220906 Django 4

1. Django Form

- 우리는 지금까지 HTML form, input 태그를 통해서 사용자로부터 데이터를 받음
- 현재 우리 Django 서버는 들어오는 요청을 모두 수용하고 있는데, 이러한 요청 중에서는 비정상적인 혹은 악의적인 요청이 있다는 것을 생각해야 함
- 이처럼 사용자가 입력한 데이터가 우리가 원하는 데이터 형식이 맞는지에 대한 **유효성 검증**이 반드시 필요
    - 이러한 유효성 검증은 많은 부가적인 것들을 고려해서 구현해야 하는데, 이는 개발 생산성을 늦출뿐더러 쉽지 않은 작업임
- Django Form은 이 과정에서 과중한 작업과 반복 코드를 줄여줌으로써 훨씬 쉽게 유효성 검증을 진행할 수 있도록 만들어 줌

1.1 Form에 대한 Django의 역할

- Form은 Django의 유효성 검사 도구 중 하나로 외부의 악의적 공격 및 데이터 손상에 대한 중요한 방어 수단
- Django는 Form과 관련된 유효성 검사를 `단순화하고 자동화` 할 수 있는 기능을 제공, 개발자가 직접 작성하는 코드보다 더 안전하고 빠르게 수행하는 코드를 작성 가능
    - 개발자가 필요한 핵심 부분만 집중할 수 있도록 돕는 프레임워크의 특성

1.2 Django는 Form에 관련된 작업의 세 부분을 처리

- 렌더링을 위한 데이터 준비 및 재구성
- 데이터에 대한 HTML forms 생성
- 클라이언트로부터 받은 데이터 수신 및 처리

2. Django Form Class

2.1 Form Class 선언

- Form Class를 선언하는 것은 Model Class를 선언하는 것과 비슷

  비슷한 이름의 필드 타입을 많이 가지고 있음 (이름이 같을 뿐 같은 필드는 아님)
- Model과 마찬가지로 상속을 통해 선언

  (forms 라이브러리의 Form 클래스를 상속받음)
- 앱 폴더에 `forms.py`를 생성 후 `ArticleForm Class` 선언

```python
# articles/forms.py
from django import forms

class ArticleForm(forms.Form):
    title = forms.CharField(max_length=10)
    content = forms.CharField()
```

  - `Charfield()`의 `max_length` `form`에서는 필수 속성값은 아님
  - `form`에는 model field와 달리 `TextField`가 존재하지 않음
  - `Form Class`를 `forms.py`에 작성하는 것은 규약이 아님

    파일의 이름이 달라고 되고 models.py나 다른 어디에도 작성 가능하지만

    더 나은 유지보수의 관점, 관행적으로 `forms.py` 파일 안에 작성을 권장

2.2 `new` view 함수 업데이트

- `articles/views.py`

```python
# articles/views.py

from .forms import ArticleForm


def new(request):
    form = ArticleForm()
    context = {
        'form': form,
    }
    return render(request, 'articles/new.html', context)
```

- `articles/new.html`

```
<!-- articles/new.html -->

{% extends 'base.html' %}

{% block content %}
  <h1>NEW</h1>
  <form action="{% url 'articles:create' %}" method="POST">
    {% csrf_token %}
    {{ form.as_p }}
    <input type="submit">
  </form>
  <hr>
  <a href="{% url 'articles:index' %}">[back]</a>
{% endblock content %}
```

2.3 From rendering options

- `label & input` 쌍에 대한 3가지 출력 옵션
  - `as_p()`

    각 필드가 단락(`<p>`)으로 감싸져서 렌더링
  - `as_ul()`
    - 각 필드가 목록 항목(`<li>`)으로 감싸져서 렌더링
    - `<ul>` 태그는 직접 작성해야 됨
  - `as_table()`
    - 각 필드가 테이블(`<tr>`) 행으로 감싸져서 렌더링

2.4 Django의 2가지 HTML input 요소 표현

- `Form fields`
  - 입력에 대한 유효성 검사 로직을 처리
  - 템플릿에서 직접 사용됨
  - `forms.CharField()`
- `Widgets`
  - 웹 페이지의 HTML input 요소 렌더링을 담당
    - input 요소의 단순한 출력 부분 담당
  - Widgets은 반드시 form fields에 할당 됨
  - `forms.CharField(widget=forms.Textared)`

2.5 Widgets

- Django의 HTML input element의 표현을 담당
- 단순히 HTML 렌더링을 처리하는 것이며 유효성 검증과 아무런 관계가 없음
    - 웹 페이지에서 input element의 단순 raw한 렌더링 만을 처리하는 것

2.5.1 widgets 응용하기

```
# articles/forms.py

class ArticleForm(forms.Form):
    NATION_A = 'kr'
    NATION_B = 'uk'
    NATIONS_CHOICES = [
        (NATION_A, '한국'),
        (NATION_B, '영국'),
    ]

    title = forms.CharField(max_length=10)
    content = forms.CharField(widget=form.Textarea())
    nation = forms.ChoiceField(choice=NATIONS_CHOICES)
    nation = forms.ChoiceField(choice=NATIONS_CHOICES,
                               widgets=forms.RadioSelect())
```

3. Django ModelForm

- 이미 Article Model Cloass에 필드에 대한 정보를 작성했는데,

  Form에 맵핑하기 위해 Form Class에 필드를 재정의 해야함
- ModelForm을 사용하면 이러한 Form을 더 쉽게 작성할 수 있음

3.1 ModelForm Class

- Model을 통해 Form Class를 만들 수 있는 helper class
- ModelForm은 Form과 똑같은 방식으로 View 함수에서 사용

3.1.1 ModelForm 선언

- forms 라이브러리에서 파생된 ModelForm 클래스를 상속받음
- 정의한 ModelForm 클래스 안에 Meta 클래스를 선언
- 어떤 모델을 기반으로 form을 작성할 것인지에 대한 정보를 Meta 클래스에 지정

```python
# articles/forms.py

from django import forms
from .models import Article

class ArticleForm(forms.ModelForm):

    class Meta:
        model = Article
        fields = '__all__'
```

3.1.2 ModelForm에서의 Meta Class

```python
# articles/forms.py

class ArticleForm(forms.ModelForm):

    class Meta:
        model = Article
        fields = '__all__'
```

```python
# articles/forms.py

class ArticleForm(forms.ModelForm):

    class Meta:
        model = Article
        exclude = ('title',)
```

- ModelForm의 정보를 작성하는 곳
- ModelForm을 사용할 경우 참조 할 모델이 있어야 하는데,

  Meta class의 model 속성이 이를 구성함
- fileds 속성에 `__all__` 를 사용하여 모델의 모든 필드를 포함할 수 있음
- 또는 exclued 속성을 사용하여 모델에서 포함하지 않을 필드를 지정할 수 있음

- [참고] `Meta data`
  - 데이터를 표현하기 위한 데이터
  - ex) 사진파일
    - 사진 데이터
    - 사진 데이터의 데이터(촬영 시각, 렌즈, 조리개 값 등)
    - 사진 데이터에 대한 데이터(== 사진의 Meta data)

- [참고] 참조값과 반환 값
  - 함수를 예시로 들면 아래와 같은 함수가 있을 때 함수의 이름을 그대로 출력하는 것과 호출 후의 결과를 비교

```python
def greeting():
    return '안녕하세요'

print(greeting) # <function greeting at 0x10761caf0>
print(greeting()) # 안녕하세요
```

    - 첫번째 결과는 함수의 참조 값을 출력
    - 두번째 결과는 함수의 반환값을 출력

- 언제 참조값을 사용할까?
  - 함수를 호출하지 않고 함수 자체를 그대로 전달하여,

    다른 함수에서 **필요한 시점**에 호출하는 경우

    ```
    urlpatterns = [
        path('', views.index, name='index'),
    ]
    ```

  - view 함수의 참조 값을 그대로 넘김으로써,

    path 함수가 내부적으로 해당 view 함수를 `필요한 시점`에 사용하기 위해서
  - Article이라는 클래스를 호출하지 않고(==model을 인스턴스로 만들지 않고) 작성하는 이유는 ArticleForm이 해당 클래스를 필요한 시점에 사용하기 위함
  - 더불어 이 경우에는 인스턴스가 필요한 것이 아닌, 실제 Article 모델의 참조 값을 통해 해당 클래스의 필드나 속성 등을 내부적으로 참조하기 위한 이유도 있음

3.2 ModelForm with view functions

- `is_valid()` method
  - 유효성 검사를 실행하고, 데이터가 유효한지 여부를 boolean 으로 반환
  - 데이터 유효성 검사를 보장하기 위한 많은 테스트에 대해 Django 는 is_valid 제공
- `save()` method
  - form 인스턴스에 바인딩 된 데이터를 통해 데이터베이스 객체를 만들고 저장
  - ModelForm의 하위 클래스는 키워드 인자 instance 여부를 통해 생성할 지, 수정할 지를 결정함
    - 제공되지 않은 경우 save()는 지정된 모델의 새 인스턴스를 만듦(CREATE)
    - 제공되면 save()는 해당 인스턴스를 수정(UPDATE)

    ```
    # CREATE
    form = ArticleForm(request.POST)
    form.save()

    # UPDATE
    form = ArticleForm(request.POST, instance=article)
    form.save()
    ```

3.2.1 CREATE

```python
# articles/views.py

def create(request):
    form = ArticleForm(request.POST)
    if form.is_valid():
        article = form.save()
        return redirect('articles:detail', article.pk)
    return redirect('articles:new')
```

- 유효성 검사를 통과하면
  - 데이터 저장 후 상세 페이지로 리다이렉트
  - 리다이렉트는 url 형태로 받아서 반환
- 통과하지 못하면
  - 작성 페이지로 리다이렉트
- form 인스턴스의 errors 속성
  - is_valid()의 반환 값이 False인 경우 form 인스턴스의 errors 속성에 값이 작성되는데, 유효성 검증을 실패한 원인이 딕셔너리 형태로 저장됨

```python
# articles/views.py

def create(request):
    form = ArticleForm(request.POST)
    if form.is_valid():
        article = form.save()
        return redirect('articles:detail', article.pk)
    context = {
        'form': form,
    }
    return render(request, 'articles/new.html', context)
```

    - 다음과 같이 코드를 작성하면 사용자에게 실패 결과 메세지를 출력 가능
  - 공백과 데이터가 없는것은 다름

3.2.2 UPDATE

- ModelForm의 인자 instance는 수정 대상이 되는 객체(기존 객체)를 저장
  - `request.POST`
    - 사용자가 form을 통해 전송한 데이터 (새로운 데이터)
  - `instance`
    - 수정이 되는 대상

- `articles/views.py`

```python
# articles/views.py

def edit(request, pk):
    article = Article.objects.get(pk=pk)
    form = ArticleForm(instance=article)
    context = {
        'article': article,
        'form': form,
    }
    return render(request, 'articles/edit.html', context)
```

- `articles/edit.html`

```
<!– articles/edit.html -->

{% extends 'base.html' %}

{% block content %}
  <h1>EDIT</h1>
  <form action="{% url 'articles:update' article.pk %}" method="POST">
    {% csrf_token %}
    {{ form.as_p }}
    <input type="submit">
  </form>
  <hr>
  <a href="{% url 'articles:index' %}">[back]</a>
{% endblock content %}
```

- `articles/views.py`

```python
# articles/views.py

def update(request, pk):
    article = Article.objects.get(pk=pk)
    form = ArticleForm(request.POST, instance=article)
    if form.is_valid():
        form.save()
        return redirect('articles:detail', article.pk)
    context = {
        'form': form,
        'article': article,
    }
    return render(request, 'articles/edit.html', context)
```

  - data는 첫번째라서 생략 가능하지만, instance는 순서가 두번째가 아니라서 생략 불가

3.2.3 Form과 ModelForm

- ModelForm이 Form보다 더 좋은 것이 아니라 각자 역할이 다른 것
- `Form`
  - 사용자로부터 받는 데이터가 DB와 연관되어 있지 않는 경우에 사용
  - DB에 영향을 미치지 않고 단순 데이터만 사용되는 경우
  - 예시 - 로그인, 사용자의 데이터를 받아 인증 과정에서만 사용 후 별도로 DB에 저장하지 않음

- `ModelForm`
  - 사용자로부터 받는 데이터가 DB와 연관되어 있는 경우에 사용
  - 데이터의 유효성 검사가 끝나면 데이터를 각각 어떤 레코드에 맵핑해야 할지 이미 알고 있기 때문에 곧바로 save() 호출이 가능

3.3 Widgets

- 오른쪽 작성 방식을 권장

```python
class ArticleForm(forms.ModelForm):

    class Meta:
        model = Article
        fields = '__all__'
        widgets = {
            'title': forms.TextInput(attrs={
                'class': 'title',
                'placeholder': 'Enter the title',
                'maxlength':10,
                }
            )
        }
```

```python
# articles/forms.py

class ArticleForm(forms.ModelForm):
    title = forms.CharField(
        label='제목',
        widget=forms.TextInput(
            attrs={
                'class': 'my-title',
                'placeholder': 'Enter the title',
            }
        ),
    )

    class Meta:
        model = Article
        fields = '__all__'
```

- Widgets 활용하기

```python
# articles/forms.py

class ArticleForm(forms.ModelForm):
    title = forms.CharField(
        label='제목',
        widget=forms.TextInput(
            attrs={
                'class': 'my-title',
                'placeholder': 'Enter the title',
                'maxlength': 10,
            }
        ),
    )
    content = forms.CharField(
        label='내용',
        widget=forms.Textarea(
            attrs={
                'class': 'my-content',
                'placeholder': 'Enter the content',
                'rows': 5,
                'cols': 50,
            }
        ),
        error_messages={
            'required': 'Please enter your content'
        }
    )

    class Meta:
        model = Article
        fields = '__all__'
```

- Widgets의 요소

```python
        label = '제목',
        widget=forms.TextInput(
            attrs={
                'class' : 'my-title',
                'placeholder' : 'Enter the title',
                'maxlength' : 10,
            }
        ),
    )

```

  - 유효성 검사와는 관련X
  - 사용자가 입력할때 제한을 거는 요소

4. Handling HTTP requests

- new-create, edit-update의 view 함수 역할
  - 공통점
    - new-create는 모두 CREATE 로직을 구현하기 위한 공통 목적
    - edit-update는 모두 UPDATE 로직을 구현하기 위한 공통 목적
  - 차이점
    - new와 edit는 GET 요청에 대한 처리만
    - create와 update는 POST 요청에 대한 처리만을 진행
- 이 공통점과 차이점을 기반으로, 하나의 view 함수에서 method에 따라 로직이 분리되도록 변경

4.1 Create

- new와 create view 함수를 합침
- 각각의 역할은 request.method 값을 기준으로 나뉨

```python
# articles/views.py

def create(request):
    if request.method == 'POST':
        form = ArticleForm(request.POST)
        if form.is_valid():
            article = form.save()
            return redirect('articles:detail', article.pk)
    else:
        form = ArticleForm()
    context = {
        'form': form,
    }
    return render(request, 'articles/new.html', context)
```

  - context의 들여쓰기 위치
    - `if form.is_valid():`에서 false로 평가 받았을 때 에러 정보가 담긴 form 인스턴스가 context로 넘어갈 수 있음

4.2 Update

- edit와 update view 함수를 합침

```python
# articles/views.py

def update(request, pk):
    article = Article.objects.get(pk=pk)
    if request.method == 'POST':
        form = ArticleForm(request.POST, instance=article)
        if form.is_valid():
            form.save()
            return redirect('articles:detail', article.pk)
    else:
        form = ArticleForm(instance=article)
    context = {
        'form': form,
        'article': article,
    }
    return render(request, 'articles/update.html', context)
```

---

5. View decorators

- 데코레이터
  - 기존에 작성된 함수에 기능을 추가하고 싶을 때, 해당 함수를 수정하지 않고 기능을 추가해주는 함수
  - Django는 다양한 HTTP 기능을 지원하기 위해 view 함수에 적용 할 수 있는 여러 데코레이터를 제공
- 데코레이터 동작 예시

```python
def hello(func):
    def wrapper():
        print('HIHI')
        func()
        print('HIHI')
    return wrapper

@hello
def bye():
    print('byebye')

bye()
```

```
# 출력

HIHI
byebye
HIHI
```

6. Allowed HTTP method

- django.views.decorators.http의 데코레이터를 사용하여 요청 메서드를 기반으로 접근을 제어할 수 있음
- 일치하지 않는 메서드 요청이라면 `405 Method Not Allowed` 를 반환
  - 요청 방법이 서버에게 전달 되었으나 사용 불가능한 상태

6.1 메서드 목록

- `require_http_methods()`
  - View 함수가 특정한 요청 method만 허용하도록 하는 데코레이터

```python
# views.py

from django.views.decorators.http import require_http_methods


@require_http_methods(['GET', 'POST'])
def create(request):
    pass


@require_http_methods(['GET', 'POST'])
def update(request, pk):
    pass
```

- `require_POST()`
  - View 함수가 POST 요청 method만 허용하도록 하는 데코레이터

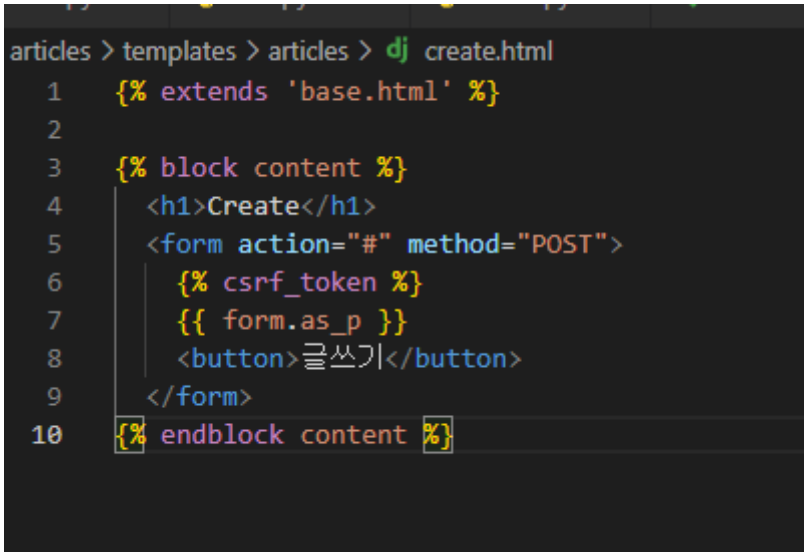
```python
# views.py

from django.views.decorators.http import require_http_methods, require_POST


@require_POST
def delete(request, pk):
    article = Article.objects.get(pk=pk)
    article.delete()
    return redirect('articles:index')
```

- `require_safe()`
  - require_GET이 있지만 Django에서는 require_safe를 사용하는 것을 권장

```python
# views.py

from django.views.decorators.http import require_http_methods, require_POST, require_safe


@require_safe
def index(request):
    ...


@require_safe
def detail(request, pk):
    ...
```

7. 마무리

- Django Form Class
  - Django 프로젝트의 주요 유효성 검사 도구
  - 공격 및 데이터 손상에 대한 중요한 방어 수단
  - 유효성 검사에 대해 개발자에게 강력한 편의를 제공
- View 함수 구조 변화
  - HTTP request 처리에 따른 구조 변화
- [참고]



  - 폼 안에 있는 버튼은 submit 역할을 함