



# Продолжение работы с ORM, шаблоны и формы

Лектор: Матвеева Светлана

# Важно



- Отмечаемся на портале
- Слушаем результаты прошлого квиза
- Проходим новый квиз



# Темы на сегодня ✓

- Django ORM
- Практики, советы и трюки Django ORM
- Шаблоны
- Формы



# Еще о Django ORM



# Q object

**Подзапрос**, или **Q объект**, позволяет делать сложные запросы к базе через ORM.

Их можно комбинировать с помощью операторов

- & – and
- | – or
- ^ – xor

Подзапрос с Q объектом может быть инвертирован с помощью ~ (not)

Q объекты через «,» объединяются через &

Примеры:

```
Q(question_startswith='Who') | Q(question_startswith='What')
```

```
Q(question_startswith='What') & ~Q(question_startswith='What about')
```

<https://docs.djangoproject.com/en/4.1/topics/db/queries/#complex-lookups-with-q-objects>

# F object

Объект F хранит преобразованное значение поля модели (аннотированный столбец).

Его значение рассчитывается на стороне базы данных (не на стороне python), что помогает увеличить производительность, сократив количество запросов.

Может быть эффективно использован для обновления данных:

```
>>> Reporter.objects.update(rating=F('rating') + 1)
```

<https://docs.djangoproject.com/en/4.1/ref/models/expressions/#django.db.models.F>

# Union

Union объединяет кверисеты.

```
>>> queryset1 = Reporter.objects.filter(...some condition...)
>>> queryset2 = Reporter.objects.filter(...some condition...)
>>> queryset = queryset1.union(queryset2)
```

Также .union() может объединять кверисеты разных моделей, но важно, чтобы у них были либо полностью идентичные атрибуты, либо было какое-то общее подмножество атрибутов.

```
>>> queryset1 = Hero.objects.all().values_list(
    "name", "gender"
)
>>> queryset2 = Villain.objects.all().values_list(
    "name", "gender"
)
```

<https://docs.djangoproject.com/en/4.1/ref/models/querysets/#union>

# Defer и Only

**Defer** исключает некоторые атрибуты из первичного вычисления кверисета, как правило - атрибуты с большими и ненужными для запроса данными. Однако «отложенные» атрибуты могут быть дovskyчислены.

```
>>> Book.objects.filter(...some condition...).defer('description')
```

**Only**, наоборот явно требует указать какие атрибуты (столбцы таблицы) нужно выгрузить.

<https://docs.djangoproject.com/en/4.1/ref/models/querysets/#django.db.models.query.QuerySet.defer>

# Subquery



В QuerySet может быть добавлен явный подзапрос - **Subquery**.

Если подзапрос должен как-то использовать какое-то значение внешнего запроса, нужно использовать **OuterRef** (по сути это ссылка на поле из внешнего запроса).

OuterRef очень похож на F-выражения.

```
>>> Author.objects.annotate(  
    rewarded_books_ids=Subquery(  
        Book.objects.filter(authors=OuterRef('pk'), givings__isnull=False).values_list('id')  
    )  
.values()
```

<https://docs.djangoproject.com/en/4.1/ref/models/expressions/#subquery-expressions>

# Рекомендации и практики работы с ORM



# Переопределение save()

```
def save(self, *args, **kwargs):  
    do_smth()  
    super(ModelName, self).save(*args, **kwargs)
```

Может быть полезно для:

- отправки уведомлений пользователю
- заполнения служебной (недоступной пользователю для изменения) информации, например, кто и когда сохранил этот экземпляр модели

# Переопределение delete()

```
def delete(self, *args, **kwargs):  
    self.books.delete()  
    super(ModelName, self).delete(*args, **kwargs)
```

Используем, когда нужно почистить связанные через m2m данные.

# Переопределение delete() на fake\_delete()

```
def fake_delete(self, *args, **kwargs):
    self.is_deleted = True
    super(ModelName, self).save(*args, **kwargs)
```

Используем, когда надо скрыть запись от пользователей, но при этом важно оставить ее в базе.

# Сигналы

```
>>> from django.db.models import pre_save, post_save # пример импорта
```

Виды сигналов:

- **pre\_save** - до сохранения
- **post\_save** - после сохранения
- **pre\_delete** - до удаления
- **post\_delete** - после удаления
- **m2m\_changed** - при изменении данных связанной m2m модели
- **request\_started** - при начале запроса
- **request\_finished** - в конце запроса

# Сигналы

Регистрация сигнала

`Signal.connect(receiver, sender=None, weak=True, dispatch_uid=None)`

Пример:

`pre_save.connect(receiver=create_booking_calendar, sender=Table)`

`post_save.connect(receiver=event_notify_author, sender=TableBooking)`

# Обновление объектов и querysets

`.update()` используется для обновления кверисетов

```
>>> queryset = Book.objects.all()
```

```
>>> queryset.update(description='test description')
```

Важно: update напрямую обновляет данные в базе, не затрагивая возможную дополнительную логику, определенную в save()

`.save()` используется для обновления инстанса модели

```
>>> book = Book.objects.get(title='On the road')
```

```
>>> book.description = 'test description'
```

```
>>> book.save()
```

# Обновление объектов и querysets

- update напрямую обновляет данные в базе сразу для всего queryset, поэтому работает сильно быстрее
- update применим когда мы знаем на что обновить поле/поля без обращения к каждой отдельной записи
- при использовании update() не выполняется логика в переопределенном методе save(), а также не вызываются сигналы

Помним, что можем использовать F-выражения для быстрого обновления кверисета с помощью update():

```
>>> queryset.update(field = F('field') + 1)
```

# Использование абстрактных моделей

```
classTimeStampModelMixin(models.Model):
    created_at = models.DateTimeField(verbose_name='Добавлена', auto_now_add=True)
    updated_at = models.DateTimeField(verbose_name='Обновлена', auto_now=True)

class Meta:
    abstract = True
```

Можно наследоваться от такого миксина везде, где нужно сохранять время создания и обновления.

# Копирование инстансов модели

```
blog = Blog(name='My blog', tagline='Blogging is easy')
blog.save() # blog.pk == 1
```

```
blog.pk = None
blog._state.adding = True #
blog.save() # blog.pk == 2
```

# Копирование инстансов отнаследованной модели

```
class ThemeBlog(Blog):
    theme = models.CharField(max_length=200)

django_blog = ThemeBlog(name='Django', tagline='Django is easy', theme='python')
django_blog.save()

django_blog.pk = None
django_blog.id = None
django_blog._state.adding = True
django_blog.save() # django_blog.pk == 4
```

# ПОИСК

```
>>> Author.objects.filter(name__contains='Terry')
>>> Author.objects.filter(name__icontains='terry') # без учета регистра
```

Поиск в PostgreSQL:

```
>>> City.objects.filter(name__unaccent="México") #ищет без учета диакритических знаков типа Ää, Öö и Üü
['<City: Mexico>']
```

```
>>> User.objects.filter(first_name__unaccent__startswith="Jerem")
['<User: Jeremy>', '<User: Jérémie>', '<User: Jérémie>', '<User: Jeremie>']
```

```
>>> Author.objects.filter(name__unaccent__icontains='Helen')
[<Author: Helen Mirren>, <Author: Helena Bonham Carter>, <Author: Hélène Joy>]
```

О том, что еще Django умеет искать вместе с PostgreSQL:

<https://docs.djangoproject.com/en/4.1/ref/contrib/postgres/search/>

# db\_index, unique, unique\_together

**db\_index** - создает индекс в БД

```
>>> name = models.CharField('название', max_length=100, db_index=True)
```

**unique** - делает содержимое поля уникальным

```
>>> codename = models.CharField('кодовое название', max_length=100, unique=True)
```

**unique\_together** - делает связку полей уникальной

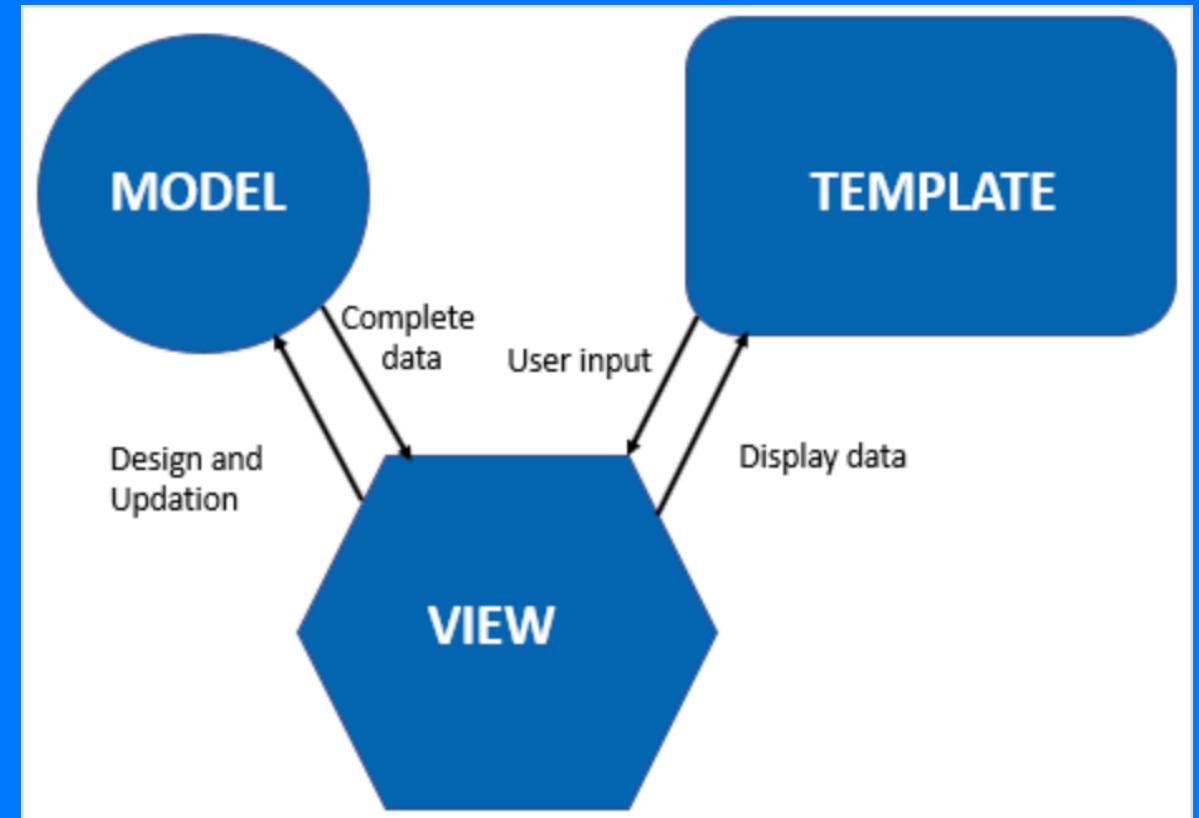
```
>>> class Reward(models.Model):  
    title = models.CharField(verbose_name='Название премии', max_length=100)  
    codename = models.CharField(verbose_name='Коднейм', max_length=100)  
    class Meta:  
        unique_together = ('title', 'codename')
```

# Django templates



# MTV (Model-Template-View)

- **Model** - отвечает за хранение данных
- **View** - отвечает за обработку запросов
- **Template** - отвечает за вывод динамического контента пользователю



# Термины

**Шаблон** - это образец для генерации веб-страницы, отправляемой в качестве ответа на запрос.

**Рендеринг** - процесс генерации веб-страницы.

**Шаблонизатор** - некоторая подсистема фреймворка, со своим синтаксисом, выполняющая рендеринг страницы на основе шаблона и контекста.

# SSR vs CSR

SSR	CSR
 Ideal for sites serving only static content	 Ideal for web apps
 Fast initial page load	 Fast rendering after initial load
 No JS dependency	 Rich site interaction
 Easy for search engine bots to crawl and index a site because the content exists before the user receives it - more straightforward SEO	 Reduces server load
 Multiple Server Requests	 Incorrect rendering & API response delays an SEO risk
 Full Page Reloads	 Slower initial load time
 Non-rich site interactions	 External library requirements
 Higher latency, Prone to vulnerability	 Higher memory consumption, Relies on capabilities of end user's browser

# SSR vs CSR

Подробно о разнице SSR и CSR тут

[https://www.youtube.com/watch?v=xg-lajRmCco&ab\\_channel=Веб-разработка-DevMagazine](https://www.youtube.com/watch?v=xg-lajRmCco&ab_channel=Веб-разработка-DevMagazine)



# Получение ответа через template

- 1) Запрос попадает в обработчик **view**
- 2) View-обработчик формирует контекст ответа, как правило словарь/queryset, который передается в **template** (шаблон)
- 3) **Template** парсит контекст и достает нужные переменные для генерации страницы

# Преимущества шаблонов

- Можно написать полноценное веб-приложение без фронтенда (не очень красивое, но вполне рабочее)
- Можно шаблонизировать не только html, а например email, CSV или XML

# Доступно из коробки Django

Django поддерживает два вида шаблонизаторов:

- **DTL (Django-Template-Language)** - используется по умолчанию
- **Jinja2**

Можно использовать другой шаблонизатор из доступный для python.

Можно написать свой.

# Шаблонизаторы для Python

- Mako (шаблонизатор для фреймворка Pyramid)
- Chameleon
- Cheetah
- Diazo
- evoque
- Genshi
- Juno
- Myghty
- pyratemp
- pystache

<https://www.fullstackpython.com/template-engines.html>

# Шаблон на DLT

Шаблон содержит **переменные**, которые заменяются значениями при рендеринге, и **теги**, управляющие логикой шаблона.

Обозначение переменных:

`{{ variable }}`

Обозначение тегов:

`{% tag %}`

# Переменные

- Точно такие же требования к названиям как в python
- Через «.» можно обратиться к
  - Ключу словаря
  - Атрибуту экземпляра класса
  - Индексу в списке
- Можно использовать фильтры для изменения отображения переменной

# Фильтры для переменных

- Применяются через «|», например `{{ name|lower }}`
- Можно скомбинировать несколько фильтров последовательно `{{ name|lower|upper }}`
- Можно передать аргументы в фильтры через «::»  
`{{ user.birth_date|date:"D d M Y" }}`
- Встроенных фильтров около 60!

<https://docs.djangoproject.com/en/4.1/ref/templates/builtins/#ref-templates-builtins-filters>

# Теги

- Одиночные `{% tag %}`
- Оборачивающие `{% tag %} ... tag contents ... {% endtag %}`

# Теги

- **if-elif-else**
- **For**
- **extends** и **block** - используются при наследовании шаблонов
- **csrf\_token** - для защиты от Cross Site Request Forgery
- **include** - для включения другого шаблона с текущим контекстом
- **lorem** - тег для текста-рыбы
- **now** - тег с текущем временем
- **regroup** - тег, группирующий список похожих объектов по общему признаку
- **with** - тег который позволяет закэшировать что-нибудь

<https://docs.djangoproject.com/en/4.1/ref/templates/builtins/#std-templatetag-block>

# Настройки для работы с шаблонами

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            # ... some options here ...
        },
    },
]
```

# Django forms



# Django формы

Используются только **GET** и **POST** запросы

На **GET** запрос:

- Пустой экземпляр формы вставляется в шаблон
- Пользователь получает шаблон для заполнения

На **POST** запрос:

- Браузер объединяет данные формы и отправляет их на сервер
- Сервер валидирует отправленную форму с данными
- В случае ошибки присыпает ошибки
- В случае успешной вариации обновляет БД

# Django формы

- `django.forms.Form`
- `django.forms.ModelForm` - подкласс `django.forms.Form`, используется для сохранения моделей

# Django формы (поля)

- BooleanField
- CharField
- ChoiceField
- DateField
- DateTimeField
- DecimalField
- DurationField
- EmailField
- FileField
- FilePathField
- FloatField
- GenericIPAddressField
- ImageField
- IntegerField
- JSONField
- MultipleChoiceField
- NullBooleanField
- RegexField
- SlugField
- URLField

<https://docs.djangoproject.com/en/4.1/ref/forms/fields/>

# Django формы (виджеты полей)

Виджеты отвечают за отображение поля на веб-странице

```
>>> comment = forms.CharField(widget=forms.Textarea)
```

<https://docs.djangoproject.com/en/4.1/ref/forms/widgets/>

# Лекция закончена!

Спасибо самым любознательным за  
внимание :)

## Домашнее задание:

Досоздать все недостающие модели и  
недостающие views (пока что  
функциональные), чтобы соединить их с  
вашим фронтеном



# Что должен уметь ваш бэкенд при проверке ДЗ:

- создавать чат
- редактировать чат
- добавлять участника в чат
- удалять участника из чата
- удалять чат
- отправлять сообщение
- редактировать сообщение
- удалять сообщение
- получать список сообщений чата
- получать список чатов
- получать информацию о пользователе