Advanced API Concepts

Api class-based view:

* Previously I used method API view where I had to implement my endpoints using API method and I had to use if statement to separate methods.
* Class API view makes code more readable organized and maintainable.
* How?
  + By simply creating a class that inherit from APIView class and I need to import it.
    - From rest\_framework.views import APIView
  + Then create and implement logic inside my rest methods, get (), post (), put (), delete () and patch ()
  + Then in the urls.py I need to import the class and call method as\_view()
    - Path (‘my/url’, MyApiView.as\_view())
  + Each method takes following parameters:
    - Self.
    - Request.
    - Parameters that I need to use it inside logic.

Mixens

* When I am creating restful API, I usually follow the same pattern,
  + Queryset.
  + Serializer.
  + Response.
* The difference is that the queryset implementation and the sertializer.
* Mixens helps us by creating class that follow the same pattern for creating rest apis.
  + It helps me get list, create object, get one object, update and delete.
* I have mixen class for each one.

Concrete mixens

Concrete mixens are more complecated than mixens because they are implements previous mixens with generic class.

* I can have combined generics like List and Create which is get list of items or create new item.
* Or retrieve item and delete or retrieve and update.
* The generic mixen have two properties that I can use to define my serializer and my query:
  + query\_set.
  + serializer\_class
* they also have methods:
  + def get\_queryset(self)
  + Def get\_serializer\_class(self)
  + Def get\_serializer\_context(self)

Return {‘request’: self.request}

* If I have straight forward logic, I only use properties.
* If I want to perform extra logic on query or serializer I use methods.
  + Extra logic like create query based on user and user access roles.
* How?
  + First import:
  + From rest\_framework,generics import ListAPIView, ListCreateAPIView …….
  + Then implement either properties or methods.

Customize generic views

* If I want to override delete or put method because I have my own logic.
* I simply create my own delete or put method inside the class, and I will override it.
* Since I provide my query without id in case of item details generic class.
  + I can use lookup\_field = ‘parameter’ to provide my own variables

Viewsets Model

* When I have generic views that have similar code, I can use viewset.
* For example, I can get list of products also product details, I will get query duplicate and serializer duplicate.
* In this case I can use viewsets to combine these both generic views.
  + This is why called viewset
* How to use?

1. Import it:
   1. From rest\_framework.viewsets import ModelViewSet.
   2. Class ProductView (ModelViewSet)
   3. Queryset
   4. Serializer\_class
   5. Custom\_methods.
   6. Lookup\_field

* This class implement get list, get item details, create item, delete item, put item and patch
* To make it only view list or item I simply inherit from readonlyModelViewSet instead of ModelViewSet

Routers:

* When we combine two views the paths in urlpatterns gets corrupted since we delete their views.
* To server the ModelViewSet I need to import router from rest framework.
  + From rest\_framework.routers import DefaultRouter, SimpleRouter
    - SimpleRouter serves my collection perfectly.
    - DefaultRouter give me two extra functionalities:
      * If I add .json to the end of my url I get json view for my items and I can consume it.
      * If I go to the root of my apis I can see links of all my apis.

Nested routers

* Nested routers allow me to create urls like this: domain/{pk}/sub\_domain/{pk}
* For example, I have an item that have reviews the nested routes allow me to add list of reviews to one item.
* To use it I have to import a package first that give me nested routes
  + This URL contains the package: <https://github.com/alanjds/drf-nested-routers>
  + Importing it:
    - From rest\_framework\_nested import routers
  + It allows me to create a domain route and register the root routers.
    - Domain\_router = routers.DefaultRouter(). Or routers.SimpleRouter()
    - Domain\_router.register(‘domain\_endpoint’, views.DomainModelView)
  + Then create sub router and add to it the sub domain url name, ModelView and the lookup field which is the item details id.
    - Sub\_domain = routers.NestedSimpleRouter(‘sub\_domain\_end\_point’, views.SubDomainModelView, lookup= ‘item’) // item is item\_id
  + Then register both domain and nested routers to urlpatterns array.
    - Urlpatterns = [

Path (‘ ’, includes(domain\_router.urls)),

Path (‘ ’, includes(dub\_domain\_router.urls))

]

Create new sub domain details

* Logically if I want to add a sub domain to a specific item the id of that item detail should not be allowed to be added to the body (not added by serializer field)
  + but I want to take it from URL parameters from ModelView because it has access to the id of that object. That I receive from the nested sub domain.
* Since my ModelView receive that id I can get it from self.kwargs[‘item\_pk’].
  + SELF IS THE MODEL VIEWSET THAT CONTAINS THE OBJECT THAT HOLDS THE INFORMATION AVOUT REQUEST SUCH AS ITE’S PARAMETERS.
    - For example, if I want to to add extra logic like render data only related to specific object, I need it’s id.
    - I can get that id from self.kwargs.
    - Also, If I need to access the self-keyword I need a method which means I am going to override my methods to do filtering
  + I can use the self.kwargs to access parameters passed to the URL. then I can pass it to the serializer using get\_serializer\_context(self).
    - Get\_serializer\_context(self):

Return { ‘item\_id’: self.kwargs[‘item\_pk’]}

* Then inside serializer I don’t provide item detail id and Instead I override the create(self) method and use the parameter that was passed from context.

Def create (self, validated\_data):

Item\_id = self.context[‘item\_id’]

Return ModelName.objects.create(item\_id = item\_id, \*\*validated\_data)

* Then I do the logic with that parameter to create my object, or whatever logic I am implementing.
* And validated\_data is the rest of parameters that are received from the body request.