**共通手順と追加コマンド:**

- `settings.py` **の設定:** データベースの設定、`INSTALLED_APPS` へのアプリケーション追加など、プロジェクト固有の設定を行う。（これがないとエラーが出る）

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'blog-sample', # ← blog-sample を追加
]
```

- `urls.py` **の設定:** URL ルーティングを設定する。URL とビューの対応付けを行う。

```
# デフォルトでインポートされているが、念のため確認
from django.urls import path
from . import views

# URL パターンとビューの対応付け
# 親 URL は http://localhost:8000/ とすると。。。
urlpatterns = [
    # すべてのユーザーがアクセスする固定された URL。
    path('', views.index, name='index'), # http://localhost:8000/ にアクセスした場合、
views.index が呼び出される
    path('hello', views.hello, name='hello'), # http://localhost:8000/hello にアクセ
スした場合、views.hello が呼び出される
    path('redirect', views.redirect_test, name='redirect_test'), #
http://localhost:8000/redirect にアクセスした場合、views.redirect_test が呼び出される

    # ユーザーがアクセスする URL に記事に ID が含まれる場合の URL パターン。動的に生成されるため、記事
の ID によって URL が変化する。
    # <int:article_id> は任意の数字を表し、article_id という引数で名前でビューに渡される。
    path('<int:article_id>/', views.detail, name='detail'), #
http://localhost:8000/[ここに任意の数字]/ にアクセスした場合、views.detail が呼び出される
    path('<int:article_id>/delete', views.delete, name='delete'), #
http://localhost:8000/[ここに任意の数字]/delete にアクセスした場合、views.delete が呼び出され
る
    path('<int:article_id>/update', views.update, name='update'), #
http://localhost:8000/[ここに任意の数字]/update にアクセスした場合、views.update が呼び出され
る
    path('<int:article_id>/like', views.like, name='like'), #
http://localhost:8000/[ここに任意の数字]/like にアクセスした場合、views.like が呼び出される


    # API 用。JS で fetch 関数などを用いてリクエストを送信するための URL。blog-sample では like
テーブルにいいね情報を保存するために使用する
    path('api/articles/<int:article_id>/like', views.api_like), #
```

```
http://localhost:8000/api/articles/[ここに任意の数字]/like にアクセスした場合、
views.api_like が呼び出される
]
```

- `models.py` **の作成:** アプリケーションで使用するデータモデルを定義する。

```python
# デフォルトでインポートされているが、念のため確認
from django.db import models

# このモジュールは、Django のタイムゾーン関連機能を提供する。自分でインポート必要な可能性あり。
from django.utils import timezone

# モデルの定義。この場合、SQL で表すと以下のテーブル構造となっている。
"""
CREATE TABLE AIrticle (
    id INT PRIMARY KEY AUTO_INCREMENT,
    title VARCHAR(200),
    body TEXT,
    posted_at DATETIME,
    published_at DATETIME,
    like INT
);
"""

# そして、日本語で書くと以下のようになる。

# id: 自動生成されるキー、主キー、auto_increment（自動増分）、整数型(int)、一意性（unique）
# title: 短い文字列型（Char）、長さ200文字まで
# body: 長い文字列型（Text）
# posted_at: 日時型（DateTime）、デフォルト値は現在時刻（timezone.now）
# published_at: 日時型（DateTime）、空白可能（blank=True）、null可能（null=True）
# like: 整数型（Int）、デフォルト値は0

class Article(models.Model):
    title = models.CharField(max_length=200)
    body = models.TextField()
    posted_at = models.DateTimeField(default=timezone.now)
    published_at = models.DateTimeField(blank=True, null=True)
    like = models.IntegerField(default=0)

    # 記事を公開した場合に呼び出されるメソッド。
    # 実行時に現在時刻を published_at に設定し、保存する。
    def publish(self):
        self.published_at = timezone.now()
        self.save()

    # この class を用いた変数を str 型に変換した際に、title を返すように設定。
    def __str__(self):
        return self.title

# このモデルは、Article モデルとの関連を持つ。
# Article モデルに対して、1対多の関係を持つ。
```

```
# 1つの記事に対して、複数のコメントが投稿されることが可能。
# on_delete=models.CASCADE は、Article モデルが削除された場合、それに関連する Comment モデルも
削除されることを意味する。

# SQL で表すと以下のようなテーブル構造となっている。
"""
CREATE TABLE Comment (
    id INT PRIMARY KEY AUTO_INCREMENT,
    text TEXT,
    posted_at DATETIME,
    article_id INT,
    FOREIGN KEY (article_id) REFERENCES Article(id) ON DELETE CASCADE
);
"""

# そして、日本語で書くと以下のようになる。

# id: 自動生成されるキー、主キー、auto_increment（自動増分）、整数型(int)、一意性（unique）
# text: 長い文字列型（Text）
# posted_at: 日時型（DateTime）、デフォルト値は現在時刻（timezone.now）
# article: Article モデルとの関連付け（ForeignKey）、Article モデルの id と関連付け

class Comment(models.Model):
    text = models.TextField()
    posted_at = models.DateTimeField(default=timezone.now)
    article = models.ForeignKey(Article, related_name='comments',
on_delete=models.CASCADE)
```

- `views.py` **の作成:** ユーザーからのリクエストを処理するビューを定義する。

```
# デフォルトでインポートされているが、念のため確認
from django.shortcuts import render, get_object_or_404, redirect
from django.http import HttpResponse
from .models import Article, Comment

# ビューの定義。この場合、以下の関数が定義されている。
# index: 記事の一覧を表示する。
# detail: 記事の詳細を表示する。
# hello: 簡単な挨拶を表示する。
# redirect_test: リダイレクトのテストを行う。
# like: 記事に「いいね」を追加する。
# api_like: API を通じて記事に「いいね」を追加する。
# 記事の一覧を表示するビュー関数

def index(request):
    # Article モデルからすべての記事を取得
    articles = Article.objects.all()
    # 'index.html' テンプレートに記事一覧を渡してレンダリング
    return render(request, 'index.html', {'articles': articles})

# 記事の詳細を表示するビュー関数
def detail(request, article_id):
```

```
    # 指定された article_id に対応する記事を取得、存在しない場合は 404 エラーを返す
    article = get_object_or_404(Article, pk=article_id)
    # 'detail.html' テンプレートに記事を渡してレンダリング
    return render(request, 'detail.html', {'article': article})

# 簡単な挨拶を表示するビュー関数
def hello(request):
    # "Hello, world!" というテキストを返す
    return HttpResponse("Hello, world!")

# リダイレクトのテストを行うビュー関数
def redirect_test(request):
    # 'index' ビューにリダイレクト
    return redirect('index')

# 記事に「いいね」を追加するビュー関数
def like(request, article_id):
    # 指定された article_id に対応する記事を取得、存在しない場合は 404 エラーを返す
    article = get_object_or_404(Article, pk=article_id)
    # 記事の「いいね」数を1増やす
    article.like += 1
    # 変更を保存
    article.save()
    # 'detail' ビュー（関数）にリダイレクト
    return redirect(detail, article_id=article_id)

# API を通じて記事に「いいね」を追加するビュー関数
def api_like(request, article_id):
    # 指定された article_id に対応する記事を取得、存在しない場合は 404 エラーを返す
    article = get_object_or_404(Article, pk=article_id)
    # 記事の「いいね」数を1増やす
    article.like += 1
    # 変更を保存
    article.save()
    # ステータスコード 204 (No Content) を返す
    return HttpResponse(status=204)
```

- **テンプレートの作成:** HTML テンプレートを作成する。ビューから渡されたデータを表示するために使用する。
    - **テンプレートの配置:** テンプレートファイルは、通常 `templates` ディレクトリに配置する。アプリケーションごとにディレクトリを分けると管理しやすい。
    - **テンプレートの例:** 以下は、記事の一覧を表示するための `index.html` テンプレートの例。

```
<!DOCTYPE html>
<html>
  <head>
    <title>記事一覧</title>
  </head>
  <body>
    <h1>記事一覧</h1>
    <ul>
```

```
      {% for article in articles %}
      <li>
        <a href="{% url 'detail' article.id %}">{{ article.title }}</a>
        <p>{{ article.body|truncatewords:30 }}</p>
        <p>いいね: {{ article.like }}</p>
      </li>
      {% endfor %}
    </ul>
  </body>
</html>
```

- **テンプレートタグ:** Django テンプレートエンジンは、`{% %}` や `{{ }}` を使用して、テンプレートタグや変数を埋め込むことができる。
- **テンプレートの継承:** テンプレートの共通部分を継承することで、コードの重複を避けることができる。例えば、共通のヘッダーやフッターを `base.html` に定義し、他のテンプレートで継承する。

```
<!-- base.html -->
<!DOCTYPE html>
<html>
  <head>
    <title>{% block title %}My Site{% endblock %}</title>
  </head>
  <body>
    <header>
      <h1>My Site</h1>
    </header>
    <main>{% block content %}{% endblock %}</main>
    <footer>
      <p>&copy; 2023 My Site</p>
    </footer>
  </body>
</html>

<!-- index.html -->
{% extends 'base.html' %} {% block title %}記事一覧{% endblock %} {% block
content %}
<h1>記事一覧</h1>
<ul>
  {% for article in articles %}
  <li>
    <a href="{% url 'detail' article.id %}">{{ article.title }}</a>
    <p>{{ article.body|truncatewords:30 }}</p>
    <p>いいね: {{ article.like }}</p>
  </li>
  {% endfor %}
</ul>
{% endblock %}
```

**その他の有用なコマンド:**

- `python manage.py createsuperuser`: 管理ユーザーの作成
- `python manage.py makemigrations`: モデルの変更を反映するためのマイグレーションファイルの作成
- `python manage.py collectstatic`: 静的ファイル (CSS, JavaScript, 画像など) の収集
- `python manage.py test`: テストの実行