**DJANGO REST FRAMEWORK (DRF)**

**WHAT IS REST?**

* **RE**presentational **S**tate **T**ransfer (REST) is an architectural style that defines a set of constraints to be used for creating web services.
* **REST API** is a way of accessing web services in a simple and flexible way without having any processing.
* It’s used to fetch or give some information from a web service. All communication done via REST API uses only HTTP request.

**WHY REST?**

* **Lightweight**- One of the main benefits of REST APIs is that they rely on the HTTP standard, which means it’s format-agonistic and you can use XML, JSON, HTML, etc. This makes REST APIs fast, and lightweight — which is necessary for mobile app projects, internet of things devices, and more.
* **Independent** - Another benefit of REST APIs is the fact that the client and server are independent. In other words, the REST protocol separates the data storage and the UI from the server. This means that developers can work on different areas of a project independently and try out multiple developer environments as needed.
* **Scalable and flexible** - One of the third benefits of REST APIs, and perhaps one of the most important, is scalability and flexibility . REST APIs can be scaled quickly primarily due to the separation between the client and the server. Additionally, developers can also easily integrate REST APIs without much added work.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**SOFTWARE SETUP**

* Install Python, Django, Django Rest Framework.
* A database like PostgreSQL and Database software like Valentina Studio.

**FUNCTION BASED VIEW**

1. **Create a Project –**
   1. Run this command on terminal “**django-admin startproject fbvSerializers**”.
   2. Change directory to that folder “**cd fbvSerializers**” and then use command “**py manage.py startapp fbvApp**”.
   3. Open fbvSerializers in VS Code and open settings.py of project fbvSerializers then configure fbvapp and rest framework there.
      1. **INSTALLED\_APPS = [**
      2. **'fbvApp',**
      3. **'rest\_framework'**
      4. **]**
   4. Then scroll down and configure database.
      1. **DATABASES = {**
      2. **'default': {**
      3. **'ENGINE': 'django.db.backends.postgresql',**
      4. **'NAME': 'studentdb',**
      5. **'USER': 'postgres',**
      6. **'PASSWORD': 'root',**
      7. **'HOST': 'localhost'**
      8. **}**
      9. **}**
2. **Create the Model –**
   1. Open fbvApp models.py and the code should be like this.
      1. ***from* django.db *import* models**
      2. ***# Create your models here.***
      3. **class Student(models.Model):**
      4. **id = models.IntegerField(*primary\_key*=True)**
      5. **name = models.CharField(*max\_length*=20)**
      6. **score = models.DecimalField(*max\_digits*=10, *decimal\_places*=3)**
      7. **def \_\_str\_\_(*self*):**
      8. ***return* *self*.id+*self*.name+*self*.score**
3. **Migration –**
   1. Before doing the migration, we have to create the database **studentdb** which we mentioned in the database section of settings.py.
   2. Open Valentina Studio and create the database named **studentdb**.
   3. Use command “**py manage.py makemigrations**” and then “**py manage.py migrate**”.
4. **Serializers –**
   1. Goto fbvApp and create a file named serializers.py.
   2. Open it and the code should be this.
      1. ***from* attr *import* field**
      2. ***from* rest\_framework *import* serializers**
      3. ***from* fbvApp.models *import* Student**
      4. **class StudentSerializer(serializers.ModelSerializer):**
      5. **class Meta:**
      6. **model = Student**
      7. **fields = ['id', 'name', 'score']**
5. **Views.py of fbvApp (Defining functions) –** 
   1. Open **views.py** of fbvApp and define the student\_list function for GET and POST method, the code should be like this.
      1. ***from* django.shortcuts *import* render**
      2. ***from* fbvApp.models *import* Student**
      3. ***from* fbvApp.serializers *import* StudentSerializer**
      4. ***from* rest\_framework.response *import* Response**
      5. ***from* rest\_framework *import* status**
      6. ***from* rest\_framework.decorators *import* api\_view**
      7. ***# Create your views here.***
      8. **@api\_view(['GET', 'POST'])**
      9. **def student\_list(*request*):**
      10. ***if* *request*.method == 'GET':**
      11. **students = Student.objects.all()**
      12. **serializer = StudentSerializer(students, *many*=True)**
      13. ***return* Response(serializer.data)**
      14. ***elif* *request*.method == 'POST':**
      15. **serializer = StudentSerializer(*data* =  *request*.data)**
      16. ***if* serializer.is\_valid():**
      17. **serializer.save()**
      18. ***return* Response(serializer.data, *status* = status.HTTP\_201\_CREATED)**
      19. ***return* Response(serializer.errors, *status* = status.HTTP\_400\_BAD\_REQUEST)**
   2. Now to implement primary key based operations we will define student\_details function in **views.py** of fbvApp for GET, PUT and DELETE method, the code should be like this.
      1. **@api\_view(['GET', 'PUT', 'DELETE'])**
      2. **def student\_detail(*request*, *pk*):**
      3. ***try*:**
      4. **student = Student.objects.get(*pk*=*pk*)**
      5. ***except* Student.DoesNotExist:**
      6. ***return* Response(*status* = status.HTTP\_404\_NOT\_FOUND)**
      8. ***if* *request*.method == 'GET':**
      9. **serializer = StudentSerializer(student)**
      10. ***return* Response(serializer.data)**
      11. ***elif* *request*.method == 'PUT':**
      12. **serializer = StudentSerializer(student, *data* = *request*.data)**
      13. ***if* serializer.is\_valid():**
      14. **serializer.save()**
      15. ***return* Response(serializer.data)**
      16. ***return* Response(serializer.errors, *status* = status.HTTP\_400\_BAD\_REQUEST)**
      17. ***elif* *request*.method == 'DELETE':**
      18. **student.delete()**
      19. ***return* Response(*status* = status.HTTP\_204\_NO\_CONTENT)**
6. **Urls.py of fbvApp and fbvSerializers project (Configure and Test) –** 
   1. First of all, create urls.py file inside the app directory which is fbvApp and import views of fbvApp there and then add path in urlpatterns, one for GET and POST method and one for primary key method like this.
      1. ***from* django.urls *import* path**
      2. ***from* fbvApp *import* views**
      3. **urlpatterns = [**
      4. **path('students/', views.student\_list),**
      5. **path('students/<int:pk>', views.student\_detail)**
      6. **]**
   2. Now open urls.py of project which is fbvSerializers and import app views there and add path for urls.py of fbvApp it should look like this.
      1. ***from* django.contrib *import* admin**
      2. ***from* django.urls *import* path, include**
      3. ***from* fbvApp *import* views**
      4. **urlpatterns = [**
      5. **path('admin/', admin.site.urls),**
      6. **path('fbvApp/', include('fbvApp.urls'))**
      7. **]**
   3. After this now it’s time for testing. To do that first we have to make sure our server is running or not, if it’s not running use command “**py manage.py runserver**” . After that visit localhost or <http://127.0.0.1:8000/fbvApp/students/> now you can see we got a UI by default by Django and we don’t see any data.
   4. To add the data, open Valentina Studio and open **studentdb** then goto SQL Editor and run query **INSERT INTO “fbvApp studentdb” VALUES(‘1’, ‘John’, ‘100’)** to insert the student data into the database, add two three students there and you can see if its added or not by using **SELECT \* FROM "fbvApp\_student"**.
   5. Now if we refresh the localhost page, we can see the all the data there and if we add 1 or 2 etc at the end of url we can get that specific students records.
   6. We can also use POSTMAN software to test rest framework.

**CLASS BASED VIEW**

1. **Create a Project –**
   1. Run this command on terminal “**django-admin startproject cbvSerializers**”.
   2. Change directory to that folder “**cd cbvSerializers**” and then use command “**py manage.py startapp cbvApp**”.
2. **Model –** Copy the same code of fbvApp.
3. **Migration –** do the same steps which was done with of fbvApp.
4. **Serializer –** Copy the same code of fbvApp.
5. **Implementing Non-Primary key base operations –**
   1. Open views.py of cbvApp and there we will create non primary key based operations in class, basically get and post method. The code will look like this.
      1. ***from* django.shortcuts *import* render**
      2. ***from* cbvApp.models *import* Student**
      3. ***from* cbvApp.serializers *import* StudentSerializer**
      4. ***from* rest\_framework.response *import* Response**
      5. ***from* rest\_framework *import* status**
      6. ***from* rest\_framework.views *import* APIView**
      7. ***# Create your views here.***
      8. **class StudentList(APIView):**
      9. **def get(*request*):**
      10. **students = Student.objects.all()**
      11. **serializer = StudentSerializer(students, *many*=True)**
      12. ***return* Response(serializer.data)**
      13. **def post(*request*):**
      14. **serializer = StudentSerializer(*data*=*request*.data)**
      15. ***if* serializer.is\_valid():**
      16. **serializer.save()**
      17. ***return* Response(serializer.data, *status*=status.HTTP\_201\_CREATED)**
      18. ***return* Response(serializer.errors, *status*=status.HTTP\_400\_BAD\_REQUEST)**
   2. After that we will create Primary Key based operations in class, basically get, put and delete method. The code will look like this.
      1. **class StudentDetail(APIView):**
      2. **def get\_object(*self*, *pk*):**
      3. ***try*:**
      4. ***return* Student.objects.get(*pk*=*pk*)**
      5. ***except* Student.DoesNotExist:**
      6. ***return* Response(*status*=status.HTTP\_404\_NOT\_FOUND)**
      7. **def get(*self*, *request*, *pk*):**
      8. **student = *self*.get\_object(*pk*)**
      9. **serializer = StudentSerializer(student)**
      10. ***return* Response(serializer.data)**
      11. **def put(*self*, *request*, *pk*):**
      12. **student = *self*.get\_object(*pk*)**
      13. **serializer = StudentSerializer(student, *data*=*request*.data)**
      15. ***if* serializer.is\_valid():**
      16. **serializer.save()**
      17. ***return* Response(serializer.data)**
      18. ***return* Response(serializer.errors, *status*=status.HTTP\_400\_BAD\_REQUEST)**
      19. **def delete(*self*, *request*, *pk*):**
      20. **student = *self*.get\_object(*pk*)**
      21. **student.delete()**
      22. ***return* Response(*status*=status.HTTP\_204\_NO\_CONTENT)**
6. **Configuring URLS and Testing –**
   1. Go to urls.py of Project cbvSerializers and add the path there for both the class-based views. The code will look like this.
      1. ***from* django.contrib *import* admin**
      2. ***from* django.urls *import* path**
      3. ***from* cbvApp *import* views**
      4. **urlpatterns = [**
      5. **path('admin/', admin.site.urls),**
      6. **path('students/', views.StudentList.as\_view()),**
      7. **path('students/<int:pk>', views.StudentDetail.as\_view())**
      8. **]**
   2. We can visit the localhost in our browser and do the testing now. If you get programming error (data not found), make sure you did the migrations.

**MIXINS**

* The mixins are a bunch of classes which provide the actions that are used to provide the basic view behavior.
* The mixin classes provide action methods rather than defining the handler methods, such as .get() and .post(), directly. This allows for more flexible composition of behavior.
* It implements the DRY (Do not repeat yourself) principle.
* ListModelMixin, CreateModelMixin, RetrieveModelMixin, UpdateModelMixin, DestroyModelMixin, these are the types of mixins.

1. **Non-Primary Key Based Operations-**
   1. We will be using cbvSerializer project for this, so open cbvApp views.py and comment all the code. Then we will import generics and mixins from rest\_framework and create new class. It will look like this.
      1. ***from* gc *import* get\_objects**
      2. ***from* django.shortcuts *import* render**
      3. ***from* cbvApp.models *import* Student**
      4. ***from* cbvApp.serializers *import* StudentSerializer**
      5. ***from* rest\_framework.response *import* Response**
      6. ***from* rest\_framework *import* status**
      7. ***from* rest\_framework.views *import* APIView**
      8. ***from* rest\_framework *import* generics, mixins**
      9. **class StudentList(mixins.ListModelMixin, mixins.CreateModelMixin, generics.GenericAPIView):**
      10. **queryset = Student.objects.all()**
      11. **serializer\_class = StudentSerializer**
      12. **def get(*self*, *request*):**
      13. ***return* *self*.list(*request*)**
      14. **def post(*self*, *request*):**
      15. ***return* *self*.create(*request*)**
2. **Primary Key Based Operations-**
   1. Now we will write the code for primary key based operations and it will look like this.
      1. **class StudentDetail(mixins.RetrieveModelMixin, mixins.UpdateModelMixin, mixins.DestroyModelMixin, generics.GenericAPIView):**
      2. **queryset = Student.objects.all()**
      3. **serializer\_class= StudentSerializer**
      4. **def get(*self*, *request*, *pk*):**
      5. ***return* *self*.retrieve(*request*, *pk*)**
      6. **def put(*self*, *request*, *pk*):**
      7. ***return* *self*.update(*request*, *pk*)**
      8. **def delete(*self*, *request*, *pk*):**
      9. ***return* *self*.destroy(*request*, *pk*)**
3. **Configuring URLs and Testing-**
   1. As we changed our old cbvApp we don’t have to do anything with the urls. We can directly do the testing.

**GENERIC VIEWS**

* Generic views are a set of commonly used patterns. They're built on top of the APIView class.
* They consist of GenericAPIView, mixins, and concrete views.
  + GenericAPIView is a more loaded version of APIView. It isn't really useful on its own but can be used to create reusable actions.
  + Mixins are bits of common behavior. They're useless without GenericAPIView.
  + Concrete views combine GenericAPIView with the appropriate mixins to create views often used in APIs.
    - CreateAPIView, ListAPIView, RetrieveAPIView, DestroyAPIView, UpdateAPIView, ListCreateAPIView, RetrieveUpdateAPIView, RetrieveDestroyAPIView, RetrieveUpdateDestroyAPIView, these are the types which we can use.
    - ListCreateAPIView can be used for all non-id or non-primary key based operations and RetrieveUpdateDestroyAPIView can be used with all id-base or primary key base operations.

1. **Views.py of cbvApp-**
   1. We will only change the code of views.py of our cbvApp and it will look like this.
      1. ***from* courseApp.models *import* Course**
      2. ***from* courseApp.serializers *import* CourseSerializer**
      3. ***from* rest\_framework *import* generics**
      4. ***#Generics / Generic Views***
      5. **class CourseList(generics.ListCreateAPIView):**
      6. **queryset = Course.objects.all()**
      7. **serializer\_class = CourseSerializer**
      8. **class CourseDetail(generics.RetrieveUpdateDestroyAPIView):**
      9. **queryset = Course.objects.all()**
      10. **serializer\_class = CourseSerializer**
2. **Configuring URLs and Testing-**
   1. It’s done we can now test it, there’s no need to do the URL configuration.

**VIEW SETS**

* ViewSet is a type of class-based view.
* ViewSet is a bunch of view or set of view that we create either by extending the ViewSet class or extending the ModelViewSet class.
* Instead of method handlers, like .get() and .post(), it provides actions, like .list() and .create().
* There are four types of ViewSets, from the most basic to the most powerful-
  + ViewSet
  + GenericViewSet
  + ReadOnlyModelViewSet
  + ModelViewSet

1. **Views.py of cbvApp-**
   1. Go to views.py of cbvApp and import ViewSets from rest\_framework there and create a StudentViewSet class, the code will look like this.
      1. ***from* cbvApp.models *import* Student**
      2. ***from* cbvApp.serializers *import* StudentSerializer**
      3. ***from* rest\_framework *import* viewsets**
      4. **class StudentViewSet(viewsets.ModelViewSet):**
      5. **queryset = Student.objects.all()**
      6. **serializer\_class = StudentSerializer**
2. **Configuring URLs and Testing-**
   1. Open urls.py of Project which is cbvSerializers and then setup the router and urlpattern there, the code will look like this.
      1. ***from* django.urls *import* path, include**
      2. ***from* cbvApp *import* views**
      3. ***from* rest\_framework.routers *import* DefaultRouter**
      4. **router = DefaultRouter()**
      5. **router.register('students', views.StudentViewSet)**
      6. **urlpatterns = [**
      7. **path('', include(router.urls))**
      8. **]**
   2. Now goto localhost and test it.

**NESTED SERIALIZERS**

1. **Create a Project –**
   1. Run this command on terminal “**django-admin startproject nestedSerializers**”.
   2. Change directory to that folder “**cd nestedSerializers**” and then use command “**py manage.py startapp nsApp**”.
   3. Open nestedSerializers in VS Code and open settings.py of project nestedSerializers then configure nsapp and rest framework there.
      1. **INSTALLED\_APPS = [**
      2. **'nsApp',**
      3. **'rest\_framework'**
      4. **]**
   4. Then scroll down and configure database.
      1. **DATABASES = {**
      2. **'default': {**
      3. **'ENGINE': 'django.db.backends.postgresql',**
      4. **'NAME': 'authordb',**
      5. **'USER': 'postgres',**
      6. **'PASSWORD': 'root',**
      7. **'HOST': 'localhost'**
      8. **}**
      9. **}**
2. **Create the Model –**
   1. Open nsApp models.py and the code should be like this.
      1. ***from* django.db *import* models**
      2. **class Author(models.Model):**
      3. **firstName = models.CharField(*max\_length*=20)**
      4. **lastName = models.CharField(*max\_length*=20)**
      5. **class Book(models.Model):**
      6. **title = models.CharField(*max\_length*=20)**
      7. **ratings = models.CharField(*max\_length*=10)**
      8. **author = models.ForeignKey(Author, *related\_name*='books', *on\_delete*=models.CASCADE)**
3. **Migration –**
4. Before doing the migration, we have to create the database **authordb** which we mentioned in the database section of settings.py.
5. Open Valentina Studio and create the database named **authordb**.
6. Use command “**py manage.py makemigrations**” and then “**py manage.py migrate**”.
7. **Serializers –**
   1. Goto nsApp and create a file named serializer.py.
   2. Open it and the code should be this.
      1. ***from* .models *import* Author, Book**
      2. ***from* rest\_framework *import* serializers**
      3. **class BookSerializer(serializers.ModelSerializer):**
      4. **class Meta:**
      5. **model = Book**
      6. **fields = '\_\_all\_\_'**
      7. **class AuthorSerializer(serializers.ModelSerializer):**
      8. **books = BookSerializer(*read\_only*=True, *many*=True)**
      9. **class Meta:**
      10. **model = Author**
      11. **fields = '\_\_all\_\_'**
8. **Views.py of nsApp –**
   1. The code should look like this.
      1. ***from* .models *import* Author, Book**
      2. ***from* nsApp.serializer *import* AuthorSerializer, BookSerializer**
      3. ***from* rest\_framework *import* generics**
      4. **class AuthorListView(generics.ListCreateAPIView):**
      5. **queryset = Author.objects.all()**
      6. **serializer\_class = AuthorSerializer**
      7. **class AuthorDetailView(generics.RetrieveUpdateDestroyAPIView):**
      8. **queryset = Author.objects.all()**
      9. **serializer\_class = AuthorSerializer**
      10. **class BookListView(generics.ListCreateAPIView):**
      11. **queryset = Book.objects.all()**
      12. **serializer\_class = BookSerializer**
      13. **class BookDetailView(generics.RetrieveUpdateDestroyAPIView):**
      14. **queryset = Book.objects.all()**
      15. **serializer\_class = BookSerializer**
9. **Configuring URLs and Testing –**
   1. Urls.py of nsApp:
      1. ***from* django.urls *import* path**
      2. ***from* nsApp *import* views**
      3. **urlpatterns = [**
      4. **path('author/', views.AuthorListView.as\_view()),**
      5. **path('author/<int:pk>', views.AuthorDetailView.as\_view()),**
      6. **path('book/', views.BookListView.as\_view()),**
      7. **path('book/<int:pk>', views.BookListView.as\_view()),**
      8. **]**
   2. Urls.py of nestedSerializerss:
      1. ***from* django.contrib *import* admin**
      2. ***from* django.urls *import* path, include**
      3. **urlpatterns = [**
      4. **path('admin/', admin.site.urls),**
      5. **path('api/', include('nsApp.urls'))**
      6. **]**
   3. Testing – Visit localhost and test it.

**PAGINATION**

* Django provides a few classes that help you manage paginated data – that is, data that’s split across several pages, with “Previous/Next” links.
* REST framework includes support for customizable pagination styles. This allows you to modify how large result sets are split into individual pages of data.

1. **PageNumberPagination (Global Configuration) –**
   1. We will be using cbvSerializer project for this, so open it and go to settings.py of project and add this code there.
      1. **REST\_FRAMEWORK = {**
      2. **'DEFAULT\_PAGINATION\_CLASS': 'rest\_framework.pagination.PageNumberPagination',**
      3. **'PAGE\_SIZE': 2**
      4. **}**
   2. That’s it, use command “py manage.py runserver” to run the server and test it.
2. **PageNumberPagination (Class Level Configuration) –**
   1. Open views.py of cbvApp and create a pagination class, code will look like this.
      1. ***from* rest\_framework.pagination *import* PageNumberPagination**
      2. **class StudentPagination(PageNumberPagination):**
      3. **page\_size=1;**
   2. After that in whichever class you like this pagination setting, assign this class name to pagination\_class variable like this.
      1. **class StudentViewSet(viewsets.ModelViewSet):**
      2. **queryset = Student.objects.all()**
      3. **serializer\_class = StudentSerializer**
      4. **pagination\_class = StudentPagination**
3. **LimitOffsetPagination (Class Level Configuration) –**
   1. This pagination style mirrors the syntax used when looking up multiple database records. The client includes both a "limit" and an "offset" query parameter. The limit indicates the maximum number of items to return, and is equivalent to the page\_size in other styles. The offset indicates the starting position of the query in relation to the complete set of unpaginated items.
   2. Open views.py and the code should be like this.
      1. ***from* rest\_framework.pagination *import* LimitOffsetPagination**
      2. **class StudentViewSet(viewsets.ModelViewSet):**
      3. **queryset = Student.objects.all()**
      4. **serializer\_class = StudentSerializer**
      5. **pagination\_class = LimitOffsetPagination**
   3. Make sure you added this code to settings.py of project.
      1. **REST\_FRAMEWORK = {**
      2. **'DEFAULT\_PAGINATION\_CLASS': 'rest\_framework.pagination.PageNumberPagination',**
      3. **'PAGE\_SIZE': 2**
      4. **}**

**FILTERING**

* The default behavior of REST framework's generic list views is to return the entire queryset for a model manager. Often you will want your API to restrict the items that are returned by the queryset.

1. **Install filter–**
   1. Open command prompt and type “pip3 install django-filter” and it will install the Django-filter in our machine.
2. **Settings.py of cbvSerializers project–**
   1. Open settings.py and add Django filters module in installed app.
   2. This is how it will look.
      1. **INSTALLED\_APPS = [**
      2. **'cbvApp',**
      3. **'rest\_framework',**
      4. **'django\_filters'**
      5. **]**
3. **Views.py of cbvApp–**
   1. Open views.py of cbvApp and import Django filter backends and write the filter code in the class, it will look like this.
      1. ***from* cbvApp.models *import* Student**
      2. ***from* cbvApp.serializers *import* StudentSerializer**
      3. ***from* rest\_framework *import* viewsets**
      4. ***from* django\_filters.rest\_framework *import* DjangoFilterBackend**
      5. **class StudentViewSet(viewsets.ModelViewSet):**
      6. **queryset = Student.objects.all()**
      7. **serializer\_class = StudentSerializer**
      8. **filter\_backends = [DjangoFilterBackend]**
      9. **filterset\_fields = ['name', 'score']**
   2. Using search filter –
      1. ***from* cbvApp.models *import* Student**
      2. ***from* cbvApp.serializers *import* StudentSerializer**
      3. ***from* rest\_framework *import* viewsets**
      4. ***from* rest\_framework *import* filters**
      5. **class StudentViewSet(viewsets.ModelViewSet):**
      6. **queryset = Student.objects.all()**
      7. **serializer\_class = StudentSerializer**
      8. **filter\_backends = [filters.SearchFilter]**
      9. **search\_fields = ['id', 'name']**
   3. There are some extras, we can do with search like if with special symbols like ^ which means starts with, = which means exact match, @ Full-text search. $ Regex search. We have to write these inside the list set of search fields. Like this [‘^id’, ‘=name’]
   4. Ordering Filter – We can order the data by default or give the option to user, code should look like this
      1. ***from* cbvApp.models *import* Student**
      2. ***from* cbvApp.serializers *import* StudentSerializer**
      3. ***from* rest\_framework *import* viewsets**
      4. ***from* rest\_framework *import* filters**
      5. **class StudentViewSet(viewsets.ModelViewSet):**
      6. **queryset = Student.objects.all()**
      7. **serializer\_class = StudentSerializer**
      8. **filter\_backends = [filters.OrderingFilter]**
      9. **ordering\_fields = ['name', 'score']**
      10. ***#always mention the ordering\_fields else user can order by sensitive fields like passwords***
      11. **ordering = ['id'] *#default ordering***

**SECURITY**

* Authentication is the mechanism of associating an incoming request with a set of identifying credentials, such as the user the request came from, or the token that it was signed with. The permission and throttling policies can then use those credentials to determine if the request should be permitted.
* REST framework provides several authentication schemes out of the box, and also allows you to implement custom schemes.
* Authentication always runs at the very start of the view, before the permission and throttling checks occur, and before any other code is allowed to proceed.

1. **Authentication –** 
   1. We will use nestedSerializers project for this and now go to views.py of nsApp and import basicauthentication and isauthenticated module and then assign those two to authentication\_classes and permission\_classes respectively, the code will look like this.
      1. ***from* .models *import* Author, Book**
      2. ***from* nsApp.serializer *import* AuthorSerializer, BookSerializer**
      3. ***from* rest\_framework *import* generics**
      4. ***from* rest\_framework.authentication *import* BasicAuthentication**
      5. ***from* rest\_framework.permissions *import* IsAuthenticated**
      6. **class AuthorListView(generics.ListCreateAPIView):**
      7. **queryset = Author.objects.all()**
      8. **serializer\_class = AuthorSerializer**
      9. **authentication\_classes = [BasicAuthentication]**
      10. **permission\_classes = [IsAuthenticated]**
   2. After that we have to create a superuser to we can login when we visit localhost and view these details, to create superuser type command “py manage.py createsuperuser” and then enter the username and password which you want to choose and its done.
2. **Authorization –** 
   1. First of all import djangomodelpermissions to nsApp views like this.
      1. ***from* .models *import* Author, Book**
      2. ***from* nsApp.serializer *import* AuthorSerializer, BookSerializer**
      3. ***from* rest\_framework *import* generics**
      4. ***from* rest\_framework.authentication *import* BasicAuthentication**
      5. ***from* rest\_framework.permissions *import* IsAuthenticated, DjangoModelPermissions**
      6. **class AuthorListView(generics.ListCreateAPIView):**
      7. **queryset = Author.objects.all()**
      8. **serializer\_class = AuthorSerializer**
      9. **authentication\_classes = [BasicAuthentication]**
      10. **permission\_classes = [IsAuthenticated, DjangoModelPermissions]**
   2. Now we have to create a user through admin panel of Django, to do that visit <http://127.0.0.1:8000/admin> and login using the superuser credentials we created in authentication part, and create a new user and then go to staff access enable it and then select auth permission of view only access and its done.
3. **Global Security –**
   1. To apply authentication and authorization for whole app or all the classes of views (which means all the endpoints), we need to configure the authentication in projects settings.py.
   2. Open settings.py and add this code.
      1. **REST\_FRAMEWORK = {**
      2. **'DEFAULT\_AUTHENTICATION\_CLASSES': ['rest\_framework.authentication.BasicAuthentication'],**
      3. **'DEFAULT\_PERMISSION\_CLASSES': ['rest\_framework.permissions.IsAuthenticated', 'rest\_framework.permissions.DjangoModelPermissions']**
      4. **}**
   3. We can test by visiting localhost:8000/api/author, Or by using Postman software.

**FLIGHT RESERVATION API**

* **We will learn some other things like Validations, Token Authentication using this example project.**

1. **Create a Project –**
   1. Run this command on terminal “**django-admin startproject FlightServices**”.
   2. Change directory to that folder “**cd FlightServices**” and then use command “**py manage.py startapp flightApp**”.
   3. Open FlightServices in VS Code and open settings.py of project FlightServices then configure flightApp and rest framework there.
      1. **INSTALLED\_APPS = [**
      2. **'flightApp',**
      3. **'rest\_framework'**
      4. **]**
   4. Then scroll down and configure database.
      1. **DATABASES = {**
      2. **'default': {**
      3. **'ENGINE': 'django.db.backends.postgresql',**
      4. **'NAME': 'flightdb',**
      5. **'USER': 'postgres',**
      6. **'PASSWORD': 'root',**
      7. **'HOST': 'localhost'**
      8. **}**
      9. **}**
2. **Create the Model –**
   1. Open flightApp models.py and the code should be like this.
      1. ***from* django.db *import* models**
      2. **class Flight(models.Model):**
      3. **flightNumber = models.CharField(*max\_length*=10)**
      4. **operatingAirlines = models.CharField(*max\_length*=20)**
      5. **departureCity = models.CharField(*max\_length*=20)**
      6. **arrivalCity = models.CharField(*max\_length*=20)**
      7. **dateOfDeparture = models.DateField()**
      8. **estimatedTimeOfDeparture = models.TimeField()**
      9. **class Passenger(models.Model):**
      10. **firstName = models.CharField(*max\_length*=20)**
      11. **lastName = models.CharField(*max\_length*=20)**
      12. **middleName = models.CharField(*max\_length*=20)**
      13. **email = models.CharField(*max\_length*=30)**
      14. **phone = models.CharField(*max\_length*=13)**
      15. **class Reservation(models.Model):**
      16. **flight = models.OneToOneField(Flight, *on\_delete*=models.CASCADE)**
      17. **passenger = models.OneToOneField(Passenger, *on\_delete*=models.CASCADE)**
3. **Serializers –**
4. Goto flightApp and create a file named serializer.py, and the code should be like this.
   * 1. ***from* rest\_framework *import* serializers**
     2. ***from* flightApp.models *import* Flight, Passenger, Reservation**
     3. **class FlightSerializer(serializers.ModelSerializer):**
     4. **class Meta:**
     5. **model = Flight**
     6. **fields = '\_\_all\_\_'**
     7. **class PassengerSerializer(serializers.ModelSerializer):**
     8. **class Meta:**
     9. **model = Passenger**
     10. **fields = '\_\_all\_\_'**
     11. **class ReservationSerializer(serializers.ModelSerializer):**
     12. **class Meta:**
     13. **model = Reservation**
     14. **fields = '\_\_all\_\_'**
5. **Viewsets (Views.py) –**
   1. Open views.py and the code should be like this.
      1. ***from* flightApp.models *import* Flight, Passenger, Reservation**
      2. ***from* flightApp.serializers *import* FlightSerializer, PassengerSerializer, ReservationSerializer**
      3. ***from* rest\_framework *import* viewsets**
      4. **class FlightViewSet(viewsets.ModelViewSet):**
      5. **queryset = Flight.objects.all()**
      6. **serializer\_class = FlightSerializer**
      7. **class PassengerViewSet(viewsets.ModelViewSet):**
      8. **queryset = Passenger.objects.all()**
      9. **serializer\_class = PassengerSerializer**
      10. **class ReservationViewSet(viewsets.ModelViewSet):**
      11. **queryset = Reservation.objects.all()**
      12. **serializer\_class = ReservationSerializer**
6. **Configuring Router –**
   1. Goto urls.py and the code should look like this.
      1. ***from* django.contrib *import* admin**
      2. ***from* django.urls *import* path, include**
      3. ***from* flightApp *import* views**
      4. ***from* rest\_framework.routers *import* DefaultRouter**
      5. **router = DefaultRouter()**
      6. **router.register('flights', views.FlightViewSet)**
      7. **router.register('passengers', views.PassengerViewSet)**
      8. **router.register('reservations', views.ReservationViewSet)**
      9. **urlpatterns = [**
      10. **path('admin/', admin.site.urls),**
      11. **path('flightApi/', include(router.urls))**
      12. **]**
7. **Migration –**
8. Before doing the migration, we have to create the database **flightdb** which we mentioned in the database section of settings.py.
9. Open Valentina Studio and create the database named **flightdb**.
10. Use command “**py manage.py makemigrations**” and then “**py manage.py migrate**”.
11. **Implementation of findFlights and saveReservation –**
    1. Go to views.py and the code should be like this. And change reservation class flight = models.OneToOneField to models.ForeignKey in models.py file.
       1. *from* flightApp.models *import* Flight, Passenger, Reservation
       2. *from* flightApp.serializers *import* FlightSerializer, PassengerSerializer, ReservationSerializer
       3. *from* rest\_framework *import* viewsets
       4. *from* rest\_framework.response *import* Response
       5. *from* rest\_framework.decorators *import* api\_view
       6. *from* rest\_framework *import* status
       7. @api\_view(['POST'])
       8. def find\_flights(*request*):
       9. flights = Flight.objects.filter(*departureCity*=*request*.data['departureCity'], *arrivalCity*=*request*.data['arrivalCity'], *dateOfDeparture*=*request*.data['dateOfDeparture'])
       10. serializer = FlightSerializer(flights, *many*=True)
       11. *return* Response(serializer.data)
       12. @api\_view(['POST'])
       13. def save\_reservation(*request*):
       14. flight = Flight.objects.get(*id*=*request*.data['flightId'])
       15. passenger = Passenger()
       16. passenger.firstName = *request*.data['firstName']
       17. passenger.lastName = *request*.data['lastName']
       18. passenger.middleName = *request*.data['middleName']
       19. passenger.email = *request*.data['email']
       20. passenger.phone = *request*.data['phone']
       21. passenger.save()
       22. reservation = Reservation()
       23. reservation.flight = flight
       24. reservation.passenger = passenger
       25. reservation.save()
       26. *return* Response(*status*=status.HTTP\_201\_CREATED)
       27. class FlightViewSet(viewsets.ModelViewSet):
       28. queryset = Flight.objects.all()
       29. serializer\_class = FlightSerializer
       30. class PassengerViewSet(viewsets.ModelViewSet):
       31. queryset = Passenger.objects.all()
       32. serializer\_class = PassengerSerializer
       33. class ReservationViewSet(viewsets.ModelViewSet):
       34. queryset = Reservation.objects.all()
       35. serializer\_class = ReservationSerializer
12. **Testing –**
    1. Before running the server make sure you added findFlights and saveReservation paths in urls.py, it will look like this.
       1. **urlpatterns = [**
       2. **path('admin/', admin.site.urls),**
       3. **path('flightApi/', include(router.urls)),**
       4. **path('flightApi/findFlights/', views.find\_flights),**
       5. **path('flightApi/saveReservation/', views.save\_reservation)**
       6. **]**
    2. Run the server using “py manage.py runserver” command and check everything is working, we can also use postman for testing.

* **Validations**
  + **Inbuilt Validations–** Django comes with inbuilt validations if we don’t pass the data as we mentioned it will give us the error and generate a warning message. These are the two examples-
    - On flightApp models.py we have set the flightNumber length to 10 digits and if we put more than that it will error out and give us a message for what’s wrong.
    - If we don’t put the value in departureCity column validation will give us the error until and unless we tell it to that we can set it to blank or empty by mention blank=True in parenthesis as we mention the max\_length etc. So, it depends on us we can even allow NULL values to by using null=True, when we will not put anything in departure field it will be null by default.
  + **Custom Validations-** To do the custom validation we have to write the code in serializers.py, it will be like this. There is two ways to do that-
    - First one is where we select a specific data which we want to validate and we have to define the function which needs to follow a convention, for example we have to validate the flight number than we will use the validate\_flightNumber name function which is exactly the same as the variable which we defined in models.py.
    - When we use the second way we can access and validate the whole data for that we have to use it like dictionary, when we call a value from dictionary using a key, we can do that same here we have to write **data[‘flightNumber’]** to get the flight number.
    - ***from* rest\_framework *import* serializers**
    - ***from* flightApp.models *import* Flight, Passenger, Reservation**
    - ***import* re**
    - **class FlightSerializer(serializers.ModelSerializer):**
    - **class Meta:**
    - **model = Flight**
    - **fields = '\_\_all\_\_'**
    - **def validate\_flightNumber(*self*, *flightNumber*):**
    - ***if*(re.match("^[a-zA-Z0-9]\*$", *flightNumber*) == None):**
    - ***raise* serializers.ValidationError("Invalid Flightnumber, Make sure it is AlphaNumeric")**
    - ***return* *flightNumber***
    - **def validate(*self*, *data*):**
    - **print(*data*['flightNumber'])**
    - **print(*data*['departureCity'])**
    - ***return* *data***
* **Token Based Authentication**
  + **Configuration-**
    - Open settings.py of project FlightServices and add rest auth app in installed apps and then we have to add rest framework block, it will look like this.
      * **INSTALLED\_APPS = [**
      * **'django.contrib.staticfiles',**
      * **'flightApp',**
      * **'rest\_framework',**
      * **'rest\_framework.authtoken'**
      * **]**
      * **REST\_FRAMEWORK = {**
      * **'DEFAULT\_AUTHENTICATION\_CLASSES' : [**
      * **'rest\_framework.authentication.TokenAuthentication'**
      * **]**
      * **}**
    - Open urls.py and add path there it will look like this.
      * ***from* django.contrib *import* admin**
      * ***from* django.urls *import* path, include**
      * ***from* flightApp *import* views**
      * ***from* rest\_framework.authtoken.views *import* obtain\_auth\_token**
      * **urlpatterns = [**
      * **path('admin/', admin.site.urls),**
      * **path('flightApi/', include(router.urls)),**
      * ***name*="api\_token\_auth")**
      * **]**
    - Open views.py and add permission\_classes to all the viewsets which you want to be authenticated. It will look like this. After this step run the migrations.
      * ***from* rest\_framework *import* status**
      * ***from* rest\_framework.permissions *import* IsAuthenticated**
      * **class FlightViewSet(viewsets.ModelViewSet):**
      * **queryset = Flight.objects.all()**
      * **serializer\_class = FlightSerializer**
      * **permission\_classes = (IsAuthenticated,)**
  + **Create Users & Tokens-**
    - Before creating the users and tokens we have to create the superuser use **“py manage.py createsuperuser”** command to do that.
    - Go to localhost/admin and login using that id pass you just created using superuser command.
    - Click on add user and create a username and password then click on add token and select username and click on save to create the token.
  + **Testing-**
    - Open Postman go to <http://127.0.0.1:8000/flightApi/flights/> and when we use GET or POST it will generate the error authentication credentials not provided.
    - To retrieve the token, open new postman tab select method POST and go to url <http://127.0.0.1:8000/api-token-auth/> it will generate the warning username and password not provided click on body then raw then select type to json after that write username and password in json format,

which is

{

“username” : “test”,

“password” : “Test#123”

}

Then click on send and it will give us the token

* + - Copy the token and then open previous tab where we were creating the flight and then click on headers and inside the key column write Authorization and on values field after writing Token give a space and paste the token. Now if we click send the new flight will be created.
  + **Automate Token Creation-**
    - Go to models.py do the imports and then define the function to do the automatic creation of token when a user is created. The code will look like this.
      * ***from* django.db *import* models**
      * ***from* django.db.models.signals *import* post\_save**
      * ***from* django.dispatch *import* receiver**
      * ***from* rest\_framework.authtoken.models *import* Token**
      * ***from* django.conf *import* settings**
      * **@receiver(post\_save, *sender*=settings.AUTH\_USER\_MODEL)**
      * **def createAuthToken(*sender*, *instance*, *created*, \*\**kwargs*):**
      * ***if* *created*:**
      * **Token.objects.create(*user*=*instance*)**
      * **class Flight(models.Model):**
      * **flightNumber = models.CharField(*max\_length*=10)**
      * **operatingAirlines = models.CharField(*max\_length*=20)**
      * **departureCity = models.CharField(*max\_length*=20, *null*=True)**
      * **arrivalCity = models.CharField(*max\_length*=20)**
      * **dateOfDeparture = models.DateField()**
      * **estimatedTimeOfDeparture = models.TimeField()**
* **ANGULAR FRONTEND**
  + First step is to install angular and to do that, We will install node.js visit <https://nodejs.org/en/download/> and download and install node.js and after that run “**npm -v**” command to check if it is working or not.
  + Now launch command prompt and run **“npm install -g @angular/cli”** to install angular.
  + After the installation we can create new app using **“ng new my-app”** command here my-app is the app name which can be anything.
  + Then use **“cd my-app”** to enter that directory and use “ng serve –open” command to run the app it will be launched and the url will be **“localhost:4200”** or **“127.0.0.1:4200”**.
  + Open models.py of flightApp and make sure you used ForeignKey, if we don’t we will not be able to repeat same flights and same passengers.
    - **class Reservation(models.Model):**
    - **flight = models.ForeignKey(Flight, *on\_delete*=models.CASCADE)**
    - **passenger = models.ForeignKey(Passenger, *on\_delete*=models.CASCADE)**
  + Now we will be running django server on 127.0.0.1:8000 and angular server on 127.0.0.1:4200 and browser will not allow both servers to communicate with each other until we use some middleware, so we will install Django cors headers.
  + Use **“pip install django-cors-headers”** command to install it, then goto settings.py of project FlightServices and in INSTALLED\_APPS block add ‘corsheaders’. It will look like this.
    - **INSTALLED\_APPS = [**
    - **'flightApp',**
    - **'rest\_framework',**
    - **'rest\_framework.authtoken'**
    - **'corsheaders'**
    - **]**
  + Scroll down and in the MIDDLEWARE block add **“corsheaders.middleware.CorsMiddleware”** and create new code block inside the settings **ALLOWED\_HOSTS =[\*]** and another one **CORS\_ORIGIN\_ALLOW\_ALL=True**. This will allow any application running on any port to access Django server. If you want to allow only single application to access Django server you can use.
    - ***#cors config***
    - **CORS\_ORIGIN\_ALLOW\_ALL = False**
    - **CORS\_ORIGIN\_WHITELIST = (**
    - **'http://localhost:4200',**
    - **)**
  + Create new project named flightReservation using **“ng new flightReservation”** command, after that change directory into that folder and now we have to create four components and for that we will use **“ng g c components/findFlights”** command like this create **“ng g c components/displayFlights” , “ng g c components/passengerDetails”** and **“ng g c components/confirm”** to confirm the reservation.
  + Now we have to create login and reservation services and to do that use **“ng g s services/login”** and **“ng g s services/reservation”** command respectively.