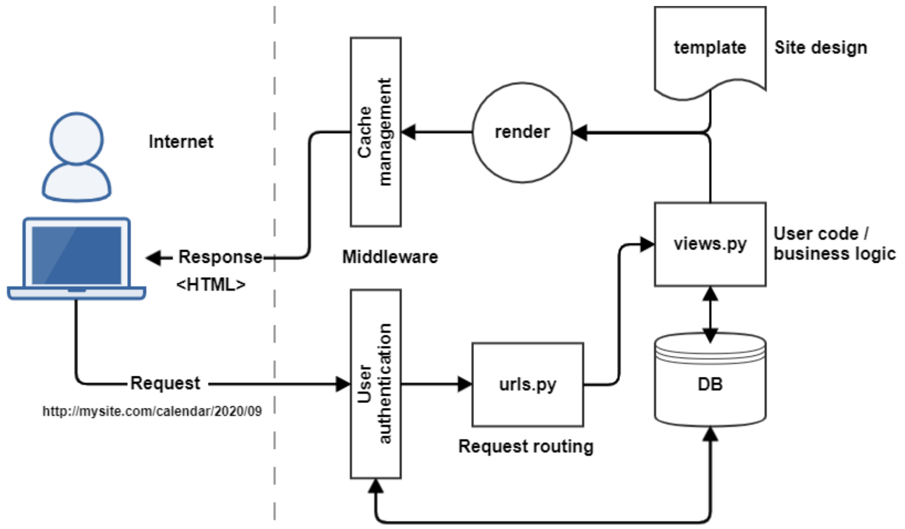
Django

Django is a "full-stack" web app system first released in 2005.

- database management
- user administration
- site data / content editing
- HTML generation
- HTTP request / response handling
- security
- URL mapping

All done by writing Python, except for HTML / CSS / JavaScript if needed (JS may not be).



Versions

Django	Python	Year
0.9	?	2005
1.0	?	2008
1.2	2.4, 2.5 and 2.6, (2.7)	2010
1.9-1.10	2.7, 3.4, or 3.5	2015-2106
1.11	2.7, 3.4, 3.5, 3.6, or 3.7	2017
2.0	3.4, 3.5, 3.6, and 3.7	2017
2.2	3.5, 3.6, 3.7, and 3.8	2019
3.0-3.1	3.6, 3.7, and 3.8	2019-2020

3.x is starting to support Python's `async` execution model.

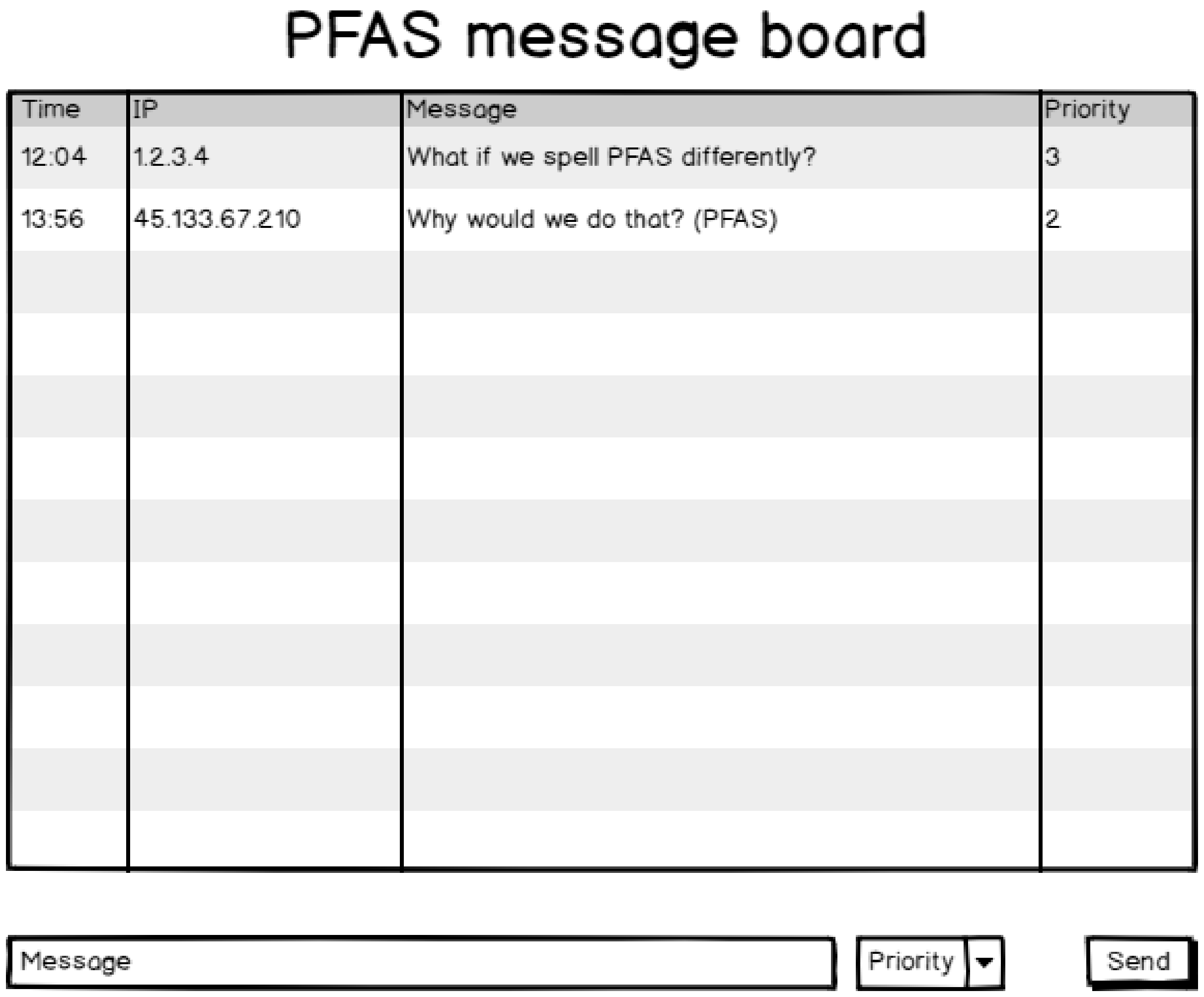
Features

Feature	Django	Flask
URL routing	✓	✓
Request middleware	✓	⊕
Response middleware	✓	⊕
ORM	✓	✓

Templates	✓	✓
Logging	✓	✓
Schema migration	✓	?
Forms	✓	✗
User authentication	✓	✗
Admin. interface	✓	✗
Geographic data	✓	✗
Lightweight	✗	✓

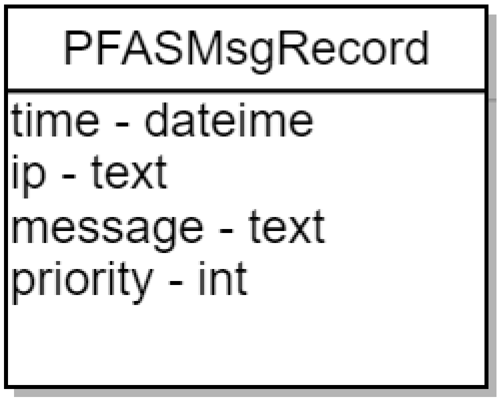
A really simple app.

Wireframe mock up


PFAS message board

Time	IP	Message	Priority
12:04	1.2.3.4	What if we spell PFAS differently?	3
13:56	45.133.67.210	Why would we do that? (PFAS)	2

Database design



API design

Application Programming Interface - apps. getting data / sending instructions to other apps.

A web service REST API uses message types GET (get data without changing anything), POST (create), PUT (modify), DELETE

`GET /api/v0/msgs` - return all messages (as JSON)

Getting started

`conda create -n djangodemo 'django>=3'`

`conda activate djangodemo`

`django-admin startproject pfasmsg`

creates

```
pfasmsg/
pfasmsg/manage.py
pfasmsg/pfasmsg
pfasmsg/pfasmsg/asgi.py
pfasmsg/pfasmsg/settings.py
pfasmsg/pfasmsg/urls.py
pfasmsg/pfasmsg/wsgi.py
pfasmsg/pfasmsg/__init__.py
```

`cd pfasmsg`

`django-admin startapp pfasmsgui`

creates

```
pfasmsgui/
pfasmsgui/admin.py
pfasmsgui/apps.py
pfasmsgui/migrations
pfasmsgui/migrations/__init__.py
pfasmsgui/models.py
pfasmsgui/tests.py
pfasmsgui/views.py
pfasmsgui/__init__.py
```

Navigation

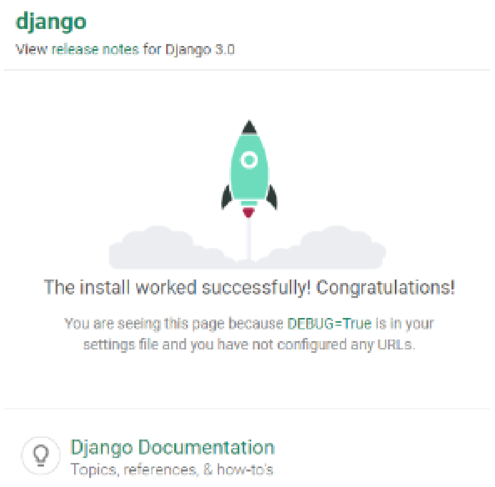
`Blue_text.py` indicates files to view in the associated source code while reading these notes.

Initial setup

edit **`settings.py`**, add `pfasmsgui` in `INSTALLED_APPS` and set up DB, but default SQLite settings ok

run the server, just to see it works

```
python manage.py runserver
```


django
View release notes for Django 3.0
Django Documentation
Topics, references, & how-to's

The install worked successfully! Congratulations!

You are seeing this page because DEBUG=True is in your settings file and you have not configured any URLs.

create **`pfasmsgui/urls.py`** and include in **`pfasmsg/urls.py`**

set up `simple_test` in **`pfasmsgui/views.py`**

Assembling the parts

At this point several pieces (model, template, view, DB setup) need to work together.
Doesn't matter too much which you start with.
You can test each is working as you go along, but in this demo all are shown as complete at once.

- add templates **`pfasmsgui/templates/pfasmsgui/base.html`** and **`pfasmsgui/templates/pfasmsgui/main.html`**
- set up MessageBoard view **`pfasmsgui/views.py`**
- edit models.py to define model **`pfasmsgui/models.py`**
- (got table PFASMsgRecord doesn't exist error)
- run migrations
    - python `manage.py makemigrations pfasmsgui`

```
Migrations for 'pfasmsgui':
pfasmsgui\migrations\0001_initial.py
- Create model PFASMsgRecord
```

- `python manage.py migrate`

migrate command output

```
Operations to perform:
Apply all migrations: admin, auth, contenttypes, pfasmsgui, sessions
Running migrations:
Applying contenttypes.0001_initial... OK
Applying auth.0001_initial... OK
Applying admin.0001_initial... OK
Applying admin.0002_logentry_remove_auto_add... OK
Applying admin.0003_logentry_add_action_flag_choices... OK
Applying contenttypes.0002_remove_content_type_name... OK
Applying auth.0002_alter_permission_name_max_length... OK
Applying auth.0003_alter_user_email_max_length... OK
Applying auth.0004_alter_user_username_opts... OK
Applying auth.0005_alter_user_last_login_null... OK
Applying auth.0006_require_contenttypes_0002... OK
Applying auth.0007_alter_validators_add_error_messages... OK
Applying auth.0008_alter_user_username_max_length... OK
Applying auth.0009_alter_user_last_name_max_length... OK
Applying auth.0010_alter_group_name_max_length... OK
Applying auth.0011_update_proxy_permissions... OK
Applying pfasmsgui.0001_initial... OK
Applying sessions.0001_initial... OK
```

The admin interface

`http://127.0.0.1:8000/admin`

`python manage.py createsuperuser`

edit **`pfasmsgui/admin.py`** to expose PFASMsgRecord in admin interface

Manage.py commands

`python manage.py -h`

List of manage.py subcommands

```
Type 'manage.py help <subcommand>' for help on a specific subcommand.

Available subcommands:

[auth]
changepassword
createsuperuser

[contenttypes]
remove_stale_contenttypes

[django]
check
compilemessages
createcachetable
dbshell
diffsettings
dumpdata
flush
inspectdb
loaddata
makemessages
makemigrations
migrate
sendtestemail
shell
showmigrations
sqlflush
sqlmigrate
sqlsequencereset
squashmigrations
startapp
startproject
test
testserver

[sessions]
clearsessions

[staticfiles]
collectstatic
findstatic
runserver
```

The API app

`python manage.py startapp pfasmsgapi`

Equivalent to `django-admin startapp pfasmsgui` used before.

Remember to add `pfasmsgapi` to `INSTALLED_APPS` in **`settings.py`**

Create `pfasmsgapi/urls.py` and include in **`pfasmsg/urls.py`**

Add `msgs` function to **`pfasmsgapi/views.py`**

No change to model / data / DB, so no need to generate / run migrations.

Details

URL parsing

URLs can be broken up into routing and parameter information.

http://mysite.com/events/calendar/2020/09

```
urlpatterns = [
    path('events/calendar/<int:year>/<int:month>', views.calendar),
]
...
def calendar(request, year, month): ...
```

Django uses Python @decorators to mark user login / rights requirements for functions in views.py.

ORM

Object Relational Model - wraps DB interaction in Python objects.

`all_msgs = PFASMsgRecord.objects.all()` - a QuerySet object, uses **lazy loading**, so you can do this without dragging a gazillion records into memory.

This fetches / saves records one at a time, **+** no memory limit, **-** a lot of DB access. Sometimes batch processing is best.

```
for msg in all_msgs:
    msg.ip = '8.8.8.8'
    msg.save()
```

The ORM uses double underscores to mark "operators" like "lte" "Less Than or Equal", "icontains" "Contains Ignoring case", etc. Full list.

`early_msgs = PFASMsgRecord.objects.filter(timestamp__lte=datetime(2000, 1, 1))`

`bad_early_msgs = early_msgs.filter(message__icontains='literally')`

or just

```
bad_early_msgs = PFASMsgRecord.objects.filter(
    timestamp__lte=datetime(2000, 1, 1),
    message__icontains='literally'
)
```

`bad_early_msgs.query` - the ORM is writing SQL to interact with the DB

```
SELECT "pfasmsgui_pfasmsgrecord"."id", "pfasmsgui_pfasmsgrecord"."ip",
       "pfasmsgui_pfasmsgrecord"."timestamp", "pfasmsgui_pfasmsgrecord"."message",
       "pfasmsgui_pfasmsgrecord"."priority" FROM "pfasmsgui_pfasmsgrecord"
 WHERE ("pfasmsgui_pfasmsgrecord"."message" LIKE %literally% ESCAPE '\' AND
        "pfasmsgui_pfasmsgrecord"."timestamp" <= 2000-01-01 00:00:00)

-- hand written SQL may be simpler...
SELECT * FROM "pfasmsgrecord"
 WHERE message LIKE %literally% ESCAPE '\' AND
       "timestamp" <= 2000-01-01 00:00:00
-- but ORM syntax is often clearer / simpler
```

The ORM also wrote SQL to create the tables during the migrations step.

```
CREATE TABLE IF NOT EXISTS "auth_group" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "name" varchar(150) NOT NULL UNIQUE
);
CREATE TABLE IF NOT EXISTS "pfasmsgui_pfasmsgrecord" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "ip" varchar(40) NOT NULL,
    "timestamp" datetime NOT NULL,
    "message" varchar(140) NOT NULL,
    "priority" integer NOT NULL
);
CREATE TABLE IF NOT EXISTS "django_session" (
    "session_key" varchar(40) NOT NULL PRIMARY KEY,
    "session_data" text NOT NULL,
    "expire_date" datetime NOT NULL
);
```

Connections to multiple databases at once are possible.

Migrations

If you change you model (add fields, tables, constraints) you can repeat the `python manage.py makemigrations` / `migrate` step to update the database. These migrations can be applied on other deployments of the app. to update their DB's as well. Migrations used be a separate component called `south` (ha ha) but are now core Django.

Validators

Fields in Forms (and Models?) can have validators that will reject bad input. Easy to make web forms that push back input until it's valid.

Forms, bound and unbound

The demo used an unbound form that was empty and accepted data to create a new record. Forms may also be bound to a particular existing record, to allow updating of that record.

Static files

`python manage.py collectstatic` efficiently collects static files from all apps in a project.

```
myApp/static/myApp/css/page1.css
myApp/static/myApp/js/do_thing.js
otherApp/static/css/main.css
otherApp/static/img/nav/arrow.png
app3/static/css/main.css
app3/static/js/do_thing.js
```

would be collected into

```
static/myApp/css/page1.css
static/myApp/js/do_thing.js
static/css/main.css
static/img/nav/arrow.png
static/css/main.css
static/js/do_thing.js
```

which is why you should put all you app's static assets in a folder named after your app, see how the two main.css files are in conflict.

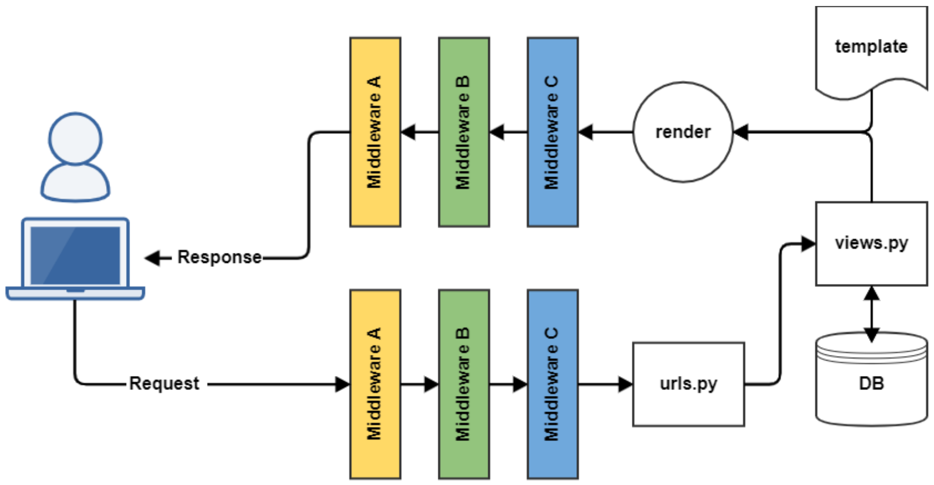
This collection of files allows a high performance web server (Apace, nginx) to serve these files natively, without pointlessly running the the Django subsystem.

DEBUG mode

By default, projects start of in `DEBUG` mode, which gives informative (i.e. insecure) traceback info. in the browser when things go wrong. Set `DEBUG = False` in `settings.py` to get uninformative standard HTTP errors (500 usually).

Middleware

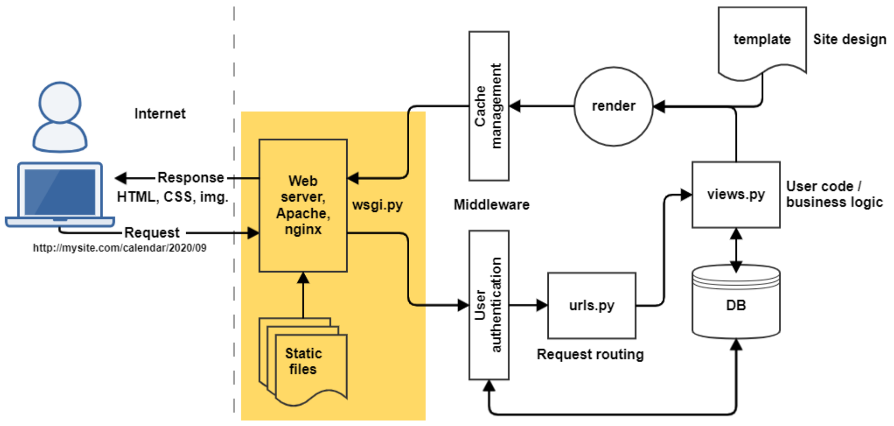
Middleware lets you alter incoming requests and outgoing responses. Useful e.g. for logging all requests.



Middleware layers take turns in reverse order on the incoming request and outgoing response. Middleware does not have to act on both request and response. Django uses several layersof middleware by default (authentication, CSRF protection, etc.). Layers are listed (and ordered) in `settings.py` .

In the demo app. you could for example append " (PFAS)" to all incoming messages that didn't mention "PFAS".

Production deployment



The Django Tutorial

The Django Tutorial builds all these pieces up one by one and is worth working through as an exercise.