

Классы представлений,  
регистрация,  
оптимизация

Классы представлений:  
ListView, DetailView,  
CreateView

Основы ORM Django за  
час

Mixins - убираем  
дублирование кода

Постраничная  
навигация (пагинация)

Регистрация  
пользователей на сайте

Делаем авторизацию  
пользователей на сайте

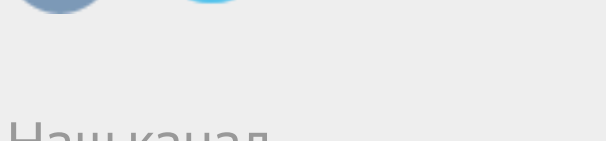
Оптимизация сайта с  
Django Debug Toolbar

Включаем кэширование  
данных

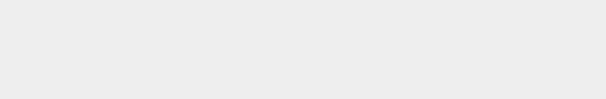
Использование капчи  
captcha

Тонкая настройка  
админ панели

Поделиться




Наш канал



Главная → Django → Классы представлений, регистрация, оптимизация

## Mixins - убираем дублирование кода

 [Смотреть материал на видео](#)



Архив проекта: [lesson-17-coolsite.zip](#)

На этом занятии мы с вами вынесем общий код классов представлений в отдельный класс, который можно воспринимать как миксин (mixin). Те из вас, кто хорошо знаком с ООП уже знают, что такое миксины и для чего они служат. Но я все же сделаю небольшую ремарку и подробнее поясню этот момент.

Вообще, миксины были придуманы для возможности единообразного оперирования объектами. Представьте, что разрабатывается интернет-магазин и для каждого товара предполагает получение следующих стандартных свойств:

- идентификатор;
- габариты;
- вес;
- цена.

Для материальных товаров все эти характеристики имеют смысл:



Но для информационных, таких как электронная книга или приложение для смартфона, габариты и вес не определены. Тем не менее, система магазина в целом, обращается к этим свойствам объектов (товаров). Как сделать так, чтобы каждый объект, представляющий товар, имел по умолчанию все нужные свойства? Конечно, мы можем непосредственно в классе их прописать:

```
class Goods:
    def getWeight(self):
        return self.weight

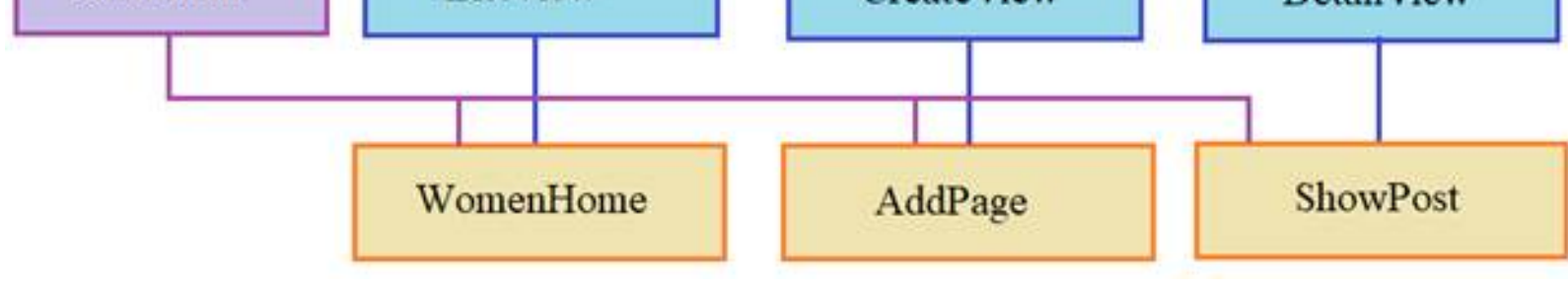
    def getPrice(self):
        return self.price
...
```

Но это начинает плохо работать, при наличии разветвленной иерархии объектов, когда каждый класс товара представляется отдельным классом:

Cars, Toys, Books, ...

Тогда пришлось бы в каждом прописывать эти методы, что нехорошо. Поэтому, как вы уже догадались, все это выносится в базовый класс, например, Goods. Но, иногда, наборы разрозненных классов не имеют единого базового, либо базовый класс находится на таком уровне абстракции, что в него писать такие конкретные методы – прямой путь к мешанине кода. Вот как раз для таких случаев, когда нужно дополнительно к уже существующей иерархии добавить какие-либо общие для разнородных классов данные и/или методы и применяется механизм миксинов.

В разных языках программирования миксины реализуются по разному. В частности, в Python, благодаря наличию механизма множественного наследования, примеси можно добавлять в виде отдельного базового класса:



Например, наши классы представлений можно дополнительно унаследовать от класса DataMixin, который мы отдельно определим. Причем, этот класс лучше прописывать первым в списке наследования:

```
class WomenHome(DataMixin, ListView):
...
```

Так как в нем могут быть атрибуты и методы, которые, затем, используются конструктором следующего класса – ListView. То есть, в Python, класс, записанный первым, первым и обрабатывается. Поэтому данные базового класса DataMixin переопределят (при необходимости) атрибуты следующего класса ListView.

Давайте теперь определим наш класс DataMixin и уберем дублирование кода из классов представлений. Первый вопрос, где прописать этот класс? Обычно, в Django все дополнительные, вспомогательные классы объявляют в отдельном файле приложения utils.py. Мы так и поступим. Создадим этот файл и в нем запишем класс DataMixin, следующим образом:

```
from .models import *

menu = [{
    'title': "0 сайте", 'url_name': 'about'},
    {'title': "Добавить статью", 'url_name': 'add_page'},
    {'title': "Обратная связь", 'url_name': 'contact'},
    {'title': "Войти", 'url_name': 'login'}
]

class DataMixin:
    def get_user_context(self, **kwargs):
        context = kwargs
        cats = Category.objects.all()
        context['menu'] = menu
        context['cats'] = cats
        if 'cat_selected' not in context:
            context['cat_selected'] = 0
        return context
```

Обратите внимание, я перенес сюда и главное меню, т.к. оно используется напрямую классом DataMixin. И вначале идет импорт моделей, т.к. мы используем класс Category для получения всех категорий.

Если вы помните, мы категории в шаблоне base.html сейчас отображаем с помощью созданного нами тега show\_categories. Это был искусственный пример, демонстрирующий возможность создания пользовательских тегов, теперь я его уберу и вместо него буду использовать переменную cats, которую передадим в шаблон. Соответственно, в шаблоне вернем строки для отображения рубрик:

```
{% for c in cats %}
    {% if c.pk == cat_selected %}
        <li class="selected">{{c.name}}</li>
    {% else %}
        <li><a href="{{ c.get_absolute_url }}">{{c.name}}</a></li>
    {% endif %}
{% endfor %}
```

Итак, что же делает класс DataMixin? Смотрите, в нем объявлен вспомогательный метод get\_user\_context() для формирования контекста шаблона по умолчанию. Также, при необходимости, мы можем передавать ему именованные аргументы, которые также будут помещаться в контекст. Благодаря этому методу, нам не придется в классах представлений каждый раз прописывать ссылки на главное меню и категории.

Итак, класс миксин объявлен, осталось прописать его в качестве базового у классов представлений. Для этого в файле views.py мы импортируем модуль utils.py:

```
from .utils import *
```

И унаследуем класс WomenHome также и от DataMixin:

```
class WomenHome(DataMixin, ListView):
...
```

Осталось изменить метод get\_context\_data(), следующим образом:

```
def get_context_data(self, *, object_list=None, **kwargs):
    context = super().get_context_data(**kwargs)
    c_def = self.get_user_context(title="Главная страница")
    context = dict(list(context.items()) + list(c_def.items()))
    return context
```

Смотрите, мы здесь вызываем метод get\_user\_context() класса DataMixin, указав, дополнительно параметр title. Получаем сформированный словарь c\_def со всеми стандартными ключами и объединяем его со словарем context. В конце, возвращаем объединенные данные. Все, дублирование в методе get\_context\_data() устранено.

По аналогии, меняю и все остальные классы представлений:

```
class AddPage(DataMixin, CreateView):
...

    def get_context_data(self, *, object_list=None, **kwargs):
        context = super().get_context_data(**kwargs)
        c_def = self.get_user_context(title="Добавление статьи")
        context = dict(list(context.items()) + list(c_def.items()))
        return context

class ShowPost(DataMixin, DetailView):
...

    def get_context_data(self, *, object_list=None, **kwargs):
        context = super().get_context_data(**kwargs)
        c_def = self.get_user_context(title=context['post'])
        return dict(list(context.items()) + list(c_def.items()))

class WomenCategory(DataMixin, ListView):
...

    def get_context_data(self, *, object_list=None, **kwargs):
        context = super().get_context_data(**kwargs)
        c_def = self.get_user_context(title='Категория - ' + str(context['posts'][0].cat),
                                     cat_selected=context['posts'][0].cat_id)

        return dict(list(context.items()) + list(c_def.items()))
```

Все, переходим на сайт и видим, что страницы отображаются также как и ранее, но теперь все работает совместно с классом DataMixin. Вот пример того, как миксины в Django позволяют устранить дублирование кода в классах представлений.

Конечно, в класс DataMixin можно прописывать не только методы, но и общие атрибуты, если они есть, то есть, выносить любую общую информацию.

## Миксины фреймворка Django

В Django есть стандартные миксины, которые можно использовать совместно с классами представлений. Использовать их достаточно просто, я покажу пример одного такого миксина:

LoginRequiredMixin

который позволяет ограничить доступ к странице для неавторизованных пользователей. Подробную информацию об этом классе можно посмотреть по этой ссылке:

<https://djbook.ru/rel3.0/topics/auth/default.html>

Давайте вначале выполним его импорт в файле views.py:

```
from django.contrib.auth.mixins import LoginRequiredMixin
```

А, затем, добавим в класс AddPage:

```
class AddPage(LoginRequiredMixin, DataMixin, CreateView):
...
```

Причем, прописывать желательно самым первым, т.к. он имеет наибольшую важность. Хотя, в нашем случае первые два миксина можно записывать в любом порядке, они никак между собой не пересекаются.

По идее все. Если теперь попробовать выйти из админки (то есть, стать не зарегистрированным пользователем) и перейти на добавление поста, то увидим страницу с кодом 404. Давайте улучшим это поведение, сделаем его более дружелюбным. Для этого, в классе AddPage (после добавления миксина LoginRequiredMixin) можно прописать специальный атрибут:

```
login_url = '/admin/'
```

который указывает адрес перенаправления для незарегистрированного пользователя. В данном случае, мы его отправляем в админ-панель. Переходим на главную страницу, затем, на добавление статьи и автоматом перенаправляемся на форму авторизации.

Конечно, прописывать конкретный URL-адрес – это не лучшая практика, поэтому, давайте, воспользуемся функцией reverse\_lazy для формирования маршрута по его имени:

```
login_url = reverse_lazy('home')
```

Также, вместо перенаправлений, можно генерировать страницу с кодом 403 – доступ запрещен. Для этого достаточно прописать атрибут:

```
raise_exception = True
```

Если похожий функционал нужно реализовать для функций представлений, а не классов, то здесь уже используется декоратор login\_required, например, так:

```
@login_required
def about(request):
    return render(request, 'women/about.html', {'menu': menu, 'title': '0 сайте'})
```

Теперь эта страница доступна только авторизованным пользователям. Я уберу его, т.к. он здесь не к месту. Это просто демонстрация того, как ограничить доступ, работая с функциями представлениями.

Наконец, давайте сделаем отображение пункта «Добавить статью» только для авторизованных пользователей. Для этого я в классе DataMixin буду удалять этот пункт, если пользователь не авторизован:

```
class DataMixin:
    def get_user_context(self, **kwargs):
        context = kwargs
        cats = Category.objects.annotate(Count('women'))

        user_menu = menu.copy()
        if not self.request.user.is_authenticated:
            user_menu.pop(1)

        context['menu'] = user_menu
        context['cats'] = cats
        if 'cat_selected' not in context:
            context['cat_selected'] = 0

        return context
```

Здесь используется объект request, у которого имеется объект user, а у того, в свою очередь, специальный булевый атрибут is\_authenticated, указывающий на авторизацию текущего пользователя (если True – авторизован, False – в противном случае). Подробнее об этом также можно посмотреть на странице:

<https://djbook.ru/rel3.0/topics/auth/default.html>

Ну и в заключение этого занятия, давайте сделаем вывод только тех рубрик, которые содержат статьи. Для этого мы будем выбирать рубрики с использованием агрегирующей функции:

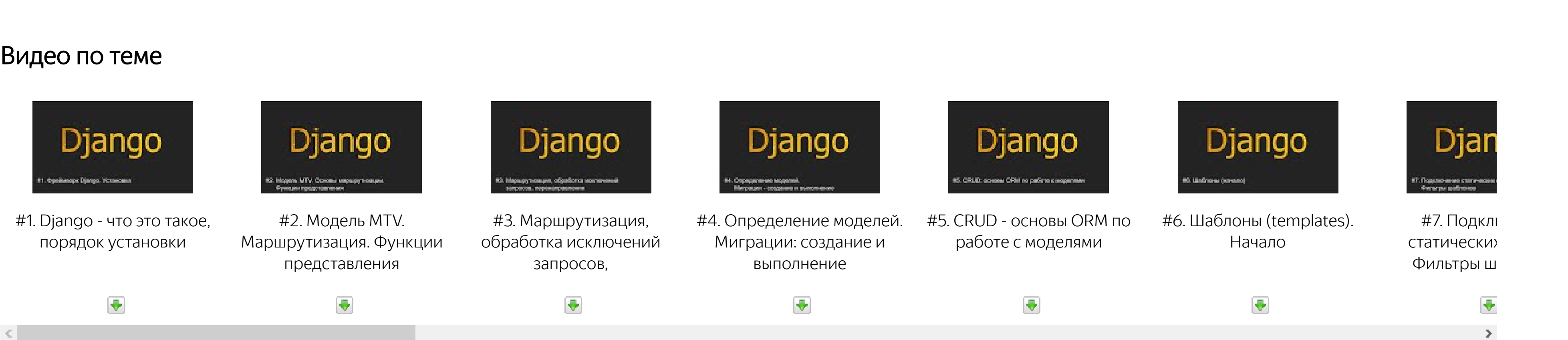
```
cats = Category.objects.annotate(Count('women'))
```

Как работает эта строка мы говорили на предыдущем занятии. Далее, в шаблоне base.html пропишем проверку при выводе рубрик:

```
{% for c in cats %}
{% if c.women__count > 0 %}
    {% if c.pk == cat_selected %}
        <li class="selected">{{c.name}}</li>
    {% else %}
        <li><a href="{{ c.get_absolute_url }}">{{c.name}}</a></li>
    {% endif %}
{% endif %}
{% endfor %}
```

Все, теперь у нас появляются только те рубрики, у которых есть статьи, что более логично.

## Видео по теме



← Предыдущая

Следующая →