.pk** (pk stands for primary key)
- We often use **pk** (instead of **id**) when retrieving objects, just in case the model uses a custom primary key field
  - e.g. `Question.objects.get(pk=12)`

Model fields

- Lots of Field types
  - BooleanField, CharField, IntegerField, DateTimeField, TextField
  - DecimalField, DurationField, EmailField, UUIDField, + more
  - ImageField, FileField

- Commonly used field arguments
  - null, default, max_length, unique
  - For DateTimeFields
    - auto_now sets to the current datetime when modified
    - auto_now_add sets to the current datetime when created

Other fields options

- You can also add validators, help text, error messages, and verbose_name to add features to the Admin and forms

- Full list of model field types and options: https://docs.djangoproject.com/en/3.2/ref/models/fields/

Blank vs null

- **blank** allows a form field to be blank
- **null** allows null values in the database
- When a CharField is left blank in the admin, it gets set to the empty string
- The book **Two Scoops of Django** has a good table that helps determine when to set **blank=True** and **null=True**

Field Type	Setting null=True	Setting blank=True
`CharField`, `TextField`, `SlugField`, `EmailField`, `CommaSeparated-IntegerField`, `UUIDField`	*Okay* if you also have set both `unique=True` and `blank=True`. In this situation, `null=True` is required to avoid unique constraint violations when saving multiple objects with blank values.	*Okay* if you want the corresponding form widget to accept empty values. If you set this, empty values are stored as `NULL` in the database if `null=True` and `unique=True` are also set. Otherwise, they get stored as empty strings.
`FileField`, `ImageField`	*Don't do this.* Django stores the path from `MEDIA_ROOT` to the file or to the image in a CharField, so the same pattern applies to FileFields.	*Okay.* The same pattern for CharField applies here.
`BooleanField`	*Okay.*	Default is `blank=True`.
`IntegerField`, `FloatField`,	*Okay* if you want to be able to set the value to `NULL` in	*Okay* if you want the corresponding form widget to

Using models

- Create an object
- Get an attribute
- Update
- Delete

Example usage in **apps.questions.examples.models.py**

Retrieving models

- Get an object by its id
- Get the first or last object
- Get a set of objects
- Filter a set of objects
- Get the number of objects in a set

Example usage in **apps.questions.examples.models.py**

Meta

- Set *ordering*
- Set *verbose_name* and *verbose_name_plural* of model
- Set db properties over groups of fields, like *index_together* and *unique_together*

- Docs: https://docs.djangoproject.com/en/3.2/ref/models/options/

- You can access `MyModel._meta.meta_field`
- Even though it is "hidden" it's very commonly used
  - They're working on making it public

Foreign keys

```
class Choice(models.Model):

    question = models.ForeignKey(
        Question,
        on_delete=models.CASCADE,
        related_name="choices"
    )

    text = models.CharField(max_length=100)
    is_correct = models.BooleanField(default=False)
    details = models.TextField(blank=True)
```

Querysets

- When Django returns a set of objects, it's called a **Queryset**
- It's like a list, but lazily evaluated
  - The db is only queried when you access an attribute
  - You can't get objects by index e.g. Question.objects.all()**[5]**

- You can make quite advanced queries
  - Look up Q, F, aggregates, and ordering
  - Optimize with select_related and prefetch_related

Managers

- A manager is what is returned if you call Question**.objects**
- You can create custom managers that can store custom queries on your models

- Custom query examples:
  - Question.objects.**with_difficulty("Easy")**
  - Question.objects.**with_images()**
  - Question.objects.**unseen_by_user(user)**

Migrations

- When a model changes, the database needs to be kept up to date

```
$ ./manage.py makemigrations
$ ./manage.py migrate
```

- You can see the SQL created for a migration

```
$ ./manage.py sqlmigrate <app_name> <migration>
```

e.g. **sqlmigrate questions 0003**

Migrating

- To apply a specific migration

```
$ ./manage.py migrate <app_name> <migration>
```

- This command will also unapply a migration if you select one "in the past"

```
$ ./manage.py migrate questions 0003
```

- If your db is behind 0003, then it will migrate up to 0003
- If your db is ahead of 0003, it will rollback until 0003
- To target the initial state, use "**zero**" instead of a #
- To "trick" Django into thinking it's at a specific migration without affecting the db, use the **--fake** flag (**migrate --fake <app> <num>**)

Migrations

- You can run arbitrary Python code in a migration
  - Usually used for updating the data inside your db
  - Tutorial

- For example, if you want to add a non-null field to an existing model with a complex default value
  1. Create the new field, but make it nullable
  2. Run Python code to set the value of the new field for existing objects
  3. Make the field non-null

Inheritance

- Inheriting Model classes in Django can be tricky
- Safest way is to create an abstract base class (not saved in db) and concrete child classes (saved in separate tables)
- You can't query over all subclasses easily, but it saves a lot of trouble in the long run

- Set **`abstract=True`** in the Meta class

User model

- It's good practice to create a custom user model from the start
- Subclassing is tricky and you usually need custom fields

Example usage in **apps.users.models.py**

- https://docs.djangoproject.com/en/3.2/topics/auth/customizing/#auth-custom-user

User login/logout/registration

- Login/logout tutorial
  - https://learndjango.com/tutorials/django-login-and-logout-tutorial
- Registration tutorial
  - https://learndjango.com/tutorials/django-signup-tutorial

- Use All Auth to handle 3rd party authentication
  - https://django-allauth.readthedocs.io/en/latest/index.html

Poll (Single choice)

- Which area do you want to cover next?
  - Settings
  - Models
  - URLs
  - Views
  - Templates
  - Admin
  - Forms
  - Other (say in chat)

URLs

URL config

- Configure which URLs your site can handle
- Other URLs return a 404 Not Found page

- The main/project URL conf can:
  - Configure its own URL endpoints
  - Include URLs from various app URL conf files
  - Include routes for static and media files
  - Configure how errors should be handled (400, 403, 404, 500)

- Docs: https://docs.djangoproject.com/en/3.2/ref/urls/

URL config

- Should contain a list called **urlpatterns**
- If it's in an app, it should also include an **app_name** variable

```
app_name = "users"
urlpatterns = [
    path("me/", view=views.profile, name="profile"),
]
```

- You can conditionally configure this list, e.g:

```
if settings.DEBUG:
    urlpatterns += []  # Add paths only for development
```

Get a URL by name

```
app_name = "users"
urlpatterns = [
    path("me/", view=views.profile, name="profile"),
]
```

- In Python files

```
from django.urls import reverse
url = reverse("users:profile")  # "/users/me/"
```

- In HTML files

```
{% url "users:profile" %}
```

```
e.g. <a href="{% url 'users:profile' %}">My profile</a>
```

Get a URL by name

```
app_name = "users"
urlpatterns = [
    path("me/", view=views.profile, name="profile"),
]
```

Here, "users" comes from the main urls.py where "users.urls" is included

- In Python files

```
from django.urls import reverse
url = reverse("users:profile")  # "/users/me/"
```

- In HTML files

```
{% url "users:profile" %}
```

```
e.g. <a href="{% url 'users:profile' %}">My profile</a>
```

Accepting arguments

- Use <> to capture URL arguments and pass them to the view

```
path('users/<username>/', views.profile)
```

- Arguments are strings by default, but you can specify the type

```
path('users/<int:pk>/, views.profile)
```

Regular expressions

- You can also capture arguments via regular expressions using **re_path()**

```
re_path(r'^users/?P<username>\w+/$', views.profile)
```

- Note:
  - This is the same **url()** in older versions of Django
  - The new **path()** function is easier to read and sufficient for most cases
- Intro to regular expressions (video)
- Regular expressions cheat sheet

Get a URL by name with arguments

```
path("<username>/bio", view=views.profile, name="profile")
```

- In Python files

```
reverse("users:profile", args=["admin"])
reverse("users:profile", kwargs= {username:"admin"})
# Returns "/users/admin/bio/"
```

- In HTML files

```
{% url 'users:profile' 'admin' %}           # Constant
{% url 'users:profile' user.username %}     # Variable
{% url 'users:profile' username='admin' %} # Kwargs
```

Combining URLconfs with include()

- URL lists can be combined

```
urlpatterns = [
    path("users/", include("apps.users.urls")),
]
```

- When Django encounters **include()**, it:
  - chops off whatever part of the URL matched up to that point
  - sends the remaining string to the included URLconf to match

Media files

- If you use any sort of FileField in your models, those are considered media files that a user can upload

- In **settings.py** (these might be different in production)

```
MEDIA_URL = '/media/'
MEDIA_ROOT = BASE_DIR / 'media'
```

- In your main **urls.py**

```
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = [  # ... other urls
] + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)]
```

Poll (Single choice)

- Which area do you want to cover next?
  - Settings
  - Models
  - URLs
  - Views
  - Templates
  - Admin
  - Forms
  - Other (say in chat)

Views

View function

- When a URL path gets matched:
  - Django calls the corresponding view function
  - Passes it the request and any captured arguments

- The view function might:
  - return some HTML or JSON data or an exception page
  - handle a GET or POST request differently
  - update and/or retrieve data from the database (via models)
  - require certain permissions from the user

View complexity

- Because certain functionality is common to many views, there are a lot of **shortcuts** you can use to create view functions
  - Django shortcuts, like **render()** and **get_object_or_404()**
  - Class based views, like **CreateView**
  - View mixins, like **LoginRequiredMixin**
  - View decorators, like **login_required**

- The more shortcuts you use, the more base knowledge is required to read your code and write your views

- The most basic view function just takes in an HTTP request and returns an HTTP response

A simple view function

```
def index(request):
    some_html = """
    <h1>Welcome to my site!</h1>
    <p>Have a look around</p>
    """
    return HttpResponse(content=some_html, status=200)
```

A more realistic view function

```
@require_GET
@login_required
def list_view(request):
    if request.user.is_staff:
        users = User.objects.all()
    else:
        users = User.objects.filter(is_staff=False)
    context = {"users": users}
    return render(request, "users/user_list.html", context=context)
```

The request object

- A view function always accepts an **HttpRequest** object
  - from django.http.request

- An **HttpRequest** has:
  - A user (as long as you're using the Django auth middleware)
    - Will be an anonymous user if they're not logged in
  - body – content of the request
  - GET/POST – query parameters
  - Headers – metadata
  - request URL

View responses

- A view function always returns an **HttpResponse** object
- An **HttpResponse** has:
  - A status code (like 200 Okay)
  - Some content (like HTML or JSON)
  - Some metadata (in headers)
- **render()** is a shortcut that finds a template, renders it with the given context, and returns it as an HttpResponse
- Compare **question_view()** and **question_view_no_shortcuts()** in trivia_site.apps.questions.views.py to see what render() does

Exceptions

- When returning an HttpResponse, you can pass in a custom status_code or return a subclass, like **HttpResponseNotFound**
  - But these require you to pass in the content as well

- In order to return consistent error pages, you can raise an **Http404** exception so Django uses your 404.html template

- **Note**: to test error templates, you must make DEBUG=False or create special URLs to serve them
  - If DEBUG = False, static files (like styles.css) will fail to load
  - Use `./manage.py runserver --insecure` to serve static files locally – Read more

View decorators

- Before a view function is called, you can do some default request handling

- Examples:
  - Require a user be logged in, redirect them to a login view if not
    - Use @login_required
  - Only accept certain HTTP methods, or return a 405 error
    - Use @require_GET, @require_POST or @require_http_methods

A more realistic view function

```
@require_GET
@login_required
def list_view(request):
    if request.user.is_staff:
        users = User.objects.all()
    else:
        users = User.objects.filter(is_staff=False)
    context = {"users": users}
    return render(request, "users/user_list.html", context=context)
```

A more realistic view function

```
@require_GET
@login_required
def list_view(request):
    if request.user.is_staff:
        users = User.objects.all()
    else:
        users = User.objects.filter(is_staff=False)
    context = {"users": users}
    return render(request, "users/user_list.html", context=context)
```

Only accept GET requests, return a 405 if other method used

A more realistic view function

```
@require_GET
@login_required
def list_view(request):
    if request.user.is_staff:
        users = User.objects.all()
    else:
        users = User.objects.filter(is_staff=False)
    context = {"users": users}
    return render(request, "users/user_list.html", context=context)
```

If a user isn't logged in, send them to the login page first, then redirect to this view

A more realistic view function

```
@require_GET
@login_required
def list_view(request):
    if request.user.is_staff:
        users = User.objects.all()
    else:
        users = User.objects.filter(is_staff=False)
    context = {"users": users}
    return render(request, "users/user_list.html", context=context)
```

If the logged in user is staff, show the all users

A more realistic view function

```
@require_GET
@login_required
def list_view(request):
    if request.user.is_staff:
        users = User.objects.all()
    else:
        users = User.objects.filter(is_staff=False)
    context = {"users": users}
    return render(request, "users/user_list.html", context=context)
```

Otherwise, show a filtered list

A more realistic view function

```
@require_GET
@login_required
def list_view(request):
    if request.user.is_staff:
        users = User.objects.all()
    else:
        users = User.objects.filter(is_staff=False)
    context = {"users": users}
    return render(request, "users/user_list.html", context=context)
```

Render a template with a context and return the HttpResponse

Function vs class-based views

- I've included some examples in the **users** app of class-based views (CBVs)
- Some Django users love CBVs and some prefer FBVs (function-based views)
- In general, I think FBVs are easier to read, understand, and write and recommend them for beginners
- As a good starting place, you can inherit from the base View

The basic View

- The previous function-based view can be rewritten as:

```
class ListView(LoginRequiredMixin, View):

    def get(self, request):
        if request.user.is_staff:
            users = User.objects.all()
        else:
            users = User.objects.filter(is_staff=False)
        context = {"users": users}
        return render(request, "users/user_list.html", context=context)
```

- Link it to the url path by using **ListView.as_view()** to get the function

The basic View

- The previous function-based view can be rewritten as:

Inherit from the basic View class

```
class ListView(LoginRequiredMixin, View):

    def get(self, request):
        if request.user.is_staff:
            users = User.objects.all()
        else:
            users = User.objects.filter(is_staff=False)
        context = {"users": users}
        return render(request, "users/user_list.html", context=context)
```

The basic View

- The previous function-based view can be rewritten as:

Use a mixin instead of a decorator

```
class ListView(LoginRequiredMixin, View):

    def get(self, request):
        if request.user.is_staff:
            users = User.objects.all()
        else:
            users = User.objects.filter(is_staff=False)
        context = {"users": users}
        return render(request, "users/user_list.html", context=context)
```

The basic View

- The previous function-based view can be rewritten as:

```
class ListView(LoginRequiredMixin, View):

    def get(self, request):
        if request.user.is_staff:
            users = User.objects.all()
        else:
            users = User.objects.filter(is_staff=False)
        context = {"users": users}
        return render(request, "users/user_list.html", context=context)
```

Define a get method to accept GETs
Don't define a post to return a 405 on POSTs

CBV Gotchas

- The order of mixins matters!
  - class ListView(**LoginRequiredMixin, View**) works ✅
  - class ListView(**View, LoginRequiredMixin**) doesn't work ❌
  - Resolution order reads left to right, so the CBV should be last since mixins should resolve before the base class
  - Detailed explanation
- Not

Poll (Single choice)

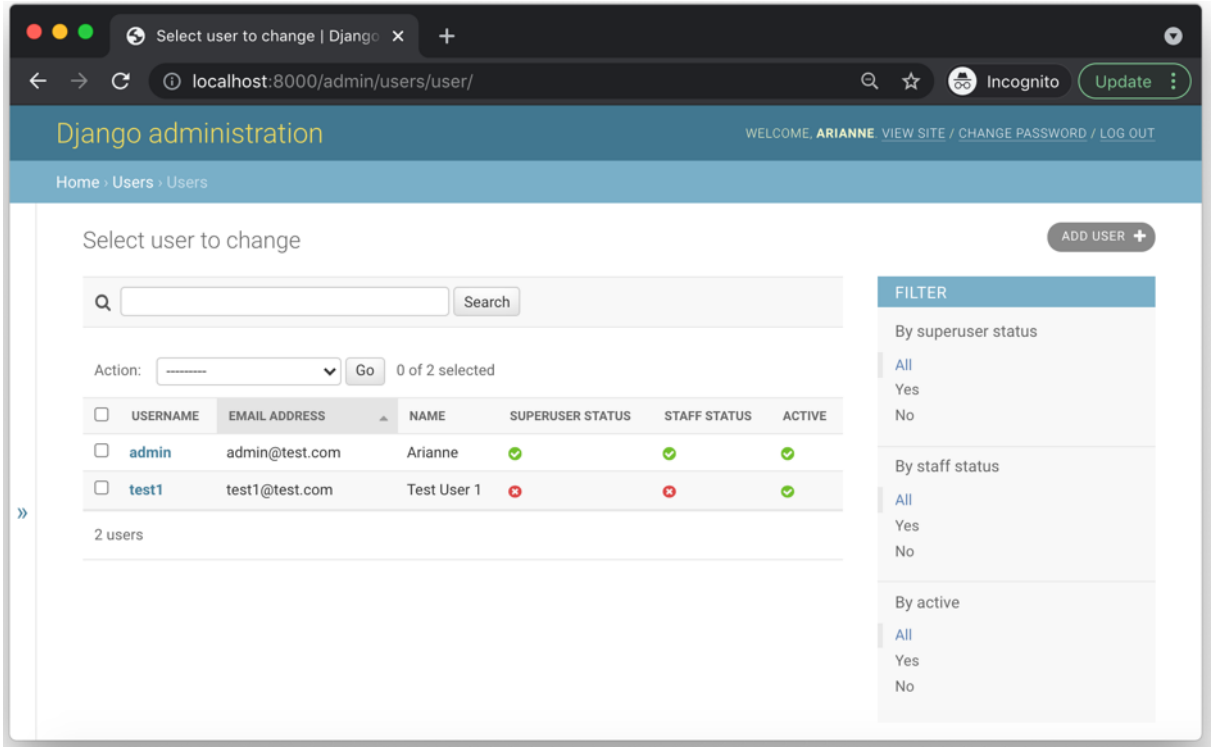
- Which area do you want to cover next?
  - Settings
  - Models
  - URLs
  - Views
  - Templates
  - Admin
  - Forms
  - Other (say in chat)

Django Admin

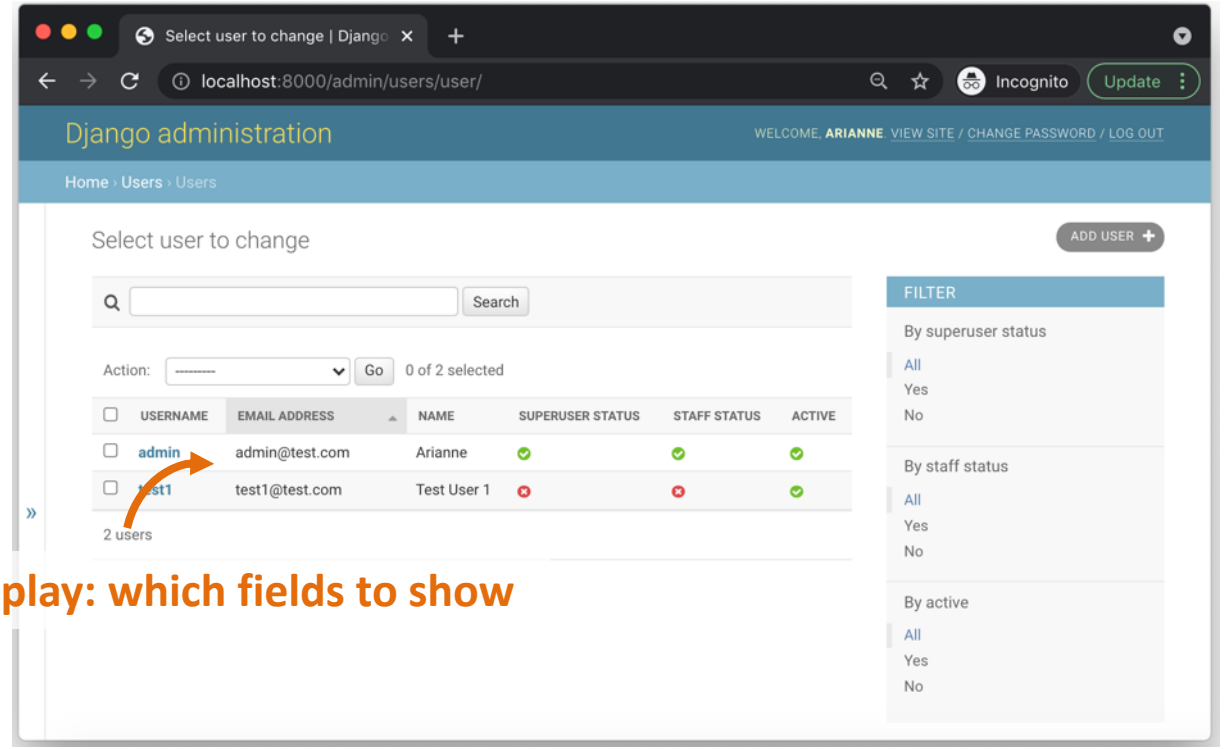
Overall design

- Index
  - Apps
    - Models
      - List view
      - Add view
      - Change view

List view

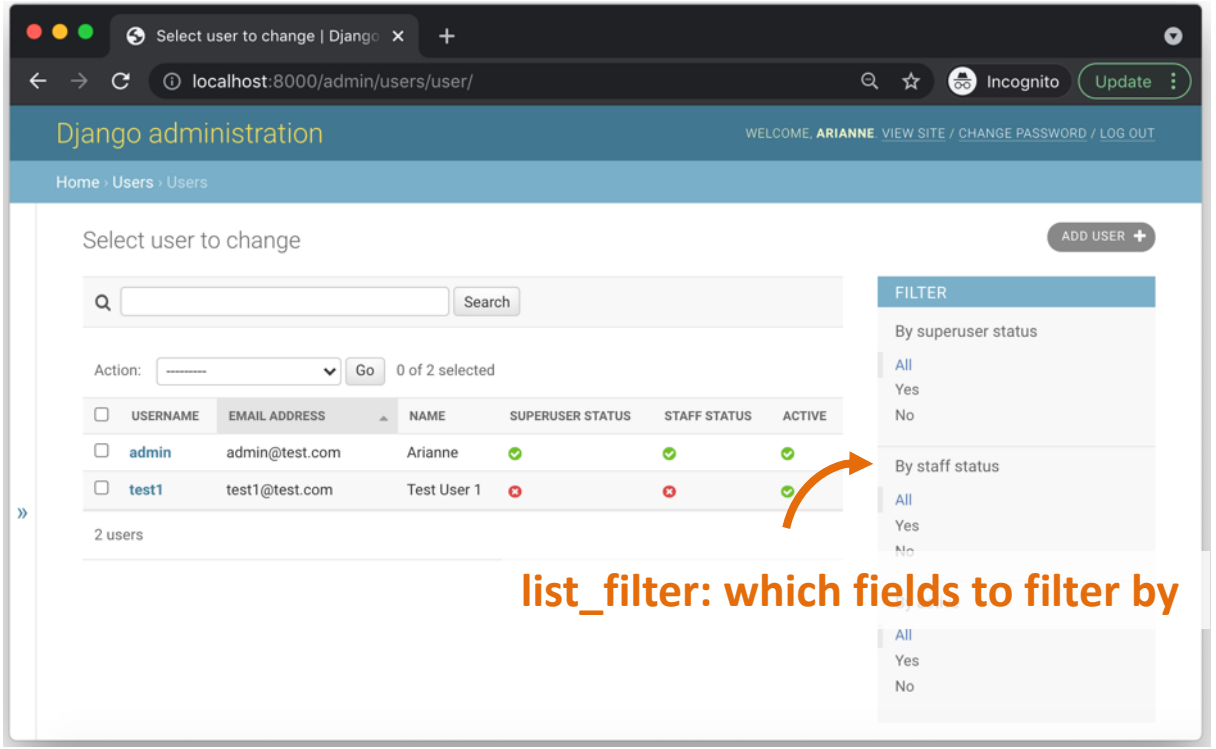

Select user to change | Django
localhost:8000/admin/users/user/
Incognito
Update
Django administration
WELCOME, ARIANNE. VIEW SITE / CHANGE PASSWORD / LOG OUT
Home › Users › Users
Select user to change
ADD USER
Search
Action:
Go
0 of 2 selected
USERNAME
EMAIL ADDRESS
NAME
SUPERUSER STATUS
STAFF STATUS
ACTIVE
admin
admin@test.com
Arianne
test1
test1@test.com
Test User 1
2 users
FILTER
By superuser status
All
Yes
No
By staff status
All
Yes
No
By active
All
Yes
No

List view

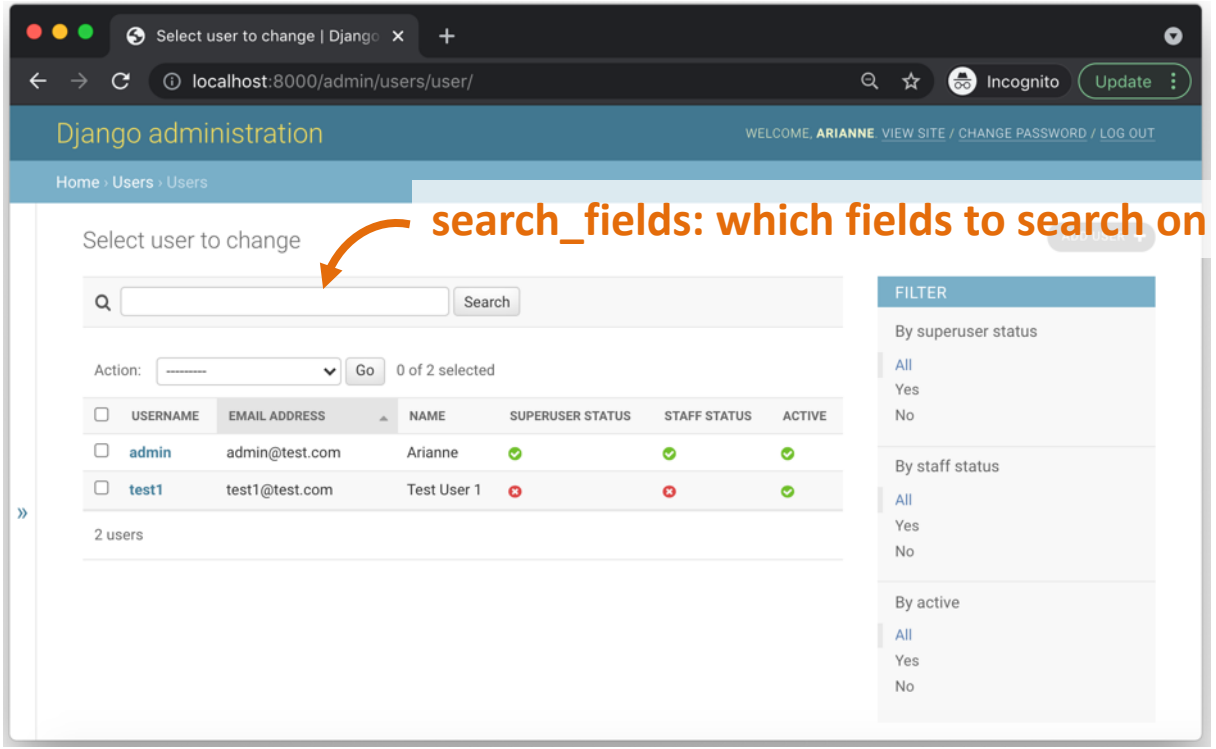


list_display: which fields to show

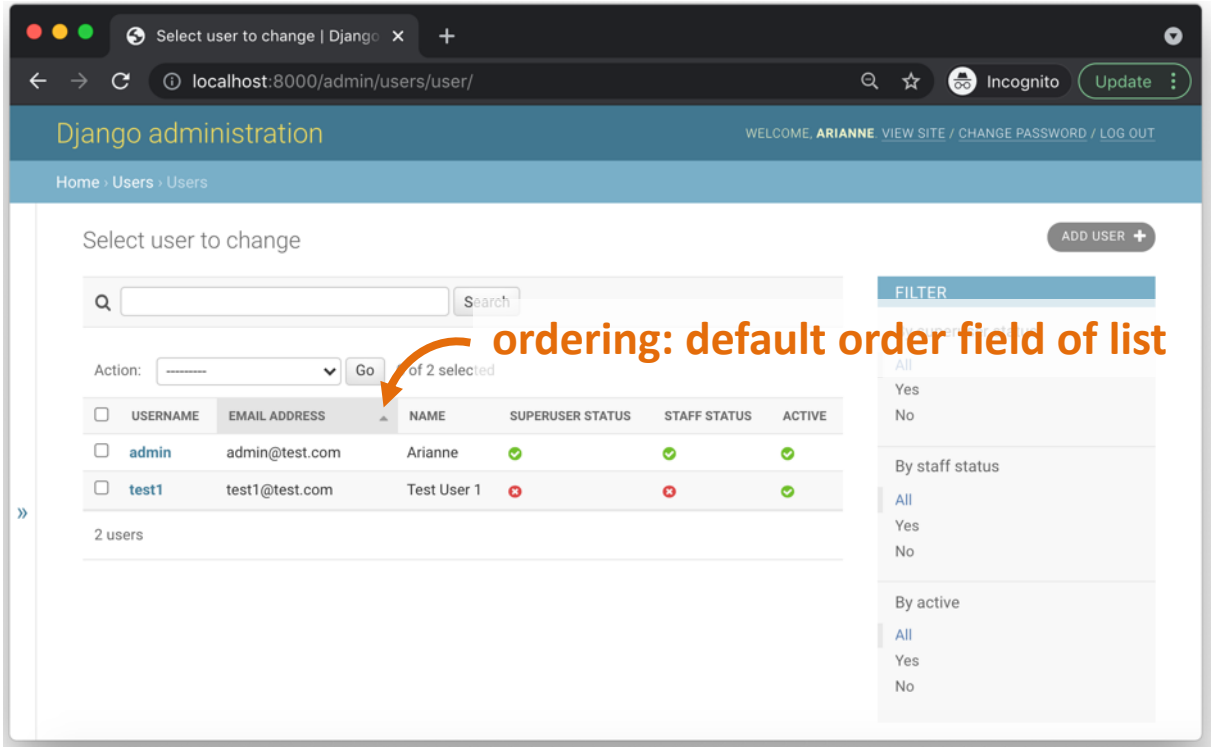
List view


Select user to change | Django
localhost:8000/admin/users/user/
Incognito
Update
Django administration
WELCOME, ARIANNE. VIEW SITE / CHANGE PASSWORD / LOG OUT
Home › Users › Users
Select user to change
ADD USER
Search
Action:
Go
0 of 2 selected
USERNAME
EMAIL ADDRESS
NAME
SUPERUSER STATUS
STAFF STATUS
ACTIVE
admin
admin@test.com
Arianne
test1
test1@test.com
Test User 1
2 users
FILTER
By superuser status
All
Yes
No
By staff status
All
Yes
No
All
Yes
No
list_filter: which fields to filter by

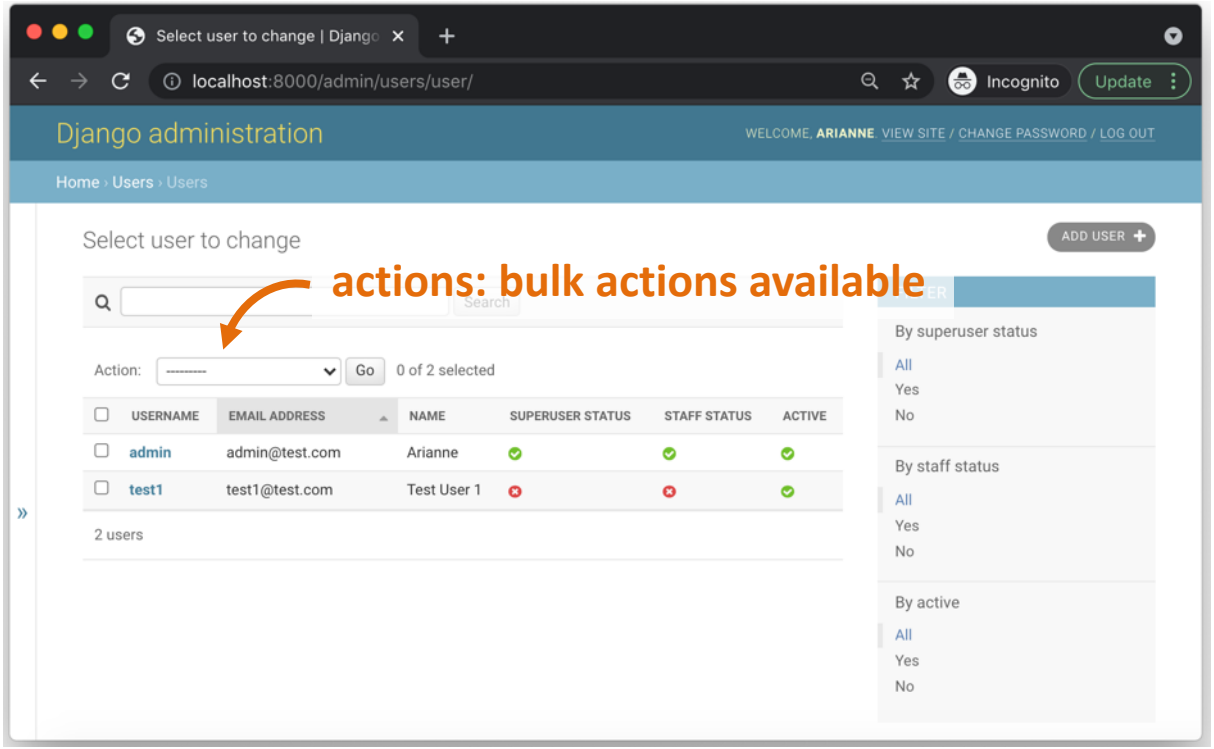
List view


Select user to change | Django
localhost:8000/admin/users/user/
Incognito
Update
Django administration
WELCOME, ARIANNE. VIEW SITE / CHANGE PASSWORD / LOG OUT
Home › Users › Users
search_fields: which fields to search on
Select user to change
Search
Action:
Go
0 of 2 selected
USERNAME
EMAIL ADDRESS
NAME
SUPERUSER STATUS
STAFF STATUS
ACTIVE
admin
admin@test.com
Arianne
test1
test1@test.com
Test User 1
2 users
FILTER
By superuser status
All
Yes
No
By staff status
All
Yes
No
By active
All
Yes
No

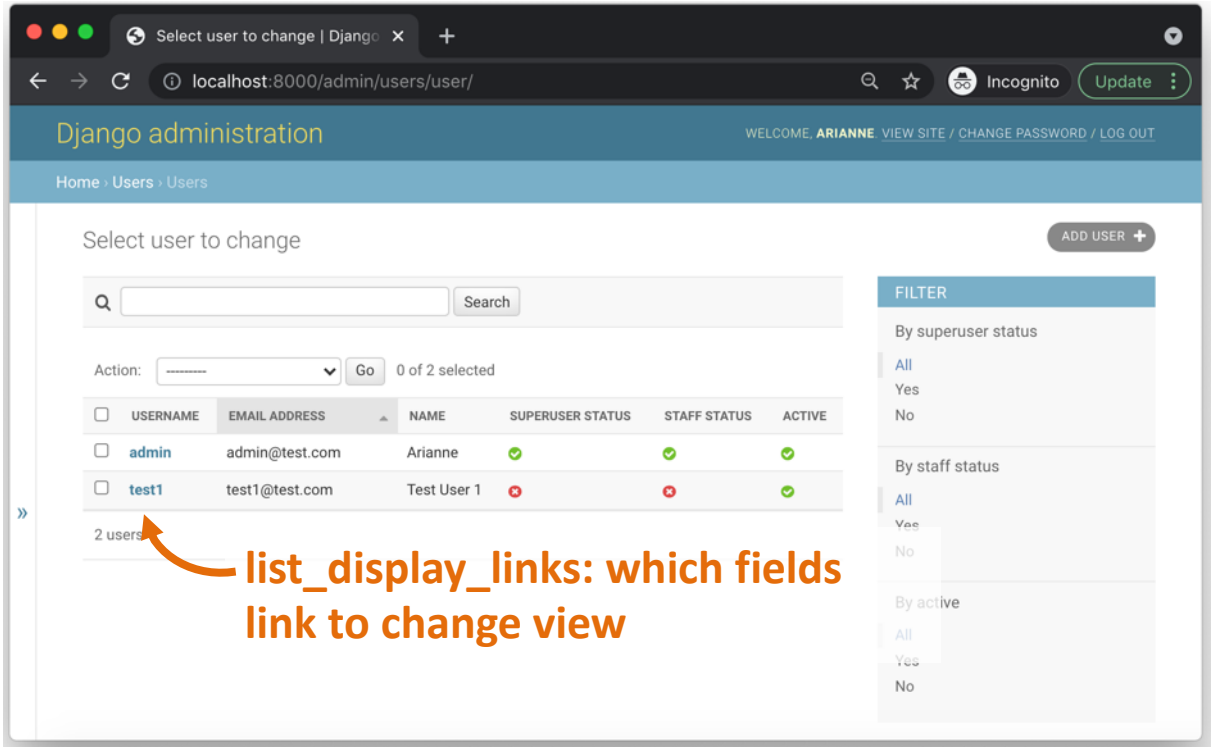
List view


Select user to change | Django
localhost:8000/admin/users/user/
Incognito
Update
Django administration
WELCOME, ARIANNE. VIEW SITE / CHANGE PASSWORD / LOG OUT
Home › Users › Users
Select user to change
ADD USER
Search
FILTER
ordering: default order field of list
Action:
Go
of 2 selected
USERNAME
EMAIL ADDRESS
NAME
SUPERUSER STATUS
STAFF STATUS
ACTIVE
admin
admin@test.com
Arianne
test1
test1@test.com
Test User 1
2 users
All
Yes
No
By staff status
All
Yes
No
By active
All
Yes
No

List view


Select user to change | Django
localhost:8000/admin/users/user/
Incognito
Update
Django administration
WELCOME, ARIANNE. VIEW SITE / CHANGE PASSWORD / LOG OUT
Home › Users › Users
Select user to change
ADD USER
actions: bulk actions available
Search
Action:
Go
0 of 2 selected
USERNAME
EMAIL ADDRESS
NAME
SUPERUSER STATUS
STAFF STATUS
ACTIVE
admin
admin@test.com
Arianne
test1
test1@test.com
Test User 1
2 users
By superuser status
All
Yes
No
By staff status
All
Yes
No
By active
All
Yes
No

List view


Select user to change | Django
localhost:8000/admin/users/user/
Incognito
Update
Django administration
WELCOME, ARIANNE. VIEW SITE / CHANGE PASSWORD / LOG OUT
Home › Users › Users
Select user to change
ADD USER
Search
Action:
Go
0 of 2 selected
USERNAME
EMAIL ADDRESS
NAME
SUPERUSER STATUS
STAFF STATUS
ACTIVE
admin
admin@test.com
Arianne
test1
test1@test.com
Test User 1
2 users
FILTER
By superuser status
All
Yes
No
By staff status
All
By active
All
No
list_display_links: which fields link to change view

Change/add view

- Create or edit your models here
- Can define readonly_fields

- Can create/edit/delete related objects through inlines
  - Inlines can be stacked or tabular
  - If you need nested inlines, use the extension **django-nested-admin**

Customizing style and templates

- Add folders called **admin** in your **templates** and **static/css/ folders**

- When Django tries to load the templates and styles for the admin site, it will check your local files first before looking in the admin app's folders

- You can override any of the admin's default files this way

Poll (Single choice)

- Which area do you want to cover next?
  - Settings
  - Models
  - URLs
  - Views
  - Templates
  - Admin
  - Forms
  - Other (say in chat)

Front-end

Front-end for Django

- The easiest set-up is to use the **Django Template Language** to create your HTML pages
- You can swap it for the **Jinja2**, which Flask uses, and can run arbitrary Python code, or write your own template backend
- This method doesn't allow you to create single-page apps
  - Limited to the old **page -> click -> request -> new page** flow

Front-end for Django

- If you want to create an interactive, single-page app, you can still use Django as the backend

- Create a REST or GraphQL API to interact with a JS front-end framework, like Angular or React
  - REST API – Use Django Rest Framework (DRF)
  - GraphQL API – Use Graphene-Django

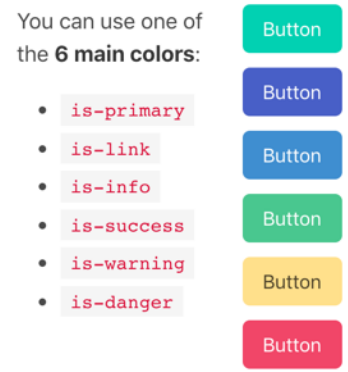
- A common middle-ground is to use JQuery and custom JavaScript to provide the amount of interactivity your app needs

CSS frameworks for Django

- Use a CSS framework to make your site responsive and interactive
- Bootstrap is common, but other options are Foundation, Tailwind, Bulma, and many more
- One thing to consider is that Django forms can be created like so:
  - `{{ form.as_p }}`
  - If you want your form to conform to the CSS framework you're using, try to find a django extension for it

Bulma – CSS Framework

- In the sample Trivia project, I'm using **Bulma**
  - https://bulma.io/
- To handle my Django forms, I'm using **django-bulma**
  - https://github.com/timonweb/django-bulma




```
<a class="button is-primary">
  Button
</a>
<a class="button is-link">
  Button
</a>
<a class="button is-info">
  Button
</a>
<a class="button is-success">
  Button
</a>
<a class="button is-warning">
  Button
</a>
<a class="button is-danger">
  Button
</a>
```

Django Template Language

Templates

- Templates are just HTML files with special syntax that the Django Template Language (DTL) uses to replace with dynamic content

- Variable: **{{** *var* **}}**
- Filter: **{{** *var*|*filter* **}}**
- Tag: **{%** *tag* **%}**
- Comment: **{#** *comment* **#}**

Example template

```
{% extends "base.html" %}

{% block content %}

  <h1>{{ user.name|upper }}</h1>

  {% for field in user_fields %}
    {% include "partials/field_display.html" %}
  {% endfor %}

{% endblock %}
```

Variables

- Some data you can use in your template
  - **request**, **user**, and anything in your **context** dict

- You can access attributes of the object with dot-notation
  - **Model field:** {{ user.email }}
  - **Model property:** {{ question.random_choices }}
  - **Model method (no arguments):** {{ user.get_absolute_url }}
  - **Dictionary value:** {{ dict.key }}
  - **List index:** {{ list.0 }} (no negative index)

Variables - cont

- To see available variables, add this to your template:
  - <**pre**> {% filter force_escape %} {% debug %} {% endfilter %} </**pre**>
  - Only look at the top section for useful info (not long list of modules)

- If you need to call a model method with arguments, you'd need to create a template tag or use a different templating library, like Jinja2
  - Read more

Filters

- Allow you to modify variables for display purposes
  - {{ user.name|**lower** }}
  - {{ my_list|**last** }}
  - {{ users|**length** }}

- You can pass arguments in
  - {{ a_value|**default**:"nothing" }}

- All built-in filters
- Creating custom filters

Tags

- Provide logic for templates
  - {% **for** u **in** users %} ... {% **endfor** %}
  - {% **if** user.is_staff %} ... {% **endif** %}
  - {% **extends** "base.html" %}
  - {% **block** title %} ... {% **endblock** %}
  - {% **include** "partials/header.html" %}

- All built-in tags
- Creating custom tags

Example template

Inherit from the base template

```
{% extends "base.html" %}

{% block content %}

  <h1>{{ user.name|upper }}</h1>

  {% for field in user_fields %}
    {% include "partials/field_display.html" %}
  {% endfor %}

{% endblock %}
```

Template inheritance

- Usually, you define a base template that includes your sites:
  - CSS styles
  - JavaScript libraries
  - general layout (header, footer, content wrappers)

- Create a base.html file
- Every other file will extend the base file
- Customize parts of the base file using {% block %} tags

Simple base.html

```
<html>
<head>
  <title>Django Trivia</title>
  <link href="/static/css/style.css" type="text/css" rel="stylesheet">
</head>
<body>
  <section class="main">
  {% block content %}
    Put some custom content here
  {% endblock %}
  </section>
  <script src="/static/js/project.js"></script>
</body>
</html>
```

Simple base.html

```
<html>
<head>
  <title>Django Trivia</title>
  <link href="/static/css/style.css" type="text/css" rel="stylesheet">
</head>
<body>
  <section class="main">
  {% block content %}
    Put some custom content here
  {% endblock %}
  </section>
  <script src="/static/js/project.js"></script>
</body>
</html>
```

Add your CSS styles

Simple base.html

```
<html>
<head>
  <title>Django Trivia</title>
  <link href="/static/css/style.css" type="text/css" rel="stylesheet">
</head>
<body>
  <section class="main">
  {% block content %}
    Put some custom content here
  {% endblock %}
  </section>
  <script src="/static/js/project.js"></script>
</body>
</html>
```

Add your JavaScript

Simple base.html

```
<html>
<head>
  <title>Django Trivia</title>
  <link href="/static/css/style.css" type="text/css" rel="stylesheet">
</head>
<body>
  <section class="main">
  {% block content %}
    Put some custom content here
  {% endblock %}
  </section>
  <script src="/static/js/project.js"></script>
</body>
</html>
```

Define a block that can be customized

Template inheritance

- Then in your child page (like index.html)
  - Override any **block** tags you want

```
{% extends "base.html" %}

{% block content %}
  <h1>Hello world!</h1>
  <p>Welcome to my website</p>
{% endblock %}
```

Everything in here will replace whatever is in the base.html block with name "content"

- Docs – Template inheritance

Example template

```
{% extends "base.html" %}

{% block content %}

  <h1>{{ user.name|upper }}</h1>

  {% for field in user_fields %}
    {% include "partials/field_display.html" %}
  {% endfor %}

{% endblock %}
```

Example template

```
{% extends "base.html" %}

{% block content %}

  <h1>{{ user.name|upper }}</h1>

  {% for field in user_fields %}
    {% include "partials/field_display.html" %}
  {% endfor %}

{% endblock %}
```

Print the variable "user.name" and transform it using the "upper" filter

Example template

```
{% extends "base.html" %}

{% block content %}

  <h1>{{ user.name|upper }}</h1>

  {% for field in user_fields %}
    {% include "partials/field_display.html" %}
  {% endfor %}

{% endblock %}
```

Loop through a list of items, e.g. user data

Example template

```
{% extends "base.html" %}

{% block content %}

  <h1>{{ user.name|upper }}</h1>

  {% for field in user_fields %}
    {% include "partials/field_display.html" %}
  {% endfor %}

{% endblock %}
```

Insert another template here

Include

- The included template will use the same context as the original template (as well as variables created in a for loop)
- I usually call these mini-templates "**partials**" and put them in a special folder
- You can pass additional variables to partials:

```
{% include "header.html" with admin=user.is_staff %}
```

Poll (Single choice)

- Which area do you want to cover next?
  - Settings
  - Models
  - URLs
  - Views
  - Templates
  - Admin
  - Forms
  - Other (say in chat)

Forms

Django Forms

- You can define forms in Django using Python
- Render them in the template in different formats
  - {{ form.as_p }}
  - {{ form.as_ul }}
  - {{ form.as_table }}

- It can take a lot of custom CSS to make any of these look good

- Use a CSS framework if you want easy form styling
  - Popular option: django-crispy-forms with Bootstrap
    - {{ form|crispy }}
  - I'm using django-bulma with Bulma
    - {{ form|bulma }}

Form

- You can create forms by defining fields and widgets

```
from django import forms

class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    message = forms.CharField(widget=forms.Textarea)
    sender = forms.EmailField()
```

ModelForm

- You can also create them magically from models

```
class QuestionForm(ModelForm):
    class Meta:
        model = Question
        fields = ("text", "details")
```

HTML

- In a template, you can use {{ form.as_p }} to render a form

- Or you can access fields individually

```
{% for field in form %}
    <div class="my-field-class">
        {{ field.errors }}
        {{ field.label_tag }} {{ field }}
    </div>
{% endfor %}
```

- It's hard to add classes to your field labels and inputs, so:
  - Use an extension built for your CSS framework (like django-crispy-forms)
  - Use a generic extension (like **django-widget-tweaks**) to add your own classes

Widgets

- You can also customize how a form field is displayed in HTML by editing its corresponding widget on the Python side

```
class Meta:
    model = Question
    fields = ("text", "details")
    widgets = {
        "details": forms.Textarea(attrs={"rows": 3})
    }
```

Cleaning

- Your form objects have certain methods that you can use and overwrite
  - form.clean()
  - form.clean_<field>()
  - form.save()

Poll (Single choice)

- Which area do you want to cover next?
  - Settings
  - Models
  - URLs
  - Views
  - Templates
  - Admin
  - Forms
  - Other (say in chat)

Course wrap up

Phew! That was a lot of info!

- Since this is my first time trying to teach Django, I put more info here than is necessary for an introductory class
- I will move some content to an intermediate class based on feedback from this one
- If you have suggestions, questions or want to discuss the content, post on the Discussions page in the GitHub repo:
  - https://github.com/ariannedee/intro-to-django/discussions

Recommended follow-up

Videos

- Web Development in Python with Django: Building Backend Web Applications and APIs with Django
  - Andrew Pinkham

Books

- A Wedge of Django – Daniel and Audrey Roy Greenfield
- Mastering Django – Nigel George

Beginner Live Trainings by Arianne

- **Introduction to Python Programming**
  - Variables, functions, conditionals, lists, loops
  - Skill level – 1/10
- **Programming with Python: Beyond the Basics**
  - Dictionaries, exceptions, files, HTTP requests, web scraping
  - Skill level – 2/10
- **Python Environments and Best Practices**
  - Virtual envs, testing, debugging, PyCharm tips, git, modules
  - Skill level – 2/10
- **Hands-on Python Foundations in 3 Weeks - *NEW***
  - Multi-week course that covers most of the above material
  - Skill level 1-3

Intermediate Live Trainings by Arianne

- **Object-Oriented Programming in Python**
  - Classes, dunder methods, and decorators
  - Skill level – 3/10

- **Introduction to Django: a web application framework for Python**
  - Building web apps in Django – starting a project and high-level overview
  - Skill level – 4/10

- **Rethinking REST: A hands-on guide to GraphQL and queryable APIs**
  - GraphQL APIs in Django and Node.js
  - Skill level – 5/10

Video courses by Arianne

- **Introduction to Python LiveLessons**
  - Very beginner content w/ brief intro to data analysis and web development in Flask
  - Link
- **Next Level Python LiveLessons**
  - Setting up Python projects with virtual environments and git
  - More fundamentals (dictionaries, exceptions, file handling)
  - Testing, debugging, and understanding modules
  - Create a web scraper
  - Link
- **Rethinking REST: A hands-on guide to GraphQL and Queryable APIs**
  - Link

Learn more

- Learn Class-based views
- REST APIs in Django Rest Framework
  - Intro tutorial
  - Video course
- GraphQL APIs in Graphene
  - Docs
  - Video tutorial
- Using Django with React
  - Tutorial

Some good Django reading

- General discussion of Django from developers
- Django Class-Based Views were a mistake