Материалы занятия

Курс: Web-разработка

Дисциплина: Создание web-приложений с использованием фреймворка Django

Тема занятия № 42: Модуль 24. Сигналы

1. ОБРАБОТКА СИГНАЛОВ

Сигнал сообщает о выполнении Django какого-либо действия: создании новой записи в модели, удалении записи, входе пользователя на сайт, выходе с него и пр. К сигналу можно привязать *обработчик* — функцию или метод, который будет вызываться при возникновении сигнала.

Сигналы предоставляют возможность вклиниться в процесс работы самого фреймворка или отдельных приложений — неважно, стандартных или написанных самим разработчиком сайта — и произвести какие-либо дополнительные действия. Скажем, приложение `django-cleanup`, рассмотренное нами ранее и удаляющее ненужные файлы, чтобы отследить момент правки или удаления записи, обрабатывает сигналы `post_init`, `pre_save`, `post_save` и `post_delete`.

Обработка сигналов

Все сигналы в Django представляются экземплярами класса `Signal` или его подклассов. Этот класс поддерживает два метода, предназначенные для привязки к сигналу обработчика или отмены его привязки.

Для привязки обработчика к сигналу применяется метод `connect()` класса `Signal`:

```
connect(<обработчик>[, sender=None][, weak=True][,
dispatch_uid=None])
```

`Обработчик` сигнала, как было сказано ранее, должен представлять собой функцию или метод. Формат написания этой функции (метода) мы рассмотрим позднее.

В необязательном параметре `sender` можно указать класс объекта, из которого отправляется текущий сигнал (*отправителя*). После чего *`обработчик`* будет обрабатывать сигналы исключительно от отправителей, относящихся к этому классу.

Если необязательному параметру `weak` присвоено значение `True` (а это его значение по умолчанию), то для связи отправителя и обработчика будет использована слабая ссылка Python, если присвоено значение `False` — обычная. Давать этому параметру значение `False` следует в случае, если после удаления всех отправителей нужда в обработчике пропадает — тогда он будет автоматически выгружен из памяти.

Необязательный параметр `dispatch_uid` указывается, если к одному и тому же сигналу несколько раз привязывается один и тот же обработчик, и возникает необходимость как-то отличить одну такую привязку от другой. В этом случае в разных вызовах метода `connect()` нужно указать разные значения этого параметра, которые должны представлять собой строки.

Если к одному и тому же сигналу привязано несколько обработчиков, то они будут выполняться один за другим в той последовательности, в которой были привязаны к сигналу.

Рассмотрим несколько примеров привязки обработчика к сигналу `post_save`, возникающему после сохранения записи модели:

```
from django.db.models.signals import post_save

# Простая привязка обработчика post_save_dispatcher() к сигналу
post_save.connect(post_save_dispatcher)
# Простая привязка обработчика к сигналу, возникающему в модели Bb
post_save.connect(post_save_dispatcher, sender=Bb)
# Двукратная привязка обработчиков к сигналу с указанием разных значений
# параметра dispatch_uid
post_save.connect(post_save_dispatcher,
                  dispatch_uid='post_save_dispatcher_1')
post_save.connect(post_save_dispatcher,
                  dispatch_uid='post_save_dispatcher_2')
```

Обработчик — функция или метод — должен принимать один позиционный параметр, с которым передаётся класс объекта-отправителя сигнала. Помимо этого, обработчик может принимать произвольное число именованных параметров, набор которых у каждого сигнала различается (стандартные сигналы Django и передаваемые ими параметры мы рассмотрим позже). Вот своего рода шаблоны для написания обработчиков разных типов:

```
def post_save_dispatcher(sender, **kwargs):
    # Тело функции-обработчика
    # Получаем класс объекта-отправителя сигнала
    snd = sender
    # Получаем значение переданного обработчику именованного параметра
    # instance
    instance = kwargs['instance']
    ...

class SomeClass:
    def post_save_dispatcher(self, sender, **kwargs):
        # Тело метода-обработчика
        ...
```

Вместо метода `connect()` объекта сигнала можно использовать декоратор `receiver(<сигнал>)`, объявленный в модуле `django.dispatch`:

```
from django.dispatch import receiver
@receiver(post_save)
def post_save_dispatcher(sender, **kwargs):
    ...
```

Код, выполняющий привязку к сигналам обработчиков, которые должны действовать всё время, пока работает сайт, обычно записывается в модуле **apps.py** или **models.py**.

Отменить привязку обработчика к сигналу позволяет метод `disconnect()` класса `Signal`:

```
disconnect([receiver=None][,][sender=None][,][dispatch_uid=None])
```

В параметре `receiver` указывается обработчик, ранее привязанный к сигналу. Если этот обработчик был привязан к сигналам, отправляемым конкретным классом, то последний следует указать в параметре `sender`. Если в вызове метода `connect()`, выполнившем привязку обработчика, был задан параметр `dispatch_uid` с каким-либо значением, то удалить привязку можно, записав в вызове метода `disconnect()` только параметр `dispatch_uid` и указав в нём то же значение. Примеры:

```
post_save.disconnect(receiver=post_save_dispatcher)
post_save.disconnect(receiver=post_save_dispatcher, sender=Bb)
post_save.disconnect(dispatch_uid='post_save_dispatcher_2')
```

2. ВСТРОЕННЫЕ СИГНАЛЫ DJANGO

Сигналы, отправляемые подсистемой доступа к базам данных и объявленные в модуле `django.db.models.signals`:

- `pre_init` — отправляется в самом начале создания новой записи модели, перед выполнением конструктора её класса. Обработчику передаются следующие параметры:
  - `sender` — класс модели, запись которой создаётся;
  - `args` — список позиционных аргументов, переданных конструктору модели;
  - `kwargs` — словарь именованных аргументов, переданных конструктору модели.

  Например, при создании нового объявления выполнением выражения:

```
Bb.objects.create(title='Дом',
                  content='Трёхэтажный, кирпич',
                  price=50000000)
```

  Обработчик с параметром `sender` получит ссылку на класс модели `Bb`, с параметром `args` — "пустой" список, а с параметром `kwargs` — словарь `{'title' : 'Дом', 'content': 'Трёхэтажный, кирпич', 'price': 50000000}`;

- `post_init` — отправляется в конце создания новой записи модели, после выполнения конструктора её класса. Обработчику передаются следующие параметры:
  - `sender` — класс модели, запись которой была создана;
  - `instance` — объект созданной записи;
- `pre_save` — отправляется перед сохранением записи модели, до вызова её метода `save()`. Обработчику передаются параметры:
  - `sender` — класс модели, запись которой сохраняется;
  - `instance` — объект сохраняемой записи;
  - `raw` — `True`, если запись будет сохранена как есть, без обращения к другим записям за дополнительными данными и без исправления других записей, и `False` — в противном случае;
  - `update_fields` — множество имён полей, заданных в параметре `update_fields` метода `save()`, или `None`, если этот параметр не был указан;
- `post_save` — отправляется после сохранения записи модели, после вызова её метода `save()`. Обработчику передаются такие параметры:
  - `sender` — класс модели, запись которой была сохранена;
  - `instance` — объект сохранённой записи;
  - `created` — `True`, если это вновь созданная запись, и `False` — в противном случае;
  - `raw` — `True`, если запись была сохранена как есть, без обращения к другим записям за дополнительными данными и без исправления других записей, и `False` — в противном случае;
  - `update_fields` — множество имен полей, заданных в параметре `update_fields` метода `save()`, или `None`, если этот параметр не был указан.

Вероятно, это один из наиболее часто обрабатываемых сигналов. Вот пример его обработки с целью вывести в консоли Django сообщение о добавлении объявления:

```
from django.db.models.signals import post_save

def post_save_dispatcher(sender, **kwargs):
    if kwargs['created']:
        print('Объявление в рубрике "%s" создано' %
              kwargs['instance'].rubric.name)
```

```
post_save.connect(post_save_dispatcher, sender=Bb)
```

- `pre_delete` — отправляется перед удалением записи, до вызова её метода `delete()`. Параметры, передаваемые обработчику:
  - `sender` — класс модели, запись которой удаляется;
  - `instance` — объект удаляемой записи;
- `post_delete` — отправляется после удаления записи, после вызова её метода `delete()`. Обработчик получит следующие параметры:
  - `sender` — класс модели, запись которой была удалена;
  - `instance` — объект удалённой записи. Отметим, что эта запись более не существует в базе данных;
- `m2m_changed` — отправляется связующей моделью при изменении состава записей моделей, связанных посредством связи "многие-со-многими".

  Связующая модель может быть явно задана в параметре `through` конструктора класса `ManyToManyField` или же создана фреймворком неявно. В любом случае связующую модель можно получить из атрибута `through` объекта поля типа `ManyToManyField`. Пример привязки обработчика к сигналу, отправляемому связующей моделью, которая была неявно создана при установлении связи "многие-со-многими" между моделями `Machine` и `Spare`:

```
m2m_changed.connect(m2m_dispatcher, sender=Machine.spares.through)
```

  Обработчик этого сигнала принимает параметры:

  - `sender` — класс связующей модели;
  - `instance` — объект записи, в котором выполняются манипуляции по изменению состава связанных записей (т. е. у которого вызываются методы `add()`, `create()`, `set()` и др.);
  - `action` — строковое обозначение выполняемого действия:
    - `"pre_add"` — начало добавления новой записи в состав связанных;
    - `"post_add"` — окончание добавления новой связанной записи в состав связываемых;
    - `"pre_remove"` — начало удаления записи из состава связанных;
    - `"post_remove"` — окончание удаления записи из состава связанных;

- "pre_clear" — начало удаления всех записей из состава связанных;
- "post_clear" — окончание удаления всех записей из состава связанных;

- reverse — False, если изменение состава связанных записей выполняется в записи ведущей модели, и True, если в записи ведомой модели;
- model — класс модели, к которой принадлежит запись, добавляемая в состав связанных или удаляемая оттуда;
- pk_set — множество ключей записей, добавляемых в состав связанных или удаляемых оттуда. Для действий "pre_clear" и "post_clear" всегда None.

Например, при выполнении операций:

```
m = Machine.objects.create(name='Самосвал')
s = Spare.objects.create(name='Болт')
m. spares.add(s)
```

обработчик сигнала m2m_changed с параметром sender получит ссылку на класс промежуточной модели (у нас — созданной самим фреймворком), с параметром instance — запись m (т. к. действия по изменению состава связанных записей выполняются в ней), с параметром action — строку "pre_add", с параметром reverse — False (действия по изменению состава связанных записей выполняются в записи ведущей модели), с параметром model — модель Spare, а с параметром pk_set — множество из единственного элемента — ключа записи s. Впоследствии тот же самый сигнал будет отправлен ещё раз, и его обработчик получит с параметрами те же данные, за исключением параметра action, который будет иметь значение "post_add". А после выполнения действия:

```
s.machine_set.remove(m)
```

обработчик сигнала m2m_changed с параметром sender получит ссылку на класс промежуточной модели, с параметром instance — запись s, с параметром action — строку "pre_remove", с параметром reverse — True (поскольку теперь действия по изменению состава связанных записей выполняются в записи ведомой модели), с параметром model — модель Machine, а с параметром pk_set — множество из единственного элемента — ключа записи m. Далее тот же самый сигнал будет отправлен ещё раз, и его обработчик получит с параметрами те же данные, за исключением параметра action, который будет иметь значение "post_remove".

Сигналы, отправляемые подсистемой обработки запросов и объявленные в модуле django.core.signals:

- request_started — отправляется в самом начале обработки запроса. Обработчик получит параметры:

- sender — класс django.core.handlers.wsgi.WsgiHandler, обрабатывающий все полученные запросы;
- environ — словарь, содержащий переменные окружения;

o request_finished — отправляется после пересылки ответа клиенту. Обработчик с параметром sender получит класс django.core.handlers.wsgi.WsgiHandler, обрабатывающий все полученные запросы;

o got_request_exception — отправляется при возбуждении исключения в процессе обработки запроса. Вот параметры, передаваемые обработчику:

- sender — None;
- request — сам запрос В виде экземпляра класса HttpRequest.

Сигналы, отправляемые подсистемой разграничения доступа и объявленные в модуле django.contrib.auth.signals:

o user_logged_in — отправляется после удачного входа на сайт. Параметры, передаваемые обработчику:

- sender — класс модели пользователя (User, если не была задана другая модель);
- request — текущий запрос, представленный экземпляром класса HttpRequest;
- user — запись пользователя, который вошёл на сайт;

o user_logged_out — отправляется после удачного выхода с сайта. Вот параметры, которые получит обработчик:

- sender — класс модели пользователя или None, если пользователь ранее не выполнил вход на сайт;
- request — текущий запрос в виде экземпляра класса HttpRequest;
- user — запись пользователя, который вышел с сайта, или None, если пользователь ранее не выполнил вход на сайт;

o user_login_failed — отправляется, если посетитель не смог войти на сайт. Параметры, передаваемые обработчику:

- sender — строка с именем модуля, выполнявшего аутентификацию;
- credentials — словарь со сведениями, занесёнными посетителем в форму входа и переданными впоследствии функции authenticate(). Вместо пароля будет подставлена последовательность звёздочек;

- `request` — текущий запрос в виде экземпляра класса `HttpRequest`, если таковой был передан функции `authenticate()`, в противном случае — `None`.

  На заметку!
  Некоторые специфические сигналы, используемые внутренними механизмами Django или подсистемами можно найти на странице https://docs.djangoproject.com/en/4.2/ref/signals/.

3. ОБЪЯВЛЕНИЕ СВОИХ СИГНАЛОВ

Сначала нужно объявить сигнал, создав экземпляр класса `Signal` из модуля `django.dispatch`. Конструктор этого класса вызывается согласно формату:

`Signal(providing_args=<`*`список имён параметров, передаваемых обработчику`*`>)`

Имена параметров в передаваемом *списке* должны быть представлены в виде строк.

Пример объявления сигнала `add_bb`, который будет передавать обработчику параметры `instance` и `rubric`:

```
from django.dispatch import Signal
add_bb = Signal(providing_args=['instance', 'rubric'])
```

Для отправки объявленного сигнала применяются два следующих метода класса `Signal`:

- `send(<`*`отправитель`*`>[, <`*`именованные параметры, указанные при объявлении сигнала`*`>])` — выполняет отправку текущего сигнала от имени указанного *отправителя*, возможно, с *именованными параметрами*, которые были указаны при объявлении сигнала и будут отправлены его обработчику.

  В качестве результата метод возвращает список, каждый из элементов которого представляет один из привязанных к текущему сигналу обработчиков. Каждый элемент этого списка представляет собой кортеж из двух элементов: ссылки на обработчик и возвращённый им результат. Если обработчик не возвращает результата, то вторым элементом станет значение `None`. Пример:

  ```
  add_bb.send(Bb, instance=bb, rubric=bb.rubric)
  ```

  Если к сигналу привязано несколько обработчиков и в одном из них было возбуждено исключение, последующие обработчики выполнены не будут;

- `send_robust(<`*`отправитель`*`>[, <`*`именованные параметры, указанные при объявлении сигнала`*`>])` — то же самое, что `send()`, но обрабатывает все исключения, что могут быть возбуждены в обработчиках. Объекты исключений будут присутствовать в результате, возвращенном методом, во-вторых элементах соответствующих вложенных кортежей.

Поскольку исключения обрабатываются внутри метода, то, если к сигналу привязано несколько обработчиков и в одном из них было возбуждено исключение, последующие обработчики всё же будут выполнены.

Объявленный нами сигнал может быть обработан точно так же, как и любой из встроенных в Django:

```
def add_bb_dispatcher(sender, **kwargs):
    print ('Объявление в рубрике "%s" с ценой %.2f создано' %
           (kwargs['rubric'].name, kwargs['instance'].price))

add_bb.connect(add_bb_dispatcher)
```