WORKFLOW ESSENTIALS & OPTIMISATION

Important bindings on VS Code:

- (standard like line duplication/movement)
- cmd+shift+k = delete line
- cmd+rn = rename
- cmd+ctrl+s = select all
- cmd+b = format all
- cmd+e = goto error
- cmd+t = terminal open
- cmd+w = close window
- cmd+m = zen work mode
- cmd+d = select current word
- shift+cmd+drag = drag a chunk of text
- ctrl+shift+r = refactor
- cmd+shift+t when in terminal = new term
- cmd+\ = split pane
- cmd+opt+l/r = jump b/w panes
- cmd+ctrl+l/r = move
- cmd+opt+u/d = add cursor
- alt+b = open in browser

cli:

- ctrl+w = delete word before cursor
- cd ../.. = move up two dirs
- !! = run my last command
- cat X = show me the output of running X
- pwd = current path
- top = current processes list

Import django snippets:

- iadmin
- register
- fint/fbool…
- iform
- Model(_full)
- qs
- mtext, mdate…
- imodels
- lv/dv
- isettings
- (in templates:)
- block
- comment
- ext
- for
- if

React snippets

- imp/imn/imd

exp
met
fr
anfn/nfn
fof/fin
cp/cs
imrc
impt
est
cdm/cdu/cwum
sst
props/state
graphql
clg/clo
—use ts if needed before these—
rfc/rcc

Git essentials:

echo = just echoes something you put in command line, used in system files to add command line functionality but also useful in the form "echo "text/changes" >> filename" to quickly add them to a file without entering it

init / clone = initializes local repo unconnected to any remote, so you have to make sure to set a remote that the local repo is associated with by using (you can also init on a sub-dir of current dir by adding dir name after init) / clone, like init, is meant to be a one-time operation, and sets the url repo from which you cloned as the remote.

add / commit / push = add puts it in the staging area, commit readies it for push, and push puts it on the remote that you're pushing to (you can also delete remote repos using "git push origin —delete branchName"

config / alias = config allows git settings changes and aliases are essentially git macros you can create to speed up your workflow eg "git config —alias.co checkout" makes co become compiled into checkout when git executes commands

remote / upstream / origin = a remote is any non-local repository / upstream is something that is further up the river that is the source for your repo, so wherever you forked from, eg saleor would be the upstream repo for artale, and you normally want to add both your upstream and an 'origin' to your local repo as remotes so that you can pull new changes from the former and push any changes you make to the latter (which is the actual final product you're working on)—you'd use git pull upstream master or git push origin branchName for this.

branch / checkout = how you make, check on, and switch between local branches, using options like -a -b -v

pull / fetch / merge = the former is the latter combined, fetch will get remote changes and add it as a branch to your master, then merge will combine that to your master so you have enacted all remote changes, and pull just abstracts those steps away

blame / diff = blame tells you who last modified a file and what changes they made, diff is better used as a cli tool to diff between two files

rebase = used for when you want to merge new changes in master with a feature branch, but use merge as rebase can be dangerous if you rebase to a public branch

tracking = just refers to which remote a certain local repo is attached to

reset = used often as "git rest —hard origin master" when you fucked up a feature and just want to go back to the last working version (but rare because normally you don't commit a feature until you're sure it works)

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
		}
	}

}
```

You send graphQL queries from your front-end to your back-end, which in turn communicates with the DB via some SQL (or noSQL) -esque language. But instead of GET/POST when a form is submitted on the front end, you can display data by button onSubmit sending a graphQL query and then manipulating the object fields you get back to, say, display products from a e-commerce DB on the front end, like I had to with Saleor.

The key advantage is you defining the structure of the data you get back—you define exactly what you want from the server, and you're given that, and nothing more, whereas you'd have to do lots of complex data wrangling and looping/manipulation to get it into the structure that you want if you were to use a REST API structure.

Important to note that you can create aliases, like the following, to get multiple pieces of information about the same object with the same argument:

```
{
	empireHero: hero(EPISODE: empire) {
		name:
	}
	jediHero: hero(EPISODE: jedi) {
		name:
	}
```

Operation types are either query, mutation or subscription (database hook) for some data). You can also create re-usable fragments of queries and use variable names that take in dynamic input to make dynamic queries (eg based on search input hitting the DB)

```
Query HeroNameAndFriends($episode: Episode = Jedi) { # like in TS
	hero(episode:  $episode) {
		name
		friends {
			name
		}
	}
}
```

Where you can remove the middle bit and abstract it away as:

```
Fragment heroAndFriends on Character {
		name
		friends {
```

```
                        name
                }
}
```

To make the original query:

```
Query HeroNameAndFriends($episode: Episode = Jedi) { # like in TS
        hero(episode: $episode) {
                …heroAndFriends
        }
}
```

So you avoid repetition. And since the whole point is to take in user input and hit the backend accordingly, and since you can't get different URLs via POST (since graphQL is centered around a single endpoint), you have to be able to dynamically change the structure of queries beyond this variability—we should be able to request different fields and sub-fields based on a certain state, and this is done using *directories.*

hero(episode: $episode) @include(if: $hero) where hero is a field in the variables JSON passed in as an argument to construct the query—make sure the query contains declarations of those variables you're going to pass in as well. But sometimes the data returned for a field is one of several types and there's no variable that you pass in to represent them (in the above examples say hero returns a Character type that is either human or droid and you want to query a different thing based on what it is, you're not passing any variable in that corresponds to human/droid) you use an *inline fragment*, as follows:

```
query HeroForEpisode($ep: Episode!) {
  hero(episode: $ep) {
    name
    ... on Droid {
      primaryFunction
    }
    ... on Human {
      height
    }
  }
}
```

And when you don't know the type that the DB will return, use __typename to get the list (see docs).

A mutation is just a convention for changing data that returns the data that was manipulated, just as POST was the convention for mutating data with RESTful APIs. An example:

```
Mutation createReviewForEpisode($ep: Episode!, $review: ReviewInput!) {
        createReview(episode: $ep, review: $view) { # remember you declared this somewhere
                stars
                commentary
        }
}
```

Which will then return the newly created review, freshly populated with the data stored in the variables that the front end passed into this mutation to hit the DB. You should also note that while queries can hit the DB in parallel, mutations are done in series to avoid race conditions.

The graphQL schema is simply a set of all things a user can request for, and when a request comes in, the fields in the request are compared against those in the schema before execution to ensure validity.

Schema and all requests are centered around the idea of types, and the vocabulary around them works as follows:

```
type Character {
  name: String!
  appearsIn: [Episode!]!
}
```

works as follows:

Whenever you send a request for a character, you can only ask for name/episode, which are its *fields*. Character itself is an *object* (type). Name is a *scalar* type which means it doesn't have sub-divisions, it's just a thing—an int, bool, float, string, enum (you define) or ID (you can also create new custom scalars that are linked to primitive scalars like Date is int/int/int etc). The way that you define these schema is what you need to do in the parent language (Python, Graphene in my case), even if you hit them using pure graphQL. In this case, appearsIn will return an array of episodes, and only episodes. You can also set arguments for the field in an object type like this, like in type Character you might add a field height(unit: HeightUnit = METERS) where setting it to meters by default. In the schema, you also need to manually define the query types that are acceptable and match them to objects you've created:

Type Query {
hero(episode: Episode): Episode # shows that inputting hero returns episode
droid(id: ID!): Droid
}

You can only call the Queries and Mutations that you have created for access to the DB—this is what Tomasz from Saleor meant about "creating mutations to manage your objects"—you have to find the file where Python is used (via Graphene) to create these mutations, and make a mutation for the Artist class that allows the front-end to change all the fields in the Artist model with arguments you want the user to pass in in the dashboard. The language you use to manage graphQL on your backend also determines things like how you define ENUMS etc, and how the manipulation of those objects is done under the hood.

Like TS, you can also define interfaces that you use to implement classes for abstraction (eg interface character used to define queries/mutations on type Droid and type Human). This relationship and sharing of traits between objects in graphQL makes it conceptually similar to an OOP language, and all of this is implemented under the hood via the concept of nodes and edges in a data structure below the hood to allow for data to move from the backend to front-end efficiently with this request structure and entry points.

When you have an object that could return multiple objects of different types (that you know), but you only want one, you can implement that with all of them returned, and just get null for the ones you're not interested in, or a more efficient way is to use *unions* where you return the union, and implement a resolver in the backend language (graphene/python in my case) to resolve the union and return the appropriate object. So if you query a field that returns your union type, you need to use the inline fragment notation (…on type1 return X, …on type2 return Y, etc) in the tutorial schema language (will vary with graphene) to return information.

You must be systematic when devising types in the schema and when making queries, to think in terms of JSON and what fields you've defined each object to return, and be incremental in only requesting things that you have explicitly defined to exist.

You can think of each field in a graphQL query as a function that takes in the type you inputted, and simply returns the field of the next type, so thinking in tiers and depth is important for understand what is returned, and why.

Although the requests you make are written in graphQL, the defining of the schema (creation of query, mutation, subscription and object types) is all done in your backend language with the help of a framework (JavaScript with Apollo, Python with Graphene, etc). You write the commands that go to the server that often you yourself write! This server is a mini-backend that takes in your command, looks at the schema, and uses various resolver functions for each of the types you've defined to return the right output based on business logic *that you implement*. The graphQL library is what you use to help implement these servers painlessly, the commands you write are in vanilla graphQL. The server here is the analogue of the backend logic you'd implement to take in the values from the front end and do ORM and model manipulation in Django via *thing = class.objects.filter(condition)* and *thing.manipulate* then *return thing* to the front-end in a RESTful analog.

In Saleor, one app name was *Collections*, which contained a models.py that defined the attributes that a Collection of products had, and when you execute this graphQL introspective query:

```
{
      __type(name: "Collection") {
    name
    fields {
      name
    }
  }
}
```

It returns all the fields that you defined in the models, which is how you'd interface between the front end (onPress=>mutation=>hits models=>hits DB=>returns through stack to make changes=>front end Reacts to change and updates automatically).

The introspection part of the documentation is supremely useful for understanding a new system, so use the commands there, as well as the codebase in atom, to fully understand the architecture of the system and how changes are made all the way through the stack, including using Graphene. Spend at least a day just mastering this so that you can make changes correctly the first time when you write your Graphene server/the mutations for managing the new models you created with Django.

Implementing a GraphQL server with Django and Graphene

Development of the server logic is centered around Schema, because that's the "common knowledge/protocol" on which the front/back-end agree on. The front end will send messages of a particular structure and the backend is hardcoded to parse messages of that structure and so will then query the DB, manipulate data and send it back to the front-end. Schema-driven development centers around defining types, and then creating resolvers to manipulate those types by hitting the DB and return a response as the query types dictate.

The graphQL tutorial used a bland "schema" language to display concepts like delaying types on the back-end of your stack, but the way it's actually implemented in Python is:

```
class Query(graphene.ObjectType) { # a graphene, not Django models, object
```

links = graphene.List(ListType) # this is in the links app of slacker-news where you can only query for existing links that have been put into the sqlite db so far, ListType was a type that was declared above as a django.ObjectType with the Meta set to the Link model in Django we created, and this is the syntax equivalent of [ListType] in the schema language of the tutorial }
You thus define all types and resolvers for those types in a schema.py file or equivalent to define both your schema and how those schema will be handled via resolver functions, and then interact with the front-end accordingly.

To create a mutation, you create a class inheriting from graphene.ObjectType and direct the field input from that to a new class you make inheriting from graphene.Mutation, and within that, write the fields you want to be outputted on calling the mutation, then define the arguments taken in, then define a *def* ***mutate****(self, info, url, description):* that will create the new object by hitting the db as well as run(self) to instantiate it based on information passed in (see both docs and the slacker-news example for clarity). When you run a mutation{} then, make sure to pass in fields after createLink(info) { fields } that represent the response you want from the mutation after changes.

You shouldn't ever use _ in a query or mutation, instead just use camelCase—just a convention that is programmed into graphQL.

**Args and **kwargs*

These are instruments to make functions and the things they act on, more versatile, general, and powerful. Args is just prepended by the unpacking operator, which means it will take the arguments you pass in, and put them into an (immutable) iterable. Kwargs = keyword args = named args which are a='2' compared to just 2 being passed in (a positional argument). This means you can do things like take in a dictionary and iterate over its keys or values.

Important use case examples:

```
# sum_integers_args_3.py
def my_sum(*args):
    result = 0
    for x in args:
        result += x
    return result

list1 = [1, 2, 3]
list2 = [4, 5]
list3 = [6, 7, 8, 9]

print(my_sum(*list1, *list2, *list3))
(gives 45)
```

```
And # extract_list_body.py
```

```
my_list = [1, 2, 3, 4, 5, 6]

a, *b, c = my_list

print(a)
print(b)
print(c)
 Gives
1
[2, 3, 4, 5]
6
```
powerful. Args is just prepended by the unpacking operator, which means it will take the arguments you pass in, and put them into an (immutable) iterable. Kwargs = keyword args = named args which are a=‘2’ compared to just 2.

Thinking in GraphQL and Relay

An important note about graphQL is that it separates product code and server logic—you can change what a React component asks for in a query without changing how that query is handled after you write the query class in the schema on the backend. The schema support any form of query (eg you can ask for just name of user or of name and email and id without having to hardcode all of these). This means that once you’ve set up the schema (which admittedly is more overhead than its REST equivalent.

If you were designing, from an architectural perspective, something like GraphQL, built with React in mind, you would want it to be declarative, as well as centered around the component architecture. Most products just need to fetch data for a view, then render the view by passing in the data fetched. You could make it so that a parent component fetches data for its children, but then you’d be coupling them—if you wanted to move a child elsewhere, you’d need to change more components than just that, architecturally/paradigmatically not quite in line with React, which touts independence between components.

The render() method jumps out as a time when we can make the API call—but of course if you did that (fetch data before rendering the whole page) then you’d have to make multiple calls (staged) because often components need certain data depending on the state/props passed into other components. We need to get all the data we need once, and up front (statically), in contrast to this. Instead, we let each component tell us what it needs as a graphQL fragment and augment them to make the call.

Relay is a JS framework for front-end data fetching from the graphQL schema and was built with react in mind. It is an alternative to Apollo, but the reason you saw the Graphene tutorial teach it is because it is made for large scale applications and requires a revamping of the API, whereas Apollo (the front-end data management framework Saleor uses) doesn’t require custom schema and therefore classic graphQL (what saleor has in its schema) is all that’s need to be understood.

In fact, Relay was designed specifically for React, by Facebook, and is the basis for quick data fetching in their new, revamped home page. If you want a listView component that needs more or less information, you simply change the front end mutation that is sent onClick or on

ComponentDidMount or whatever, and you get different data back without having to change any of the logic in the endpoints (django views, in this case).

DjangoObjectType is a graphene-django library function that takes in a django model and converts it to an ObjectType recognizable by graphene—but you're not limited to objects to query in Graphene, you can create custom types that can also be queried simply by declaring graphene.ObjectType—but since you inevitably have to fetch some data from the DB you end up using these as abstraction layers between the models and the external API, if you do end up using them ever at all.

Filtering functionality is taken from a library called django-filter, and is used when a React component has some data it wants to narrow its search by (eg if there are a thousand species in a DB and it just needs to list the ones in a genus in the component's children, it would filter by genus, and the ability to accommodate this filter would be need to be in the schema). Relay.Nodes and FilterConnectionFields are surprisingly part of the django-filter documentation/functionality.

You will see the concepts of "nodes" and "connections" and "edges" mentioned a lot around graphQL, often in the context of Relay. All of these are misleading names that arise from a central need of every web app: pagination and stuttering the fetching of data (from e-commerce store lists to the Facebook news feed). There are two main ways of pagination: cursor-based and limit/offset, where the latter involves breaking data in static chunks and serving them, which can be problematic (user can re-see or not-see certain data) if data is altered in between serves. Alternatively, cursor based pagination simply keeps track of where the user is in the data list, and that solves most problems, and so is a little more robust than the latter method of pagination, and is used at places like FB.

The reason the names for these pagination concepts are as they are is because Facebook, the developer of both GraphQL and Relay, centers its internal data design around graphs (particularly suited to social networks). If you think in terms of nodes as users connected by edges that signify friendships, it's clear that if you took a particular user in the social network and wanted to get their name, you could also query another property: their friends, which would be done by looking at the *name* property inside all of the *edges* it has connecting to it in the graph! And therefore when querying data with GraphQL and relay you see nodes with information (friends' info, in this case) inside the edges {}, as the edges{} refers to a particular property of the original node (the friend has other friends, and that's the property, in this case!).

And finally, a connection is a list of edges, just as how one type of connection to the friends is that of it to its friends, and another type of connection might be to all the posts they have liked.

If you think about how Facebook would've implemented these, the nodes would've had the properties of users, the edges would have had metadata (when did they like a post, was it a like, laugh, love, or what, etc, as well as storing information about the node it points to, etc). Now you can see how previously nebulous functions like DjangoFilterConnectionField are important (from django-filter) because they let you filter by the different pieces of information a node (User in FB's case) is related to such as "likes" or "friends" all of which are just lists of edges in the end. You can thus filter the data you get back from the API by connection: likes and things like that to only get data you need. In the case of users and posts, the implementation:

```
Class Query(graphene.ObjectType):
        all_posts = DjangoFilterConnectionField(PostNode)
```
Now makes sense because you're returning all the posts attached to a particular user.

Relay draws together the idea of a component and its data, so that you specify *within the component* what data it needs, and when components are compiled together the schema are collated.