Models and their Fields (Django models map pretty much 1 to 1 onto SQL tables and fields onto columns. For more information, read [this](https://docs.djangoproject.com/en/5.0/topics/db/models/)).

If something is in italics it is not yet implemented and/or we haven't decided on whether we should implement it in that way.

The default primary key for a model is id, but you can always access an object's primary key by using <object>.pk. id is an integer field that is just sequential as objects are added to a model.

account application models:

**member:** the member model is used to store user account information relevant to authorization moments and content attribution.

* name: CharField, maximum length = 35
* ranking: SmallIntegerField, default = 1, choices =
  + 1 : "member"
  + 2 : "trusted member"
  + *3 : "organization"*
  + 98 : "moderator"
  + 99 : "admin"
* created: DateTimeField, auto\_now = false, auto\_now\_add = true
* email: EmailField
* pic = ImageField, default = "default/blankprof.png"
  + ImageField stores a string that is a filepath. right now when you provide an image to the model, it stores it to MEDIA\_ROOT / users / <pk> / profile.<file extension>
* about = TextField, default = ""
* *location*

**gLogIn**: The gLogIn model is used in the authentication moment when users log in via google one touch. The structure of this model can be replicated to implement other authentication methods

* googleID: CharField, max length = 255, unique = true
  + This ID is provided by google via the google one touch API and is unique to a particular google account
* refersTo: ForeignKey("Member")
  + this is the primary key of a Member instance

boiler models:

**message:** the message model is for anonymous contact with site admin

* sender: CharField, max\_length=75
* email: EmailField
* subject: CharField, max\_length=75
* message: CharField, max\_length=300
* created: DateTimeField, auto\_now=False, auto\_now\_add=True

Current Filterer/Widget setup (Jordan's work)

Filterer models

**PostFilters**: this is where the possible tags for posts are stored

* filter\_name: CharField, max length = 200
* filter\_desc: CharField, max length = 1000

**PostFilterAssoc**: in SQL, many to many associations can be difficult. you need the two tables that will have relationships and a third table, the "through" table, in which you store the myriad of relationships between the two tables you are trying to [here](https://www.sankalpjonna.com/learn-django/the-right-way-to-use-a-manytomanyfield-in-django) is a good explanation of how this works and how Django deals with it. in Django, you should be able to hide the through table unless you need to store additional information about a given relationship. Jordan has this set up to add a name to every relationship between a label and a post, which seems unnecessary. I think there's a simpler way to do this and I'll make a section for what I think we should do at the end of the document

* AssociationName: CharField, max length = 200
* Entry\_MTM: many to many field, associates with MapWidget
* Filter\_MTM: many to many field, associates with PostFilter

MapViewer models

**MapWidget:** this is where posts are currently stored. very barebones

* title: CharField, max length = 100
* description: TextField
* latitude: FloatField
* longitude: FloatField

I think we should get rid of the filterer application and integrate the functionality it has into the MapViewer application. It would streamline things and make the code easier to follow. The following is the schema as I would make it.

MapViewer models:

**PostTag**: just the list of tags, mod/admin would be able to add/remove tags as they saw fit

* name: CharField, max length = 25

**MapPost**: posts would be stored in this.

* title = models.CharField(max\_length=100)
* created: DateTimeField, auto\_now=False, auto\_now\_add=True
* content = models.TextField()
* author = models.ForeignKey("member", on\_delete=models.CASCADE)
* description = models.TextField() #will be derived from content. whenever this stuff gets working. first x characters of content stored here and displayed on map. necessary to be its own field for ease of use with the Django JSON Serializer
* *address*
* latitude = models.FloatField()
* longitude = models.FloatField()
* tags = models.ManyToManyField(PostTag, related\_name="mapposts")
* visible= models.BooleanField(default = false)

**PostComment:** haven't thought much about how this would work, so this is all hypothetical

* author: ForeignKey, member instance
* created: DateTimeField, auto\_now=False, auto\_now\_add=True
* content: TextField, max length = 150
* parentPost: ForeignKey, points to Post instance
* parentComment: ForeignKey, points to 'self' (points to comment model), allowed to be null, default is null

One of the things that I've been trying to think about is how to deal with the fact that we also want to allow user file uploads to their posts. I'm wondering if the clients are envisioning us building a collaborative blogging platform and are hoping to come out the other end with what is essentially an integrated word processor with in post embeds and such. If that's the case, we are most likely going to have to overhaul a lot of \this (maybe there's a django plug in we could use)

One way we could proceed is using a model like so:

**PostMedia**

* ParentPost: ForeignKey, relates to a MapPost instance
* Image: ImageField, stores an image
* *Other fields for video/other filetypes as needed*

Then when loading a post, check this model to see if it has associated media. if it does, render a slideshow type widget above the post's text. this would let us display multiple files in one post without worrying about how to dynamically display text for lots of files: ie display one at a time