COURSE WORK REPORT

EDIFUC – 2023

MARK SEMENOV

LEV NIKITIN

Object-Oriented Programming in Real Estate Data Management System

Introduction

The project is a real estate data management system designed to streamline operations for realtors and data managers. This system facilitates the management and marketing of real estate properties by automating data entry, price retrieval, and social media content generation. The main part of the project consist of control of all properties, creating and edit of price list automatically, connection to social media API with service and TG bot. Last one helps to provide correct communication between client and service.

Body / Analysis

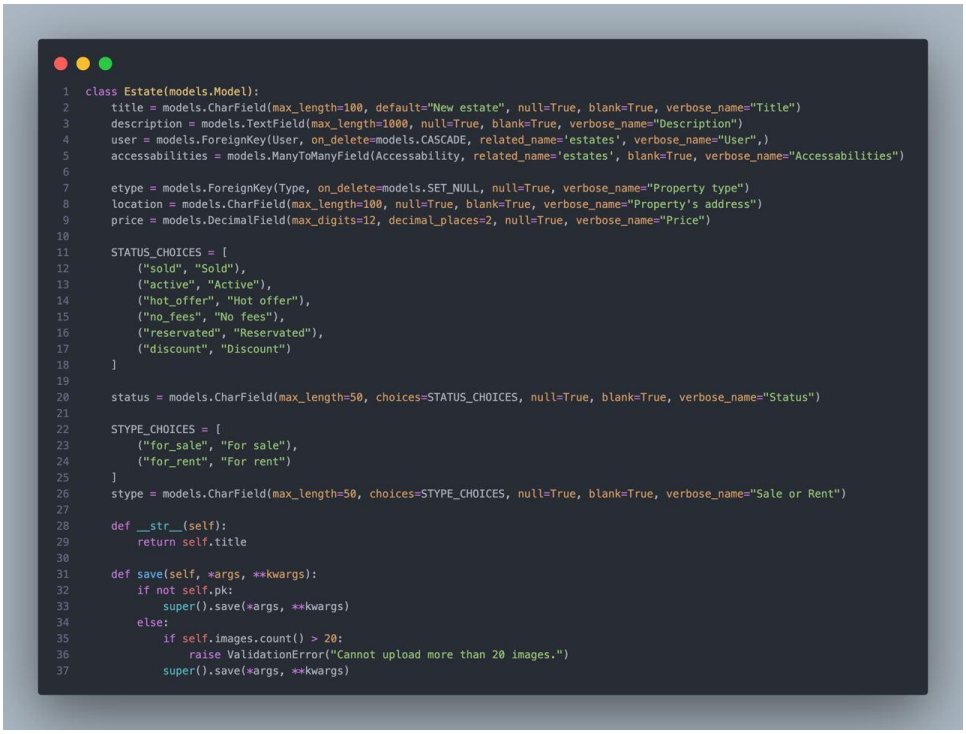
Object-Oriented Programming Principles

1. Encapsulation: Encapsulation involves bundling the data and methods that operate on the data into a single unit or class, and restricting access to some of the object's components. This ensures that the internal state of the object is shielded from unintended interference.

2. Abstraction: Abstraction means hiding the complex reality while exposing only the necessary parts. It helps in reducing programming complexity.

3. Inheritance: Inheritance is a mechanism where a new class is derived from an existing class. The new class inherits attributes and methods of the existing class.

4. Polymorphism: Polymorphism allows methods to do different things based on the object it is acting upon. This means the same method or property name can be used on objects of different classes.



```python
class Estate(models.Model):
    title = models.CharField(max_length=100, default="New estate", null=True, blank=True, verbose_name="Title")
    description = models.TextField(max_length=1000, null=True, blank=True, verbose_name="Description")
    user = models.ForeignKey(User, on_delete=models.CASCADE, related_name='estates', verbose_name="User",)
    accessabilities = models.ManyToManyField(Accessability, related_name='estates', blank=True, verbose_name="Accessabilities")

    etype = models.ForeignKey(Type, on_delete=models.SET_NULL, null=True, verbose_name="Property type")
    location = models.CharField(max_length=100, null=True, blank=True, verbose_name="Property's address")
    price = models.DecimalField(max_digits=12, decimal_places=2, null=True, verbose_name="Price")

    STATUS_CHOICES = [
        ("sold", "Sold"),
        ("active", "Active"),
        ("hot_offer", "Hot offer"),
        ("no_fees", "No fees"),
        ("reservated", "Reservated"),
        ("discount", "Discount")
    ]

    status = models.CharField(max_length=50, choices=STATUS_CHOICES, null=True, blank=True, verbose_name="Status")

    STYPE_CHOICES = [
        ("for_sale", "For sale"),
        ("for_rent", "For rent")
    ]
    stype = models.CharField(max_length=50, choices=STYPE_CHOICES, null=True, blank=True, verbose_name="Sale or Rent")

    def __str__(self):
        return self.title

    def save(self, *args, **kwargs):
        if not self.pk:
            super().save(*args, **kwargs)
        else:
            if self.images.count() > 20:
                raise ValidationError("Cannot upload more than 20 images.")
            super().save(*args, **kwargs)
```

Design Patterns

Singleton Pattern: Used to ensure that only one instance of the bot's configuration is active at any given time, centralizing bot settings management.

```python
class SingletonModel(models.Model):
    class Meta:
        abstract = True

    def save(self, *args, **kwargs):
        if not self.pk and self.__class__.objects.exists():
            raise ValidationError(f"An instance of {self.__class__.__name__} already exists.")
        super(SingletonModel, self).save(*args, **kwargs)

    def delete(self, *args, **kwargs):
        raise ValidationError(f"Deletion of {self.__class__.__name__} instance is not allowed.")

    @classmethod
    def load(cls):
        obj, created = cls.objects.get_or_create(pk=1)
        return obj

class BotAdmin(SingletonModel):
    icon = models.ImageField(upload_to="tgbot/", verbose_name="Icon")
    name = models.CharField(max_length=50, verbose_name="Name")
    url = models.URLField(max_length=200, verbose_name="URL")
    token = models.CharField(max_length=255, unique=True, verbose_name="Token")

    class Meta:
        verbose_name = "TG Bot"
        verbose_name_plural = "TG Bot"

    def __str__(self):
        return self.token
```

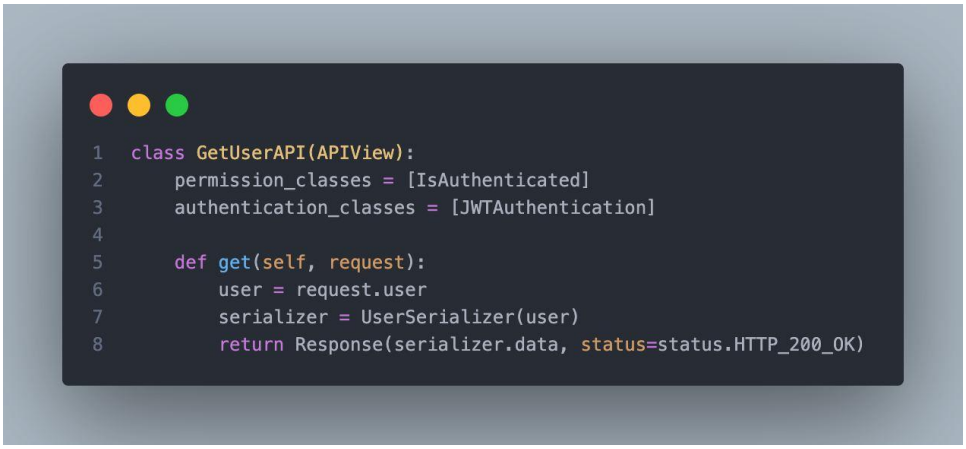
Decorator

```python
@api_view(['GET'])
def bot_info(request):
    singleton_token = BotAdmin.load()
    serializer = BotAdminSerializer(singleton_token)
    return Response(serializer.data, status=status.HTTP_200_OK)

```

File Handling

Detail the methods used for reading from and writing to files, possibly for saving configurations or logging. We use on production PostgreSQL on the production server to store real-time data. This database ensures reliability and performance for data processing in the production environment. On the local machine for development and testing, we use SQLite3, a lightweight database that simplifies the development and debugging process of the application before deployment to the server.



```python
class GetUserAPI(APIView):
    permission_classes = [IsAuthenticated]
    authentication_classes = [JWTAuthentication]

    def get(self, request):
        user = request.user
        serializer = UserSerializer(user)
        return Response(serializer.data, status=status.HTTP_200_OK)
```

Testing

Test for estate:

```python
class Image(models.Model):
    estate = models.ForeignKey('Estate', related_name='images', on_delete=models.CASCADE)
    image = models.ImageField(upload_to='estate_images/')
    uploaded_at = models.DateTimeField(auto_now_add=True)

class Accessability(models.Model):
    name = models.CharField(max_length=50)

    def __str__(self):
        return self.name


class Type(models.Model):
    name = models.CharField(max_length=50)

    def __str__(self):
        return self.name

class Estate(models.Model):
    title = models.CharField(max_length=100, default="New estate", null=True, blank=True, verbose_name="Title")
    description = models.TextField(max_length=1000, null=True, blank=True, verbose_name="Description")
    user = models.ForeignKey(User, on_delete=models.CASCADE, related_name='estates', verbose_name="User",)
    accessabilities = models.ManyToManyField(Accessability, related_name='estates', blank=True, verbose_name="Accessabilities")

    etype = models.ForeignKey(Type, on_delete=models.SET_NULL, null=True, verbose_name="Property type")
    location = models.CharField(max_length=100, null=True, blank=True, verbose_name="Property's address")
    price = models.DecimalField(max_digits=12, decimal_places=2, null=True, verbose_name="Price")

    STATUS_CHOICES = [
        ("sold", "Sold"),
        ("active", "Active"),
        ("hot_offer", "Hot offer"),
        ("no_fees", "No fees"),
        ("reservated", "Reservated"),
        ("discount", "Discount")
    ]

    status = models.CharField(max_length=50, choices=STATUS_CHOICES, null=True, blank=True, verbose_name="Status")

    STYPE_CHOICES = [
        ("for_sale", "For sale"),
        ("for_rent", "For rent")
    ]
    stype = models.CharField(max_length=50, choices=STYPE_CHOICES, null=True, blank=True, verbose_name="Sale or Rent")

    def __str__(self):
        return self.title

    def save(self, *args, **kwargs):
        if not self.pk:
            super().save(*args, **kwargs)
        else:
            if self.images.count() > 1:
                raise ValidationError("Cannot upload more than 20 images.")
            super().save(*args, **kwargs)
```

```python
class EstateModelTests(TestCase):
    def setUp(self):
        self.user = User.objects.create_user(
            email='owner@example.com',
            password='password123',
            name='Owner',
            surname='Estate'
        )
        self.etype = Type.objects.create(name='Apartment')
        self.accessability = Accessability.objects.create(name='Elevator')

    def test_create_estate(self):
        estate = Estate.objects.create(
            title='Beautiful Estate',
            description='A beautiful estate with a nice view.',
            user=self.user,
            etype=self.etype,
            location='123 Estate Ave',
            price=1000000.00,
            status='active',
            stype='for_sale'
        )
        estate.accessabilities.add(self.accessability)
        self.assertEqual(estate.title, 'Beautiful Estate')
        self.assertEqual(estate.user, self.user)
        self.assertEqual(estate.etype, self.etype)
        self.assertEqual(estate.status, 'active')
        self.assertEqual(estate.stype, 'for_sale')
        self.assertIn(self.accessability, estate.accessabilities.all())

    def test_image_upload_limit(self):
        estate = Estate.objects.create(
            title='Limited Estate',
            user=self.user,
            etype=self.etype,
            location='456 Estate Blvd',
            price=500000.00
        )
        for i in range(20):
            Image.objects.create(estate=estate, image=f'image_{i}.jpg')

        with self.assertRaises(ValidationError):
            Image.objects.create(estate=estate, image='image_21.jpg')
            estate.save()

```

Test for singleton bot:

```python
class BotAdminModelTests(TestCase):
    def test_singleton_creation(self):
        bot_admin = BotAdmin.objects.create(
            icon='icon.jpg',
            name='Test Bot',
            url='https://example.com',
            token='sometoken123'
        )
        with self.assertRaises(ValidationError):
            BotAdmin.objects.create(
                icon='icon2.jpg',
                name='Test Bot 2',
                url='https://example2.com',
                token='anothertoken456'
            )

        self.assertEqual(BotAdmin.load(), bot_admin)

    def test_singleton_deletion(self):
        bot_admin = BotAdmin.objects.create(
            icon='icon.jpg',
            name='Test Bot',
            url='https://example.com',
            token='sometoken123'
        )
        with self.assertRaises(ValidationError):
            bot_admin.delete()

```

```python
class SingletonModel(models.Model):
    class Meta:
        abstract = True

    def save(self, *args, **kwargs):
        if not self.pk and self.__class__.objects.exists():
            raise ValidationError(f"An instance of {self.__class__.__name__} already exists.")
        super(SingletonModel, self).save(*args, **kwargs)

    def delete(self, *args, **kwargs):
        raise ValidationError(f"Deletion of {self.__class__.__name__} instance is not allowed.")

    @classmethod
    def load(cls):
        obj, created = cls.objects.get_or_create(pk=1)
        return obj

class BotAdmin(SingletonModel):
    icon = models.ImageField(upload_to="tgbot/", verbose_name="Icon")
    name = models.CharField(max_length=50, verbose_name="Name")
    url = models.URLField(max_length=200, verbose_name="URL")
    token = models.CharField(max_length=255, unique=True, verbose_name="Token")

    class Meta:
        verbose_name = "TG Bot"
        verbose_name_plural = "TG Bot"

    def __str__(self):
        return self.token
```

Test for user :

```python
class UserModelTests(TestCase):
    def test_create_user(self):
        # Test creating a regular user
        user = User.objects.create_user(
            email='test@example.com',
            password='password123',
            name='John',
            surname='Doe'
        )

        self.assertEqual(user.email, 'test@example.com')
        self.assertTrue(user.check_password('password123'))
        self.assertEqual(user.name, 'John')
        self.assertEqual(user.surname, 'Doe')
        self.assertEqual(user.username, 'John Doe')
        self.assertIsNotNone(user.ref_code)
        self.assertFalse(user.is_staff)
        self.assertFalse(user.is_active)
        self.assertFalse(user.is_superuser)
        self.assertTrue(user.is_confirmed_requirements)
        self.assertTrue(user.is_confirmed_news)

    def test_create_superuser(self):
        # Test creating a superuser
        admin_user = User.objects.create_superuser(
            email='admin@example.com',
            password='admin123',
            name='Admin',
            surname='User'
        )

        self.assertEqual(admin_user.email, 'admin@example.com')
        self.assertTrue(admin_user.check_password('admin123'))
        self.assertEqual(admin_user.name, 'Admin')
        self.assertEqual(admin_user.surname, 'User')
        self.assertIsNotNone(admin_user.ref_code)
        self.assertTrue(admin_user.is_staff)
        self.assertTrue(admin_user.is_active)
        self.assertTrue(admin_user.is_superuser)
        self.assertTrue(admin_user.is_confirmed_requirements)
        self.assertTrue(admin_user.is_confirmed_news)

```

```python
class CustomUserManager(BaseUserManager):
    def _create_user(self, email, password=None, **extra_fields):
        if not email:
            raise ValueError("Email must be provided")
        # if not password:
        #     raise ValueError('Password is not provided')

        user = self.model(
            email=self.normalize_email(email),
            **extra_fields
        )

        if password:
            user.set_password(password)
        user.save(using=self._db)

        return user

    def create_user(self, name=None, surname=None, email=None, username=None, password=None, **extra_fields):
    # def create_user(self, *args):
        extra_fields.setdefault('is_staff', False)
        extra_fields.setdefault('is_active', False)
        extra_fields.setdefault('is_superuser', False)

        if username is None and name is not None and surname is not None:
            username = f"{name.capitalize()} {surname.capitalize()}"

        return self._create_user(email, password, name=name, surname=surname, username=username, **extra_fields)

    def create_superuser(self, email, password, **extra_fields):
        extra_fields.setdefault('is_staff', True)
        extra_fields.setdefault('is_active', True)
        extra_fields.setdefault('is_superuser', True)
        return self._create_user(email, password, **extra_fields)


class User(AbstractBaseUser, PermissionsMixin):
    # General
    avatar = models.ImageField(upload_to='avatars/', verbose_name="Avatar", null=True, blank=True, default=None)
    username = models.CharField(max_length=100, null=True, blank=True, verbose_name="Username")
    email = models.EmailField(db_index=True, unique=True, max_length=254, verbose_name='Email')

    #Own
    name = models.CharField(max_length=50, null=True, blank=True, verbose_name="First name")
    surname = models.CharField(max_length=50, null=True, blank=True, verbose_name="Second name")
    location = models.CharField(max_length=100, null=True, blank=True, verbose_name="Office's address")

    #Business
    website = models.URLField(max_length=500, null=True, blank=True, verbose_name="Website")
    instagram = models.URLField(max_length=500, null=True, blank=True, verbose_name="Instagram")
    facebook = models.URLField(max_length=500, null=True, blank=True, verbose_name="Facebook")

    #Technical
    is_active = models.BooleanField(default=True,
                                    verbose_name='Active')  # must needed, otherwise you won't be able to loginto django-admin.
    is_staff = models.BooleanField(default=False,
                                   verbose_name='Staff')  # must needed, otherwise you won't be able to loginto django-admin.
    is_superuser = models.BooleanField(default=False,
                                       verbose_name='Admin')  # this field we inherit from PermissionsMixin.
    ref_code = models.CharField(max_length=10, verbose_name="Referal code", unique=True, null=True, blank=True)

    is_confirmed_requirements = models.BooleanField(default=True, verbose_name="Confirmed terms")
    is_confirmed_news = models.BooleanField(default=True, verbose_name="Confirmed receiving emails")

    def save(self, *args, **kwargs):
        if not self.ref_code:
            ref_code_length = 8
            characters = string.ascii_letters + string.digits
            self.ref_code = ''.join(random.choices(characters, k=ref_code_length))
        super(User, self).save(*args, **kwargs)


    objects = CustomUserManager()

    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = ['password', 'name', 'surname', ]

    class Meta:
        verbose_name = 'User'
        verbose_name_plural = 'Users'

    def __str__(self):
        return self.email if self.email else 'email not confirmed'

```

Compile results:

This testing indicates that the code is functioning correctly. All tests have been executed successfully.

Results and Summary

Functional Coverage: How well the program meets the requirements of automating data entry and retrieval.

Challenges Encountered: Discuss any issues faced during implementation, such as integration with external APIs or data consistency.

Conclusion

Achievements: Recap what the program successfully accomplishes and the benefits it offers.

Future Prospects: Explore potential enhancements like adding more property types, integrating more social platforms and improving the user interface. Also, add functions that we don't have.

Additional Resources

List any libraries, frameworks, or external resources that were utilized, providing links and documentation references where applicable.