Djangoまとめ

環境構築，テンプレートについては説明しない.
HTTP通信，webシステム，Pythonの基本的知識を前提とする.
コードは主にDjangoドキュメントから引用した.
参考にしたのは，『Python Django3　超入門』(秀和システム出版　掌田津耶野著)とDjangoドキュメント

目次

1. Djangoの概要
2. urlの設定
3. modelの設定
4. form(selializer)の設定
5. viewの設定

1.Djangoの概要

Djangoは，データアクセス関係の処理を担当するModel，画面表示関係を担当するView，全体の制御を担当するControllerという部分により構成されるMVCアーキテクチャーに基づいて設計されている.
実際のDjangoの中ではそれぞれのMVC関係のプログラムが一つになったアプリケーションを状況に応じて追加していくことになる.そのアプリケーションの中のmodel.pyファイルを編集することでModelの機能を，view.pyファイルを編集することでViewの機能とアクセス時の処理というControllerの機能の一部を担うことになる.画面の表示というControllerの役割はTemplateフォルダを作ることで実現される.単にバックエンドとしての役割でDjangoを使う際はTemplateフォルダは必要としない.

2.urlの設定

url.pyファイルを編集することで行う.ユーザーがどのページをリクエストした時にどんな処理をするかを設定する.
./urls.py

```
from django.urls import path,include,re_path

from . import views

urlpatterns = [
    path('', views.index, name='index'),
    path('app/', include('app.urls')),
    path('myview/',views.MyView.as_view())
    path('articles/<int:year>/<int:month>/', views.archive),
    re_path(r'^articles/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/$',
views.archive),
]
```

上のようにpath(route, view, kwargs=None, name=None)と設定できる.それぞれのrouteは，Baseurlからのrouteを記述する.分かりやすくするためにプロジェクトのurls.pyに全ての設定を記述するのでなく，それぞれのアプリケーションごとにurls.pyを作成して設定し，include()を使いプロジェクトのurls.pyでまとめるという記述の仕方をす

る.re_path()を用いることで正規表現を用いることができる．またnameをつけることもできる．詳しくは以下を参照
django.urls functions for use in URLconfs

viewのところにはリクエストがあった際の処理を行う関数やクラスベースビューのas_view()で呼んだ関数を記述する．.as_view()はdjango.views.generic.base.Viewクラスのメソッドで，行っている処理はクラスを初期化したのちrequestを受け取ったのちdispatch()を呼ぶことでHTTPメソッドに応じた処理を行う．詳しくは以下を参照．
Base views

また，urlのpathとしてviewに記述した関数に引数を渡すことができる．その際，< int:- >などの形で型を指定できる．結果として，viewの関数にはHttpRequestのインスタンスとurlのpathとして記述した引数，それに加えpathやre_pathのオプションとして渡したkwargsが渡されることになる．詳しくは以下を参照．
URL dispatcher

3.modelの設定，説明

modelについて説明する．modelの設計の際にしかmodels.pyファイルは編集しないことに注意．

modelの設計

model.pyファイルを編集することで行う．このmodel.pyを介してデータベースを操作する．それぞれのモデルのクラスを作ることはデータベースのテーブルを作ることに対応し，レコードはそのクラスのインスタンスに対応する．

```
from django.db import models

class Student(models.Model):
    YEAR_IN_SCHOOL_CHOICES = [
    ('FR', 'Freshman'),
    ('SO', 'Sophomore'),
    ('JR', 'Junior'),
    ('SR', 'Senior'),
    ('GR', 'Graduate'),
]
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
    year_in_school = models.CharField(
        max_length=2,
        choices=YEAR_IN_SCHOOL_CHOICES,
        default=FRESHMAN,
    )

    def __str__(self):
        return '%s %s' % (self.first_name, self.last_name)
```

全てのmodelはdjango.db.models.Modelのサブクラスとして定義する．メソッド__str__(self)を定義することでインスタンスにstr()が呼ばれた時の処理を記述でき，有用である．

modelの設計の際必要となるのがfieldであり，これはFieldクラスのインスタンスである．Fieldのインスタンスとすることでバリデーションを行うことやそのモデルに基づいたフォームがレンダリングされる際の振る舞いを記述することができる．例えば，max_lengthを定義することで文字列の長さの最大値のバリデーションを行うことができ，また，CharFieldにchoicesを指定することレンダリングされた時にはselect boxとなる．

また，バリデーションにはバリデータというクラスのインスタンスや自作のバリデーション関数を用いることができ，model.CharField(varidators=[バリデータのインスタンス or 自作関数])のリストを指定して複数指定することもできる．また，バリデーションはModelのメソッドである.save()が呼ばれた際に実行される．詳しくは以下を参照．
Model field reference

modelの管理

django.db.models.Modelはデータベースでのmodelの管理として主に次の二つのメソッドをもつ．
.save()

```
s1 = Student(first_name="Fred", last_name="Flintstone",
year_in_school="FR")
s1.save()
s1.id     # Returns the ID of your new object.
```

このように用いる．この際，primary keyが自動的に追加されidという名前で保存される．
.delete()

```
s1.delete() # Returns the number of objects deleted and a dictionary with
the number of deletions per object type.
```

deleteした際にそのオブジェクトのprimary keyとオブジェクトの値を辞書として返す．これによりデータベースから削除される．

modelを通じたデータベースへのアクセス

データベースからデータの取得などのアクセスは，django.db.models.ModelのもつManagerクラスを通じてQuerySetを構成することで行う．QuerySetクラスはそれぞれのレコードの集まりのようなものである．
Managerクラスはそれぞれのmodelに対して少なくとも一つはあり初期設定ではobjectsという名前になっている．

```
all_students = Student.objects.all()
all_freshmans = Student.objects.filter(year_in_school="FR")
all_john = Student.objects.get(last_name="John")
fresh_notjohn =
Student.objects.filter(year_in_school="FR").exclude(last_name="John")
all_students_dict = Student.objects.all().values()
```

上のようにall()やfilter()をManagerクラスから呼び出すことでQuerySetを取得することができる．filterやexcludeを使って条件にあうデータを絞り込める．特に上の例でもあるようにメソッドを続けて記述するメソッドチェーンは有用．.values()を用いることで辞書の形式で取り出すことができる．getでもfilterと同じように取り出せるがgetでは条件に合うものがない時にエラーを出力する点で異なる．詳しくは以下を参照．
Making queries
QuerySet API reference
Managers

テーブル同士の関係

modelを設計する際にはそれぞれのテーブル同士の関係が重要になる．例えばsnsのようなアプリを作る際には，メッセージの情報を保管するテーブルとユーザーの情報を保管するテーブルの間のどのメッセージがどのユーザーのものかといった関係が重要になる．
modelのfieldにForeignKey fieldを設定することで関係性を記述する．次の3つの関係性がある．どの関係性においても一方にしか関係についての記述をしないがそれぞれのデータに双方向的にアクセスできることに注意．

Many-to-one relationships
数の多い方(従モデル)にForeignKeyを指定する．例えば，

```
class Message(models.Model):
    owner = models.ForeignKey(User, on_delete=models.CASCADE)

john = User.objects.create(name='John')
msg = Message.objects.create(owner=john)
john.delete()
# (2, {'User': 1, 'Message': 1})
```

主モデルはUserである．Userの発信するMessageの方にForeignKeyを指定している．オプションでon_delete=models.CASCADEを指定できる．この指定により主モデル側が削除された時にそれに関係を持つ従モデルも削除される．CASCADEではModel.delete()は連鎖的に発生しないことに注意．QuerySet.delete()などは連鎖的に発生する．詳しくは以下を参照．
ForeignKey
次の例が参考になる．
Many-to-one relationships

Many-to-many relationships
どちらかを従モデルにしてManyToManyFieldを設定する．

```
from django.db import models

class Publication(models.Model):
    title = models.CharField(max_length=30)

class Article(models.Model):
    headline = models.CharField(max_length=100)
    publications = models.ManyToManyField(Publication)
```

次の例を見るのが参考になる．
Many-to-many relationships

One-to-one relationships
どちらかを従モデルにしてOneToOneFieldを設定する．

```
from django.db import models
```

```
class Place(models.Model):
    name = models.CharField(max_length=50)
    address = models.CharField(max_length=80)

class Restaurant(models.Model):
    place = models.OneToOneField(
        Place,
        on_delete=models.CASCADE,
    )
    serves_hot_dogs = models.BooleanField(default=False)
    serves_pizza = models.BooleanField(default=False)
```

次の例を見るのが参考になる.
One-to-one relationships

4.form(selializer)の設定

forms.pyファイルを編集することで設定する. Formクラスはdjangoの処理とフロントエンドでのHTMLにおけるformタグによる処理の連携を簡単にする. DRFではsilializerが似たような役割を果たす. 例えば, レンダリングのためにデータを再構成したりクライアントから送信されたデータやフォームを受け取って処理することができる.実際, フォームのデータを受け取る, またフォームにデータを入れて送信することは重要な処理となる.

```
from django import forms
class OptionalPersonForm(forms.Form):
    first_name = forms.CharField()
    last_name = forms.CharField()
    nick_name = forms.CharField(required=False)
data = {'first_name': 'John', 'last_name': 'Lennon'}
f = OptionalPersonForm(data)
f.is_valid()
```

上のようにForm fieldを設定することで使える. バリデーションなども行える. 次がFormの役割について参考になる.
Working with forms
The Forms API
実際には, modelに基づいたフォームを用いることが多いため次のModelFormが有用.
ModelForm

```
from django.forms import ModelForm
from .models import Student

class StudentForm(ModelForm):
    class Meta:
        model = Student
        fields = ['first_name', 'last_name', 'year_in_school']

s1 = StudentForm(kwargs)
new_student = s1.save()
```

```
# Save a new Student object from the form's data.
s = Student.objects.get(pk=1)
s2 = StudentForm(kwargs, instance=s)
s2.save()
# Populate the form with kwargs
```

上のようにmodelを指定し，fieldsにリストを与えることでmodelのfieldに基づきformのfieldを構成する．また，ModelFormの.save()メソッドはフォームのデータからオブジェクトを作成しデータベースに保存することができる．すでにあるインスタンスを指定した場合，そのインスタンスをupdateすることになる．詳しくは以下を参照．
ModelForm

5.Viewの設定

view.pyを編集することで設定する．viewの役割はrequestを受け取って様々な処理を行ったのちに，responseを返すことである．urlの設定で記述したように，viewの関数はrequestと幾らかの引数を受け取る．

requestの処理

viewの関数の受け取るrequestはHttpRequestクラスのインスタンスである．特に次のrequest.method, request.POST,request.GETを取り扱う．

```
from .forms import StudentForm
from .models import Student
from django.http import HttpResponse

def index(request):
    if (request.method == 'POST'):
        obj = Student()
        student = StudentForm(request.POST, instance=obj)
        student.save()
        return HttpResponse('You saved the data!')
    elif (request.method == 'GET'):
        n = request.GET['number']
        sn = Student.object.get(pk=n)
        name = sn.first_name + sn.last_name
        return HttpResponse(name)

def getparamater(request,year,month):
    result = str(year) + '/' + str(month)
    return HttpResponse(result)
```

まず，request.methodがrequestの種類を持っている．それに応じて処理を書いていくことになる．
request.POST，request.GETはQueryDictのオブジェクトである．QueryDictは辞書のサブクラスであるため，辞書と同じように扱うことができる．上の例ではModelFormのkwargsとしてrequest.POSTを渡したり，keyを指定することでデータを取り出したりしている．
また，urlの設定でも記述したように，urlのpathとして与えられた引数を，requestに続けて受け取ることができる．詳しくは以下を参照．
Request and response objects

クラスを用いたview

クラスを用いることで様々なHTTPメソッドをひとまとめに処理することができる．これを用いることで複雑な処理が記述しやすくなりviewのコードの可読性も向上する．

```
from django.http import HttpResponse
from django.views import View

class MyView(View):

    def get(self, request, *args, **kwargs):
        return HttpResponse('Hello, World!')

    def post(self, request, *args, **kwargs):
        return HttpResponse('You post!')
```

上で継承しているのは django.views.generic.base.Viewクラスで他の全てのクラスベースビュー，例えばTemplateViewやDetailViewなど，はこのクラスを継承している．
このクラスはsetup()というメソッドをもち，インスタンス変数としてrequest，args，kwargsを設定する．
またdispatch()というメソッドをもち，requestメソッドに応じた処理を呼び出す．これが実質的にviewの機能を担う．実際次のようになっている．
django/views/generic/base.py

```
...

http_method_names = ['get', 'post', 'put', 'patch', 'delete', 'head',
'options', 'trace']

...

 def dispatch(self, request, *args, **kwargs):
        # Try to dispatch to the right method; if a method doesn't exist,
        # defer to the error handler. Also defer to the error handler if
the
        # request method isn't on the approved list.
        if request.method.lower() in self.http_method_names:
            handler = getattr(self, request.method.lower(),
self.http_method_not_allowed)
        else:
            handler = self.http_method_not_allowed
        return handler(request, *args, **kwargs)
```

また，as_view()というメソッドをもち，urlの設定で説明したようにクラスを初期化してsetup()を呼び出したのちdispatch()を呼び出す．
そして，実際にはListViewのようなモデルに基づいたクラスが有用となる．

```
from django.views.generic import ListView
from .models import Student

class StudentDetail(ListView):
    model = Student
```

getした際のデータなどは，初期設定では(モデル名)_list.htmlに送られることに注意．ListViewではmodelを設定するだけでそのmodelのレコード一覧を取得することができる．これは，MultipleObjectMixinを継承しているため，getの際にそのmodelからQuerySetを取得するためである．実際，次のようになっている．(一部分のみ抜粋)

django/views/generic/list.py

```
class MultipleObjectMixin(ContextMixin):
    queryset = None
    model = None
...
    def get_queryset(self):
        """
        Return the list of items for this view.
        The return value must be an iterable and may be an instance of
        `QuerySet` in which case `QuerySet` specific behavior will be
enabled.
        """
        if self.queryset is not None:
            queryset = self.queryset
            if isinstance(queryset, QuerySet):
                queryset = queryset.all()
        elif self.model is not None:
            queryset = self.model._default_manager.all()
        else:
            raise ImproperlyConfigured(
                "%(cls)s is missing a QuerySet. Define "
                "%(cls)s.model, %(cls)s.queryset, or override "
                "%(cls)s.get_queryset()." % {
                    'cls': self.__class__.__name__
                }
            )
        ordering = self.get_ordering()
        if ordering:
            if isinstance(ordering, str):
                ordering = (ordering,)
            queryset = queryset.order_by(*ordering)

        return queryset
...

class BaseListView(MultipleObjectMixin, View):
    """A base view for displaying a list of objects."""
    def get(self, request, *args, **kwargs):
        self.object_list = self.get_queryset()
        allow_empty = self.get_allow_empty()
```

```
...
```

クラスベースビューについては詳しくは次を参照.

Class-based views

クラスの継承関係については次をが参考になる.

Djangoのクラスベースビューを完全に理解する