Packages/Modules:

- Package/Module names should be always in **lowercase** letters only.
- Try to define a name with a **single word** only
    - **Ex. users, employees**
- When multiple words are needed, an **underscore** should separate them
    - **Ex. user_management**

Classes:

- Class names should follow the **UpperCaseCamelCase** convention
- Define meaningful class names from there anyone can identify the usage of the class.
    - If create class for custom exception or error, add "**Exception**" at end of name
    - **Ex. UserNotFoundException**
- Add some documentation after the class declaration line.

```
class UserManagement:
    """
    Use to manage user related information
    """
```

Methods/Functions:

- Method names should follow the **lowercase** convention only.
- Ex.
    - **get_user() -** (Valid)
    - **getuser(), getUser()** - (Invalid)
- Always use **self** for the first argument to instance methods.
- Always add block comments (document strings) for the purpose of the method with parameters and return details.

Variable Names:

- Variable names should be always in **lowercase** letters only.
- When multiple words are needed, an **underscore** should separate them.

Views/Viewsets:

- All view/viewsets class names should be **UpperCaseCamelCase** convention.
- Create separate class view/viewset for separate endpoint feature.
- It should always extend with Base Class (We will always use base class for common use).
- **Ex.**
    - We have one base class named "**BaseModelViewSet**".

- If we create one api view named **UserViewSet,** it should be extended with **BaseModelViewSet.**
-
```
class UserViewSet(BaseModelViewSet):
    """
    Users feature endpoints.
    """
```

- All endpoints data should be validated through serializers. If serializer data is valid, then continue with rest of logic otherwise raise the exception with valid message/format.
- Required to add valid commenting for every endpoint action.
- Basic business logic will be inside views/viewsets class.
- If we require to create a separate @actions method for endpoint, we need to define valid action method name and also url path.

```
@action(methods=["get"], detail=True, url_path="customer/users")
def customer_users(self, request, pk):
    """
    Fetch all customer users data from database.
    """
```

- When some endpoint contains more business logic, then we need to move those business logic to **Manager classes** (If logic will be based on particular models) or **Service classes** (if we have logic which contains to work with multiple data models).
- All endpoints should be returned with a valid response format with valid http response status.

```
return Response(
    data={}, status=status.HTTP_200_OK
)
```

Serializers:

- Seriazliers should be classes, the class naming convention applies here.
- Create a separate serializer class for different endpoints to validate fields.
- If possible also extends serializer class with base serializer classes, so we use common features/methods from parent class.

Models:

- Model class name should be the **UpperCaseCamelCase** convention.
- Create base model class which contains audit fields as below:
    - created_at
    - updated_at

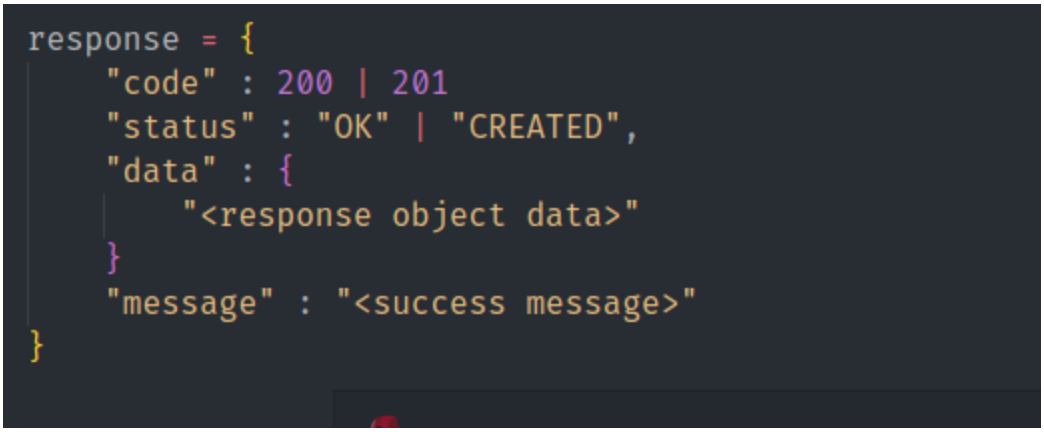
    - is_delete
    - created_by (if require)
    - updated_by (if require)
- All model classes should be extended with **BaseModel** class.
- Create common methods in base class and use it in child model classes.

Constants:

- Constants are defined on a module level and written in camel case letters with underscores separating words.
- Ex.
    - MAX_OVERFLOW
    - TOTAL

Endpoint Response Format:

- All endpoint response should fix format standard as below:
    - **Success Response:**



```
response = {
    "code" : 200 | 201
    "status" : "OK" | "CREATED",
    "data" : {
        "<response object data>"
    }
    "message" : "<success message>"
}
```

    - **Error Response:**

```
response = {
    "code" : 400 | 403 | 404
    "status" : "BAD_REQUEST" | "NOT_FOUND",
    "message" : "<error message>"
    "errors" : {
        "<error message with fields>"
    },
}
```

Spaces between code/module:

- Require one line space to differentiate logic between core logic/feature in any kind of business logic.