Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования «Южно-Уральский государственный университет»

(Национальный исследовательский университет)

Высшая школа электроники и компьютерных наук

Кафедра ИКТ

**ОТЧЕТ**

**по производственной практике**

Выполнил:

Студент группы КЭ-311

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Шилов П.A.

«\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_2024 г.

Принял:

Руководитель практики от предприятия

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Волков Е.А.

«\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_2024 г.

Челябинск

2024

Оглавление

[ВВЕДЕНИЕ 3](#_Toc168685693)

[1.1 Краткое описание Django и его применения 3](#_Toc168685694)

[1.2 Что такое сериализация данных и почему она важна 3](#_Toc168685695)

[1.3 Обзор сериализаторов Django 4](#_Toc168685696)

[2. ОСНОВНЫЕ ТИПЫ СЕРИАЛИЗАТОРОВ DJANGO 5](#_Toc168685697)

[2.1 ModelSerializer 5](#_Toc168685698)

[2.2 JSONSerializer 5](#_Toc168685699)

[2.3 Serializer 5](#_Toc168685700)

[3. ВАЛИДАТОР 5](#_Toc168685701)

[4. АНАЛИЗ КОДА СЕРИАЛИЗАТОРА 6](#_Toc168685702)

[4.1 Библиотеки 6](#_Toc168685703)

[4.2 Первая часть кода 7](#_Toc168685704)

[4.3 Вторая часть кода 11](#_Toc168685705)

[4.4 Третья часть кода 14](#_Toc168685706)

[4.5 Четвертая часть кода 16](#_Toc168685707)

[4.6 Шестая часть кода 18](#_Toc168685708)

[6. ЗАКЛЮЧЕНИЕ 28](#_Toc168685709)

[Преимущества использования сериализаторов Django 28](#_Toc168685710)

[Рекомендации по выбору сериализатора 28](#_Toc168685711)

# **ВВЕДЕНИЕ**

### 1.1 Краткое описание Django и его применения

Django - это высокоуровневый веб-фреймворк Python, созданный для быстрого и легкого создания веб-приложений. Он известен своей простотой использования, надежностью и масштабируемостью. Django используется для создания широкого спектра веб-приложений, от простых блогов до сложных корпоративных систем.

**Некоторые из ключевых функций Django:**

**Модульная архитектура:** Django построен на модульной архитектуре, что делает его простым в расширении и настройке.

**Система URL-адресов:** Django имеет мощную систему URL-адресов, которая позволяет легко сопоставлять URL-адреса с представлениями.

**Система шаблонов:** Django имеет систему шаблонов, которая позволяет легко создавать HTML-код.

**Объектно-реляционное отображение (ORM):** Django имеет ORM, который позволяет легко взаимодействовать с базами данных.

**Административная панель:** Django имеет встроенную административную панель, которая позволяет легко управлять контентом веб-приложения.

### 1.2 Что такое сериализация данных и почему она важна

Сериализация данных - это процесс преобразования данных из одного формата в другой. Это часто необходимо для обмена данными между различными системами или приложениями.

**Существует несколько распространенных форматов сериализации данных:**

**JSON:** JSON (JavaScript Object Notation) - это легкий текстовый формат обмена данными, основанный на JavaScript.

**XML:** XML (Extensible Markup Language) - это текстовый формат данных, основанный на тегах.

**YAML:** YAML (YAML Ain't Markup Language) - это человекочитаемый формат данных, основанный на indentation.

**Сериализация данных важна по ряду причин:**

**Обмен данными:** Сериализация данных позволяет легко обмениваться данными между различными системами или приложениями.

**Хранение данных:** Сериализованные данные можно легко хранить в файлах или базах данных.

**Передача данных:** Сериализованные данные можно легко передавать по сети.

### 1.3 Обзор сериализаторов Django

Сериализаторы Django - это мощный инструмент для сериализации и десериализации данных. Они позволяют легко преобразовывать объекты Django в различные форматы данных, такие как JSON, XML и YAML.

**Некоторые из основных функций сериализаторов Django:**

**Автоматическое создание полей:** Сериализаторы Django могут автоматически создавать поля на основе модели Django.

**Валидация данных:** Сериализаторы Django могут валидировать данные, чтобы убедиться, что они действительны.

**Преобразование типов:** Сериализаторы Django могут преобразовывать типы данных, например, преобразовывать дату в строку.

**Вложенные сериализаторы:** Сериализаторы Django могут использоваться для сериализации вложенных объектов.

## 2. ОСНОВНЫЕ ТИПЫ СЕРИАЛИЗАТОРОВ DJANGO

В Django REST framework существует три основных типа сериализаторов, каждый из которых предназначен для различных целей:

### 2.1 ModelSerializer

ModelSerializer является наиболее часто используемым типом сериализатора. Он автоматически генерирует поля на основе модели Django, а также валидаторы и методы create() и update(). Это делает его идеальным выбором для сериализации и десериализации экземпляров моделей Django.

### 2.2 JSONSerializer

JSONSerializer используется для сериализации объектов Python в JSON. Он не связан с какой-либо конкретной моделью Django и обеспечивает больше гибкости при настройке выходных данных.

### 2.3 Serializer

Serializer является базовым классом для всех сериализаторов Django. Он обеспечивает ручной контроль над сериализацией и десериализацией данных.

## 3. ВАЛИДАТОР

Валидатор - это механизм проверки данных, который используется в Django REST Framework для проверки корректности значений перед их сохранением или использованием. Валидаторы позволяют определять правила, которым должны соответствовать данные, и в случае несоответствия генерировать сообщения об ошибках.

## 4. АНАЛИЗ КОДА СЕРИАЛИЗАТОРА

### 4.1 Библиотеки

from django.contrib.auth import get\_user\_model: Этот импорт позволяет вам получить модель пользователя Django, которая используется в нашем проекте. Это может быть как стандартная модель пользователя Django, так и пользовательская модель, которую вы определили.

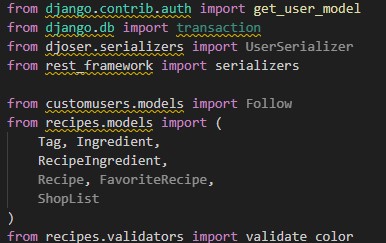
from django.db import transaction: Этот импорт позволяет вам использовать декоратор @transaction.atomic, который обеспечивает атомарность транзакций в базе данных. Это означает, что если вы выполняете несколько операций базы данных, которые должны выполняться как одна операция, то если какая-либо из них завершается неудачей, все изменения откатываются.

from djoser.serializers import UserSerializer: Этот импорт позволяет вам использовать сериализатор UserSerializer из библиотеки djoser. Сериализаторы в Django REST Framework используются для преобразования данных в JSON и обратно.

from rest\_framework import serializers: Этот импорт позволяет вам использовать различные классы и функции из библиотеки rest\_framework, которая является основной библиотекой для создания API в Django.

Остальные импорты относятся к приложению и импортируют различные модели, которые используются в нашем приложении. Например, Tag, Ingredient, RecipeIngredient, Recipe, FavoriteRecipe, ShopList и Follow - это модели, которые, похоже, определены в нашем приложении recipes и customusers соответственно.

from recipes.validators import validate\_color: Этот импорт позволяет использовать функцию validate\_color из модуля validators в пакете recipes. Функция validate\_color использована для валидации цвета в приложении.



### 4.2 Первая часть кода

1. В коде на рисунке выполняются две основные задачи:

Получение пользовательской модели пользователя: get\_user\_model()

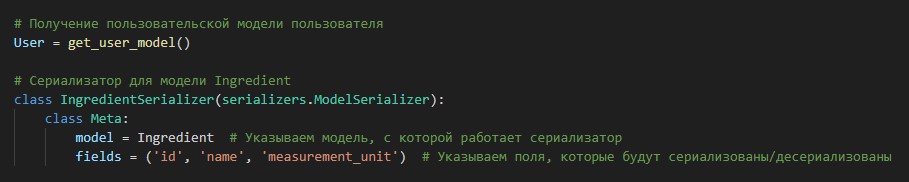
Эта строка использует функцию get\_user\_model() из django.contrib.auth, чтобы получить класс модели пользователя, который используется в нашем проекте.

Определение сериализатора для модели Ingredient:

Здесь мы создаем класс IngredientSerializer, который является подклассом serializers.ModelSerializer.

Внутри класса IngredientSerializer мы определяем вложенный класс Meta, который содержит метаданные для сериализатора. В этом случае мы указываем, что сериализатор должен работать с моделью Ingredient и сериализовать/десериализовать только поля id, name и measurement\_unit.

Таким образом, сериализатор IngredientSerializer будет преобразовывать экземпляры модели Ingredient в формат JSON и обратно, сериализуя/десериализуя только указанные поля.



2) В данной строчке кода определяется сериализатор для модели RecipeIngredient, который используется для преобразования экземпляров модели в формат JSON и обратно.

Вот что делает каждый компонент нашего кода на рисунке:

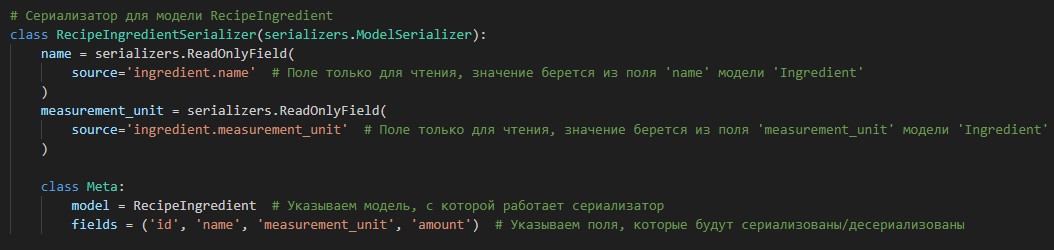
Определение сериализатора RecipeIngredientSerializer:

В данном коде мы добавляем два дополнительных поля только для чтения в сериализатор. Они используются для отображения имени и единицы измерения ингредиента, которые связаны с RecipeIngredient. source='ingredient.name' и source='ingredient.measurement\_unit' указывают, что значения этих полей должны браться из полей name и measurement\_unit соответственно модели Ingredient, связанной через внешний ключ в модели RecipeIngredient.

Определение метаданных сериализатора:

Внутри класса Meta мы указываем, что сериализатор должен работать с моделью RecipeIngredient и сериализовать/десериализовать поля id, name, measurement\_unit и amount.

Таким образом, сериализатор RecipeIngredientSerializer будет преобразовывать экземпляры модели RecipeIngredient в формат JSON и обратно, сериализуя/десериализуя указанные поля, включая дополнительные поля name и measurement\_unit, которые берут значения из связанной модели Ingredient.



3) В коде на рисунке выполняются две основные задачи:

В нашем коде определяется сериализатор для модели Tag, который используется для преобразования экземпляров модели в формат JSON и обратно. Сериализатор наследуется от serializers.ModelSerializer, что позволяет автоматически создавать сериализаторы на основе полей модели.

Определение сериализатора TagSerializer:

Мы создаем класс TagSerializer, который является подклассом serializers.ModelSerializer.

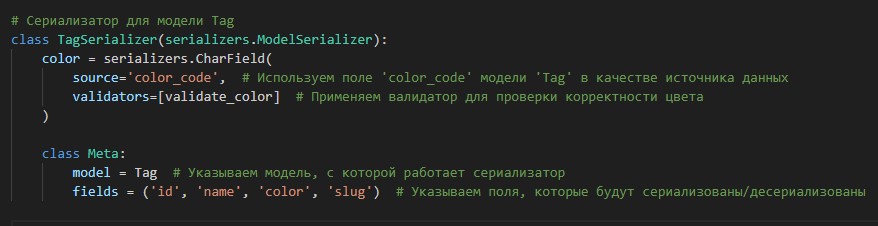
Добавление поля color с кастомным источником данных и валидаторами:

В этом коде мы добавляем поле color в сериализатор. Оно является экземпляром serializers.CharField, который представляет текстовое поле. Мы указываем, что значение этого поля должно браться из поля color\_code модели Tag. Кроме того, мы применяете валидатор validate\_color, который проверяет, является ли введенный цвет корректным.

Определение метаданных сериализатора:

Внутри класса Meta мы указываете, что сериализатор должен работать с моделью Tag и сериализовать/десериализовать поля id, name, color и slug.

Таким образом, сериализатор TagSerializer будет преобразовывать экземпляры модели Tag в формат JSON и обратно, сериализуя/десериализуя указанные поля, включая поле color, которое использует поле color\_code модели Tag в качестве источника данных и применяет валидатор validate\_color для проверки корректности цвета.



### 4.3 Вторая часть кода

1) В коде на рисунке выполняются две основные задачи:

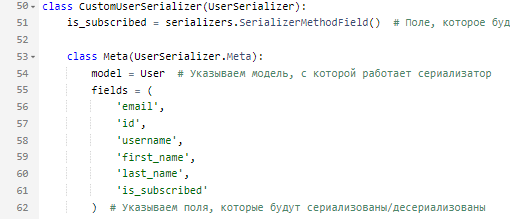
В данном отрезке кода создается класс CustomUserSerializer, который наследуется от UserSerializer.

Добавляется поле is\_subscribed, которое будет вычисляться с помощью метода get\_is\_subscribed. Используется SerializerMethodField() для определения этого поля.

Внутри CustomUserSerializer объявляется вложенный класс Meta, который наследуется от Meta класса базового сериализатора.

В Meta классе указывается модель User, которая будет использоваться сериализатором.

В поля перечисляются поля модели, которые будут сериализованы/десериализованы: электронная почта, id, имя пользователя, first\_name, last\_name, и is\_subscribed.



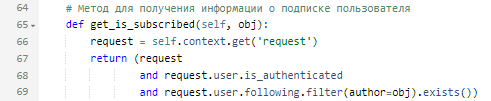
2) В коде на рисунке выполняются две основные задачи:

Определяется метод get\_is\_subscribed, который будет вызываться для вычисления значения поля is\_subscribed.

Метод принимает два аргумента: self (ссылка на экземпляр класса) и obj (объект, который сериализуется).

Получается запрос из контекста сериализатора. А далее проверяется, что запрос не пуст, и что пользователь в запросе аутентифицирован.

И проверяется, есть ли у текущего пользователя подписка.



3) В коде на рисунке выполняются две основные задачи:

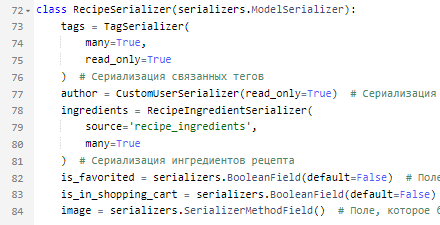
Создается класс RecipeSerializer, который наследуется от serializers.ModelSerializer.

Поле tags сериализуется с помощью TagSerializer. Это указывает, что каждый рецепт может иметь несколько тегов, и они только для чтения (read\_only=True).

Поле author сериализуется с использованием CustomUserSerializer. Это означает, что информация об авторе рецепта будет включена в сериализованные данные рецепта, и это поле только для чтения.

Поле ingredients сериализуется с использованием RecipeIngredientSerializer. Поле source='recipe\_ingredients' указывает, что данные для этого поля берутся из связанной модели recipe\_ingredients. Это поле также может содержать несколько записей (many=True).

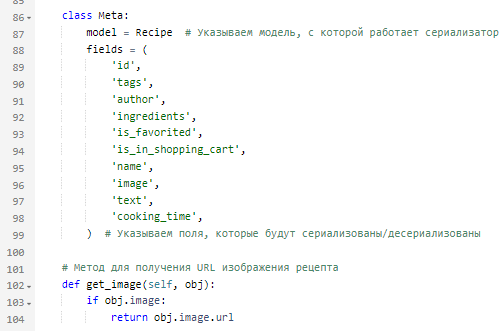
Поле is\_favorited указывает, находится ли рецепт в избранном. Поле is\_in\_shopping\_cart указывает, находится ли рецепт в списке покупок. Поле image используется для получения URL изображения рецепта.



4) В коде на рисунке выполняются две основные задачи:

Внутри RecipeSerializer объявляется вложенный класс Meta. Указывается модель Recipe, с которой работает сериализатор. В fields перечисляются поля модели, которые будут сериализованы/десериализованы.

Определяется метод get\_image, который принимает два аргумента: self (ссылка на экземпляр класса) и obj (объект, который сериализуется). Если у объекта obj есть изображение (obj.image), возвращается URL этого изображения.



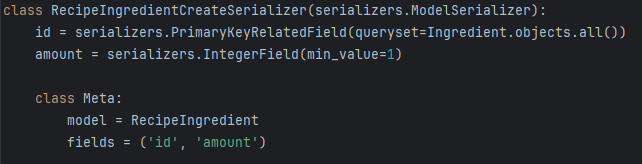
### 4.4 Третья часть кода

1) В коде на рисунке выполняется описание сериализатора:

Определяем класс RecipeIngredientCreateSerializer, наследующийся от класса serializers.ModelSerializer. Это значит, что мы используем модельный сериализатор, который автоматически генерирует набор полей на основе модели Django.

В качестве ключевых полей сериализатора установим поля: id и amount, где id – это поле, которое представляет собой внешний ключ на модель Ingredient. Оно позволяет выбрать существующий объект из базы данных Ingredient, а amount представляет собой целочисленное значение и задаёт количество ингредиентов.

class Meta: Этот класс используется для указания метаданных для сериализатора. Далее мы должны указать модель базы данных, чтобы этот сериализатор работал с моделью RecipeIngredient, и поля для которых будет происходить сериализация. В данном случае это id и amount они определяют, какие поля должны быть включены в сериализованный вывод.



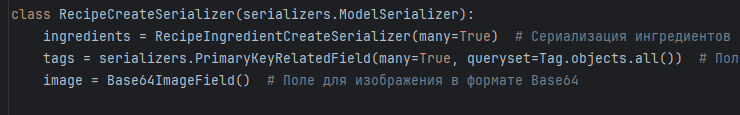
Описание сериализатора RecipeCreateSerializer на рисунке:

Определяем ключевые поля сериализатора: ingredients, tags, image

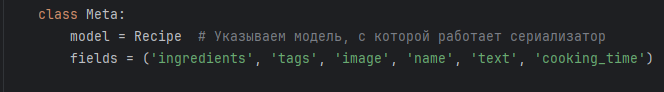
Ingredients: Поле, которое сериализует список ингредиентов.

Tags: Поле для связи с моделью Tag, принимающее множество тегов.

Image : Поле для фотографии в формате Base64



Указываем модель, с которой работает серилизатор в данном случае это Recipe и указываем поля по которым будет происходить сериализация и десериализация на рисунке:



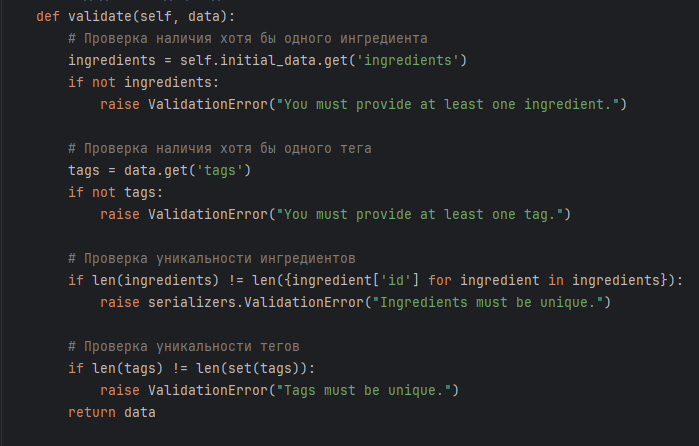
Создам метод для валидации данных, который выполняет следующий функционал на рисунке::

Проверка наличия хотя бы одного ингредиента. Если ингредиенты не указаны, выбрасывается ошибка валидации.

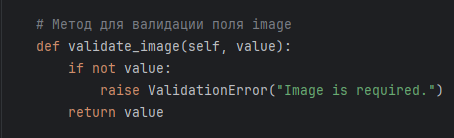
Проверка наличия хотя бы одного тега. Если теги не указаны, выбрасывается ошибка валидации.

Проверка уникальности ингредиентов. Если ингредиенты не уникальны, выбрасывается ошибка валидации.

Проверка уникальности тегов. Если теги не уникальны, выбрасывается ошибка валидации.



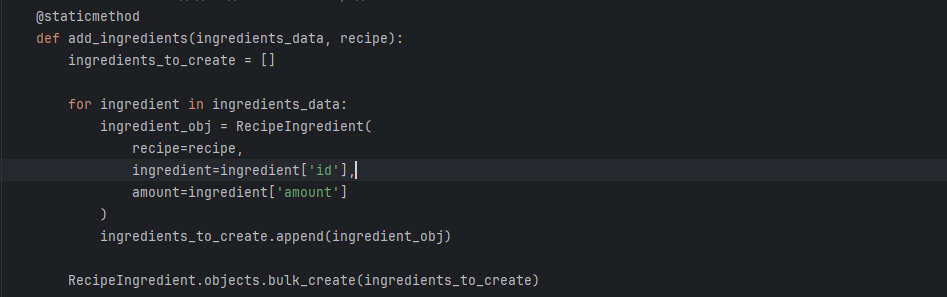
Далее создаем отдельный метод для проверки поля image, который проверяет наличие изображения (на рисунке). Если изображение не указано, выбрасывается ошибка валидации.



### 4.5 Четвертая часть кода

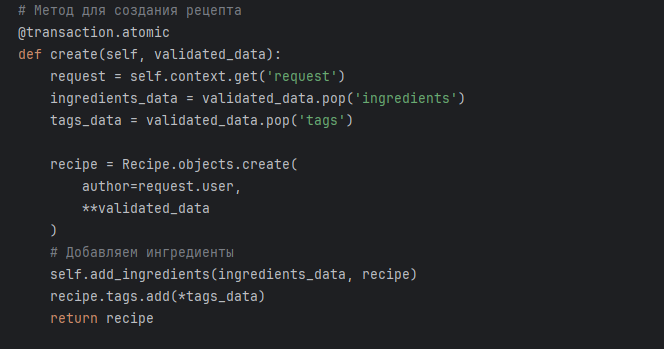
1) В коде (на рисунке) выполняются создание списка ingredients\_to\_create: Инициализируем пустой список для хранения объектов RecipeIngredient.

Заполнение списка: Итерируем ingredients\_data, создаём объект RecipeIngredient с указанием рецепта (recipe), ингредиента (ingredient['id']) и количества (ingredient['amount']). Затем добавляем объект в ingredients\_to\_create.

С помощью bulk\_create сохраняем все объекты из списка в базу данных. 

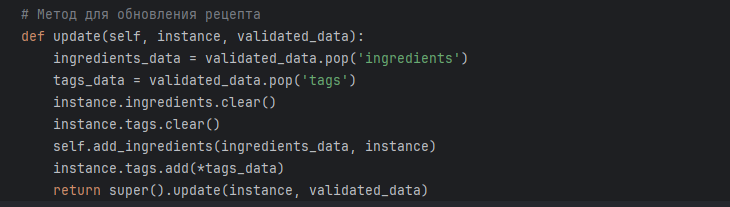
2) В коде на рисунке выполняется @transaction.atomic: Декоратор, обеспечивающий выполнение метода в транзакции для целостности данных.

1. Извлекаем объект запроса из контекста сериализатора и сохраняем в request.
2. Удаляем и сохраняем данные ингредиентов и тегов из validated\_data.
3. Создаём новый объект Recipe, устанавливая автора и остальные поля из validated\_data.
4. Вызываем метод add\_ingredients для добавления ингредиентов к рецепту.
5. Добавляем теги к рецепту.
6. Возвращаем созданный объект Recipe.



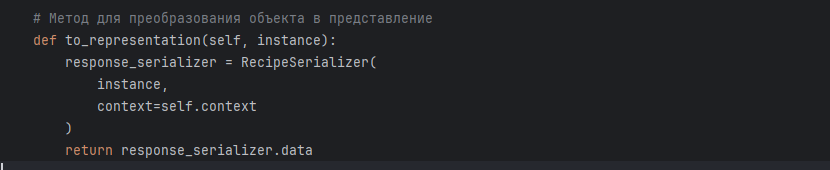
3) В коде на рисунке выполняется:

1. Удаляем и сохраняем данные ингредиентов и тегов из validated\_data.
2. Очищаем текущие ингредиенты и теги у экземпляра рецепта.
3. Вызываем метод add\_ingredients для добавления новых ингредиентов к рецепту.
4. Добавляем новые теги к рецепту.
5. Вызываем метод родительского класса для обновления оставшихся данных в экземпляре и возвращаем обновленный объект.



4) В коде на рисунке выполняем:

1. Создаём экземпляр RecipeSerializer с переданным экземпляром instance и контекстом self.context. Это позволяет использовать другой сериализатор для преобразования объекта в его представление.
2. Возвращаем данные сериализованного объекта, полученные от response\_serializer.



### 4.6 Шестая часть кода

1)Сериализатор краткого представления рецепта

Назначение: Данный сериализатор используется для преобразования объектов модели "Рецепт" в формат JSON с краткой информацией. Он преобразует объекты модели Recipe в формат JSON, содержащий только идентификатор рецепта, название, URL изображения и время приготовления. Он предназначен для использования в ситуациях, когда требуется представить рецепт в упрощенной форме, например, в списках или в предпросмотре.

Ниже, на рисунке 1.1 представлен код данного сериализатора в проекте:

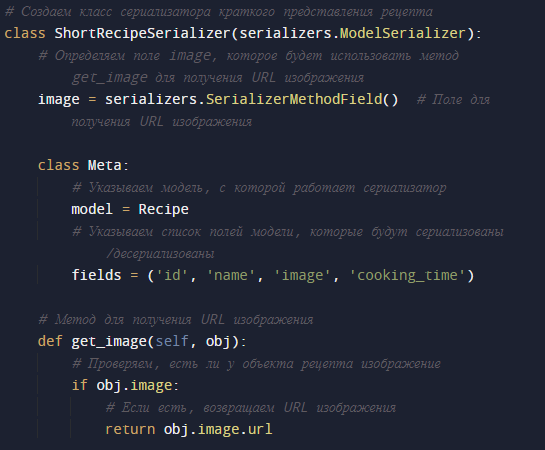


Рисунок 1.1 – код сериализатора ShortRecipeSerializer

Класс ShortRecipeSerializer определяет сериализатор для краткого представления рецепта. Он включает поле image, которое использует метод get\_image для получения URL изображения. В методе get\_image проверяется, есть ли у объекта рецепта изображение, и возвращает его URL, если оно существует.

Поля:

id: Идентификатор рецепта.

name: Название рецепта.

image: URL изображения рецепта.

cooking\_time: Время приготовления рецепта.

Методы:

get\_image(self, obj): Метод для получения URL изображения. Если у рецепта есть изображение, возвращает его URL, иначе возвращает None.

Создаем класс сериализатора краткого представления рецепта: Здесь определяется новый класс с названием "ShortRecipeSerializer". Этот класс используется для сериализации данных о рецептах в кратком формате. Ниже, на рисунке 1.2 приведен код для реализации этого:



Рисунок 1.2 – создание класса ShortRecipeSerializer

Определяем поле image, которое будет использовать метод get\_image для получения URL изображения: Создается поле с именем "image". Это поле будет использовать метод "get\_image", чтобы получить URL изображения рецепта. Ниже, на рисунке 1.3 приведен код, как это реализовано:

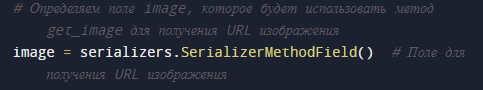


Рисунок 1.3 – определение поле image

Указываем модель, с которой работает сериализатор:

Список полей модели, которые будут сериализованы/десериализованы: Здесь определяются поля модели, которые будут включены в сериализацию и десериализацию. Это включает в себя идентификатор рецепта, название, изображение и время приготовления. Ниже, на рисунке 1.4 приведен код для реализации этого:

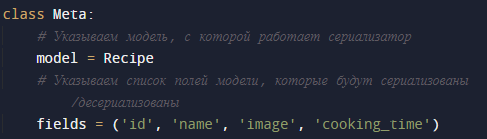


Рисунок 1.4 – определение модели, списка полей

Метод для получения URL изображения: Этот метод используется для получения URL изображения рецепта. Он проверяет, есть ли изображение для рецепта, и, если есть, возвращает его URL.

Проверяем, есть ли у объекта рецепта изображение: Эта строка кода проверяет, есть ли у рецепта изображение.

Ниже, на рисунке 1.5 приведен код для реализации данного метода:

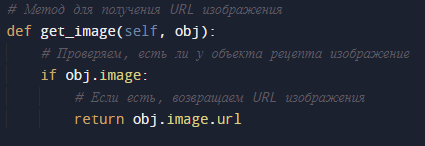


Рисунок 1.5 – метод для получения URL изображения

Таким образом, этот код определяет сериализатор для представления краткой информации о рецептах, включая их изображения, в формате JSON.

2) Сериализатор подписок пользователя

Назначение: Этот сериализатор используется для представления информации о подписках пользователя. Его целью является включение в выходные данные основной информации о пользователе, а также списка его рецептов в кратком формате.

Ниже, на рисунке 2.1 представлен код данного сериализатора в проекте:



Рисунок 2.1 – код сериализатора FollowUserSerializer

Дополнительные поля:

recipes: Рецепты пользователя в кратком формате.

recipes\_count: Количество рецептов пользователя.

Методы:

get\_recipes(self, obj): Метод для получения рецептов пользователя. Если у запроса есть параметр recipes\_limit, ограничивает количество рецептов до указанного лимита. Использует ShortRecipeSerializer для сериализации рецептов.

get\_recipes\_count(self, obj): Метод для получения общего количества рецептов пользователя.

Сериализатор для представления подписок пользователя: Этот комментарий вводит новый класс, который используется для сериализации данных о подписках пользователя. Ниже, на рисунке 2.2 приведен код для реализации этого:



Рисунок 2.2 – Сериализатор для представления подписок пользователя

Определяется поле recipes для получения рецептов пользователя: Здесь создается поле с именем "recipes". Оно будет использовать метод get\_recipes для получения списка рецептов пользователя. Ниже, на рисунке 2.3 приведен код для реализации этого:



Рисунок 2.3 – Сериализатор для представления подписок пользователя

Определяется поле recipes\_count для получения количества рецептов пользователя: Создается поле с именем "recipes\_count", которое будет использовать метод get\_recipes\_count для получения общего количества рецептов пользователя. Ниже, на рисунке 2.4 приведен код для реализации этого:



Рисунок 2.4 – Определяется поле recipes\_count для получения количества рецептов пользователя

Указываем поля, которые будут сериализованы/десериализованы: В этой строке указываются поля, которые будут включены в сериализацию и десериализацию. Это включает поля из базового класса CustomUserSerializer, а также поля "recipes" и "recipes\_count". Ниже, на рисунке 2.5 приведен код для реализации этого:



Рисунок 2.5 – Указываем поля, которые будут сериализованы/десериализованы

Метод для получения рецептов пользователя: Этот метод используется для получения списка рецептов пользователя. Он извлекает список рецептов из объекта пользователя, а также может ограничить количество рецептов в соответствии с параметром recipes\_limit запроса. Для сериализации рецептов используется ShortRecipeSerializer. Ниже, на рисунке 2.6 приведен код для реализации этого:

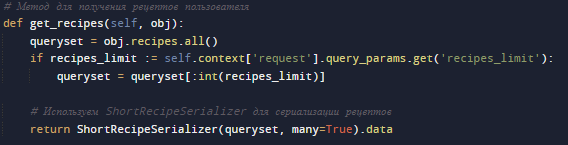


Рисунок 2.6 – Метод для получения рецептов пользователя

Метод для получения количества рецептов пользователя: Этот метод возвращает общее количество рецептов, принадлежащих пользователю. Он извлекает количество рецептов из объекта пользователя и возвращает это значение. Ниже, на рисунке 2.7 приведен код для реализации этого:

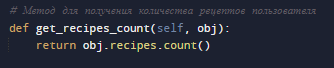


Рисунок 2.7 – Метод для получения количества рецептов пользователя

3)Сериализатор создания подписки

Назначение: Данный сериализатор применяется для создания новых подписок на пользователей. Он обеспечивает валидацию вводимых данных и представление созданной подписки.

Ниже, на рисунке 3.1 представлен код данного сериализатора в проекте:

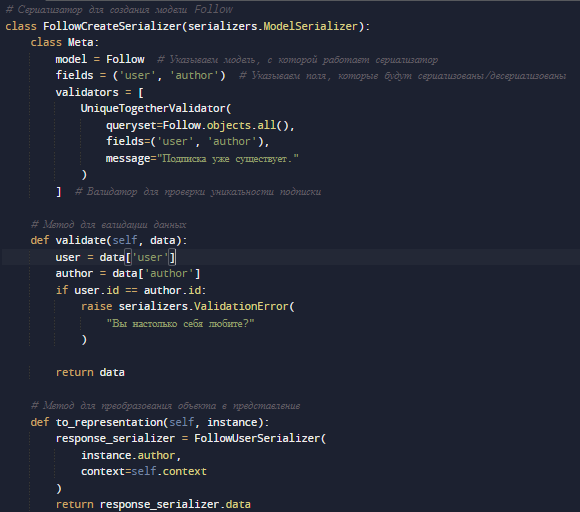


Рисунок 3.1 – код сериализатора FollowCreateSerializer

Этот сериализатор используется для создания подписки на пользователя. Проверяет уникальность подписки и валидирует данные.

Поля:

user: Пользователь, который подписывается.

author: Пользователь, на которого подписываются.

Валидация:

Убеждается, что пользователь не пытается подписаться на самого себя.

Методы:

validate(self, data): Метод для валидации данных перед созданием подписки.

to\_representation(self, instance): Метод для преобразования созданной подписки в представление. Создает представление автора подписки с помощью FollowUserSerializer.

Создаем класс сериализатора для создания подписки: Этот комментарий вводит новый класс, который используется для создания сериализатора данных для создания подписок Ниже, на рисунке 3.2 представлен код данного сериализатора в проекте:



Рисунок 3.2 – Создаем класс сериализатора для создания подписки

Указываем модель, с которой работает сериализатор: Здесь определяется модель, с которой будет работать сериализатор. В данном случае это модель Follow, представляющая сущность подписки. Ниже, на рисунке 3.3 представлен код данного сериализатора в проекте:



Рисунок 3.3 – Указываем модель, с которой работает сериализатор

Указываем поля, которые будут сериализованы/десериализованы: Этот код определяет, какие поля модели будут включены в процесс сериализации и десериализации. В данном случае это поля user и author. Ниже, на рисунке 3.4 представлен код данного сериализатора в проекте:



Рисунок 3.4 – Указываем поля, которые будут сериализованы/десериализованы

Валидатор для проверки уникальности подписки: Здесь создается валидатор, который проверяет уникальность подписки, чтобы избежать дублирования. Если подписка уже существует для указанных пользователей, будет сгенерировано сообщение об ошибке.

В конкретном коде, UniqueTogetherValidator является типом валидатора, который обеспечивает уникальность комбинации значений полей user и author для модели Follow. Он гарантирует, что не будет создано двух одинаковых записей подписок для одной и той же пары пользователей. Если валидация не пройдет (то есть если такая подписка уже существует), будет сгенерировано сообщение об ошибке с текстом "Подписка уже существует." Ниже, на рисунке 3.5 представлен код данного сериализатора в проекте:

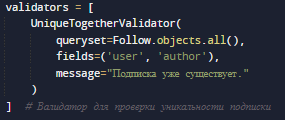


Рисунок 3.5 – Валидатор для проверки уникальности подписки

Метод для валидации данных: Этот метод выполняет валидацию данных, введенных пользователем при создании новой подписки. Он проверяет, что пользователь не пытается подписаться на самого себя. Ниже, на рисунке 3.6 представлен код данного сериализатора в проекте:

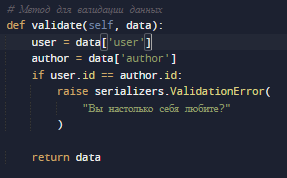


Рисунок 3.6 – Метод для валидации данных

Метод для преобразования объекта в представление: Этот метод используется для преобразования созданной подписки в представление, которое будет возвращено клиенту. Он создает сериализатор для представления пользователя, на которого оформлена подписка, и возвращает данные этого сериализатора. Ниже, на рисунке 3.7 представлен код данного сериализатора в проекте:

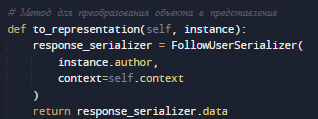


Рисунок 3.7 – Метод для преобразования объекта в представление

1)Сериализатор для выполнения действий с рецептами

НазначениеКласс BaseActionSerializer используется в качестве базового сериализатора для операций с рецептами, таких как добавление в избранное или список покупок. Он переопределяет метод to\_representation, чтобы представлять данные рецепта в кратком формате.

ИспользованиеЭтот сериализатор применяется для сериализации объектов, содержащих информацию о рецепте и связанных пользователях. Он предоставляет краткое представление рецепта, используя ShortRecipeSerializer.

Как работает

Поля: user и recipe указывают на пользователя и рецепт соответственно.

Валидатор UniqueTogetherValidator: Проверяет уникальность сочетания user и recipe в таблице ShopList, чтобы предотвратить дублирование.

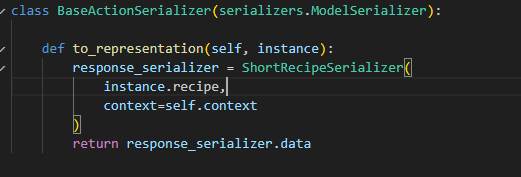


Рисунок – код сериализатора BaseActionSerializer

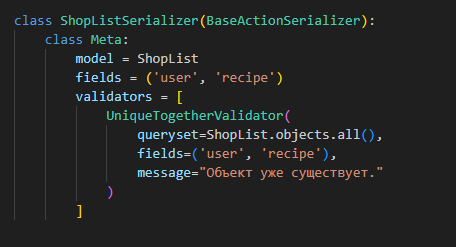
2)Сериализатор для списка покупок

#### Назначение ShopListSerializer используется для сериализации данных о списке покупок пользователя. Этот сериализатор включает в себя информацию о пользователе и рецепте, который был добавлен в список покупок.

#### Использование Применяется при работе со списком покупок пользователей, обеспечивая уникальность добавленных рецептов для каждого пользователя.

**Как работает**

* Поля: user и recipe указывают на пользователя и рецепт соответственно.
* Валидатор UniqueTogetherValidator: Проверяет уникальность сочетания user и recipe в таблице ShopList, чтобы предотвратить дублирование.



### Рисунок – код сериализатора ShopListSerializer

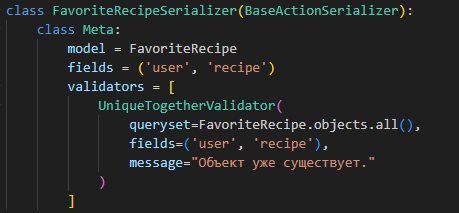
### Сериализатор для избранных рецептов

#### Назначение FavoriteRecipeSerializer используется для сериализации данных о рецептах, которые были добавлены пользователями в избранное. Он обеспечивает уникальность сочетания пользователя и рецепта.

#### Использование Этот сериализатор применяется для работы с избранными рецептами, позволяя пользователям добавлять рецепты в список избранного и проверять уникальность этих действий.

#### Как работает

* Поля: user и recipe указывают на пользователя и рецепт соответственно.
* Валидатор UniqueTogetherValidator: Проверяет уникальность сочетания user и recipe в таблице FavoriteRecipe, чтобы предотвратить дублирование.



### Рисунок – код сериализатора FavoriteRecipeSerializer

### Описание методов и валидаторов

#### Метод to\_representation Метод to\_representation преобразует экземпляр модели в представление данных, используя ShortRecipeSerializer для представления данных рецепта в кратком формате.

#### Валидатор UniqueTogetherValidator Валидатор UniqueTogetherValidator обеспечивает уникальность сочетания полей user и recipe в моделях ShopList и FavoriteRecipe. Если объект с такими значениями полей уже существует, будет сгенерировано сообщение об ошибке.

## 6. ЗАКЛЮЧЕНИЕ

### Преимущества использования сериализаторов Django

Сериализаторы Django предлагают ряд преимуществ, которые делают их ценным инструментом для разработки REST API:

* **Простота использования:** Сериализаторы Django просты в использовании и настройке, что позволяет быстро создавать API.
* **Гибкость:** Поддерживается сериализация различных типов данных, включая модели Django, объекты Python и базовые типы данных.
* **Валидация данных:** Сериализаторы предоставляют встроенные функции валидации данных, помогая гарантировать целостность данных.
* **Документация:** Сериализаторы автоматически генерируют подробную документацию API, упрощая ее использование разработчиками.

### Рекомендации по выбору сериализатора

Выбор подходящего типа сериализатора зависит от ваших конкретных потребностей:

* **ModelSerializer:** Используйте ModelSerializer для сериализации и десериализации моделей Django, если вам нужна простота и автоматизация.
* **JSONSerializer:** Используйте JSONSerializer, если вам нужно сериализовать объекты Python в JSON с большей гибкостью.
* **Serializer:** Используйте Serializer, если вам нужен полный контроль над сериализацией и десериализацией данных.