Testing Notes

1. Django Tests

Django Tests is based on unittest. Shares the same syntax but extends the functionality into as below

1. Allows you to simulate a request to see what response is returned by a particular endpoint. Also allows you to check (a) status code of endpoint (b) context (c) what content is actually being returned

Note: When you are creating tests don’t need to be exhaustive. The idea is that when bugs are discovered, you will add an extra test to make sure that that bug will never turn up again

1. Selenium

Django Tests does not have the ability to test code that occurs on the client side. I.E. Any event handler that is attached to HTML elements cannot be checked whether that interactivity is correct.

Common way to conduct browser testing is via Selenium. Selenium can simulate a web browser and a user interacting with that browser via a web driver. A web driver allows a developer to programmatically control what goes on to a browser.

1. Selenium Set-Up

Each browser requires a special web driver to be installed. Please download these from the Selenium download page.

1. Selenium Commands

**File\_uri(uriName)**

Selenium needs a page’s URI in order to test it. Simple use the above method to get it.

**Driver.get(uri)**

This would tell the driver to actually go to that page

**driver.<htmlElement> and driver.page\_source**

Once you have the uri attached, then you can access any part of the DOM simply by just dot notation.

**Driver.find\_element\_by\_id(“increase)**

This works too to find HTML elements. You can assign this element to a python variable. If it is a button you can use increase.click().

1. CI/CD

Continuous Integration

* Frequent merges to main branch
  + Catch and resolve conflicts quickly
* Automated unit testing
  + Make sure each component – like function by function – works well.

Continuous Delivery

* Short release schedule
  + Incremental changes means faster time to market plus catching bugs earlier

1. Github Actions

Allows for certain actions to automatically trigger once something has been merged to a repo. I.E. A style check for code or running unit tests and having an email sent if they fail

Github actions is written in YAML. YAMlL has the following:

1. Key-value pairs
2. Arrays
3. Nesting between key and values i.e. within 1 key you can have 1 array which was values
4. Docker

Docker helps to standardize environments across servers and development machines to ensure that application actually works.

Docker creates a container where each container contains its own configurations. Applications are ran inside these containers instead on separate developer machines.

Docker is similar to a VM however a VM runs an entirely new OS while docker containers run on Host OS but has a layer in between applications and OS. Hence, they are lighter weight than VMs

Docker files describe instructions to create a docker image. Docker image contains all the configurations you want the container to have.

A common problem docker fixes is the issue of having a separate database. Databases in production are hosted on their own server such as in POSTGRSQL or MYSQL. However, your development database might be SQLite. Docker fixes this by having separate containers for your web application and your database.

Docker compose allows us to compose various service. I.E. allow 2 containers one running Postgres another running the web application to talk to one another

**Project Notes**

1. New Post

Index route should show a form that allows user to write text-base post. This post should contain 2 things (i) textarea for post content (ii) submit button with name post

Index route should also display the author’s post below

After writing a new post, then user should be redirected to their home page. Which should contain a feed of all their post

Tests required

1. Unit test on Django Model Method create\_post to ensure that tests are created
2. Integration test with Client to show that route is working and correct template is used with get request
3. Integration test with Client to show that form can successfully post to route and save data to model.
4. All Posts

New template where all posts should be shown in chronological order with most recent first. Each post should contain (i) username of poster (ii) post content (iii) date and time of post (iv) number of likes the post has which defaults to 0

Must apply pagination here as well

There should be logic to obtain all posts. Should this be in views? No. Can it be on Model? No cause it is not related to model. Perhaps a new package called PostReader -> Get all post.

Tests required

1. Check that method get\_all\_post is working
2. Check that the right template is being returned
3. Check number of likes
4. Username

Username should lead to profile page. Need the following information

* Number of followers of the user, number of people the user follows
* All posts in reverse chronological order i.e. most recent first
* IF current user is not host of page, have follow or unfollow button
  + Means require logic to determine whether (i) user is signed in (ii) user is viewing his own profile

1. Following

Same as All Posts but they can only see posts by users that the current user follows. Apply pagination here

1. Edit Post

Data Model

1. Tables to have are (i) User (ii) Post
2. User should contain the following
   1. Followers -> Many to many relationship. This should be self-referencing. A single user can have many followers. A follower can have many following.
   2. Following -> Same as above
   3. Check this
3. Posts
   1. Author -> User\_ID foreign key
   2. Post content -> Text
   3. Timestamp -> Date and Time

Logic -> Create new URL which accept user ID -> new view that takes a user\_id. -> Call the following (i) number of followers the user has (ii) number of people the user follows (iii) all post for user in reverse chronological order -> Should also check if user id in request is the user\_id in request page is the same as request. If yes, then add button to follow or unfollow -> Use method to check whether user is following or not following currently viewed user

Template -> (i) Template takes the following arguments (1) whether it is the same user (2) whether viewed\_user is being followed by viewing\_user (3) number of followers (4) number of following (5) all posts by that user

Tests

1. Get request should return a 200 status code
2. Check whether follow status is correct
3. Check whether posts found is correct [There should be 2 based on database]
4. Check whether templates called is correct

Selenium Tests

1. Check whether follow or unfollow button is displayed correct.y
2. Likes
   1. User\_Id
   2. Post\_id
   3. Timestamp

**Follow button**

Logic -> User press follow -> send the following information to api\_endpoint (a) user\_id of person viewing (b) user\_id of profile\_viewed. Information should also be sent as a POST request because you are trying to update the database -> endpoint should receive these 2 information and do the following (i) check whether they are current following each other. If they are then send error message back (ii) if they aren’t then follow each other and send back success message

Tests

1. Send get request to this endpoint with correct data. Then search database again to check whether it is correct

**Following**

Logic -> create new following view that takes no argument -> Within view first find all the people that the user follows. Then query Post model with all these arguments and add them to a list. Then arrange them according to chronological order. -> Pass these posts into a template called following.html -> Following html should just be a post-container [maybe rename to feed?] which then displays all the posts

**Like Button**

Logic -> Click like button. -> Create new like view that takes no arguments. API call in body should contain the post ID. If the user currently likes the post, then unlike it. If the user currently is not liking the post then like it. Return a JSON response of Error Or Message. Error if no such post -> Create event handler on button. It should call the API. Add the like to the page ->

OOP Design

1. User Object
   1. Given a post should like that post
   2. Given a post unlike that post
   3. Given another user, follow that user
   4. Given a user unfollower that user
2. Post object

Django Concepts

1. Making queries

QuerySet -> Reprsents a collection of objects from database. It can contain filters which narrow down the query results based on the given parameters. QuerySets roughly correspond to the SQL keyword SELECT and a filter is a limiting clause such as WHERE or LIMIT

Manager -> Every model has 1 manager and it is called objects by default. A manager is the main source of QuerySets for a model. They are basically an interface which database query operations are provided to Django models.

Queries therefore start like this

modelName.objects.filter(…..)

Example:

**>>>** Entry.objects.filter(

**...**  headline\_\_startswith='What'

**...** ).exclude(

**...**  pub\_date\_\_gte=datetime.date.today()

**...** ).filter(

**...**  pub\_date\_\_gte=datetime.date(2005, 1, 30)

**...** )

**Using get**

You can use the get method on manager if you know that only a single object will match your query. This would return a QuerySet containing a single element.

If there are no results, get will raise a DoesNotExists exception. If more than 1 it will raise a MultipleObjectsReturned

**Django Models**

1. On\_delete
2. Through
3. Related\_name

ForeignKey has a method called related\_name. This name specifies the name to use for the relation from related object back to the one that it is currently defined on. I.E. ForeignKey is defined on a Django model and it is used to create a many\_to\_one relationship with a another table where the model containing the foreign key is the parent. Therefore, using childTable.relatedKeyName would result in a row in the parent table that contains a reference to the child table.

It is also the default valkue for related\_query\_name which is the name to use for thereverse filter name from target model.

1. Get\_or\_create()

A method defined on Manager. Used to lookup an object with given arguments – can be empty wth models has default values for all fields – or can create one if unable to find one.

Returns a tuple of (object, created) where object is the retrieved or created object and created is a Boolean specifying whether a new object was created.

Meant to prevent duplicate objects from being created when requests are made in parallel and shortcut boilerplate code.

1. Create()

This is a method defined on Manager to create a new object, save it and put it on a related object set. Returns a newly created object. I.E. you don’t need to use the word save

**Django Model Form**

1. Validation

2 steps involved in validating ModelForm.

1. Validating the form
2. Validating the model instance

Validation for both is triggered implicitly when calling is\_valid() or accessing the errors attribute and explicitly when calling full\_clean() but using full\_clean() is normally not needed.

Any cleaning method will raise VAlidationError.

**Django Testing Tools**

1. Test Client

Client is a Python class that acts as a dummy Web browser. Allows testing of views and interact with Django powered app programmatically. Use the test client to

* Establish that the correct template is being rendered and the template is passed the correct context data

However, use Selenium to

* Test the rendered HTML and the behaviour of the web page such as their JS functionality.

1. How to Test Forms

Client can simply send post request to selected URL to test forms with information in the context. Then the context will check

1. How to Test Views
2. General Notes

**How does Django urls work?**

**Django Pagination**

Django has a built-in pagination class. It defines the number of posts to be displayed per page and retrieves those post that correspond to the page requested by the user.

Steps in views.py

1. First you have to instantiate the Paginator class with the number of objects that is to be displayed on each page
2. Get the page GET parameter which indicates the current page number -> i.E. This is done via page = request.GET.get(‘page’)
   1. Note request.GET is a QueryDict object that contains all the given HTTP GET parameters. By default it has the key page which is the page which
3. After getting requested page number, obtain the objects for desired page by calling page method on Paginator class.
4. If page parameter in request is not an integer, retrieve the first page of results. If parameter higher than the last page of results, retrieve the last page.
5. Pass page number and retrieved objects to template

Steps in templates

1. Create a template to display paginator so that it can be included in any template that uses pagination. Just copy the standard for Django documentation <https://docs.djangoproject.com/en/3.2/topics/pagination/>
   1. This template expects a Page object in order to render the previous and next links and display the current page and total pages of results.
2. Within whatever template that you want to add pagination, just add {% include "pagination.html" with page=posts %} This would add the template that we just defined.
3. Note when we say pagination we literally only mean the page number. The display of any contents on the paginated page still needs to be done manually.

**Django Queries Spanning Relationships**

Assuming

**from** **django.db** **import** models

**class** **Blog**(models.Model):

name = models.CharField(max\_length=100)

tagline = models.TextField()

**def** \_\_str\_\_(self):

**return** self.name

**class** **Author**(models.Model):

name = models.CharField(max\_length=200)

email = models.EmailField()

**def** \_\_str\_\_(self):

**return** self.name

**class** **Entry**(models.Model):

blog = models.ForeignKey(Blog, on\_delete=models.CASCADE)

headline = models.CharField(max\_length=255)

body\_text = models.TextField()

pub\_date = models.DateField()

mod\_date = models.DateField()

authors = models.ManyToManyField(Author)

number\_of\_comments = models.IntegerField()

number\_of\_pingbacks = models.IntegerField()

rating = models.IntegerField()

**def** \_\_str\_\_(self):

**return** self.headline

1. Spanning Single Value Relationships

In SQL, if you have a JOIN table, you would have to use the JOIN statement to Join the various tables that the JOIN table has references to make a new table and then query that table. Django ORM abstracts this.

Use the field name of related fields across models, separated by double underscores until you get the field that you want.

Example: To get all Entry objects with a Blog whose name is “Beatles Blog”

Entry.objects.filter(blog\_\_name=”Beatle Blog”)

modeName.objects.filter(fieldname\_\_fieldName)

In our above example, Entry table has a ForeignKey to the Blog table under the name blog. Blog table has another field called name. Hence, this is how you query a relationship across tables.

This spanning can be as deep as you like i.e. if Blog has another foreing key you can simply add \_\_ to span that relationship again. When you reverse a relationship in a lookup use the lowercase name of the model.

For example, retrieve all Blog objects which have a least one Entry whose headline contains Lennon

Blog.objects.filter(entry\_\_headline\_\_contains=“Lennon”)

1. Spanning muli-valued relaionships

When filtering based on ManyToManyField or ForeignKey, 2 types of filters exists. First, assuming you have a one-to-many relation, we might want to find 1 row in the parent table which has multiple rows in the child table with a specific filter. For example, Blog/Entry relationship above we want to find blogs that have an entry “Lennon” in the headline as well as entries that are published in 2008.

There are 2 types of queries

1. We want to find entries that have headline=”Lenon” AND published in 2008
2. We want to find entries that have headline=”Lenon” OR published in 2008

Django handles both based on specific logic in filter. Everything inside a single filter() call is applied to smultanously to filter out items matching all those requirements. Successive filter() calls further restrict the set of objects. However, for multi-valued relations, they apply to any object linked to the primary model and not only those that were previously selected by an earlier filter() call.

E.g.

For 1) The query is like this

Blog.objects.filter(entry\_\_headline\_\_contains=”Lennon”, entry\_\_pub\_date\_\_year=2008)

For 2) The query is like this

Blog.objects.filter(entry\_\_headline\_\_contains=”Lennon”).filter(entry\_\_pub\_date\_\_year=2008)

Note: In the above case, Blog model does not have a field with ForeignKey. However, Entry does not exactly 1 column with ForeignKey for Blog. Hence, you can just specify the table name entry.

The idea here is that when we use entry, we get a reference to the Entry table. With headline, we look at a specific field within the Entry table.

A good way to think of it is this. On the model that you are currently on, if you refer to a field that is a foreign key to another table, then you would get a reference to that other table. Once there, from that table, if you refer to another field, you would get a reference to another table.

1. Querying through relationships by reverse

In the above examples, you directly queried the table that contained the information you wanted. I.E. if you wanted all the posts by a specific user, you would query it like this:

Posts = Post.objects.filter(….)

However, you can query by reverse. This means given a User, how you find all the posts

User1 = User.objects.get(…)

Posts = user1.post\_set.all()

varName = <userInstance>.<modelName>\_set.all()

This assumes that the child table – Post in this case – only has 1 foreign key related to User.

Alternatively, you can use related\_name.

1. Querying Many To Many Relationship

Tag model is related to Product with a ManyToManyField. I.E. In Product, there is a field called tags which is a ManyToManyField with Tag model.

Products = Product.objects.filter(tags\_\_name=”Sports”)

When querying ManyToMany relationship you MUST query base on attribute name instead of model name.

**Django defining ManyToMany Fields with through\_name**

Normally when you define a ManyToMany field on a table, Django will automatically determine which fields of the intermediary model to use. However, sometimes Django will meet an ambiguous definition

Example

Assuming the following

1. There exists 3 models. (a) Person (b) Group (c) Membership
2. Group model contains an attribute called member which is a ManyToManyField that links to Person. Through an intermediary model called Membership.
3. Membership contains 3 attributes. (a) Group -> ForeignKey to Group (b) person -> ForeignKey to Person (c) Inviter -> ForeignKey to Person with related\_name=”membership\_invites”

In the above, Membership has 2 foreign keys to Person in the attributes person and inviter. This makes relationship to Group ambiguous. This is because when Django tries to instantiate the attribute member, it does not know which foreign key belongs Group (along with the group ID) and which foreign key belongs to Person independently.

Must use the through\_fields in Group to explicitly define which foreign Key to Person is linked to Group. Through\_fields takes a tuple, where the first element is the name of the foreign key to the model that ManyToManyField is defined on, and 2nd argument is the name of the foreign key to target model. In our case, target model is Person and model that ManyToManyField defined on is Group.

1. When to use through\_name

When you have more than 1 foreign key on intermediary model to any of the models participating in a many-to-many relationship you must specify through\_fields.

Applies to recurtsive relationships when an intermediary model is used and there are more than 2 foreign keys to the model or you want to explicitly specify which 2 Djanog should use.

I.E. In recursive relationship you need MORE THAN 2 so there must be 3 foreign keys.

**Tips on Writing APIs**

1. When you put parameters in a URL
2. CSRF

By default any POST request in Django would require a CSRF token. POST request to endpoints would not contain this so important to add a csrf-exempt decorator.

1. Setting up and organizing static files

In settings.py:

STATIC\_URL = “<appName>/static/”

1. When to put logic in JS vs. Backend
2. Format for sending and receiving APIs

Right now, we are calling the API with JSON.stringfy in Javascript. Basically this method converts a Javascript value to a JSON string.

Note: Although JSON says JavaScript object as its name, JSON is language agnostic. Syntax of JSON inspired by JS Object literal notation but differences exists.

Namely

1. All JSON keys must be quoted while object literals in JS are not necessary.
2. JSON only has the following values (a) string (b) number (c) object (d) array (e) literals such as (i) true (ii) false (iii) null
3. JSON quotes must be “” while JS can have both ‘’ and “”
4. JS accepts hexadecimals while JSON only allows decimal literals

In our POST request, the request body changes a JS value to a JS String. Hence, body is a JSON string.

1. **What is JSON string?**

All JSON exists as a string. It needs to be converted to a native JS object when attempting to access the data OR JS data needs to be converted to JSON in order to send across a network.

JSON.strinfy is basically a method to convert JS values into a JSON object.

1. **In views.py**

We must json.loads in python as it is a method to parse a valid JSOn string and convert it to a Python dictionary. Mainly used as deserialization.

Opposite of json.loads is json.dumps. Method used to convert a python value to JSON object.

1. **Summary**

Whenever we send information to an api view or an api view is sending information to JS, we are sending it in JSON format. This means that on both ends we need a process of serialization and deserialization.

Serialization and deserialization need to be done as well when we are testing responses. When we use client, we must specify the content\_type is JSON, the data must be JSON. The response we get back will also be a JSON object.