Python Notes

Django Notes

Django separates things into different projects, each containing one or more applications which can interface and execute certain functions to build out the overall functionality that you're looking for. It automatically creates a bunch of files that have some innate functionality for you to use from the get go, and this contributes to the "overhead" that you have to get used to when starting to use it, more powerful though it may be. These sub-files immediately created are called "project components"

after you've initialised all the django files by doing "django-admin startproject name", you need to initialise an actual app within that project via "python manage.py startapp name". at this point, you'll have a directory called "name" inside your project, and when you navigate into it, a bunch of files set up for you, including migrations (for ORM), and models/tests, amongst others. Here, you have views.py (already created), which is the equivalent of app.py when making a flask application (that is, where you write you main back-end code).

you'll have an extra urls.py file for the application as well as one for the project as a whole. this is so that you can create the equivalent of app.routes; that is, you can tell django you want to associate a certain route/ path with a certain view (or, more specifically, a certain function inside views.py). there, you can write:

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

urlpatterns = [
    path("", views.index)
]
```

and then link the *project's* URLs.py with that of the application. this is because each app within a project will have its different routes, and so this project urls.py is where you get routed to the correct app in the first place when hitting an endpoint. something you have to be very careful about is the naming and placing of files with django. django looks for specific filenames by default, and if your views is in the wrong directory (of which there are may layers), you'll run into an error—so you have to be very systematic about these things.

you can do the ORM equivalent of Flask-SQLAlchemy in the models.py of your django application. You'd create a SQL database by writing:

```
class Flight(models.Model):
    origin = models.CharField(max_length=64)
    destination = models.CharField(max_length=64)
    duration = models.IntegerField()
```

and then going to CLI and migrating over the changes to the file (make sure your project knows about the models via the settings file by adding *'flights.apps.FlightsConfig'*). This is done by python manage.py makemigrations which then gets ready to update the database. To actually make the change, just do *"python3 manage.py migrate".*

you can *def __str__(self):* to define what the string representation of a certain object looks like when displayed in a shell or on an html template (when otherwise you might be returned "query Object 2", which isn't very informative).

it is convention to pass information into the front end (eg an html template) as a variable (dictionary) called context, which is passed in as a final argument in the render() function, of which you should make sure not to forget including the request.

the process of 'namespacing' involves adding the directory name before a file—so /flights/x.html instead of / x.html because you'd have multiple apps in one large templates file then and you could have multiple x.htmls for each app—so make sure you put x inside its app 'namespace', in this case, '/flights'

managing databases and constructing functionality around those in Flask was non-trivial, where you had to thin through the data processing and build web pages carefully around it, maybe with a little help from Adminer, a GUI. Similarly, but more powerfully, you have admin on Django. Just import then admin.site.register(className) to have it up on the interface. you can even customise the interface to be able to make certain addition operations easier and more intuitive by changing the settings/adding code to admin.py, where previously we only registered certain classes with the admin interface.

you have the *try: except*: clauses where you can do something and have validation readymade incase something doesn't work (where the error follows the except clause, by doing something like *raise*).

Django is so powerful because it abstracts so much away, so you can create much more complex and functional applications with the same lines of code compared to Flask. An example of this is in how it handle database manipulation—you create classes in your models.py and then just migrate them, specifying as an argument in the class properties how you want the classes to be related (eg, have flights be related to passengers and vice versa so you can access passengers on a flight as well as flights for a passenger) and it will understand that, and create 2 tables under the hood to keep track of the many to many relationship, so you can simply do f.passengers() to get at what you're looking for in the future. With flask, you would've had to create those tables after thinking about which to create and with what columns and how, and spent a lot more time "in the weeds".

errors will always have the stupidest origins. for example, the reason I was having so much database trouble was because sqlite wasn't told to cascade the deletion of things in one table to things in another table because my on_delete argument of the class declaration was in quotes when it shouldn't have been. this is why, when you're constructing your code in the first place, you have to be super careful from the get-go to avoid hours spent painfully debugging afterwards.

the "return" function breaks out of the current code set you're in—so in the case of a django app it breaks out of the current function/route you're operating in and moves onwards if it's triggered

To paginate the flights we created in the DB, we simply needed to 1) make a URL path for that in the flights.url corresponding to some function we wanted to call in views, then 2) create that function that takes in the flight ID from the URL int in paths, and queries the DB to get the flight as an object then pass that via a context variable to the front end and 3) make a template that fleshes out all the information in the way we want.

Besides the authentication, CMS, chat, and other out-of-the-box functionality, Django is great because it abstracts so much database manipulation away. When thinking about the schematic relationship between passengers and flights, we don't have to worry about designing the table and fields that specify the relationship, only the *type* of relationship between the objects (Passenger/Flight object is ManytoMany) and then it knows all the right tables to create behind the scenes, and does that seamlessly. Isn't that awesome!

Interesting Django feature for ORM search: Question.objects.filter(question_text__startswith='What')

Django automatically creates a "n_set" that you can use .create() with when you establish a ForeignKey relationship; for example, if you make the Choice object a ForeignKey to question within it, then you can access q.choice_set.create(choice_text='', votes=) because you set question to be the ForeignKey of choice.

If you want to by filter or get something specific, like the *year* of a question's publication date, you can use double underscore to go down layers: Question.objects.filter(pub_date__year=current_year)

Views are not just to do with what pages you see, but also POST functionality with forms that you *don't* see —the voting functionality of the polls app doesn't correspond to any particular page but instead to the computation of registering a vote itself.

The *request* object is so necessary and ubiquitous because it contains embedded information about the session and cookies that each function and page needs to be able to make all the internet-y back-end things work seamlessly.

All a single view is responsible for is outputting either an HttpResponse of some sort, or raising an error. The business logic in between can use advanced algorithms, fancy libraries, cool file formats, but on an abstract

level, just needs to return an HttpResponse or an error (eg 404) of some sort so the site knows what to do when that URL pattern is tried.

Some neat functionality is using the “order_by” extension on the ORM, and you can just add a - to reverse the order, then splice off the latest questions in typical pythonic fashion via [:5]. Knowing how to hit the database API via ORM correctly while making full use of all the commands and filters at your disposal is crucial so you can select and order data from the database correctly and quickly.

You need to create the templates file in the app you want them in, since each app (web app as a black box) will have its own interesting templates, and an index for each of them.

You need to make sure to raise an exception after attempting to get any object in the business logic of a view in case there is an error in hitting the database or some other thing, then you need to error handle. For this purpose, the try: and except: is in place, or alternatively you can just use get_object_or_404(class, criteria to find instance) to abstract that away (get_list_or_404 serves a similar purpose).

When using HTML forms and moving data throughout the stack, the important attributes you should know are: have the data destination set by the <form> tag, with attributes entailing action=“” the URL you want to send it to, and action=“post”, obviously. If you’re going to use a list of radio buttons (done via Jinja-like for loop) then make the name= whatever you need to request.POST[‘name’] and the *value* is the value that that will then take on (you can use for.loopcounter if need be).

The KeyError exception is built in for the case that some input isn’t provided at all.

A neat feature/detail is *vote{{ choice.votes|pluralize }}* which makes the vote become votes based on the number of the them.

Django comes with list and detail views out of the box. Since hitting a database to fetch data based on paginated URLs is a very common web dev paradigm (virtually every project I’ve done involves it to some degree), it’s abstracted away to the point where you can literally write basic web apps without including *any* actual python you’ve written yourself! The basic method I’ve used so far in the early stages of this tutorial as well as in CS50W is just learning the underlying functionality so that you know when is a good idea to use this abstraction at all.

For each generic.ListView or generic.DetailView, you need to give the model you’re talking about (listing products, questions, or what), and for the detail view give information via the pk= specifies built in. These views also default to using templates of the given format: <app_name>/<model_name>_detail.html, and that can be overridden by simply pointing to (and same for list views): template_name = ‘pollsapp/results.html'

Because we’re using the word question as our context variable in the templates we can leave that to the DetailView default which just uses the model name, but with the ListView of the Question model, we’ve decided to use “latest_question_list” and so have to specify that via context_object_name = ‘latest_question_list' otherwise it defaults to question_list because of the same reason.

You have to include a second argument specifying the namespace attribute in the ‘include()’ function in the parent URLConf otherwise it’s not automatically recognized (a recent Django feature inclusion).

Small bugs will fuck you—I labored over why I kept getting a polls app is not a namespace error, and it turned out to be partly the above, and partly because in my app URLConf I used appname= instead of app_name=, this is why you have to be careful *while* writing code so you can debug as you go along while you understand what you’re doing instead of having to parse swathes of legacy code and want to kill yourself.

Using the CLI testing utility in the shell:

```
from django.test.utils import setup_test_environment

>>> setup_test_environment()
```

You can debug things as if you were a viewer of the page, and get response status codes, HTML, variables passed between the front and back-end, and get a lot of insight into the mechanisms that you wouldn’t be

able to get form a visual interface easily. This is to test entire views (pages)—to test individual functions just write unit tests in tests.py

get_queryset() is a built-in function you can override by defining explicitly yourself when using generic views to determine what data is passed into the front-end as context.

Wherever you put your templates, make the static folder int he same place, name spacing in the same way. Then to include style.css in states just add these two to a HTML template:

{% load static %}
<link rel="stylesheet" type="text/css" href="{% static 'pollsapp/style.css' %}" />

It's wonderful to behold how much Django abstracts away—with one:

search_fields = ['question_text']

In the admin.py inside of a model class, you can add search functionality by a certain field on data in your database out-of-the-box, and it handles the LIKE query behind the scenes, creation and management of databases, and more. sure, lots hasn't been abstracted away, but quite frankly, most that can be—since it isn't specific to any one type of app in particular—has, optimizing for ease of use without sacrificing versatility.

Since the existing Django templates were built using Django itself, you can navigate into the downloaded Django project, which includes these templates, and rewrite them yourself (which is how you change the text on the admin site header, for example)

A package is a set of self-contained files that can be reused for a specific function—a project can be a package (given it has an __init__.py file) and all apps are, by their very nature, also reusable. If you want to browse packages on python, you can do it on PyPI and on djangoprojects.org, as well as package your own project by putting it in a parent *django-projectName* directory and adding LICENSE, setup.py, setup.cfg, manifest.in as ancillary files. Even the very act of creating pretty documentation has been abstracted away with Django project *readthedocs. setup.py* has the sdist function you use for actually converting the whole thing to a package, which can then be shared with the world (pip3 install it in the future if you need to use it in one of your own python projects).

Artale Notes (Art Gallery E-commerce)

Kevin bai approach:

start by designing it ground up using photoshop with visual look
has experience translating look or image to html/css
he has his own library and process since he's built all this stuff from scratch himself
go on find a bootstrap template from online and then go from there
backend processes and creates data and just returns that to the backend
build on top of the current theme and pretend all existing code doesn't exist
first time setting up a domain name etc. is annoying first time—takes an hour up front and then a few days to load
writing backend myself is beneficial as a learning process
backend: figure out tables/rows for products/transactions and determining all relationships in database and setup database then make RESTful API
if you use AWS/GCP they can provide authentication procedures—they store all customer data
it takes him around 1 month of full time to build a food delivery app (incomplete)
there are libraries for shopping card, CMS
make an admin page w/ a password, specifically for this kind of use
CMS has a back and front-end part, need to factor it in from the beginning, what kind of stuff does each option help with? some are specific for e-commerce functionality, others are more general and will require work on your end

Artale core features needed:

- artists and products models, where artists make products (one to many)
- products have category tags as they are of different types (painting, sculpture, etc)
- shopping cart to put products into (w/ OAuth)
- checkout, billing & delivery functionality, with taxes and discounts/codes, w/ stripe and paypal
- PDF invoice generation
- chinese/english multilingualism supported
- join us form that gets sent to her/admin

Possible additional features for practise:

- recommendation of new products based on user history (ML)
- sorting products by name, age, price, etc on “our products”

Saleor pros/cons:

- easier to modify front-end, which I will have to do a bunch of
- too heavyweight? Has A/B, ML recommendations support, etc
- uses React and GraphQL, both of which I’ll have to then learn

Oscar pros/cons:

- inflexible—people complain it’s difficult to make small changes to back-end functionality like currency conversion

Main things I want to learn through building Artale:

- how to choose a framework and learn it fast in production
- how to architect a system without structured guidance
- learn implementation difficulties that come with deploying on local server and in production
- learn difficulties of working with a client continuously
- how to know what features to build and make design decisions on how to build them and which ones are cost and time-effective

Why I chose Saleor:

- quicker to get up and running, small project so need something with less learning curve
- headless and more customizability needed since this isn’t a standard e-commerce site, on the front-end too as she wants lots of custom javascript functionality
- opportunity for me to practice good, scalable design as well as learn new skills—integrating ML, honing my React, learning GraphQL, and more.

REST vs GraphQL API structure

REST API is really just the front-back end communication method I’ve learned from the beginning: the user uses the web client to click some button (like submitting a form) or access a page, and the browser issues a GET or POST request to the server, alongside a path (like the /vote path on POSTing a vote in django-polls) alongside some context, then the server uses this to make some change to the database (like the votes on the choices of a question) and replies with something (returning a redirect to another page, for example). This GET and POST with context and paths is what is referred to as the RESTful API structure. When someone says they’re “building a REST API” they just mean they’re building back-end business logic for some purpose that can be seen as a black box taking in structured data of some sort and outputting responses like the business output (to plug into a front-end) or exceptions like 404.

An alternative paradigm, developed by Facebook, is the GraphQL method, whereby you don’t have multiple verbs or paths, just one method—posting to the graphql endpoint. All the information is

present in the query you send, kind of like how you query a DB (except you're talking to the back-end, not a DB). A graphQL query may look like:

```
{
        query # this shows what type of request you're making—get/post/etc {
                user # analog of the endpoint you want to hit {
                        name
                        age
                }
        }

}
```

SECRET_KEY = H231JB29A7

Node.js v12.16.1 to `/usr/local/bin/node`
npm v6.13.4 to `/usr/local/bin/npm`

Make sure that `/usr/local/bin` is in your `$PATH`.

```
/Users/tanishqkumar/Library/LaunchAgents/
homebrew.mxcl.postgresql.plist -> /usr/local/opt/postgresql/
homebrew.mxcl.postgresql.plist
```