

2-1. デバッグツールバーの導入

2019年4月4日 [コメントをする](#)

今回のテーマは「デバッグツールバーの導入」です。第二章がスタートしました。今回は開発に便利なツールを導入することにします。開発中にテンプレートに渡されている変数や、リクエストで受けた値を確認したいことは多いと思います。今回導入するdjango-debug-toolbarはサードパーティ製ライブラリですが、非常に使いやすいですよ。

※本ページは[コメント投稿機能を付与する](#)まで読まれた方を対象としています。そのためサンプルソースコードが省略されている場合があります。

ツールバーの導入

まずはpipでdjango-debug-toolbarを導入しましょう。

```
(venv)$ pip install django-debug-toolbar
```

次にmysite/settings.pyの設定に入ります。

デバッグツールバー表示のための設定

mysite/settings.py(一部抜粋)

```

DEBUG = True # Trueでないとデバッグツールは表示されない。
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
+   'debug_toolbar',
    'base',
    'thread',
]

+ INTERNAL_IPS = [
+     '127.0.0.1',
+ ]

MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
+   'debug_toolbar.middleware.DebugToolbarMiddleware', # Deubg tool bar
]

```

今回はローカル環境内で確認することを前提としてINTERNAL_IPSのIPにはローカルのIPを当てています。もし、開発用サーバーを別に立てる場合はアクセスする側のIPを設定してください。

そしてmysite/urls.pyにも変更を加えます。

mysite/urls.py

```
from django.contrib import admin, auth
from django.urls import path, include
+ from django.conf import settings

urlpatterns = [
    path('admin/', admin.site.urls),
    path('accounts/', include('django.contrib.auth.urls')),
    path('', include('base.urls')),
    path('thread/', include('thread.urls')),
]

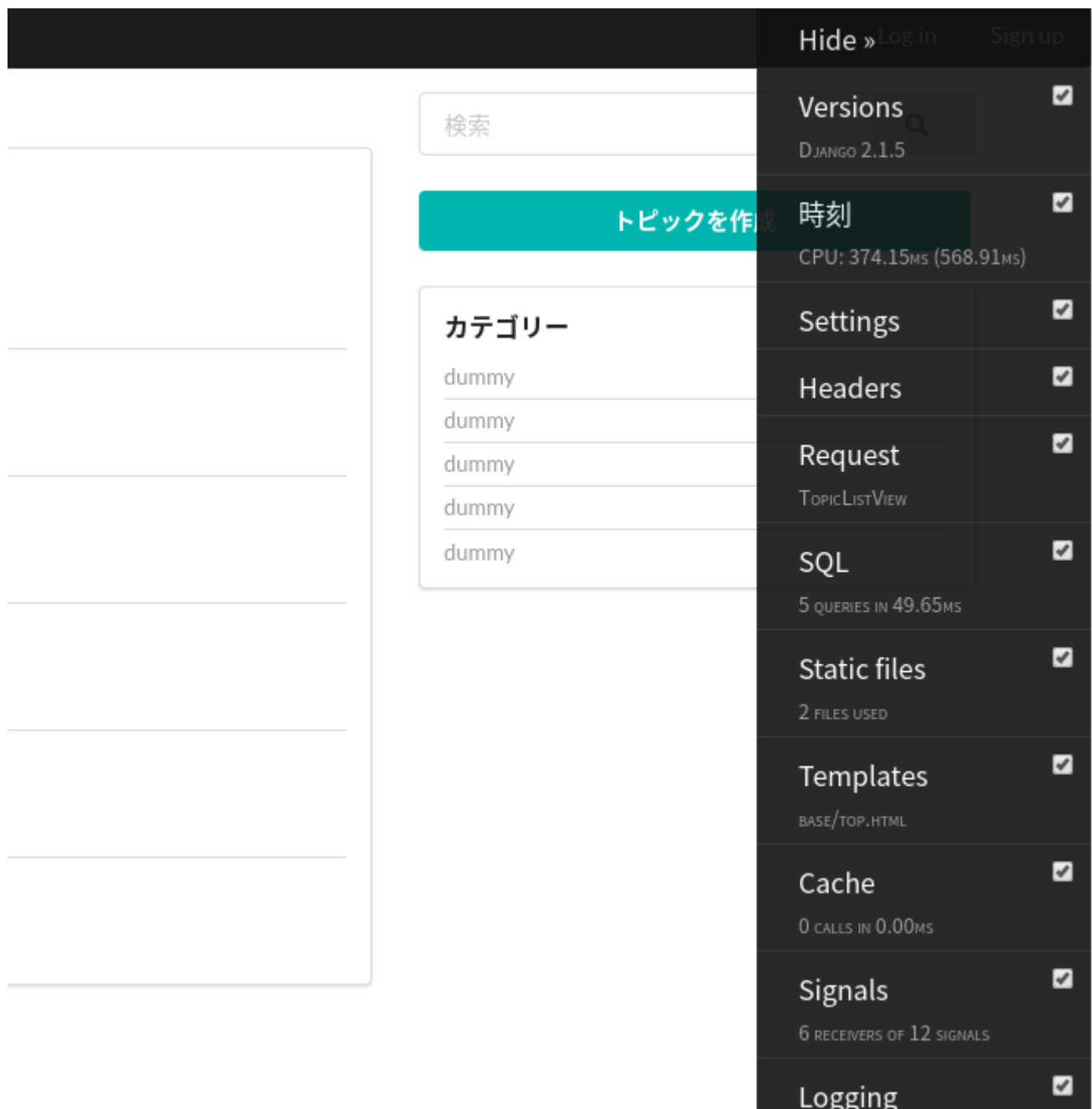
+ if settings.DEBUG:
+     import debug_toolbar
+     urlpatterns = [
+         path('__debug__/', include(debug_toolbar.urls)),
+     ] + urlpatterns
```

これで設定は完了です。

localhost:8080にアクセスしてブラウザで確認してみましょう。(ブラウザでの確認方法は[プロジェクトの作成](#)をご確認下さい)



'D,DT'をクリックする。



デバッグツールで分かること

デバッグツールは様々な情報を表示してくれる大変便利なツールです。よく使う機能としてはPOSTやGETで送信された情報の確認やテンプレートに渡された情報の確認だと思います。いくつかユースケースを紹介します。

テンプレートに渡されたコンテキストを確認する

まずテンプレートに渡したコンテキストの中身を確認したい場合ですが、デバッグツールバーの'Templates'の項を確認します。各テンプレートのToggle contextを開けるとコンテキストが確認出来ます。

thread/detail_topic.html

/mysite/templates/thread/detail_topi

▼ Toggle context

```
{'False': False, 'None': None, 'True': True}
{'DEFAULT_MESSAGE_LEVELS': {'DEBUG': 10,
                             'ERROR': 40,
                             'INFO': 20,
                             'SUCCESS': 25,
                             'WARNING': 30},
 'csrf_token': <SimpleLazyObject: '0bJ50JHue0Ueu448RRAYhl
 'debug': True,
 'messages': <django.contrib.messages.storage.fallback.F
 'perms': <django.contrib.auth.context_processors.PermWr
 'request': '<<request>>',
 'sql_queries': <function debug.<locals>.<lambda> at 0x7
 'user': <SimpleLazyObject: <User: admin>>}}
{'comment_list': '<<queryset of thread.Comment>>',
 'form': <CommentModelForm bound=False, valid=Unknown, f
 'topic': <Topic: Djangoモデル作成について>,
 'view': <thread.views.TopicViewAndCommentCreateView obj
```

リクエスト内容を確認する

GETやPOSTでリクエストされた内容、或いはセッションの内容を確認したい場合は'Request'の項を確認します。

Session data

Variable	Value
'_auth_user_backend'	'django.contrib.auth.ba
'_auth_user_hash'	'137e0e6e6e27df79d8475f
'_auth_user_id'	'1'

No GET data









POST data

Variable	
'category'	['1']
'csrfmiddlewaretoken'	['sK9yCMfrEL0z2FWjvRS52
'message'	['テスト投稿']
'next'	['confirm']
'title'	['テスト']
'user_name'	['名無し']

発行されたSQLを確認する

クエリセットが評価された後に発行されるSQL文を確認する際にもデバッグツールは役に立ちます。'SQL'の項を開くとSQLコマンドに関する情報を閲覧することが出来ます。クエリセットの評価結果が予想外の結果を返す場合などに発行されているSQLを確認することはとても重要ですので、活用して下さい

い。

Query	Timeline	Time (ms)	操作
 SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED		0.13	
 SELECT ... FROM `django_session` WHERE (`django_session`.`expire_date` > '2019-03-11 02:38:49.465958' AND `django_session`.`session_key` = 'utjvjlG0zs605agdZ7t8uZx0Gj7q7ubb')		0.53	Sel Expl
 SELECT ... FROM `auth_user` WHERE `auth_user`.`id` = 1		0.46	Sel Expl
 SELECT `thread_topic`.`id`, `thread_topic`.`user_name`, `thread_topic`.`title`, `thread_topic`.`message`, `thread_topic`.`category_id`, `thread_topic`.`created`, `thread_topic`.`modified` FROM `thread_topic` ORDER BY `thread_topic`.`created` DESC		0.42	Sel Expl

最後に

本当は一章の先頭で紹介しようかと思っていたのですが、Djangoを機能を紹介する本章の内容としました。次回はテンプレートタグを使ってサイドバーのカテゴリー部分を作成していきます。

Sponsored Link

1-20. コメント投稿機能を付与する

2019年4月3日 [コメントをする](#)

今回のテーマは「コメント投稿機能を付与する」です。第一章もいよいよ最後です。まだまだ不完全な掲示板アプリではありますが、今回を終えると以下の機能を満たすウェブアプリになる予定です。

- トピックの登録
- トピックの一覧表示
- トピックのカテゴリ毎の表示
- コメントの一覧表示
- コメントの追加

※本ページは[1-19. カテゴリ毎のトピック一覧画面を作る](#)まで読まれた方を対象としています。そのためサンプルソースコードが省略されている場合があります。

ここまで手を動かしてきた方はなんとなくDjangoというフレームワークを使う感覚が分かってきたのではないのでしょうか？（細かい機能はまだ紹介しきれていません）今回はこれまでの内容を理解していれば難しくありません。Djangoでスピーディに実装する感覚を味わいましょう。

※本ページは[カテゴリ毎のトピック一覧画面を作る](#)まで読まれた方を対象としています。そのためサンプルソースコードが省略されている場合があります。

コメント投稿用のフォーム作成

ユーザー入力画面といえばフォームの作成ですね。今回もモデルベースのフォームで大丈夫ですのでModelFormを継承して作っていきます。

`thread/forms.py`(一部抜粋)

```
class CommentModelForm(forms.ModelForm):
    class Meta:
        model = Comment
        fields = [
            'user_name',
            'message',
        ]

    def __init__(self, *args, **kwargs):
        kwargs.setdefault('label_suffix', '')
        super().__init__(*args, **kwargs)
        self.fields['user_name'].widget.attrs['value'] = '名無し'
```

[FormとModelForm](#)で扱ったとおり__init__関数をオーバーライドして各種設定をしています。

トピック詳細表示テンプレートの修正

では早速トピック詳細表示テンプレートであるthread/detail_topic.htmlをコメント表示・登録用に修正しましょう。

templates/thread/detail_topic.html


```

{% extends 'base/base.html' %}
{% block title %}トピック作成 - {{ block.super }}{% endblock %}
{% block content %}
<div class="ui grid stackable">
  <div class="eleven wide column">
    <div class="ui breadcrumb">
      <a href="{% url 'base:top' %}" class="section">TOP</a>
      <i class="right angle icon divider"></i>
      <a href="{% url 'thread:category' url_code=topic.category.url_code %}" class="
      <i class="right angle icon divider"></i>
      <a class="active section">{{topic.title}}</a>
    </div>
    <div class="ui segment">
      <div class="content">
        <div class="header"><h3>{{topic.title}}</h3></div>
        <p>{{topic.user_name}} - {{topic.created}}</p>
        <div class="ui segment">
          <p><pre>{{topic.message}}</pre></p>
        </div>
      </div>
    </div>
    <!--コメント表示-->
    <div class="ui segment">
      {% if comment_list %}
      {% for comment in comment_list %}
      <div class="ui segment secondary">
        <p>{{comment.no}}. {{comment.user_name}}<br>{{comment.created}}</p>
        {% if comment.pub_flg %}
        <p><pre>{{comment.message}}</pre></p>
        {% else %}
        <p style="color: #aaa">このコメントは非表示となりました。</p>
        {% endif %}
      </div>
      {% endfor %}
      {% else %}
      <div class="ui warning message"><p>まだコメントはありません</p></div>
      {% endif %}
    </div>
    <!--//コメント表示-->
    <!--コメント投稿-->
    <h4>コメント投稿</h4>
    <div class="ui segment">
      <form class="ui form" action="" method="POST">
        {% csrf_token %}
        {{form.as_p}}
        <button class="ui button orange" type="submit">コメント投稿</button>
      </form>
    </div>

```

```
<!--//コメント投稿-->
</div>
{% include 'base/sidebar.html' %}
</div>
{% endblock %}
```

テンプレートの中で条件分岐をしています。1つ目はcomment_listの有無で分岐、2つ目はコメントのpub_flgによって表示を分岐しています。Djangoのテンプレートはループや分岐等のある程度ロジカルな操作が可能です。あくまで表示に関する部分に限り見せたいデータの整形はViewで行った方が良いと考えます。また、コメント投稿欄については{{form.as_p}}でHTML出力しました。

ビューの作成

では次にビューを作っていきます。TopicDetailViewをカスタマイズしても良いのですが、練習なので新たにクラスを作成しましょう。TopicAndCommentViewとします。(ダサい命名でスミマセン)この画面はトピックの詳細表示、コメントのリスト表示、コメントの作成が行われる画面です。コメントの投稿は確認画面は作りません。さて、クラスベースビューで実装する際、どのベースビューを継承するのが良いでしょうか？これが正解というのはないのですが、今回の場合だとFormViewを継承したクラスを作るのが手間が少ないかと思います。CreateViewを使わない理由は入力した情報だけでなく、他の情報も付与してオブジェクトを保存したいからです。これについては実際のコードを見たほうが早いと思います。ではthread/views.pyを見ていきましょう。

```
class TopicAndCommentView(FormView):
    template_name = 'thread/detail_topic.html'
    form_class = CommentModelForm

    def form_valid(self, form):
        comment = form.save(commit=False) #保存せずオブジェクト生成する
        comment.topic = Topic.objects.get(id=self.kwargs['pk'])
        comment.no = Comment.objects.filter(topic=self.kwargs['pk']).count() + 1
        comment.save()
        return super().form_valid(form)

    def get_success_url(self):
        return reverse_lazy('thread:topic', kwargs={'pk': self.kwargs['pk']})

    def get_context_data(self):
        ctx = super().get_context_data()
        ctx['topic'] = Topic.objects.get(id=self.kwargs['pk'])
        ctx['comment_list'] = Comment.objects.filter(
            topic_id=self.kwargs['pk']).order_by('no')
        return ctx
```

簡単に解説します。template_nameについてはもう説明不要かと思います。GETでアクセスされた場合に表示するテンプレート名ですね。この時テンプレートに渡すコンテキストがget_context_data関数で定義されています。ここでは表示するトピックとリスト表示するコメント一覧をDBから取得していま

す。'topic_id'は'topic__id'の誤りではないのか？とお気づきになった方は鋭いです。外部キーのIDは例外扱いなのです。（参考；[公式ドキュメント](#)）get_success_url関数はフォームのデータ検証成功後のリダイレクト先のURLを定義しています。

さて、問題はform_valid関数の中身ですね。これまでの話であればform.save()メソッドを呼んで保存すれば良かったのですが、トピックとコメント番号は後から付与したいために、上記のような記述になっています。saveメソッドにcommit=Falseと引数を渡すことで、保存せずにオブジェクトだけを生成出来ます。このオブジェクトに情報を付与して改めて保存しています。よく使う手段なので覚えておくと便利です。

thread/urls.pyも書き換えましょう。

thread/urls.py

```
from django.urls import path

from . import views
app_name = 'thread'

urlpatterns = [
    path('create_topic/', views.TopicCreateView.as_view(), name='create_topic'),
    path('<int:pk>', views.TopicAndCommentView.as_view(), name='topic'),
    path('category/<str:url_code>', views.CategoryView.as_view(), name='category'),
]
```

あらためてMVTモデルについて考える

コメントを投稿し、保存するという機能は満たしました。この処理このようにビューで保存処理をしている参考サイトはとても多くて、おそらくこのように書く場合も多いのだと思います。しかし、改めて考えるとcommentオブジェクトの保存処理はビューの仕事ではありません。あくまで「ユーザーからこんな情報でコメント作成するって依頼が来ましたよ」とモデルに伝えて、その結果として「ユーザーに見せる情報」を返すことが（Djangoの）ビューの仕事です。なので、commentの保存処理はモデルにお任せしましょう。今回のようにフォームから受けた情報の保存処理はモデルのマネージャに書くか、フォームに書くことが多いです。フォームがモデルとは意外かも知れません。Djangoにおけるフォームの役割はMVTを横断しているため分かりづらいのですが、特にModelFormはモデルとしての役割が強いです。今回はCommentオブジェクトの保存処理はCommentModelForm内で処理することにします。もちろん、CommentManager内で書いても良いと思います。（話がDjangoとは逸れますが、MVCモデルでもビューやコントローラにビジネスロジックを書きまくる初心者の方が意外に多い気がします…）

thread/forms.py(一部抜粋)

```
class CommentModelForm(forms.ModelForm):
    class Meta:
        model = Comment
        fields = [
            'user_name',
            'message',
            'image',
        ]

    def __init__(self, *args, **kwargs):
        kwargs.setdefault('label_suffix', '')
        super().__init__(*args, **kwargs)
        self.fields['user_name'].widget.attrs['value'] = '名無し'

    def save_with_topic(self, topic_id, commit=True):
        comment = self.save(commit=False)
        comment.topic = Topic.objects.get(id=topic_id)
        comment.no = Comment.objects.filter(topic_id=topic_id).count() + 1
        if commit:
            comment.save()
        return comment
```

CommentModelFormクラスにsave_with_topic関数を定義しました。トピックIDを引数にとりコメントを保存します。ビューからはこの処理をメソッドとして呼び出すように書き換えましょう。

thread/views.py(一部抜粋)

```
class TopicAndCommentView(FormView):
    template_name = 'thread/detail_topic.html'
    form_class = CommentModelForm

    def form_valid(self, form):
        # comment = form.save(commit=False)
        # comment.topic = Topic.objects.get(id=self.kwargs['pk'])
        # comment.no = Comment.objects.filter(topic=self.kwargs['pk']).count() + 1
        # comment.save()
        # コメント保存のためsave_with_topicメソッドを呼ぶ
        forms.save_with_topic(self.kwargs.get('pk'))
        return super().form_valid(form)

    def get_success_url(self):
        return reverse_lazy('thread:topic', kwargs={'pk': self.kwargs['pk']})

    def get_context_data(self):
        ctx = super().get_context_data()
        ctx['topic'] = Topic.objects.get(id=self.kwargs['pk'])
        ctx['comment_list'] = Comment.objects.filter(
            topic_id=self.kwargs['pk']).order_by('no')
        return ctx
```

これでビューは本来の仕事に専念できますね。「どうやってcommentを保存するか」はモデルにお任せすべきと考えます。

第一章の終わりに

お疲れ様でした。ここまで手を動かしてコードを書いてきた方はDjangoの使い方が何となく見えてきたでしょうか？見えてくると良いなと思っています。作ってきたサンプルアプリの確認をしてみましょう。

Djangoモデル作成について

名無し - 2019年3月6日 15:15

サンプルサンプルサンプルサンプル
サンプルサンプルサンプル
サンプルサンプル
サンプル

まだコメントはありません

コメント投稿

お名前

名無し

投稿内容

テストテスト
テスト

コメント投稿

Djangoモデル作成について

名無し - 2019年3月6日 15:15

サンプルサンプルサンプルサンプル
サンプルサンプルサンプル
サンプルサンプル
サンプル

1. 名無し
2019年3月8日 15:41

テストテスト
テスト

コメント投稿

お名前

名無し

投稿内容

これで、取り敢えず「掲示板」と呼べそうなものが出来上がってきました。サイドバーの表示がまだダミーのままですが、その処理も第二章で扱います。まだまだお伝えしたいことはありますが第一章に関しては筆を置きたいと思います。しばらく鋭気を養ったら第二章を書き始めます。どうぞ宜しくお願いいたします。

おまけ：サンプルアプリの補足

この章の終わりにもう少し頑張って、途切れているリンクの修正や静的なページを追加しちゃいましょう。面倒な方は作業不要です。追加ページは「プライバシーポリシー」、「このサイトについて」です。リンクやCSSを一部修正したバージョンを下記しておきます。Djangoを解説するという本筋からは逸れるので本文では放置してきた部分です。

`templates/base/base.html`

```

{% load static %}
<!DOCTYPE html>
<head>
    <meta charset="UTF-8">
    <meta http-equiv="content-language" content="ja">
    <meta name="viewport" content="width=device-width, initial-scale=1.0, maximum-scale=1.0">
    {% block meta_tag %}{% endblock %}
    <link href="{% static 'css/semantic.css' %}" rel="stylesheet">
    {% block css %}{% endblock %}
    <title>
        {% block title %}IT学習ちゃんねる{% endblock %}
    </title>
</head>
<body>
    <div class="ui stackable inverted menu">
        <a href="{% url 'base:top' %}" class="header item">
            IT学習ちゃんねる
        </a>
        <a href="{% url 'base:about' %}" class="item">
            このサイトはなに？
        </a>
        <a class="item" href="{% url 'thread:create_topic' %}">
            トピック作成
        </a>
        <div class="right menu">
            <a class="item">
                Log in
            </a>
            <a class="item">
                Sign up
            </a>
        </div>
    </div>

    <div class="ui container" style="min-height:100vh;">
        {% block content %}
        {% endblock %}
    </div>

    <div class="ui inverted stackable footer segment">
        <div class="ui container center aligned">
            <div class="ui horizontal inverted small divided link list">
                <a href="{% url 'base:top' %}" class="item">© 2019 IT学習ちゃんねる(仮)</a>
                <a href="{% url 'base:terms' %}" class="item">利用規約</a>
                <a href="{% url 'base:policy' %}" class="item">プライバシーポリシー</a>
            </div>
        </div>
    </div>

    <script src="https://code.jquery.com/jquery-3.1.1.min.js"></script>

```



```
<script type="text/javascript" src="{% static 'js/semantic.js' %}"></script>
{% block js %}{% endblock %}
</body>
```

templates/base/sidebar.html

```
<div class="five wide column">
  <div class="ui action input" style="width: 100%;">
    <input type="text" placeholder="検索">
    <button class="ui button"><i class="search icon"></i></button>
  </div>
  <div class="ui items">
    <div class="item">
      <a href="{% url 'thread:create_topic' %}" class="ui fluid teal button">トピック
    </div>
  </div>
  <div class="ui segment">
    <div class="content">
      <div class="header"><h4>カテゴリー</h4></div>
      <div class="ui relaxed list small divided link">
        <a class="item">dummy</a>
        <a class="item">dummy</a>
        <a class="item">dummy</a>
        <a class="item">dummy</a>
        <a class="item">dummy</a>
      </div>
    </div>
  </div>
</div>
```

templates/base/policy.html

```
{% extends 'base/base.html' %}
{% block title %}プライバシーポリシー - {{ block.super }}{% endblock %}
{% block content %}
<div class="ui grid stackable">
  <div class="eleven wide column">
    <div class="ui breadcrumb">
      <a href="{% url 'base:top' %}" class="section">TOP</a>
      <i class="right angle icon divider"></i>
      <a class="active section">プライバシーポリシー</a>
    </div>
    <div class="ui segment">
      <div class="content">
        <div class="header"><h3>プライバシーポリシー</h3></div>
        <p>あんなことやこんなことに情報を使います。などなど</p>
        <p>.....</p>
        <p>.....</p>
        <p>.....</p>
      </div>
    </div>
  </div>
  {% include 'base/sidebar.html' %}
</div>
{% endblock %}
```

templates/base/about.html

```

{% extends 'base/base.html' %}
{% block title %}このサイトはなに？ - {{ block.super }}{% endblock %}
{% block content %}
<div class="ui grid stackable">
  <div class="eleven wide column">
    <div class="ui breadcrumb">
      <a href="{% url 'base:top' %}" class="section">TOP</a>
      <i class="right angle icon divider"></i>
      <a class="active section">このサイトはなに？</a>
    </div>
    <div class="ui segment">
      <div class="content">
        <div class="header"><h3>このサイトはなに？</h3></div>
        <p>このサイトはDjangoのサンプルアプリです。などなど</p>
        <p>.....</p>
        <p>.....</p>
        <p>.....</p>
      </div>
    </div>
  </div>
  {% include 'base/sidebar.html' %}
</div>
{% endblock %}

```

base/urls.py

```

from django.urls import path
from django.views.generic import TemplateView
from . import views

app_name = 'base'

urlpatterns = [
    path('', views.TopicListView.as_view(), name='top'),
    # path('', views.top, name='top'),
    path('terms/', TemplateView.as_view(template_name='base/terms.html'), name='terms'),
    path('policy/', TemplateView.as_view(template_name='base/policy.html'), name='policy'),
    path('about/', TemplateView.as_view(template_name='base/about.html'), name='about'),
]

```

Sponsored Link

1-19. カテゴリー毎のトピック一覧画面を作る

2019年4月2日 [8件のコメント](#)

今回のテーマは「カテゴリー毎のトピック一覧画面を作る」です。ここでは簡単なDjangoのORM(Object Relational Mapper)の使い方を確認しておきましょう。

※本ページは[確認画面付きのトピック作成画面を作る](#)まで読まれた方を対象としています。そのためサンプルソースコードが省略されている場合があります。

データベースからのデータ取得

Djangoではクエリセットというイテレーション可能なオブジェクトを生成して、それを評価することでデータベースと情報をやり取りします。プログラマは直接SQLを書かずともクエリセットを生成する処理を行うことで、内部的にSQLが発行されてデータが処理されます。クエリセットに関しては[QuerySet API](#)が用意されており、プログラマはモデルが持つマネージャを介してクエリセットを操作します。

参考：[Django 1.0のAPIリファレンス](#)：情報が古いのですが日本語なので理解しやすいかも知れませんが、記載しておきます。

既にトップページでトピック一覧を作っているのでデータベースからのデータの取得については扱っていませんが、以前はorder_by関数で並び替えをただけでしたので、ここで基本的な操作を少し見てみます。全てのAPIに関してはAPIリファレンスを見ていただくとして、ここではよく使いそうなものをピックアップします。

クエリセットを生成し返す関数

all()：全てのオブジェクトが入ったクエリセットを返す。
filter(**kwargs)：指定の照合パラメータに一致するオブジェクトの入った新たなクエリセット返す。
exclude(**kwargs)：filterのNOT検索版
order_by(*fields)：指定パラメータで昇順に並び替える。'-'をつけると降順に並び替え
annotate(*args, **kwargs)：検索結果に付帯情報をもたせる時に使用する。集計結果やサブクエリ使用時に使う。

クエリセットを生成する関数はドットでつないで記載することが出来ます。その場合はAND条件で結合されます。

クエリセット以外を返す関数

get(**kwargs)：照合パラメータに合致するオブジェクトを返す
first()：クエリセットの最初のオブジェクトを返す
last()：クエリセットの最後のオブジェクトを返す
count()：クエリセットのオブジェクトの個数を返す

カテゴリー毎のリスト表示テンプレート作成

前置きが長くなりました。まずはテンプレートを作成します。

templates/thread/category.html

```
{% extends 'base/base.html' %}
{% block title %}{{category.name}} - {{ block.super }}{% endblock %}
{% block content %}
<div class="ui grid stackable">
  <div class="eleven wide column">
    <div class="ui breadcrumb">
      <a class="section">TOP</a>
      <a class="active section">{{category.name}}</a>
    </div>
    <div class="ui segment">
      <div class="content">
        <div class="header"><h3>{{category.name}}</h3></div>
        <div class="ui divided items">
          {% if topic_list %}
          {% for topic in topic_list %}
          <div class="item">
            <div class="content">
              <div class="header">
                <a href="{% url 'thread:topic' pk=topic.id %}"><h4>{{topic
              </div>
              <div class="meta">
                <span class="name">{{topic.user_name}}</span>
                <span class="date">{{topic.created}}</span>
              </div>
            </div>
          </div>
          {% endfor %}
          {% else %}
          <div class="ui warning message">トピックが存在しません</div>
          {% endif %}
        </div>
      </div>
    </div>
  </div>
  {% include 'base/sidebar.html' %}
</div>
{% endblock %}
```

もう特別解説は必要ないと思います。賢明な読者はすでに渡すパラメータがどのようなものか想像ついていると思います。次にビューを作ります。今回はリスト表示ということでListViewを継承したクラスビューを作ります。

thread/views.py(一部抜粋)

```

from django.views.generic import (
    CreateView, FormView, DetailView, TemplateView, ListView)

class CategoryView(ListView):
    template_name = 'thread/category.html'
    context_object_name = 'topic_list'

    def get_queryset(self):
        return Topic.objects.filter(category__url_code = self.kwargs['url_code'])

    def get_context_data(self, **kwargs):
        ctx = super().get_context_data(**kwargs)
        ctx['category'] = get_object_or_404(Category, url_code=self.kwargs['url_code'])
        return ctx

```

さて、ListViewについては良いと思います。今回はmodelパラメータがないですね。ListViewはmodelもしくはquerysetが必要です。今回はURLパラメータとして'url_code'を受ける必要があったのでget_queryset関数ないでquerysetを規定しました。肝心の中身ですが、前述のfilter関数が使われていますね。ここでトピックが持つカテゴリーの'url_code'を検索条件にしているところがキーポイントです。トピックのプロパティなら話が簡単です。id=3やtitle='hogehoge'などで指定すれば良いのです。しかし今回は一度Categoryオブジェクトを取得するサブクエリを使いたくなるような場面ですよね。Djangoではモデルを跨ぐ検索も'__'でつなぐことで照合条件に指定できます。初心者が意外に躓くポイントなので解説しておきます。

また、get_context_data関数内ではget_object_or_404を使っています。以前に使っているので詳細は省きますがこれは存在しないurl_codeが指定された場合の対策です。

最後にURLを規定しましょう。thread/urls.pyは以下のようになります。

thread/urls.py(一部抜粋)

```

urlpatterns = [
    path('create_topic/', views.TopicCreateView.as_view(), name='create_topic'),
    path('<int:pk>', views.TopicTemplateView.as_view(), name='topic'),
    path('category/<str:url_code>', views.CategoryView.as_view(), name='category'),
]

```

URLのパラメータとして今回は文字列を受けますのでstrで型を指定しています。では確認してみましょう。localhost:8080/thread/category/web_app/にアクセスしてみます。

[TOP](#) > [WEB技術](#)

WEB技術

[Djangoモデル作成について](#)

名無し 2019年3月6日15:15

[Djangoの分かりやすいページ](#)

名無し 2019年3月7日15:55

今度はトピックが存在しないカテゴリを表示してみましょう。localhost:8080/thread/category/os/にアクセスしてみます。

[TOP](#) > [OS関連・インフラ](#)

OS関連・インフラ

トピックが存在しません

本題とは外れますが、トピック詳細ページのパンくず(breadcrumbs)のURLをカテゴリーページのURLに直しておきましょう。

templates/thread/detail_topic.html(一部抜粋)

```
<div class="ui breadcrumb">
  <a href="{% url 'base:top' %}" class="section">TOP</a>
  <i class="right angle icon divider"></i>
-  <a class="section">{{topic.category.name}}</a>
+  <a href="{% url 'thread:category' url_code=topic.category.url_code %}" class="section
  <i class="right angle icon divider"></i>
  <a class="active section">{{topic.title}}</a>
</div>
```

最後に

クエリセットの作成は慣れが必要だと思います。今回の例だけでは練習不足だと思いますので、出会う度にAPIリファレンスを読み想定する挙動をするクエリセットが作成できるように練習しましょう。次回で一章は終了予定です。トピック詳細画面にコメント表示と登録機能を付与していきます。

Sponsored Link

1-18. 確認画面付きのトピック作成画面を作る

2019年4月1日 [5件のコメント](#)

今回のテーマは「確認画面付きのトピック作成画面を作る」です。確認画面つき画面とはユーザーの入力した内容を一度表示して必要に応じて入力画面に戻ることができる画面のことです。アンケートページや申込みページでは頻繁に使われますね。これまで使ってきたCreateViewに少し手を加えるだけで簡単に確認画

面つきの登録画面が作れますよ。

はじめにお断りしておくと、確認画面の作り方は色々な手法があつて、今回ご紹介する方法はあくまで1つの例と捉えていただければと思います。データ保持にセッションを使ったり、URLをページ毎に分ける場合もありますし、ページ遷移はフロントのみ（バックエンドはAPIのみ担当）で対応するなど、ケースごとに対応が異なります。（これは確認画面に限った話ではないですが）

※本ページは[FormViewとCreateViewを使う](#)まで読まれた方を対象としています。そのためサンプルソースコードが省略されている場合があります。

確認事項

まず、現状を確認しましょう。ここまで説明のためにビューに色々なクラスを作成してきたので、少し整理します。まず、ビューですが、`thread/views.py`には`TopicCreateView`クラスが以下のように書かれている筈です。

`thread/views.py`(一部抜粋)

```
class TopicCreateView(CreateView):
    template_name = 'thread/create_topic.html'
    form_class = TopicModelForm
    model = Topic
    success_url = reverse_lazy('base:top')
```

そして`thread/urls.py`は`localhost:8080/create_topic/`にアクセスされた場合にトピック作成画面を表示するように以下の様になっています。

`thread/urls.py`(一部抜粋)

```
urlpatterns = [  
    path('create_topic/', views.TopicCreateView.as_view(), name='create_topic'),  
    # path('create_topic/', views.topic_create, name='create_topic'),  
    path('<int:pk>', views.TopicTemplateView.as_view(), name='topic'),  
]
```

登録が成功した場合にはbaseアプリケーションのTOPが表示され、登録されたトピックが一番上に表示されるようになっていますね。

確認画面テンプレートの作成

では確認用の画面を作るわけですからテンプレートを追加しましょう。今回は `templates/thread/confirm_topic.html` と名付けましょう。

`templates/thread/confirm_topic.html`

```

{% extends 'base/base.html' %}
{% block title %}トピック作成 - {{ block.super }}{% endblock %}
{% block content %}
<div class="ui grid stackable">
  <div class="eleven wide column">
    <div class="ui breadcrumb">
      <a href="{% url 'base:top' %}" class="section">TOP</a>
      <i class="right angle icon divider"></i>
      <a class="active section">トピック作成</a>
    </div>
    <div class="ui segment">
      <div class="content">
        <div class="header"><h3>トピック作成</h3></div>
        <p>内容を確認してください</p>
        <table class="ui celled table table table-hover" >
          <tr><td>タイトル</td><td>{{form.title.value}}</td></tr>
          <tr><td>お名前</td><td>{{form.user_name.value}}</td></tr>
          <tr><td>カテゴリー</td><td>{{form.cleaned_data.category}}</td></tr>
          <tr><td>本文</td><td><pre>{{form.message.value}}</pre></td></tr>
        </table>
        <form class="ui form" action="{% url 'thread:create_topic' %}" method="POST" >
          {{ csrf_token }}
          {% for field in form %}
            {{field.as_hidden}}
          {% endfor %}
          <button class="ui button grey" type="submit" name="next" value="back">
          <button class="ui button orange" type="submit" name="next" value="create">
        </form>
      </div>
    </div>
  </div>
  <div>
    {% include 'base/sidebar.html' %}
  </div>
{% endblock %}

```

注目点は3つあります。まず、カテゴリーのところだけ書き方が変わっていますね。form.category.valueにはカテゴリーIDの数値しか入っておらず、確認画面で表示してもユーザーには分かりません。そこでform_valid関数通過後に生成される検証済みのデータ（cleaned_data）からカテゴリーを取得して表示しています。

二つ目は{{field.as_hidden}}ですね。これは{{form.as_p}}と同様にHTMLを返す関数で、インプットタグのタイプをhiddenとしてくれます。確認画面を作る場合セッションにデータを保持する方法もありますが、今回はhiddenのインプットタグで再度POSTする方式を取ります。

3つ目ですが、戻るボタンと作成ボタンのname属性とvalue属性ですね。つまりPOSTする度にnextパラメータに次にどちらに進むかを司令を出すというわけです。

次に既に作成済みのtemplates/thread/create_topic.htmlにも手を加えましょう。

templates/thread/create_topic.html(一部抜粋)

```
- <button type="submit" class="ui button">作成</button>
+ <button type="submit" class="ui button" name="next" value="confirm">作成</button>
```

先程と同様にname属性とvalue属性を追加しました。つまり確認画面に進めという指示を送るということです。

ビューの作成

ではビューの作成に入っていきます。既に作成済みのTopicCreateViewを改良していきます。TopicCreateViewはDjangoのクラスベースビューであるCreateViewを継承したクラスです。

thread/views.py (一部抜粋)

```
class TopicCreateView(CreateView):
    template_name = 'thread/create_topic.html'
    form_class = TopicModelForm
    model = Topic
    success_url = reverse_lazy('base:top')

    def form_valid(self, form):
        ctx = {'form': form}
        if self.request.POST.get('next', '') == 'confirm':
            return render(self.request, 'thread/confirm_topic.html', ctx)
        if self.request.POST.get('next', '') == 'back':
            return render(self.request, 'thread/create_topic.html', ctx)
        if self.request.POST.get('next', '') == 'create':
            return super().form_valid(form)
        else:
            # 正常動作ではここは通らない。エラーページへの遷移でも良い
            return redirect(reverse_lazy('base:top'))
```

ソースコードを見たままなのですが、基本的に手を加えたのはform_valid関数のオーバーライドのみです。POSTされたnext値によって場合わけして表示するテンプレートを分けています。'confirm'の場合は先程作成したconfirm_topic.htmlを表示しています。'back'であった場合には受けたformをそのまま渡して入力画面であるtopic_create.htmlを表示しています。これにより、ユーザーの入力値は保持されたまま表示されます。'create'であった場合にはCreateViewのform_valid関数を呼び出して保存処理を行いsuccess_urlに遷移します。これらどれも無い場合は異常系動作となりますが、今回はトップページに飛ばしました。エラーページなどを作成してそちらに飛ばしても良いと思います。

確認してみましょう。

入力画面

[TOP](#) > [トピック作成](#)

トピック作成

タイトル

Djangoの分かりやすいページ

お名前

名無し

カテゴリー

WEB技術 ▼

本文

サンプルサンプルサンプルサンプル
サンプルサンプルサンプル
サンプルサンプル
サンプル

作成

トピック作成

内容を確認してください

タイトル	Djangoの分かりやすいページ
お名前	名無し
カテゴリー	WEB技術
本文	サンプルサンプルサンプルサンプル サンプルサンプルサンプル サンプルサンプル サンプル

[戻る](#)[作成](#)

新着スレッド

[Djangoの分かりやすいページ](#)

名無し 2019年3月7日15:55

[素晴らしきネコの世界](#)

ネコ 2019年3月7日14:53

[Python入門サイトを教えて下さい](#)

名無し 2019年3月6日15:17

最後に

データ検証をフォームが担ってくれているので、基本的にはビューの仕事は見せるべきデータを揃えてテンプレート渡すだけ、という分かりやすい例かなと考えています。一章もそろそろ大詰めですね。次回はカテゴリ毎にトピック一覧表示するページを作しましょう。

1-17. FormViewとCreateViewを使う

2019年4月1日 [コメントをする](#)

今回のテーマは「FormViewとCreateViewを使う」です。前回まで、トピックの作成処理はthread/views.pyにtpic_create関数で処理を書きました。今回はフォームで受けたデータを保存する処理をFormViewをCreateViewを使って書いていきましょう。

※本ページは[FormとHTMLレンダリングの関係を理解する](#)まで読まれた方を対象としています。そのためサンプルソースコードが省略されている場合があります。

FormViewを使う

FormViewはTemplateViewを継承したクラスで受け取ったデータの精査の成功と失敗での別々の処理をします。まずはtopic_create関数をFormViewを使って書き直してみましょう。

thread/views.py

```

from django.shortcuts import render, redirect
from django.views.generic import CreateView, FormView
from django.urls import reverse_lazy

from . forms import TopicCreateForm
from . models import Topic

class TopicFormView(FormView):
    template_name = 'thread/create_topic.html'
    form_class = TopicCreateForm
    success_url = reverse_lazy('base:top')

    def form_valid(self, form):
        form.save()
        return super().form_valid(form)

def topic_create(request):
    template_name = 'thread/create_topic.html'
    ctx = {}
    if request.method == 'GET':
        ctx['form'] = TopicCreateForm()
        return render(request, template_name, ctx)

    if request.method == 'POST':
        topic_form = TopicCreateForm(request.POST)
        if topic_form.is_valid():
            topic_form.save()
            return redirect(reverse_lazy('base:top'))
        else:
            ctx['form'] = topic_form
            return render(request, template_name, ctx)

```

thread/urls.pyも書き直します。

thread/urls.py

```

urlpatterns = [
    path('create_topic/', views.TopicFormView.as_view(), name='create_topic'),
    # path('create_topic/', views.topic_create, name='create_topic'),
]

```

これでlocalhost:8080/thread/create_topic/にアクセスしてみましょう。フォームに入力して「作成」ボタンを押すと作成が成功すればTOP画面に遷移するはずです。現段階では作成したトピックが見れないので管理画面で確認しましょう。

トピック作成

タイトル:

お名前:

カテゴリー: ▼

本文:

作成

管理画面で確認

topic を変更

お名前:	<input type="text" value="名無し"/>
タイトル:	<input type="text" value="素晴らしいワンコの世界"/>
本文:	<div>サンプルサンプルサンプルサンプル サンプルサンプルサンプル サンプルサンプル サンプル</div>
カテゴリー:	<div>業界ネタ・憩いの場 ▼   </div>

削除

ここで行っている処理自体は書き直す前の関数と同じ処理です。FormViewはTemplateViewを継承しているのでGETで受けた場合にはtemplate_nameで指定されたテンプレートを表示します。その際にform_classで指定されたフォームを'form'という名前でコンテキストとして渡します。POSTされた際にはform_valid関数が呼ばれデータの精査が行われ、成功すればsuccess_urlに遷移します。もし失敗した場合はエラー情報をフォームに格納して再度template_nameのテンプレートを表示します。今回はform_valid関数をオーバーライドして保存処理を行っています。

CreateViewを使ってみる

さて、thread/views.pyの内容をクラスベースビューの1つであるCreateViewを使って書き直してみましよう。CreateViewはFormViewを継承しているのでもっとシンプルに書くことができますよ。

thread/views.py

```
from django.shortcuts import render, redirect
from django.views.generic import CreateView
from django.urls import reverse_lazy

from . forms import TopicCreateForm
from . models import Topic

class TopicCreateView(CreateView):
    template_name = 'thread/create_topic.html'
    form_class = TopicCreateForm
    model = Topic
    success_url = reverse_lazy('base:top')
```

当然urls.pyも変更します。

thread/urls.py

```
from django.urls import path

from . import views
app_name = 'thread'

urlpatterns = [
    path('create_topic/', views.TopicCreateView.as_view(), name='create_topic'),
    # path('create_topic/', views.topic_create, name='create_topic'),
]
```

CreateViewはGETメソッドで受けた場合にtemplate_nameで指定されたテンプレートを表示します。その際にform_classで指定されたフォームを‘form’という名前でコンテキストとして渡します。POSTでデータを受けた際にはformのデータを精査して正しいデータであれば保存処理を行い、success_urlにリダイレクトします。もし正しいデータでなければ、エラー内容を含んだformを渡してtemplate_nameで指定されたテンプレートを表示する。という処理を行います。この処理は最初に出てきたcreate_topic関数と同じ処理を行っていますよね。

最後に

ちょっと混乱してしまったでしょうか？ Djangoは関数でガリガリ自分で書くことも出来ますが、目的に応じたプリセットクラスを使うことでよりスマートに書くことも出来ます。次回は確認画面がある登録画面を作成していきます。