

Djangoチュートリアルで学ぶ Web開発入門

by DjangoBrothers

第3版

はじめに

本書の特徴

本書の特徴は大きく2つあります。

- チュートリアル形式の実践的な流れでWeb開発の基本を学ぶ
- 人気のWebフレームワークDjangoの数少ない日本語解説書籍

プログラミングの知識は、実際に自分で手を動かして開発していく過程で理解が深まります。そのため本書では、実際に動くWebサービスを作りながら、技術や知識が必要になったタイミングで解説をするという形式をとります。これによって、なぜその機能が必要なのか、どうやったらその機能を実現できるのかなどの知識を学習することができます。

また、本書ではDjangoというWebフレームワークをもとにWebサービスを作っていきます。Djangoは世界的に非常に人気のWebフレームワークなのですが、残念ながら日本語の情報・教材はあまり多くないというのが現状です。Djangoの入門的知識を解説している日本語書籍は少ないため、皆さんの学習の一助になれば幸いです。

なお、本書の最後にアンケートを設置しておりますので、ご協力いただける方はご回答をお願いいたします。

対象読者

本書の対象読者は以下のように想定しています。

- HTML、CSS、Pythonの基礎文法は学習済みで、Web開発の基礎を学びたい方
- Web開発の経験があり、Djangoに入門したい方
- プログラミング学習サービスやDjangoの公式チュートリアルからステップアップしたい方

プログラミングの基礎を学習した方を対象に、Djangoを使ったWebサービスの開発について、解説をじっくり進めています。

また、自分で使っているOS環境の基本的なコマンドを理解していることを前提にしています。本書ではMacOS環境での説明を行います。とはいっても、フォルダやファイルを作成したり、ワーキングディレクトリを移動したりといった簡単なコマンドしか扱いません。

動作環境

本書は以下の動作環境に準拠しています。

Python3.6 / Django2.2.5

実際にチュートリアルを進める際にはこちらの環境に合わせることを推奨します。

目次

- はじめに
 - 本書の特徴
 - 対象読者
 - 動作環境
- 目次
- 第1章 Djangoの概要
 - 1.1 Djangoとは
 - 1.2 Webフレームワークの役割
- 第2章 Web開発の前提知識とプロジェクトの初期設定
 - 2.1 Web開発の前提知識
 - 2.2 開発環境の設定
 - 2.3 Djangoプロジェクトの初期設定
- 第3章 チュートリアル1 ブログサイト
 - 3.1 アプリケーションを作ろう
 - 3.2 トップページを作ろう
 - 3.3 ViewとURLを紐づける
 - 3.4 ブログモデルの作成
 - 3.5 Adminページを作ろう
 - 3.6 トップページに記事を表示しよう
 - 3.7 記事詳細ページを作ろう
 - 3.8 CRUDを理解しよう
 - 3.9 記事の削除と更新を実装しよう
- 第4章 チュートリアル2 写真投稿サイト
 - 4.1 開発準備をしよう
 - 4.2 Template拡張を利用する
 - 4.3 Django標準のUserモデルを使う
 - 4.4 ImageFieldで画像をアップロードしよう
 - 4.5 ForeignKeyでモデル同士を紐づける
 - 4.6 ユーザー認証機能を作ろう
 - 4.7 ユーザー登録機能を作ろう
 - 4.8 写真投稿機能と削除機能の実装
 - 4.9 includeテンプレートタグでリファクタリング
 - 4.10 カテゴリーで絞り込む
- 第5章 チュートリアル3 ECサイト
 - 5.1 カスタムユーザーモデルを作ろう
 - 5.2 ログイン/サインアップ処理を作ろう
 - 5.3 必要なモデルを作ろう
 - 5.5 ManyToManyフィールドで多対多の関係を作る
 - 5.4 商品をページに表示しよう
 - 5.6 セッションを使ってカートに商品を追加しよう
 - 5.7 カートページから決済する
 - 5.8 注文履歴ページを作ろう

第1章 Djangoの概要

1.1 Djangoとは

Djangoとは、人気のプログラミング言語PythonをベースにしたWebフレームワークです。

Webフレームワークとは、「Webサービスでよく使われる機能を簡単に実装できるようにしたツール」のことです。Djangoだけでなく様々なフレームワークが存在します。

PythonベースのWebフレームワークだと、「Flask」や「Bottle」などがあり、別のプログラミング言語のRubyベースだと「Ruby on Rails」などが人気です。

そんな数あるWebフレームワークの中でもDjangoをおすすめできる理由は主に以下の通りです。

- 人気の汎用プログラミング言語Python製
- フルスタックフレームワーク
- Pythonベースのフレームワークで最も人気がある

DjangoはWeb開発以外にも汎用的に使えるPythonをベースにしているので、Pythonを利用したことのある方には非常にとっつきやすく、Pythonはプログラミングの入門用の言語としても人気のため、利用した経験のない方にとってもハードルが低いです。

また、DjangoはフルスタックWebフレームワークと言われています。これは、Webサービスを作る際に必要な基本的な機能はほとんど全てDjangoの機能だけで済んでしまうという意味です。例えば、Webサービスでよく必要となる会員登録・ログイン機能、管理ユーザー用画面、データベース操作などが全てDjangoの一機能として提供されています。そのため、「これさえ覚えれば一通りのWeb開発には困らない」とも言えるくらいの威力を発揮してくれるフレームワークです。（当然、「フロントエンド開発やインフラ周りの知識なども状況に応じて必要にはなりますが）

さらに、PythonベースのWebフレームワークの中で最も歴史が長く、人気のあるフレームワークなので、情報の多さや、バグの少なさ、セキュリティの堅牢さという面でもメリットがあります。

「Web開発初心者にもハードルが低く、業務レベルでもしっかり使える」という点でおすすめできるWebフレームワークです。

1.2 Webフレームワークの役割

Webフレームワークとは、「Webサービスでよく使われる機能を簡単に実装できるようにしたツール」と説明しました。フレームワークを使わない場合と比較して圧倒的に短期間で、スケーラブルなWebサービスを堅牢に作ることができます。そのためWebフレームワークは、普段みなさんがよく利用しているサービスの開発にも広く使われています。

また、副次的な効果ですが、Webフレームワークは仕様が決まっているので、共通のフレームワークの知識を持ったエンジニアであればスムーズにプロジェクトにジョインすることができます。そういう意味でも、より多くのエンジニアに指示されているWebフレームワークのスキルを身につけておくのは大きなアドバンテージになります。

テンプレートで、よく使うページを使いまわすことができる

チュートリアル内で詳しく解説しますが、Webフレームワークでは「テンプレート」という概念が使われることが多いです。普段みなさんが利用しているWebサービスでは、ページの内容は異なっていても、ヘッダーやフッターの部分は全く同じ構成を取っているサービスが多いと思います。こういった、各ページで共通している部分を切り出して、個別のページから呼び出すことで、冗長な実装を省くことができます。

また、テンプレートを利用すると、複数のページで共通している部分を修正する際に、1つのファイルを修正するだけでよくなりるので、メンテナビリティや開発速度が大幅に向上します。

データベースからのデータの取得や操作ができる

通常、データベースからのデータの取得は「SQL」と呼ばれるデータベースを操作する言語を利用し、その後にアプリ側の実装と連携する必要があります。また、データベースにはいくつか種類があり、その種類ごとに操作の方法や実現できることが異なっています。

Webフレームワークでは、こういったデータベースの操作に対して統一されたAPIを提供していることが多く、「異なるデータベースに対して同じ操作で同じことを実現できる」ようになっています。

また、オブジェクト指向のフレームワークでは、データの操作もアプリ側で直感的に記述できるようになっており、モデルとデータベースの連携も簡単にできるようになっています。

セキュリティを堅牢に保ちやすくなる

DjangoなどのWebフレームワークでは、データを入力するフォームの処理やユーザーの認証周りなどの実装時に、デフォルトでセキュリティの対策が行われるようになっています。例えば、ユーザー登録をする時などは、利用者は個人情報を送信することになるので、セキュリティ周りにも当然気をつけなければなりません。そういった実装は当たり前に必要になるものではあるものの、毎回実装するのは面倒ですし、実装し忘れたりすると大変なことになってしまいます。

こういった「絶対に必要だけど、実装が面倒な部分」を大幅に楽にしてくれる点はWebフレームワークの大きな魅力です。

このように、Webサービスに必要不可欠な様々な機能の開発を高速化する工夫が随所に盛り込まれています。

Djangoのようなフレームワークを使いこなすことで、堅牢でスケーラブルなWebサービスを短期間でつくることができるようになります。

そんなDjangoの開発について学ぶために、次章では開発環境と基本的な設定を整えていきましょう。

第2章 Web開発の前提知識とプロジェクトの初期設定

この章では、Web開発に必要となる前提知識について説明します。Djangoの開発に必要な開発環境や、今後の各チュートリアルに共通で必要になる初期設定を行います。

この章で行う初期設定の説明は、各チュートリアルの章では割愛することになります。初期設定を進める場合にはこの章に戻って確認してください。

2.1 Web開発の前提知識

この節では、今後のWeb開発を始める前提として必要な知識を簡単に解説します。

1. HTML

Djangoは「Webフレームワーク」ですので、Webブラウザ上に表示するページをたくさん作ることになります。そのページを作るのに使われるのがHTMLという言語です。

HTMLは**Hyper Text Markup Language**の略で、Webページの骨組みを作るための言語になります。みなさんが普段ブラウザでアクセスしているWebサイトでは基本的に全てHTMLをベースにして文字や画像が表示されています。

この本で紹介しているWebサービスを作るためには、`<h1>`、`<p>`、`<a>` タグなど基礎的なタグが理解できていれば十分です。HTMLを全く触ったことがない人は、オンライン学習サイトや参考書で、ある程度の基礎を身につけておきましょう。

2. CSS

CSSは、**Cascading Style Sheets**の略で、HTMLで表示された文字や画像をデザインするものです。文字の大きさや色、要素の配置を変えたりするために使われます。

CSSはデザインのための言語ですので、極論を言えばWebページの表示には必ずしも必要のないものです。

ただ、Webサイトを作る上で基礎的なデザインの知識は重要となりますので、HTMLと一緒に一通りの知識は学習しておいてください。

3. Python

DjangoはPythonをベースに作られたWebフレームワークなので、Djangoのコードを書くときはPythonの文法に従って書いていくことになります。

Pythonは世界的に非常に高い人気を誇るプログラミング言語で、近年需要が高まっている機械学習や自然言語処理、数値計算などの分野でも活躍する言語です。もちろん、Web系の分野にも強く、誰もが知っている有名なWebサービスもDjangoで実装されていました。シンプルな文法で学習しやすいので、プログラミング入門者が初めて学ぶ言語としてもよく利用されます。

この本を進める上では、Pythonの知識が必須となります。特別難しい知識は必要ありません。入門書のレベルが身についていれば大丈夫です。if文、for文、リスト、関数の作成、オブジェクト指向(クラスやメソッド)についての基礎知識があれば大丈夫です。

Pythonの基礎知識については、DjangoBrothersのブログ記事でもまとめているので、自信のない方は事前に一読しておくことをおすすめします。

- [【Python】オブジェクト指向を理解するための超重要ワードまとめ](#)

4. コマンドライン

コマンドラインはプログラミングをする上で必須の知識です。

皆さんがあなたが普段パソコンを操作するときには、マウスでカーソルを動かして、指定の場所でクリックすることで画面を操作します。

コマンドラインはそのような操作を、文字を打ち込むことで実現する仕組みです。

文字で操作を指示するよりも、マウスで操作した方が楽だと思われるかもしれません。実際にプログラミングをしてみると、コマンドラインの利点が見えてきます。

コマンドで動作を指示できるようになると、多少複雑な操作や、人力では非常に労力がかかるような操作もまとめてコンピュータに任せることができます。

コマンドラインに関しても、多くを学ぶには専門書が1冊必要ですが、今回のチュートリアルで必要になる最低限のレベルの知識はこちで紹介しておきます。

なお、「対象読者」でも述べたように、この本で使うコマンドは全てMacOSの操作方法ですので、Windowsなどの別のOSでの操作は適宜読み替えてください。「Mac Windows コマンド 対応表」のようなキーワードで検索すると、OSごとのコマンドについて説明している記事が出てくるので参考にできるでしょう。

コマンドラインの基礎

MacOSの場合は「ターミナル」というアプリケーションを起動します。

うまく起動できていたら、真っ白または真っ黒の画面が出てきていると思います。

画面には\$(ドルマーク)が表示されていると思いますが、このマークの後に文字列(コマンド)を入力していくことでコンピュータを操作していきます。

\$の前に色々文字が表示されているかもしれません。この部分は個人の設定によって変わるものなので、このチュートリアルでは\$のみを表記します。

それでは、さっそく初めてのコマンドを打ってみましょう!

\$の後に `pwd` と入力し、Enter を押してください。

```
$ pwd  
/Users/user_name
```

`/Users/user_name` のような文字が表示されましたか? `user_name` のところはあなたのユーザー名になっているかもしれませんし、もしかしたらちょっと違う文字が表示されているかもしれません。

`pwd` というのはPrint Working Directoryの略で、現在作業しているディレクトリ(カレントディレクトリ)を表示してくれるコマンドです。

次に、`ls` というコマンドを打ちます。これはlistの略で、カレントディレクトリの中に保存されているディレクトリを一覧として表示するコマンドです。

```
$ ls
Applications Music
Desktop Documents
Downloads Library
Movies Pictures
Projects Public
```

このようにたくさんのディレクトリがリストとして表示されます。表示される内容は個人の環境によって違うので、一致していなくても大丈夫です。

今度は、いま表示したリストの中にある「Desktop」ディレクトリに移動してみましょう。移動したい時は`cd` コマンド(Change Directory)を使います。なお、これ以降は`(~/)` のように括弧内にカレントディレクトリやファイル名を表記するようにします。

`(~/)`

```
cd Desktop
```

これで、「Desktop」ディレクトリに移動できているはずなので、`pwd` コマンドで確認してみましょう。

`(~/Desktop/)`

```
$ pwd
/Users/user_name/Desktop
```

カレントディレクトリが「Desktop」になっていますね!このように表示されれば成功です。

ちなみに、`cd ..` というコマンドで1つ上の階層のディレクトリに移動することができます。`..` は、1つ上の階層を表すもので、`cd .../..` と書けば、2つ上の階層まで移動できます。

`(~/Desktop/)`

```
$ cd ..
$ pwd
/Users/user_name/
```

ここまでで、ディレクトリの表示や移動に関するコマンドを紹介しました。

続いては、ディレクトリを新たに作成するためのコマンドです。新しいディレクトリをデスクトップに作ってみましょう。

まずは、`cd` コマンドで Desktop ディレクトリに移動します。

(~/)

```
$ cd Desktop
```

デスクトップに移動できたら、`mkdir` コマンドを使って「DjangoBros」という名前のディレクトリを作ってみます。`mkdir` は、**Make Directory** のことで新しいディレクトリを作成するコマンドです。

(~/Desktop/)

```
$ mkdir DjangoBros
```

これでデスクトップに、「DjangoBros」というディレクトリが作成されました。自分のデスクトップを見て確認してみましょう。`ls` コマンドでも確認できます。

次は、このディレクトリの中にHTMLファイルを作ってみましょう。ファイルを作るコマンドは `touch` です。

`cd` コマンドで「DjangoBros」ディレクトリに移動してから、ファイルを作ります。

(~/Desktop/)

```
$ cd DjangoBros  
$ touch sample.html
```

`touch` コマンドで「sample.html」というhtmlファイルを作ることができました。CSSファイルや、Pythonファイルを作る時も同様に `touch` コマンドを使います。

これで基本的なコマンドの紹介は終わりです。最後に、今回作ったディレクトリは練習用ですので一旦削除しておきましょう。`rm -r` コマンドを使えばディレクトリを削除することができます。ファイルを削除したいときは `rm` コマンドです。

削除するディレクトリの1つ上の階層/Desktop)まで移動して `rm -r` コマンドを打ってください。

(~/Desktop/)

```
$ rm -r DjangoBros
```

これで、Desktopから DjangoBros ディレクトリが削除されていると思います。

ここまでで、コマンドラインの基本を紹介しました。今後チュートリアルを進めていく中で、わからないコマンドが出てきたらこちらに戻って確認してみてください。

2.2 開発環境の設定

続いて、Djangoで開発を進めるために必要な環境を構築します。

Djangoで開発をするためには、お使いのパソコンにPythonとDjangoがインストールされている必要があります。本書のチュートリアルでは、Python3.7とDjango2.2を使って開発します。3.7や2.2という数字はバージョンを表しています。

Pythonのインストール

まだパソコンにPython3.7がインストールされていなければインストールしましょう。「ターミナル」アプリを開いて、`python3 --version` コマンドを実行してください。

```
$ python3 --version
Python 3.7.3
```

このように、`Python 3.7.-` と表示されれば必要なバージョンのPythonはインストールされています。最後の一桁はどの数字でも大丈夫です。

これが表示されない場合はインストールをしましょう。

まずは、Pythonの公式ページ(<https://www.python.org/downloads/>)の「Download」を押してダウンロードしてください。

次に、ダウンロードしたファイルをクリックして開き、インストールを実行してください。

ここまで完了したら、もう一度先ほどのコマンドを打ち、`Python 3.7.-` と表示されることを確認してください。

Pythonを使ってみよう

Python をインストールできたら、少しだけ使ってみましょう!

コマンドラインで`python3` というコマンドを打つといくつか文字が出てきた後に`>>>` という文字が表示されるはずです。ここにPythonのコードを書くことができます。

```
$ python3
...
>>>
```

まずは簡単な計算をしてみましょう。`4 + 5` と入力してEnterキーを押してください。

```
$ python3
...

```

```
>>> 4 + 5
```

```
9
```

計算結果が返ってきましたね。Pythonのコードですので、もちろん掛け算や割り算、剰余の計算結果も調べることができますし、文字列を出力することも可能です。

```
>>> 3 * 5  
15  
>>> 10 % 3  
1  
>>> print('DjangoBrothers')  
DjangoBrothers
```

うまくできましたか?うまくできたならいいのですが、残念ながらプログラミングをしていく上では、自分の思った通り動かずに入力がでてしまうことがあります。そんな時は、落ち着いてエラーメッセージを読むようにしましょう。上手くいかない原因は、エラーメッセージに隠されています。

試しにエラーメッセージを表示させるため、`print(color)`と打ってEnterを押してみてください。

```
>>> print(color)  
Traceback (most recent call last):  
File "", line 1, in  
NameError: name 'color' is not defined  
>>>
```

このようなエラーが出たと思います。1番下の行に「NameError」とメッセージが表示されていますね。メッセージを見ると `name 'color' is not defined` と言っています。これは変数名「color」が定義されていないので、出力することができませんよという意味です。このエラーを解消するためには、変数「color」に何か文字列などを代入してあげれば良いことがわかりました。

```
>>> color = "Red"  
>>> print(color)  
Red
```

このように変数「color」を定義することでエラーが解消されました!

Djangoでの開発もそうですが、うまくいかないときはエラーメッセージを読んで、エラーを解消しながら開発をしていきます。エラーメッセージの意味がわからないときは、検索して調べたり、プログラミングの先輩に相談してみたりしてください。あなたがこれから遭遇するかもしれないエラーのほとんどは、既に他の人が経験済みのものですので、先輩に頼るのが習得への1番の近道です。

では、一旦Pythonの入力モードを閉じましょう。`quit()` コマンドで終了します。

```
>>> quit()
$
```

仮想環境を作ろう

続いて、開発を進めるにあたって必要となる**仮想環境**というものを作ります。

通常であればパソコンごとにPythonの開発環境は1つしか作れませんが、仮想環境を利用することで複数の開発環境を共存させることができます。

たとえば、Pythonを使って複数のプロジェクトを同時進行で開発する場合などに、プロジェクトAではPython2を使用、プロジェクトBではPython3を使用したいことがあります。こういった場合、各プロジェクトに応じてそれぞれの開発環境を用意し、環境を切り替えながら開発していくのが一般的です。この開発用に作るそれぞれの環境を、「仮想環境」と呼びます。仮想環境を複数作ることで、プロジェクトに応じて開発環境の切り替えができるようになります。

また、開発を進めていく中で、コンピュータに様々なライブラリ(便利な機能を簡単に使えるようにしたプログラムのこと)をインストールすることがあるのですが、そのライブラリを必要な仮想環境にだけインストールすることで、全てのプロジェクトに影響を及ぼすことを防ぐこともできます。

それでは、実際に仮想環境を作っていきます。

仮想環境を作る前にまずは、このチュートリアル用にディレクトリを作りましょう。ディレクトリを作る場所はデスクトップ等、どこでも構いません。

```
$ mkdir DjangoBros
$ cd DjangoBros
$ pwd
~/DjangoBros
```

「DjangoBros」というディレクトリを作ってそこに移動しました。現在のカレントディレクトリは「DjangoBros」です。現時点では、この中に何にもディレクトリやファイルを作成していないので、ここで `ls` コマンドを実行しても何も表示されないことを確認してください。

それでは、このディレクトリの中に仮想環境を作ります。

仮想環境を作る方法はいくつかありますが、今回は `python3 -m venv` コマンドを使います。

コマンドラインに `python3 -m venv djangobros_venv` と入力してください。

これは、Python3を使って「djangobros_venv」という名前の仮想環境を作る、という意味のコマンドです。「djangobros_venv」の部分は自分の好きな名前を指定しても大丈夫です。ちなみに `venv` は、**Virtual Environment(仮想環境)** の略です。

(~/DjangoBros/)

```
$ python3 -m venv djangobros_venv
```

ここでもう一度 `ls` コマンドを実行すると、「`djangobros_venv`」というディレクトリができていることが確認できるはずです。

これで DjangoBrothers用の環境を作ることができました!この環境に対してDjangoをインストールしていきます。

Djangoをインストールしよう

今は、環境を作っただけですので、まずはこの環境の中に入る必要があります。以下コマンドで環境の中に入るることができます。

このコマンドは、仮想環境(`djangobros_venv`)がある、1つ上の階層で実行してください。今回の場合だと、「`DjangoBros`」ディレクトリをカレントディレクトリにした状態で実行してください。

(~/DjangoBros/)

```
$ source djangobros_venv/bin/activate  
(djangobros_venv) $
```

コマンドがうまく実行されると、\$の前に `(djangobros_venv)` といった文字が表示されます。これが「仮想環境の中にいる」状態を表しています。

この状態でDjangoをインストールします。

Djangoをインストールするためには、`pip`というパッケージ管理システムを使います。パッケージ管理システムとは、ライブラリやパッケージを簡単にインストールしたり、バージョン管理したりできるツールのことです。`pip`は、Pythonのパッケージ管理システムです。

まずは、`pip`を最新のものにアップグレードします。

```
(djangobros_venv) $ pip install --upgrade pip
```

次に`pip`を使ってDjangoをインストールします。`pip install django` コマンドで、最新のバージョンの Djangoをインストールすることができます。

```
(djangobros_venv) $ pip install django
```

また、下のように `==` を使えば、バージョンを指定してDjangoをインストールすることができます。今回は Djangoのバージョンは2.2を使いたいので以下コマンドでインストールしてください。

```
(djangobros_venv) $ pip install django==2.2.5
```

インストールしたライブラリの一覧は `pip freeze` コマンドで確認できます。

```
(djangobros_venv) $ pip freeze
Django==2.2.5
pytz==2020.4
sqlparse==0.4.1
```

`python` コマンドでPythonのコマンドプロンプトを起動することでも、Djangoがインストールされていることを確認できます。以下の手順でコマンドを打つことで確認できます。

```
(djangobros_venv) $ python
>>> import django
>>> django.get_version()
'2.2.5'
```

2.2.5が現在使っているDjangoのバージョンです。確認できたら `quit()` と打ってコマンドプロンプトを終了します。

```
>>> quit()
(djangobros_venv) $
```

これで環境構築は完了です!

現在いる仮想環境から抜け出したい場合は、`deactivate` コマンドを使います。

`deactivate` コマンドにより、`(djangobros_venv)` の表示が消えていれば、ちゃんと仮想環境から抜け出せています。仮想環境から抜け出した状態だと、`pip freeze` としても `Django==2.2.5` が表示されないと思います。

Djangoは仮想環境内にインストールしていますので、当然仮想環境から抜け出した状態では Djangoを使うことができません。開発する際は必ず仮想環境内に入ってください。

2.3 Djangoプロジェクトの初期設定

開発環境が整ったところで、いよいよ Djangoで Webサービスを作っていきましょう!

まずは、Djangoのプロジェクトを実際に立ち上げて、初期設定を行います。なお、ここで実行する初期設定は **これ以降の各チュートリアルで共通のスタート地点**になるので、わからなくなったらこちらを見直してください。

プロジェクトとアプリケーション

いよいよDjangoでの開発を進めていきましょう。
まずはDjangoにおける基本的な概念の説明から始めます。

Djangoでの開発では、「プロジェクト」と「アプリケーション」と呼ばれる2つのディレクトリを管理することになります。

まず、「アプリケーション」とは、**Webサービスに必要な機能を実現するための部分**のことを指します。例えば、「ブログをWebページに表示させる機能」や「自分の好きなブログをお気に入り登録する機能」といった、なにか特定の役割を持つ機能を担っている部分のことです。

そして「プロジェクト」とは、**作成しているWebサービス全体に関する設定やそのWebサービスの各機能を構成する「アプリケーション」を1つにまとめたもの**です。基本的に、DjangoでWebサービスを1つ作る場合には、全体をまとめる「プロジェクト」が1つと、そのプロジェクトに対応する複数の「アプリケーション」で構成されることになります。

実際に作ってみないと中々イメージしづらいと思いますが、チュートリアルを進める中で理解できてくるはずです。

基本的な概念が分かったところで、さっそくDjangoのプロジェクトを作成してみましょう。

まずは、2.2で作った「DjangoBros」ディレクトリに移動して、仮想環境を有効にしてください。

(~/DjangoBros)

```
$ source djangobros_venv/bin/activate  
(djangobros_venv) $
```

今回は、第3章で学ぶチュートリアル①に合わせて、「django_blog」という名前のプロジェクトを作ります。カレントディレクトリが「DjangoBros」であることを確認して、以下のコマンドを実行してください。

なお、今後の各チュートリアルでは、このプロジェクトの名前(**django_blog**)を各チュートリアルで指定された名前に置き換えてプロジェクト作成を行ってください。

(~/DjangoBros/)

```
$ django-admin startproject django_blog
```

これで「django_blog」という名前のプロジェクトが作成されました。1つ下の階層に作成された「django_blog」という名前のディレクトリに移動して、`ls`コマンドでその中のディレクトリ構成を確認してみてください。

(~/DjangoBros/)

```
$ cd django_blog  
$ ls  
django_blog  
manage.py
```

基本的なディレクトリ構成を確認しよう

ディレクトリを確認してみると、以下のような構成になっています。

(~/DjangoBros/django_blog/)

```
django_blog  
manage.py
```

プロジェクト名と同じ `django_blog` というディレクトリと、`manage.py` というファイルがありますね。`.py` の拡張子がついたファイルは、Python形式で書かれたファイルであることを示しています。

まず、`manage.py` ですが、こちらはその名の通り Django プロジェクトを manage[管理・運営]する時に使うファイルです。サーバーを立ち上げたり、プロジェクトの管理者情報を作成したりする時に使います。こちらのファイルの中身を編集することは基本的にありません。

`django_blog` というディレクトリはこのプロジェクトの Python パッケージです。中には 4 つのファイルが含まれています。

- `__init__.py`: django_blog が Python パッケージであることを示すための空ファイルです。
- `settings.py`: Django プロジェクトの設定ファイルです。今後様々な場面で利用します。
- `urls.py`: ウェブサイトの各ページの URL を定義するファイルです。
- `wsgi.py`: サーバーの設定などを行うファイルです。

少し難しく感じるかもしれません、今後必要な場面で説明を加えるので、現時点では各ファイルの役割を覚える必要はありません。

Django の初期ページを表示しよう

どんなプロジェクトでも、まずは Django の 初期ページ を表示することから始めるのが良いでしょう。

Django には、ローカル環境(自分のパソコン)で簡単にサーバーを立てることができる機能が備わっているので、すぐにプロジェクトの内容を確認することができます。

はじめに、プロジェクトのルートディレクトリに移動してください。ルートディレクトリとは、1 番上の階層という意味です。現在、DjangoBros ディレクトリの中に django_blog というプロジェクトを作っていますので、(DjangoBros/django_blog) がプロジェクトのルートディレクトリとなります。

(~/DjangoBros/django_blog/)

```
$ pwd  
~/DjangoBros/django_blog
```

ルートディレクトリに移動したら、`ls` コマンドで 1 つ下の階層に `manage.py` ファイルがあることを確認した上で、`python manage.py runserver` コマンドを打ってください。これはローカルサーバーを起動するコマンドです。

(~/DjangoBros/django_blog/)

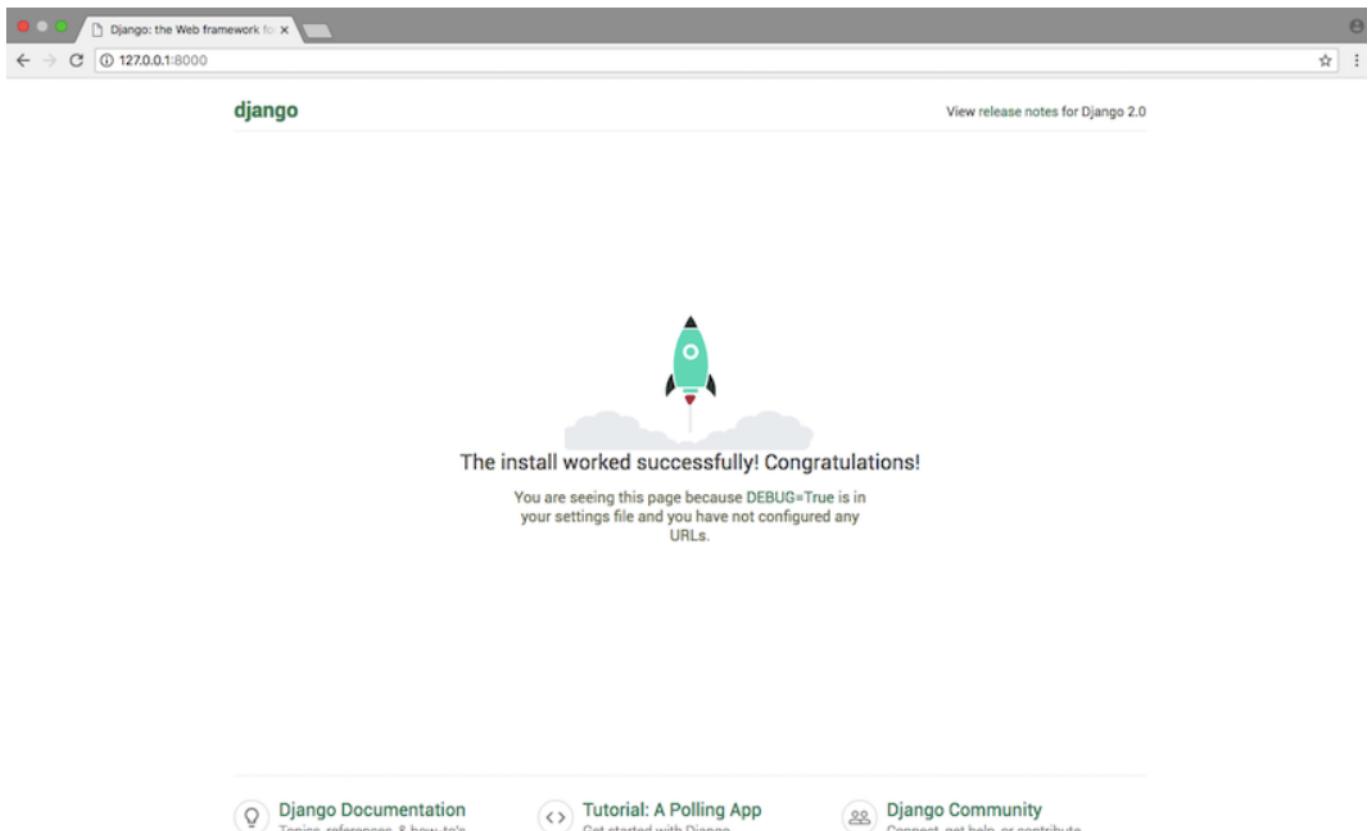
```
$ ls
django_blog manage.py
$ python manage.py runserver
```

今後、`python manage.py ~` というコマンドを頻繁に使うことになりますが、このコマンドは `manage.py` ファイルの1つ上の階層にいなければ使えませんので注意してください。

`python manage.py runserver` コマンドにより、コンピュータ内でそのDjangoプロジェクトのサーバーが立ち上がります。最初は、ターミナルに「マイグレートしてください」という内容のメッセージが英語で表示されると思いますが、一旦気にしなくて大丈夫です。

この状態で、`http://127.0.0.1:8000/` というURLにアクセスすると、Djangoデフォルトの初期ページが表示されると思います。このURLはローカル環境を表しており、Djangoではデフォルトで8000番のポートを利用するようになっています。

(Djangoプロジェクトの初期ページ)



これで初めてDjangoのページを表示させることができました!

サーバーを停止するときは、ターミナルで `control + c` を押してください。サーバーが停止されるとページも表示されなくなります。

また、サーバーを起動した状態だと同じターミナルのウィンドウではコマンドを実行できなくなります。新たにコマンドを打ちたい時は、ターミナルで新しいウィンドウを開き、もう一度仮想環境に入ってから作業するように注意してください。

settings.pyで設定を変更しよう

初めてDjangoのページを表示させることはできましたが、全て英語なのでちょっとわかりづらいですよね。これは、このプロジェクトで使用する言語に英語が設定されているからです。設定ファイルで、使用する言語を日本語に変えてみましょう。

エディタで `settings.py` ファイルを開き、`LANGUAGE_CODE = 'en-us'` と書かれている場所を探してください。ここで使用言語を設定しています。日本語に変更するために、ここを `'ja'` と書き換えましょう。

(~/DjangoBros/django_blog/django_blog/settings.py)

```
LANGUAGE_CODE = 'ja'
```

書き換えたら変更を保存して、もう一度 `http://127.0.0.1:8000/` にアクセスしてください。すると、表示が日本語になっているはずです。



続いて、`TIME_ZONE` も書き換えてみましょう。ここでは時間に関する設定をしています。

Djangoでは、`now` という文字を書くだけで現在の日時を取得する便利な機能があるのですが、現在の時刻といつても、どこの地域のタイムゾーンを使うかで時刻が変わってきます。時刻を日本基準にしたい場合は `'Asia/Tokyo'` と設定しましょう。

(~/DjangoBros/django_blog/django_blog/settings.py)

```
TIME_ZONE = 'Asia/Tokyo'
```

現在時刻の表示だけでなく、ブログの投稿日時などもこの設定が基準となるので、初めに日本のタイムゾーンを設定しておくとよいでしょう。

データベースを初期化しよう

Webサービスでは、ユーザー情報や投稿されるブログの記事など、たくさんのデータを管理しなくてはなりません。そのデータを保管するのが「データベース」です。

データベースにはいくつか種類がありますが、Djangoではデフォルトで**sqlite3**というデータベースを使うように設定されており、`settings.py`ファイルの中の `DATABASES` の部分で設定されています。

(~/DjangoBros/django_blog/django_blog/settings.py)

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

プロジェクトを立ち上げただけの現段階では、デフォルトで使用するデータベースの指定がされているだけで、`python manage.py migrate` というコマンドで実際にデータベースを作成(初期化)する必要があります。ルートディレクトリに移動して、このコマンドを打ってください。

(~/DjangoBros/django_blog/)

```
$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying sessions.0001_initial... OK
```

たくさん文字が出てきますがこのような文字が表示されていれば成功で、無事データベースが作成されています。

ウェブサービスの開発を進めていく中で、データベースの設計は隨時更新していくことになりますが、その変更をデータベースに反映するのが `python manage.py migrate` コマンドです。詳しくは、今後のチュートリアルの中で説明します。

第3章 チュートリアル1 ブログサイト

いよいよ1つ目のチュートリアルです。この章では、簡易的なブログサイトの作成を通して、Djangoでの開発の基本を学びます。ここで作成するブログサイトは、要件的に実際のサービスとしては使用できないと思いますが、DjangoやWeb開発の基本的な概念の説明を行うので、しっかりと押さえておきましょう。

このチュートリアルは第2章で作成した「django_blog」というプロジェクトを使って進めていきます。これ以降のサンプルコードでは、仮想環境への出入りを扱わないため、**それぞれのプロジェクト用の仮想環境(djangobros_venvなど)に入ってから作業するのを忘れないでください。**

3.1 アプリケーションを作ろう

第2章では、Djangoのプロジェクトを作成しました。それに続いて、**アプリケーションを作成していきましょう。**プロジェクトが「Webサービス全体に関わる設定」をまとめているのに対して、アプリケーションは「Webサービスに必要な機能を実現する部分」のことでしたね。

チュートリアル①で作るブログサイトでは、大まかに以下のような機能が必要になります。

- トップページに、ブログ記事の一覧が投稿日時の降順で表示される
- トップページから、新規記事作成ページにアクセスできる
- トップページから、各記事の詳細ページにアクセスできる
- 新規記事作成ページから、新規ブログ記事の投稿ができる
- 各記事の詳細ページでは、ブログのタイトルと内容が表示される
- 各記事の詳細ページから、その記事の編集ページにアクセスできる
- 各記事の編集ページから、その記事の内容を更新できる
- 各記事の詳細ページから、その記事を削除できる

1つのプロジェクトに対して、いくつも機能がある場合には、複数のアプリケーションを作成して役割を明確に分割することが多いですが、今回は簡易的に「blogs」というアプリケーションを1つだけ作成して開発を進めましょう。

アプリケーションを作るときは `python manage.py startapp` コマンドを使います。プロジェクトのルートディレクトリで以下のコマンドを実行してください。

(~/DjangoBros/django_blog/)

```
$ python manage.py startapp blogs
```

このコマンドにより、ルートディレクトリ内に `blogs` というディレクトリができます。

これが、このプロジェクトにおける1つのアプリケーションです。この `blogs` ディレクトリの中には、すでにたくさんOfFileがありますが、これらのファイルにコードを書いていくことで、様々な機能を実現していきます。

さて、アプリケーションを作成しましたが、実はこのままではDjangoのプロジェクトはこの**blogs**ディレクトリをアプリケーションとして認識してくれません。アプリケーションを作成したら、まずは `settings.py` ファイルの `INSTALLED_APPS` に追加する必要があります。

(~/DjangoBros/django_blog/django_blog/settings.py)

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'blogs', # これを追加する
]
```

こうすることによって、`blogs`アプリケーションが`django_blog`プロジェクトのアプリケーションとして認識されるようになります。この設定がないと、アプリケーションをいくら編集してもプロジェクトに反映されませんので、アプリケーションを作ったら、`INSTALLED_APPS`に追加することを忘れないようにしてください。

3.2 トップページを作ろう

まずは最初にシンプルなトップページを作成して表示してみましょう。最終的にこのページには、ブログ記事の一覧や新規記事作成ページへのリンクを表示します。

ページを表示するときは、通常のホームページ等と同様にHTMLファイルを作成して表示させますが、Djangoで扱うHTMLファイルは、動的なコードを扱うことができます。つまり、**HTMLファイル内でPythonコードを書く**ことができ、**変数やfor文などを扱う**ことができます。Djangoでは、このようなHTMLファイルを**Templates**と呼びます。

では、トップページ用に最初のテンプレートを作成しましょう。テンプレート用のフォルダを作るために、**blogs**ディレクトリに移動して以下のようにコマンドを打ってください。

(~/DjangoBros/django_blog/blogs/)

```
$ mkdir templates
$ cd templates
$ mkdir blogs
```

上記のコマンドでは以下のことを行いました。

- `django_blog/blogs/`内に「`templates`」というディレクトリを作成
- 作成した「`templates`」ディレクトリに移動
- `django_blog/blogs/templates/`内に「`blogs`」というディレクトリを作成

この一連の作業で、`django_blog/blogs/templates/blogs/` というディレクトリ階層ができました。

Djangoでは、このように各アプリケーション内に作成した `templates/<アプリケーション名>/` というディレクトリ内にテンプレート(HTMLファイル)を配置していきます。

次に、`django_blog/blogs/templates/blogs/` 内に、`index.html` というファイルを作成してください。

(~/DjangoBros/django_blog/blogs/templates/blogs/)

```
$ touch index.html
```

`index.html` を作成できたらエディタでそのファイルを開いて少しだけ編集しましょう。コードが書けたら忘れずにファイルを保存してください。

(~/DjangoBros/django_blog/blogs/templates/blogs/index.html)

```
<h1>ブログサイト</h1>
<p>ここはトップページです。</p>
```

これでトップページに表示させるテンプレートを作ることができました!

しかし、ファイルを作成しただけでは、「どのURLにアクセスされたときにこのファイルを表示するのか」を設定できていない状況です。ですので、今度はこのテンプレートファイルとURLを結びつけていきましょう。

3.3 ViewとURLを紐づける

トップページを表示するために、まずViewという概念を理解しましょう。

Viewでは、ユーザーから送られてきたアクセス情報(リクエスト)をもとに、どのHTMLファイルを表示させるか、どういった内容のデータを表示させるかなどを決めてレスポンスを返す処理をしています。

ユーザーは、特定のURLにアクセスすることによって、Djangoのプロジェクトが置いてあるWebサーバーに対してリクエストを送っています。そのリクエスト情報をもとに、表示内容を決めているのです。

言葉で説明すると難しく感じますが、慣れるとそれほど難しくはありません。後ほど解説しますが、これはMTV(Model/Template/View)モデルと呼ばれ、非常にシンプルかつ綺麗にプロジェクトの設計ができるようになっているのです。

それではまずは簡単なViewファイルを作って感覚を掴んでいきましょう。blogsアプリケーションの中にある `views.py` ファイルを開き、以下のようにコードを記述していきましょう。

(~/DjangoBros/django_blog/blogs/views.py)

```
from django.shortcuts import render

def index(request):
    return render(request, 'blogs/index.html')
```

これがViewの基本形です。ここでは `index()` という関数を作っています。

このView関数では `request` を第1引数にとっています。`request`とは、ユーザーがURLを入力してサーバーにアクセスする時に送られる情報のことです。`request`には、例えばログインしている人のユーザー情報などの様々な情報が含まれています。

返り値としてreturnしているrenderメソッドは、`request`情報を元にして `blogs/index.html` を表示することを意味しています。つまり、このindex関数は「ユーザーからの`request`情報を元に、`index.html`を返す」という内容のView関数になります。

続いて、このView関数を特定のURLに紐づける設定をしましょう。この設定によって、ユーザーがそのURLにアクセスした時にindex関数が実行されるようになり、ブラウザ上ではindex.htmlが表示されるようになります。URLの設定は `urls.py` ファイルで行います。

URLの設定は、プロジェクトディレクトリの`urls.py`ファイルで定義することができますが、通常はアプリケーション側にも`urls.py`ファイルを作成し、プロジェクト側の`urls.py`では各アプリケーションの `urls.py` ファイルをインポートすることで管理することが多いです。例えば、`blogs`アプリケーションの`urls.py`では

`http://127.0.0.1:8000/blogs/<...>/` のように`blogs`に紐づくURLだけを管理できるようにします。

それではまずプロジェクト側の `urls.py` ファイルを編集します。

(~/DjangoBros/django_blog/django_blog/urls.py)

```
from django.contrib import admin
from django.urls import path, include # includeを追加

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('blogs.urls')), # 追加
]
```

`path('', include('blogs.urls'))` では何をしているかというと、

`http://127.0.0.1:8000/` (ルートURL)にアクセスされたときは、`blogs.urls` ファイルを参照することを意味しています。`blogs.urls` とは、この後自分で作成する `django_blog/blogs/urls.py` のことです。

例えばこれを、`path('blogs/', include('blogs.urls'))` と書いたとすると、

`http://127.0.0.1:8000/blogs/` にアクセスされたときに `blogs.urls` ファイルを参照することになります。

続いて、`blogs`アプリケーション内のURLの設定を行います。`blogs`ディレクトリに移動して以下の内容の`urls.py`ファイルを作成しましょう。

(~/DjangoBros/django_blog/blogs/urls.py)

```
from django.urls import path
from . import views

app_name = 'blogs'
```

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

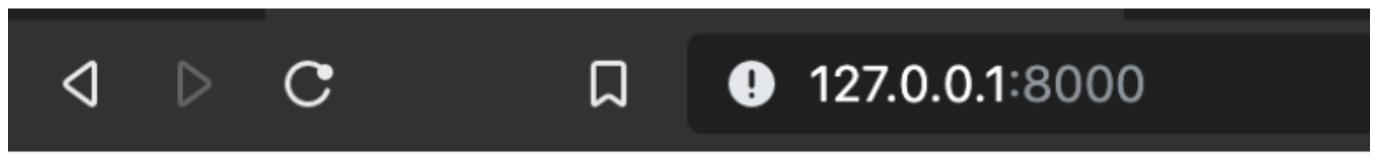
2行目の `from . import views` という部分では、同じ階層(ここではblogs/ディレクトリ)にあるviews.pyファイルをインポートしています。ドットは、同じ階層という意味です。

path関数では、第1引数で空の文字列を指定し、第2引数で、`views.index` を指定することで、URL(`http://127.0.0.1:8000/`)にアクセスした時は、views.pyのindex関数を実行するように設定をしています。

例えば、ここを `path('top/', views.index, name='index')` のように書き換えたとすると、`http://127.0.0.1:8000/top/` にアクセスしたときにindex関数が実行されることになります。このように、第1引数ではURLのパスを設定しています。

第三引数の `name='index'` という部分は、このURLパスに名前をつけており、他のファイルからこのURLへの参照をできるようにしています。`app_name = 'blogs'` も同様の意味があります。

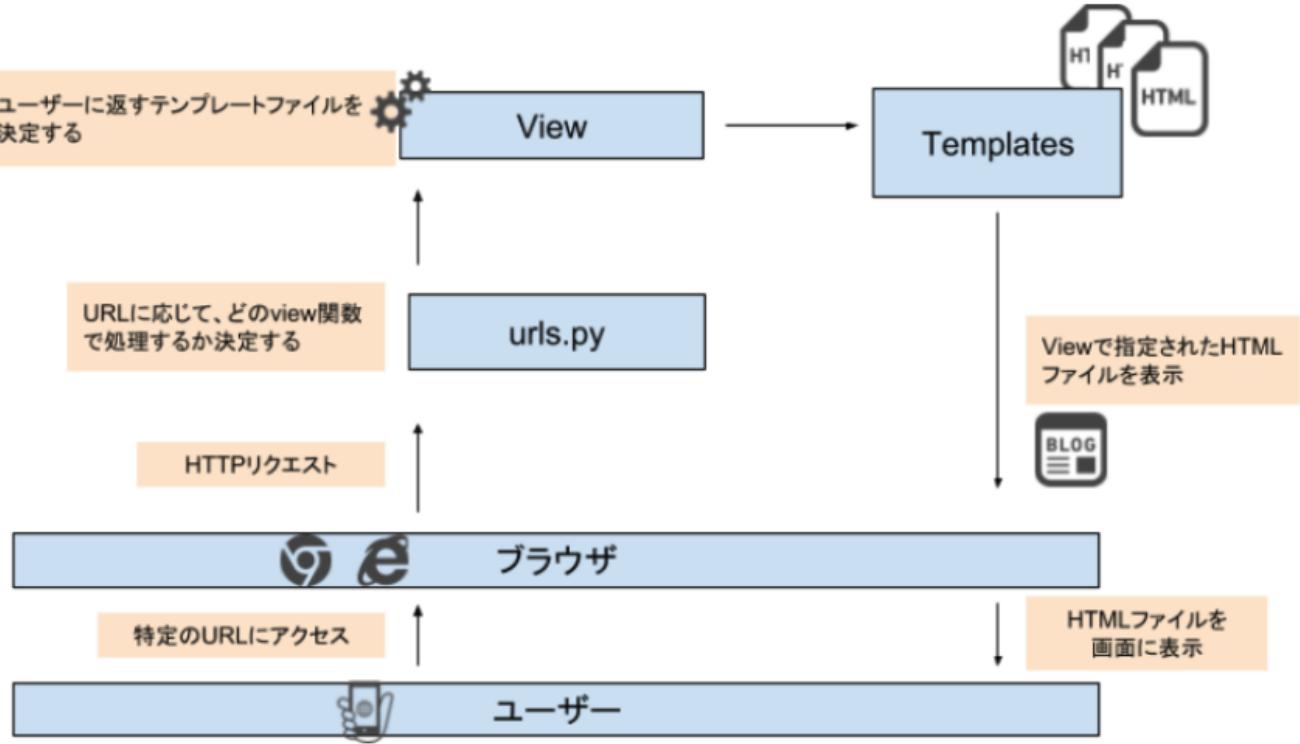
これでページを表示する準備が整いました。ローカルサーバーを立ち上げて、`http://127.0.0.1:8000/` にアクセスすると、トップページが表示されるはずです。



ブログサイト

ここはトップページです。

ここまで学習した流れを図で見るとこのようになります。



URL→View→Templateの流れはDjangoアプリケーションにおいてベースとなるものなので、この流れをしっかり抑えおきましょう。

どうして**templates**ディレクトリ内にアプリケーションと同名のディレクトリを作るのは？

テンプレート用のディレクトリを作成する際に、`/blogs/templates/blogs/` のよう
に、ディレクトリ内にアプリケーションと同名のディレクトリを作成しました。一見、`/blogs/templates/`
だけでもよさそうですが、どうしてこのようにしているのでしょうか。

これは、Djangoがテンプレートファイルを探す仕組みに関係しています。Djangoのデフォルトの設定では、テンプレートファイルを探す際に、`templates` という名前のディレクトリを参照するようになっています。

例えばviews.pyに書いたrender関数では、第2引数で表示するテンプレートファイルを指定しています。第2引数は`'blogs/index.html'` となっていますが、これは`templates/blogs/index.html` を指定していることになるのです。

ディレクトリ構成を`blogs/templates/index.html` のようにtemplates直下にindex.htmlファイルを置き、第2引数を`'index.html'` とする書き方もできます。

この方がシンプルで良さそうですが、この書き方だと他のアプリ内にもindex.htmlがあった場合、意図しない結果となってしまうので望ましくありません。

例えば、以下のようにapp1とapp2という2つのアプリがあったとします。ここでは、それぞれがindex.htmlファイルという同名のファイルを保有しているとします。このとき、`render(request, 'index.html')` すると、render関数は常にapp1内のtemplatesディレクトリを参照することになり、app2内のindex.htmlは指定できなくなります。

※仮にapp2の中にあるviews.pyで `render(request, 'index.html')` と記述してもapp1のindex.htmlを参照します。これは、テンプレートファイルを参照するときに、上のtemplatesディレクトリから探していくという決まりがあるからです。

(悪い例)

```
app1/
  templates/
    index.html
app2/
  templates/
    index.html
```

この状況を防ぐために、あえてtemplates内にディレクトリを作ることで、同名ファイルがあっても明示的に指し示せるようにします。

(良い例)

```
app1/
  templates/
    app1/
      index.html
app2/
  templates/
    app2/
      index.html
```

このようにすれば、`'app1/index.html'` や `'app2/index.html'` のように記述でき、それぞれのアプリ内に同名ファイルがあっても大丈夫になります。

つまり、アプリケーションのtemplatesディレクトリ内に、アプリケーション名と同名のディレクトリを作成する理由は、別々のアプリケーションの間で同名のHTMLファイルを作成してもバッティングしないようにするためということです。

ちなみにですが、app内のtemplatesを自動的に参照するのは、settings.pyファイルの `'APP_DIRS'` がTrueになっているからです。

(settings.py)

```
TEMPLATES = [
{
  'BACKEND': 'django.template.backends.django.DjangoTemplates',
  'DIRS': [],
  'APP_DIRS': True,
  'OPTIONS': {
    'context_processors': [
      'django.template.context_processors.debug',
      'django.template.context_processors.request',
```

```

        'django.contrib.auth.context_processors.auth',
        'django.contrib.messages.context_processors.messages',
    ],
},
],
]

```

3.4 ブログモデルの作成

トップページを表示することができたので、実際に表示するブログ記事をデータベースに登録できるように、ブログモデルを作りましょう。モデルとは、特定のデータの設計図の役割を果たし、データベースとやりとりするクラスになります。

実際にモデルを作成する前に、データベースの基本的な部分について解説します。Webサービスを運営していると、様々なデータを扱うことになりますが、それを保管している場所がデータベースです。

例えば、新規にユーザー登録した人がいればその人の名前や年齢などのユーザー情報を保管しなければなりませんし、新しくブログ記事が投稿されたときはその文章などを保管する必要があります。

データベースがあればデータを保管するだけではなく、データを操作したり検索したりすることも可能となります。

では、具体的にデータベースはどのような構成になっているのでしょうか。

データベースは、以下のような表形式になっており、この表のことを**テーブル**、横の行の1つ1つを**データ**(または**レコード**)、縦の列を**フィールド**(または**カラム**)といいます。

1つの行につき1つのインスタンス(個別のデータ)を保管し、各フィールドでそのインスタンスのプロパティ(実際の値)を保管しています。

Users テーブル

フィールド (列)						
データ (行) →		id	名前	年齢	性別	趣味
1	ジョブス	32	男	プログラミング		
2	ローラ	26	女	サッカー		
3	ジョン	18	男	ゲーム		
4	ケイ	52	男	読書		
5	キャサリン	29	女	料理		

ここまで整理できたら実際にモデルを作ってみましょう。今回作るブログモデルは、タイトル、テキスト(ブログ記事の本文)、作成日時、更新日時の4つの情報(フィールド)を持たせます。

ブログサイトでは、たくさんのブログ記事がデータとして保存されることになりますが、ブログ記事に必要な情報(フィールド)をブログクラスで定義してあげます。

本来、データベースを操作するときはSQLという言語を使うのですが、この「モデル」を作成すると**Pythonのコード**でデータベースの構造を記述し、簡単に操作できるようになります。

また、Djangoにおけるモデルとは、データベースに保存したいデータの構造を指定したもののことです。通常、1つのモデルに対して1つのデータベーステーブルが割り当てられます。モデルは各アプリケーション内の `models.py` に記述していきます。

(~/DjangoBros/django_blog/blogs/models.py)

```
from django.db import models

class Blog(models.Model):
    title = models.CharField(blank=False, null=False, max_length=150)
    text = models.TextField(blank=True)
    created_datetime = models.DateTimeField(auto_now_add=True)
    updated_datetime = models.DateTimeField(auto_now=True)

    def __str__(self):
        return self.title
```

先頭で、`from django.db import models` をインポートします。`models.Model` はDjangoにおけるモデルの型の基本形であり、それを継承させたBlogクラスを作成しています。このクラスを元に作られたインスタンスはデータベースに保存されるオブジェクトになります。

1つ注意点ですが、モデルを作るときは、「Blogs」の様に複数形で書かずに「Blog」と単数形で書きます。これによって「Blogs」という名称のテーブルを自動生成してくれます。また、Pythonでクラスを作る場合と同様、最初の文字は大文字で始めます。

次に、「title」や「text」の部分では、Blogsテーブルのフィールドを定義しています。この記述により、以下のよくなテーブルを作ることができます。

Blogs テーブル

	id	title	text	created_datetime	updated_datetime
(例)	1	読書	今日は読書をしました。	2018/03/18	2018/03/21
	2				
	3				
	4				
	⋮				
	⋮				
	⋮				

各フィールドについてもう少し詳しく説明します。

(~/DjangoBros/django_blog/blogs/models.py)

```
title = models.CharField(blank=False, null=False, max_length=150)
```

まず、フィールドに `title` という名前をつけて、右辺で `models.CharField` という型を指定しています。このようにフィールドにはいくつか型の種類があり、**型を指定することによって意図しないデータが入力されることを防いでいます。** この `models.CharField` は文字列を入力できるようにした型です。当然、ブログ記事のタイトルには文字列が入ることになるからです。(CharField は、Characterを意味しています)

さらに `models.CharField` の中にいくつか条件を指定することができます。例えば、`blank=False, null=False` は、このフィールドが空欄であってはいけないということを示しています。こうすることで、ブログ記事のタイトルが空欄の状態では、データベースに保存できないことになります。

`blank`と`null`の違いは、入力フォームでの空欄を許容するかどうかと、データベース上での空欄を許容するかどうかの違いです。この点は、DjangoBrothersのブログ記事([Djangoモデルフィールドのnullとblankの違いを理解する](#))でも解説しています。

そして、`max_length=150` ではタイトルの文字数を最大150字に制限しています。

`created_datetime`、`updated_datetime` のフィールドでは、`models.DateTimeField` という型を使って、日時情報を保存するように指定しています。もちろん、日時情報を自分で手動入力して保存することもできますが、今回は `auto_now_add`、`auto_now` という引数を与えることで、インスタンスが作成された日時、更新された日時を自動的に保存するよう設定しています。

最後の `def __str__(self)` の部分では、このモデルで作成されたインスタンス(1つ1つのブログ記事)自体を指し示すときに利用する文字列を指定しています。この設定により、管理ページなどで各インスタンスを表示するときはブログ記事のタイトルで表示されることになります。(この説明だけだとわからないと思いますので、管理ページを作成するときに詳しく説明します)

このように、モデルの作成時には、フィールドの型と各種条件をつけてデータベースのデータの形を設計していきます。

モデルで使用できる型は、こちらの公式ドキュメントでも確認することができます。

<https://docs.djangoproject.com/ja/2.2/ref/models/fields/>

さて、Blogモデルを作成することができました。ただ、モデルはあくまで「設計図」であるため、現段階だとデータベースの設計図を作ったに過ぎません。この設計図の内容を実際にデータベースに反映するには、「マイグレーション」という作業が必要になります。

マイグレーションとは、`models.py`ファイルで定義したデータベースの設計を、実際にデータベースに反映させることです。マイグレーションをするためには、以下2つのコマンドを実行します。

(~/DjangoBros/django_blog/)

```
$ python manage.py makemigrations  
$ python manage.py migrate
```

`python manage.py makemigrations` コマンドでは「マイグレーションファイル」というものを作っています。このコマンドによって、`django_blog/blogs/migrations/` ディレクトリの中にマイグレーションファイルが新たに追加されます。マイグレーションファイルとは、`models.py`ファイルで作成したデータベースの設計情報がまとめられたファイルです。このファイルには、「データベースに対してどの様な変更を加えるか」の情報が記載されています。

`python manage.py migrate` コマンドは、マイグレーションファイルの情報をデータベースに反映させるコマンドです。これにより、models.pyファイルで作成したBlogモデルをデータベースに反映させることができました。

このように、データベースに変更を加えるときは、**models.py**ファイルで設計 → マイグレーションでデータベースに反映という手順で行います。models.pyファイルに何か変更を加えたとき(例えばタイトルフィールドの文字数制限を150文字から200文字に変えたとき)は、その都度マイグレート処理を行い、変更内容をデータベースに反映させることを忘れないようにしてください。

3.5 Adminページを作ろう

続いて、Adminページというものを設定していきます。Adminページとは管理者のみが使用できるページのことです、通常は自作しなければならないものなのですが、**Djangoはこの管理ページを自動的に生成してくれる**という素晴らしい特徴を持っています。

Djangoでは、models.pyで定義したモデルを元に自動的に管理画面を作ってくれます。これにより、**新しいデータの追加や編集、削除などの機能が管理者ページから簡単に利用できる**のです。

Adminページはデフォルトだと、`/admin` というURLからアクセスすることができ、最初に何のモデルも作成していない状態でもデフォルトのページを利用することができます。今回は、作成したBlogモデルをAdmin画面で確認できるように設定してみます。また、実際にブログ記事をAdmin画面から作成してみましょう。

まずは、管理ページにアクセスするために管理者アカウントを作成します。Djangoでは、**管理者のことをスーパーユーザー**と呼びます。スーパーユーザーのアカウントはコマンドラインから作成することができます。

(~/DjangoBros/django_blog/)

```
$ python manage.py createsuperuser
```

コマンドを実行するとユーザー名、メールアドレス、パスワードを聞かれるので、自分の好きなように入力してください。パスワードは、入力してもターミナルには何の文字も表示されませんが、表示されないだけでちゃんと入力されているので心配しないでください。

管理者アカウントが作成されたら、実際にAdminページにアクセスしてみましょう。ローカルサーバーを立ち上げて、`http://127.0.0.1:8000/admin` というURLからアクセスすることができます。

最初にログインが要求されると思いますが、作成した管理者アカウントでログインできます。ログインすると次のような画面が表示されます。

The screenshot shows the Django Admin site's main dashboard. On the left, there are two main sections: '認証と認可' (Authentication and Authorization) containing 'グループ' (Groups) and 'ユーザー' (Users), each with '追加' (Add) and '変更' (Change) buttons; and '最近行った操作' (Recent Operations) which is currently empty. The URL in the browser is 127.0.0.1:8000/admin/.

管理するためのページといつても、まだデフォルトの状態なのでユーザー情報しか管理することができません。次は、この画面でBlogsテーブルの情報を管理できるように設定ていきましょう。ちなみに、この管理ページが日本語で表示されているのは、`settings.py`ファイルで `LANGUAGE_CODE = 'ja'` と設定しているからです。

Adminページを編集するためには`admin.py`ファイルを使います。まずはBlogモデルを`admin.py`に登録します。`admin.py`ファイルを開いて、以下のように編集してください。

(~/DjangoBros/django_blog/blogs/admin.py)

```
from django.contrib import admin
from .models import Blog

admin.site.register(Blog)
```

`from .models import Blog` では、`admin.py` ファイルと同じ階層にある`models.py`ファイルで定義した Blogモデルをインポートして、このファイルで扱えるようにしています。そして `admin.site.register(Blog)` の部分でインポートしたBlogモデルを、Adminページで利用できるようにしています。これにより、AdminページでBlogの情報を見ることができます。ファイルを保存して、再度Adminページにアクセスしてください。

The screenshot shows the Django Admin site's main dashboard again. This time, the 'BLOGS' section is highlighted. It contains a single entry 'Blogs' with '追加' (Add) and '変更' (Change) buttons. Below this, the '認証と認可' (Authentication and Authorization) section is visible, identical to the previous screenshot. The URL in the browser is 127.0.0.1:8000/admin/.

Blogsというパートが表示されるようになっているはずです。これでブログ記事の管理ができるようになりました! このように、Blogモデルを作ると自動的にBlogsと複数形にしてテーブルを作ってくれます。

続いて、Admin ページから記事を投稿してみましょう。Blogsという欄の「追加」ボタンをクリックしてください。個別のブログ記事の新規追加ページに遷移します。

blog を追加 | Django サイト管理

Django 管理サイト

ようこそ `USER_NAME`. サイトを表示 / パスワードの変更 / ログアウト

ホーム > Blogs > Blogs > blog を追加

blog を追加

Title:

Text:

保存してもう一つ追加 保存して編集を続ける 保存

「Title」と「Text」という `models.py` ファイルで定義した `Blog` クラスのフィールド名が表示されています。ここには文字列型を入れられる設定にしていましたので、このように文字を入れるフォームが表示されます。

このページから新規のブログ記事を投稿できるので、実際に1つブログを投稿してみましょう。好きなようにタイトルと本文を入力し、右下の「保存」ボタンで保存しましょう。ちなみに、タイトルを入力しないまま保存ボタンを押すとエラーメッセージが表示されます。このフィールドは `blank=False` を指定しているため、入力が必須のフォームになっています。

ここで `Blog` クラスを元に `Blog` インスタンスを作ることができました。クラスは「オブジェクトを量産するための設計図」、インスタンスは「クラスを元に作られた個々のオブジェクトのこと」です。

テスト用にもう1つか2つ程度、記事を投稿してみてください。

ここで先ほど少し触れた、`def __str__(self)` の意味について説明をします。ブログオブジェクトの一覧ページ(`http://127.0.0.1:8000/admin/blogs/blog/`)にいくと、それぞれのブログのタイトルが表示されています。ここにタイトルが表示されるのは以下のように `self.title` と記述しているからです。

(~/DjangoBros/django_blog/blogs/models.py)

```
...
def __str__(self):
    return self.title
```

例えば、ここを `self.text` と書き換えると、ブログの本文が見出しとして表示されるようになります。このように `__str__` メソッドでそのオブジェクト指し示す時に使用されるフィールドを指定することができます。

また、このページには現在タイトルだけが表示されていますが、ここには複数のフィールドを表示させることもできます。 `admin.py` ファイルを以下のように書き換えてみてください。

(~/DjangoBros/django_blog/blogs/admin.py)

```

from django.contrib import admin
from .models import Blog

class BlogAdmin(admin.ModelAdmin):
    list_display = ('id', 'title', 'created_datetime', 'updated_datetime')
    list_display_links = ('id', 'title')

admin.site.register(Blog, BlogAdmin)

```

`list_display` で指定したフィールドが管理ページに表示されるようになります。

`list_display_links` で指定したフィールドはリンクがつくようになります。また、ここで指定している `id` というフィールドはモデルの作成時には定義していましたが、これはインスタンスがデータベースに保存されるときに自動的に割り振られる番号のことです。**1つのインスタンスに対して1つの番号(id)が割り振られます**。このIDがインスタンス同士で重複することはありませんので、「出席番号」や「マイナンバー」と同じようなものと考えることができます。

ID	TITLE	CREATED DATETIME	UPDATED DATETIME
3	読書	2018年3月18日19:15	2018年3月18日19:15
2	サッカー	2018年3月18日19:15	2018年3月18日19:15
1	初めてのブログ記事です。	2018年3月18日19:15	2018年3月18日19:15

Adminページのカスタマイズは他にもいろいろできるので、公式ドキュメントなどを参考に、使いやすいうように改良してみてください。

<https://docs.djangoproject.com/ja/2.2/ref/contrib/admin/>

3.6 トップページに記事を表示しよう

この節では、先ほどAdminページから作成した記事を、トップページに表示してみましょう。まずは、データをDjango上でどのように扱うかの基本を確認したのちに、テンプレート上にデータを表示する方法を学びます。

クエリセットを理解しよう

クエリセットとは、データベースから取得したモデルインスタンスの一連の情報のことで、先ほどAdmin画面で作成したブログ記事等のデータをまとめて処理するための集合体と考えると良いかと思います。

例えば、`Blog.objects.~` とコードを書くことで、データベースから取得するデータを定義することができます。`objects`は様々なメソッドを持っていて、特定の条件でフィルターをかけることができます。ここで取得したデータのリストがクエリセット(QuerySet)です。

実際に使ってみるとよく理解できると思いますので、コンソールからこのクエリセットの取得を試してみましょう。

プロジェクトのルートディレクトリで、`python manage.py shell` というコマンドを打つと、Pythonのコマンドを入力できる画面になります。`python3` というコマンドでも似たような画面が起動することはすでに学習済みですが、こちらのコマンドではDjangoのプロジェクトと対話的にやりとりができるコマンドプロンプトが立ち上がります。

(~/DjangoBros/django_blog/)

```
$ python manage.py shell  
>>>
```

まずは、先ほど作成したBlogモデルを扱うために、モデルをインポートしましょう。Blogモデルは、`django_blog/blogs/models.py` で定義しているので以下のようにインポートできます。

(Django shell)

```
>>> from blogs.models import Blog
```

インポートできたら、実際にデータベースにアクセスしてBlogsテーブルからデータを取得してみます。まずは今データベースに保存されている全てのBlogインスタンスを取得してみます。データを全て取得するためには `all()` メソッドを使います。

(Django shell)

```
>>> Blog.objects.all()
```

先ほど作成したブログ記事を一覧表示できたと思います。`all()` の他にも便利なメソッドがあるので、簡単に紹介しておきます。

(Django shell)

```
# idが1のインスタンスを取得  
>>> Blog.objects.get(id=1)  
  
# titleフィールドに文字列"Django"を含む記事を全て取得  
>>> Blog.objects.filter(title__contains="Django")  
  
# idの順に取得  
>>> Blog.objects.order_by('id')  
  
# フィールド名の前に - をつけると降順で取得できる  
>>> Blog.objects.order_by('-id')
```

また、複数のメソッドを繋げて利用することもできます。

(Django shell)

```
# titleフィールドに"Django"を含む記事をidの降順で取得  
>>> Blog.objects.filter(title__contains="Django").order_by('-id')
```

他にもクエリセットの指定方法はたくさんあり、様々な形で条件を指定してインスタンスを取得することができます。クエリセットの取得方法については、他のチュートリアルでも適時解説していきますので、少しずつ知識をつけていきましょう。

次に、`create()` メソッドを紹介します。このメソッドで、コマンドプロンプトからインスタンスを作成することができます。

(Django shell)

```
# 各フィールドに値を指定してBlogインスタンスを作成  
>>> Blog.objects.create(title="コマンドから作られたブログ", text="createメソッド  
でブログを作ってみました。")
```

日時に関するフィールドは、自動で値が入るので指定する必要はありません。これでコマンドラインからインスタンスを作ることができました。Adminページから確認することもできますし、先ほど学習した`all()` メソッドでも、新しいインスタンスが追加されているのを確認できると思います。

実際のコード内では、クエリセットの取得結果は変数に代入して利用することが多いです。また、`インスタンス.フィールド名`と書けば、そのインスタンスのプロパティを取得することもできます。

(Django shell)

```
# 取得結果(id=1のインスタンス)をblog変数に代入  
>>> blog = Blog.objects.get(id=1)  
  
# <インスタンス.フィールド名>でプロパティを取得  
>>> blog.title  
  
# Blogインスタンスを全件取得しblogs変数に代入  
>>> blogs = Blog.objects.all()  
  
# blogsの1番目のインスタンスの本文を表示  
>>> blogs[0].text  
  
# for文で全Blogインスタンスのタイトルを表示  
>>> for blog in blogs:  
...     print(blog.title)
```

テンプレートタグを使ってクエリセットをHTMLに表示

クエリセットを理解することで、データベースに保管されたインスタンスを自分が指定した条件で取得できるようになりました。次は、取得したインスタンスをHTMLで表示させましょう。

すでに少しだけ紹介しましたが、HTMLで表示する内容はViewで定義することができます。先ほどターミナルに直接打ったコードをViewに書いてクエリセットを取得し、そのクエリセット HTMLファイルに渡して表示させるという流れになります。

それでは、クエリセットをHTMLで表示させる準備をしていきます。Blogsアプリケーションのviews.pyを開いて、このように書き換えてください。

(~/DjangoBros/django_blog/blogs/views.py)

```
from django.shortcuts import render
from .models import Blog # 追加

def index(request):
    blogs = Blog.objects.order_by('-created_datetime')
    return render(request, 'blogs/index.html', {'blogs': blogs})
```

これまでのコードと比べて、3箇所修正されています。

1つ目は、2行目でBlogモデルをmodels.pyからインポートしている点です。このファイルでBlogモデルを利用できるようにしています。

2つ目は、`blogs = Blog.objects.order_by('-created_datetime')` の部分です。データベースに保存されているBlogインスタンスを作成日の降順に並べたクエリセットを取得し、`blogs` という変数に代入しています。

3つ目は、`render()` の第3引数に `{'blogs': blogs}` を追加している部分です。このようにrender関数の中で{}と書かれている部分では、テンプレートに渡したいデータを自由に定義することができます。Pythonの辞書型と同様に定義しますので、キーと値を書かなければなりません。今回の場合、キーは文字列の `'blogs'` で、値は変数 `blogs` (つまり、1つ上の行で定義したクエリセット)となります。

これにより、`blogs/index.html` では、キー名の `blogs` と書くだけでクエリセットで取得したデータを表示することができるようになります。

今回は、キーと値の名前を同じ(blogs)にしましたが、キーの名前は自由に定義することができます。注意しなければならないのは、キーは文字列なのでシングルクオートかダブルクオートで囲み、値は今回の場合は変数名をそのまま書いている点です。

これで、クエリセットのデータをテンプレートに渡すことができるので、実際に表示させてみましょう。HTMLファイルを開いてこのように書いてください。

(~/DjangoBros/django_blog/blogs/templates/blogs/index.html)

```
<h1>ブログサイト</h1>
<p>ここはトップページです。</p>
{{ blogs }}
```

このように、Viewから渡されたデータはキーに指定した文字列を `{} {}` で囲むことで表示することができます。 `{} {}` のことを **テンプレートタグ**といいます。

コードを書いたら、保存してトップページを表示してみましょう。このように、クエリセットが表示されるはずです。



ブログのクエリセットを表示させることができましたが、これだとBlogインスタンスの一覧を表示させているだけです。1つ1つの要素を表示するために、Pythonのfor文を使いましょう。以下のように、**テンプレートタグを使うことでHTMLに直接Pythonコードを書くことができます**。このfor文では、変数「blog」に、blogsのインスタンスを順番に代入して表示させています。

(~/DjangoBros/django_blog/blogs/templates/blogs/index.html)

```
<h1>ブログサイト</h1>
<p>ここはトップページです。</p>
{% for blog in blogs %}
    {{ blog.title }}
{% endfor %}
```

`{% %}` という新しい表現が出てきましたが、これもテンプレートタグです。`{} {}` と違って、`{% %}` では、ロジックの部分を扱うため、中身は実際のページには表示されません。

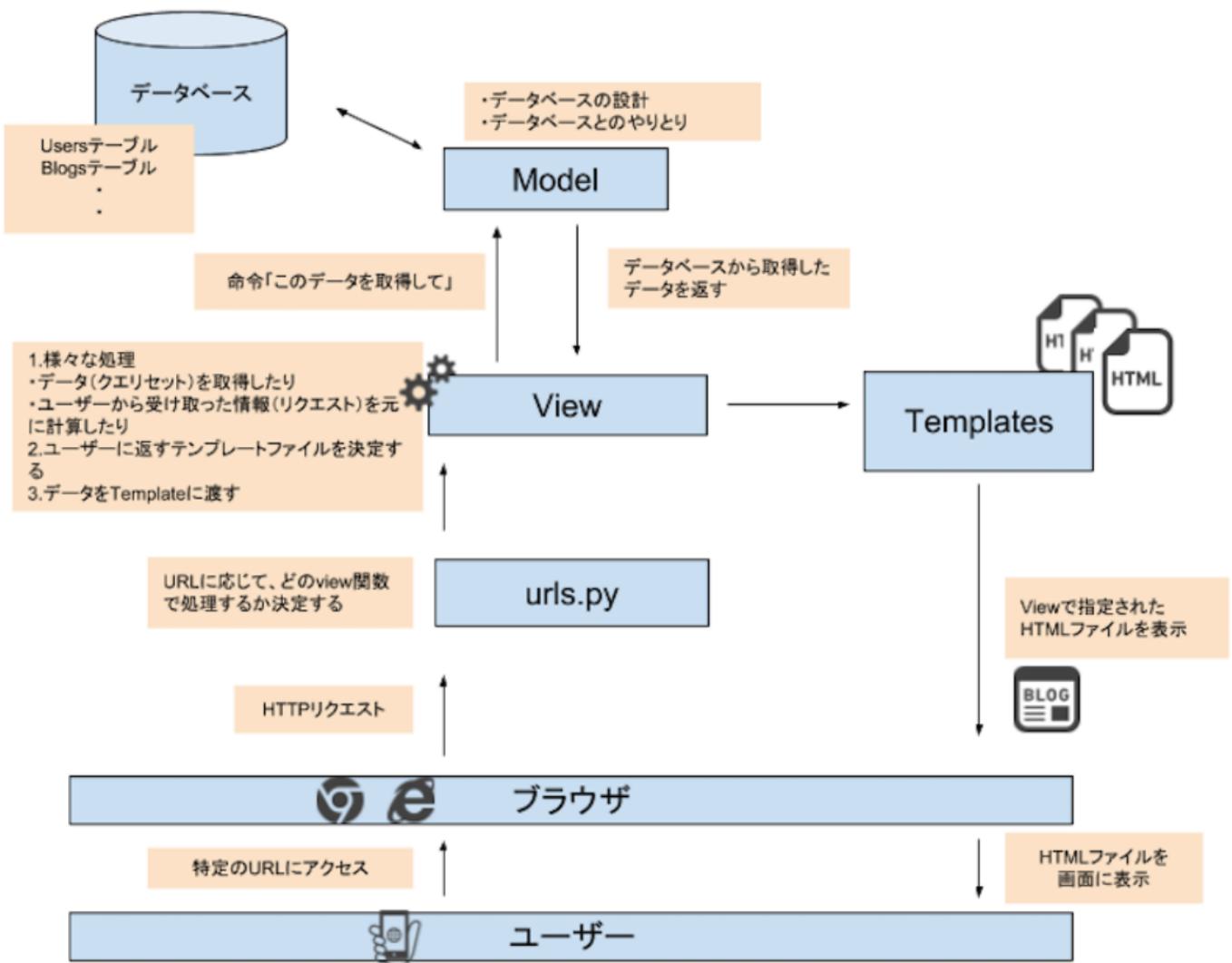
また、通常のPythonコードと違い、for文の終わりを表す `{% endfor %}` というコードが必要になるので、忘れないように注意してください。

上のコードでは、`{{ blog.title }}` のように「インスタンス.フィールド名」の形式で書くことで、各インスタンスのタイトルを表示させています。`{{ blog.text }}` と書けば、各記事の本文が表示されるようになります。

なお、フィールド名を省略して、`{{ blog }}` とだけ書く場合、models.pyファイルの `__str__` 関数で `return self.title` と指定しているので、タイトルが表示されます。

これでクエリセットのデータをちゃんと1つ1つ表示できるようになりました!

ここまで流れを図にすると以下のようになります。これが理解できていれば、Djangoの基礎的な流れは完璧です!



最後に、もう少し見た目を整えておきましょう。実際にはCSSを使ってデザインできますが、このチュートリアルでは単純に、HTMLタグにstyle属性を指定することで最低限のデザインを整えるだけにします。

ついでに、`{{ blog.text }}` とコードを追加して各記事の本文も表示させます。実際のコードには、`{{ blog.text |truncatechars:100 }}` と書いてください。`|truncatechars:100` の部分はテンプレートタグフィルターと呼ばれるもので、表示上の条件をつけることができるオプションです。`truncatechars` フィルターは表示する文字数を制限するもので、本文の最初の100文字だけを表示するように条件をつけています。フィルターは他にもありますので、別の機会に紹介します。

(~/DjangoBros/django_blog/blogs/templates/blogs/index.html)

```
<!DOCTYPE html>
<html>
<head>
    <title>ブログ管理サイト</title>
</head>
<body>
    <h1 style="text-align: center;">My Blog</h1>
    <div style="width: 70%; margin: 0px auto;">
        <hr />
        {% for blog in blogs %}
            <div>
                <h3>{{ blog.title }}</h3>
```

```
<div>{{ blog.text | truncatechars:100 }}</div>
</div>
<hr />
{% endfor %}
</div>
</body>
</html>
```

My Blog

読書

今日は家で読書しました。

サッカー

今日は友達とサッカーをしました。

初めてのブログ記事です。

はじめてのテキストです。

3.7 記事詳細ページを作ろう

この節では、先ほど表示したトップページからリンクを張って、記事詳細ページを表示しましょう。以下の変更を加えて、トップページにリンクを表示します。

(~/DjangoBros/django_blog/blogs/templates/blogs/index.html)

```
<!DOCTYPE html>
<html>
<head>
    <title>ブログ管理サイト</title>
</head>
<body>
    <h1 style="text-align: center;">My Blog</h1>
    <div style="width: 70%; margin: 0px auto;">
        <hr />
        {% for blog in blogs %}
        <div>
            <h3>{{ blog.title }}</h3>
            <div>{{ blog.text | truncatechars:100 }}</div>
            <!-- 追加 -->
            <div style="text-align: right;">
                <a href="{% url 'blogs:detail' blog_id=blog.id %}">記事を読む</a>
            </div>
        </div>
    {% endfor %}
</body>
</html>
```

```

    </div>
</div>
<hr />
{% endfor %}
</div>
</body>
</html>

```

aタグのhrefの中身に注目してください。hrefの中には通常、リンク先のURLを記述しますが、ここでも、`{% %}`のテンプレートタグを使っています。つまりここではDjango特有の書き方でURLを指定していることになります。

具体的にそれぞれのコードが何を意味しているのかを説明します。まず、`url 'blogs:detail'`の部分では、このリンクがblogsアプリケーション内のurls.pyで「detail」と名前をつけたURLにリンクすることを示しています。urls.pyには、まだこのURLを作成していないのであとで作ります。

`blog_id=blog.id`では、変数`blog_id`に各ブログのidである数字を代入しています。`blog.title`で各ブログのタイトルを取得しているのと同様に、`blog.id`で各ブログのidが取得できるので、それを変数に代入しています。こうすることでid情報(数字)をurls.pyに渡すことができます。

次に、記事詳細ページを作っていきましょう。新しいページを追加するために必要なことを覚えていませんか？以下3つのステップが必要になります。

- URLを設定する(urls.py)
- Viewを作る(views.py)
- HTMLファイルを作る

まずは、URLを作成しましょう。urls.pyを開いて、このようにpathを追加してください。

(~/DjangoBros/django_blog/blogs/urls.py)

```

from django.urls import path
from . import views

app_name = 'blogs'
urlpatterns = [
    path('', views.index, name='index'),
    path('detail/<int:blog_id>', views.detail, name='detail'),
]

```

追加したpathは、1つ目のpathよりちょっとだけ複雑になっています。

第1引数の`'detail/<int:blog_id>'`ではURLを指定しています。`int`は、ここに数字が入ることを示しています。つまり、`http://127.0.0.1:8000/detail/100/`のようなURLを想定しています。

第2引数は、1つ上のpathと同様のことを行っています。このURLでは、views.pyファイルのdetail関数が実行されることを意味しています。また、detail関数には`<int:blog_id>`の部分に対応する値が渡されることになります。例えば、`http://127.0.0.1:8000/detail/100/`にアクセスされたとき、detail関数内では変数`blog_id`の値が100となります。

第3引数は、このURLにdetailという名前をつけています。名前をつけることで、`記事を読む`の部分でこのpathを参照することができるようになっています。`blog.id`の値が100だった場合、`http://127.0.0.1:8000/detail/100/`のURLに遷移することを意味します。

これで、`http://127.0.0.1:8000/detail/1`にアクセスした際は、id=1の記事の詳細ページを表示するURLの設定ができました。ただし、まだView関数やHTMLファイルを作っていないので、現段階でこのURLにアクセスしてもエラーが出ます。

今度は、View関数を作りましょう。views.pyファイルのdetail関数で記事詳細ページを表示する処理がされるように設定しましたので、detail関数を作ります。

(~/DjangoBros/django_blog/blogs/views.py)

```
from django.shortcuts import render
from .models import Blog

def detail(request, blog_id):
    blog = Blog.objects.get(id=blog_id)
    return render(request, 'blogs/detail.html', {'blog': blog})
```

index関数と違って、request以外にもblog_idという引数を指定しています。先ほど設定したpathの第1引数の、変数blog_idを受け取っています。`http://127.0.0.1:8000/detail/100/`にアクセスされた場合、blog_idの値は100となります。

`blog = Blog.objects.get(id=blog_id)`では、idがblog_idの数字と一致するBlogインスタンスをデータベースから取得して、変数「blog」に代入しています。
render関数で、変数「blog」をdetail.htmlに渡しています。

これで、View関数は完成です。それでは最後に、記事詳細ページ(detail.html)を作りましょう。

トップページ(index.html)を作ったときと同様、`templates/blogs`の中にhtmlファイルを作ってください。今回作るのは、detail.htmlというファイルになります。

記事の詳細ページですので、まずは記事が持つ情報を全て表示させてみましょう。Blogインスタンスは「タイトル、本文、作成日、更新日」のフィールドを持っています。「インスタンス.フィールド名」で情報を表示させていきましょう。先ほど作ったView関数で変数blogにインスタンスを代入していますので、「blog.フィールド名」で表示できます。

(~/DjangoBros/django_blog/blogs/templates/blogs/detail.html)

```
{{ blog.title }}
{{ blog.text }}
{{ blog.created_datetime }}
{{ blog.updated_datetime }}
<a href="{% url 'blogs:index' %}">トップページに戻る</a>
```

これでブログ記事の情報が表示されたと思います。urls.pyでトップページにはindexという名前をつけていますので、トップページに戻るURLは `{% url 'blogs:index' %}` で表現できます。

記事が取得できない場合は404ページを表示する

記事詳細ページの内容を表示させることができましたが、詳細ページのURLに、まだ存在していない記事のIDを入力するとエラーが出ると思います。例えば、id=100の記事が存在しないのに

`http://127.0.0.1:8000/detail/100` にアクセスするとエラーになります。

こういった場合は、エラー画面を出すのではなく、「指定されたIDの記事は存在しませんでした」という内容のメッセージを画面に出してあげるべきです。こういったページを**404ページ**と呼びます。

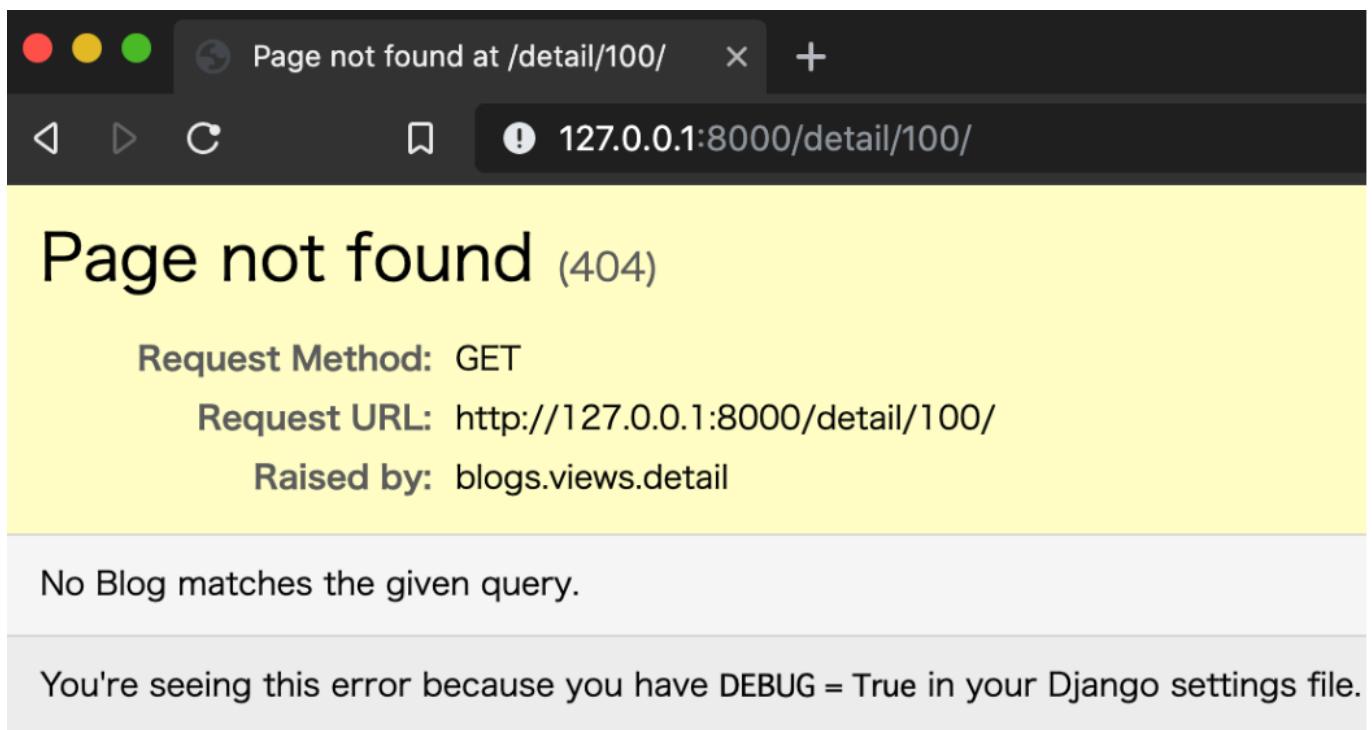
Djangoには特定のインスタンスの存在確認と取得を同時にやってくれる**get_object_or_404**という便利なメソッドがあります。views.pyを以下のように書き換えましょう。

(~/DjangoBros/django_blog/blogs/views.py)

```
from django.shortcuts import render, get_object_or_404 # 追加
from .models import Blog

...
def detail(request, blog_id):
    blog = get_object_or_404(Blog, id=blog_id) # 修正
    return render(request, 'blogs/detail.html', {'blog': blog})
```

この変更によって、ブログ記事が存在しないIDを指定された時には、404ページを表示してくれるようになります。



404ページのデザインを自分でカスタマイズすることもできます。ブログ記事で実装方法を紹介しています。

- Djangoでカスタマイズした404ページを表示する

基本的な情報の表示の仕方は理解できたでしょうか?最後に、ページのデザインを整えておきましょう。

(~/DjangoBros/django_blog/blogs/templates/blogs/detail.html)

```
<!DOCTYPE html>
<html>
<head>
  <title>ブログサイト</title>
</head>
<body>
  <h1 style="text-align: center;">{{ blog.title }}</h1>
  <div style="width: 70%; margin: 0px auto;">
    <div style="margin: 60px 20px;">
      {{ blog.text }}
    </div>
    <div style="text-align: center; margin-top: 50px">
      <a href="{% url 'blogs:index' %}">トップページに戻る</a>
    </div>
  </div>
</body>
</html>
```

3.8 CRUDを理解しよう

この章では、ブログアプリの作成を通して、Djangoにおける基本的なWeb開発のスキルを学びました。

最後の節では、Web開発に共通して重要な概念である**CRUD**について学びましょう。

Create



Read



Update



Delete



CRUDとは、**Create**、**Read**、**Update**、**Delete**の4つの操作を指します。これらは、Web開発における基本的な機能を表したものです。

たとえば、今回のブログアプリでも、記事を新しく作成して投稿したり(Create)、個別の記事のデータを読み込んで表示したり(Read)、記事の内容の更新や削除(Update、Delete)という操作が必要になります。

皆さんご普段利用している多くのWebサービスやアプリでも、このCRUDを基本とした操作がたくさんあると思います。DjangoのようなWebフレームワークでは、このCRUDをベースにした操作が簡単に実装できるようになっています。

これまでのチュートリアルでは、記事を表示するための機能(Read)を実装したのと、Admin画面からではありますが、新規記事の作成(Create)もできるようになっています。

このあとは、ブログアプリの要件で必要になる「新規ブログ記事の投稿」と「記事の編集・削除」をWebページ上からできるようにしながら、DjangoでのCRUDの基本を学びます。

HTTPメソッドを理解する

3.6節ではコマンドラインからBlogインスタンスの作成を行いましたが、この操作をブラウザ上に表示される投稿フォームからできるようにしたいです。そのためにはまず、ブラウザとサーバー間の通信プロトコルである**HTTP**について理解しましょう。

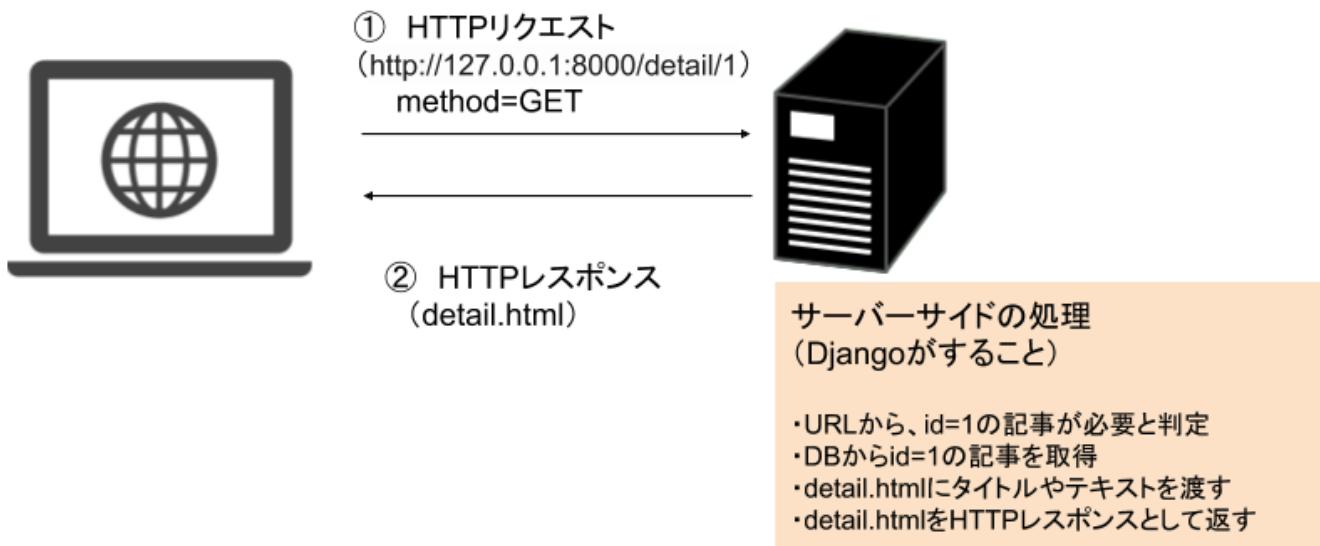
ブラウザとサーバーの通信では、HTTPというプロトコルが使われています。プロトコルとは規約という意味があり、「HTTPという決まった方式・規約に従ってブラウザとサーバーがやり取りをしている」ということです。

ユーザーはブラウザ上で特定のURLにアクセスすることによって、サーバーに対して**HTTPリクエスト**を送っています。リクエストを受け取ったサーバーは、その内容に基づいて情報を返します(**HTTPレスポンス**)。

例えば、今回のブログアプリでは、ユーザーが `http://127.0.0.1:8000/detail/1` にアクセスした場合、サーバーに対して「id=1の記事を読みたい」というリクエストが送られます。リクエストを受け取ったサーバーは、URLからid=1の記事を返す必要があると判断します。そしてデータベースからid=1のブログ記事を探し出し、その情報をdetail.htmlに載せてブラウザに返します。このサーバー上の処理をDjangoで実装しているということです。

この例の場合、情報を読み取って返しているので、CRUDの中のReadに該当する機能となります。

Read(読み取り)



HTTPリクエストの中には様々な情報が含まれていますが、その中にはmethod(メソッド)という項目があります。メソッドは、そのHTTPリクエストがどんな種類のものであるかを表します。

上の図にある**GET**メソッドは、サーバーからデータを返して欲しいときに使われます。ブラウザにURLを直接入力したときや、aタグで囲まれたリンクをクリックしたときなどは、サーバーに対してmethod=GETのリクエストが送られます。

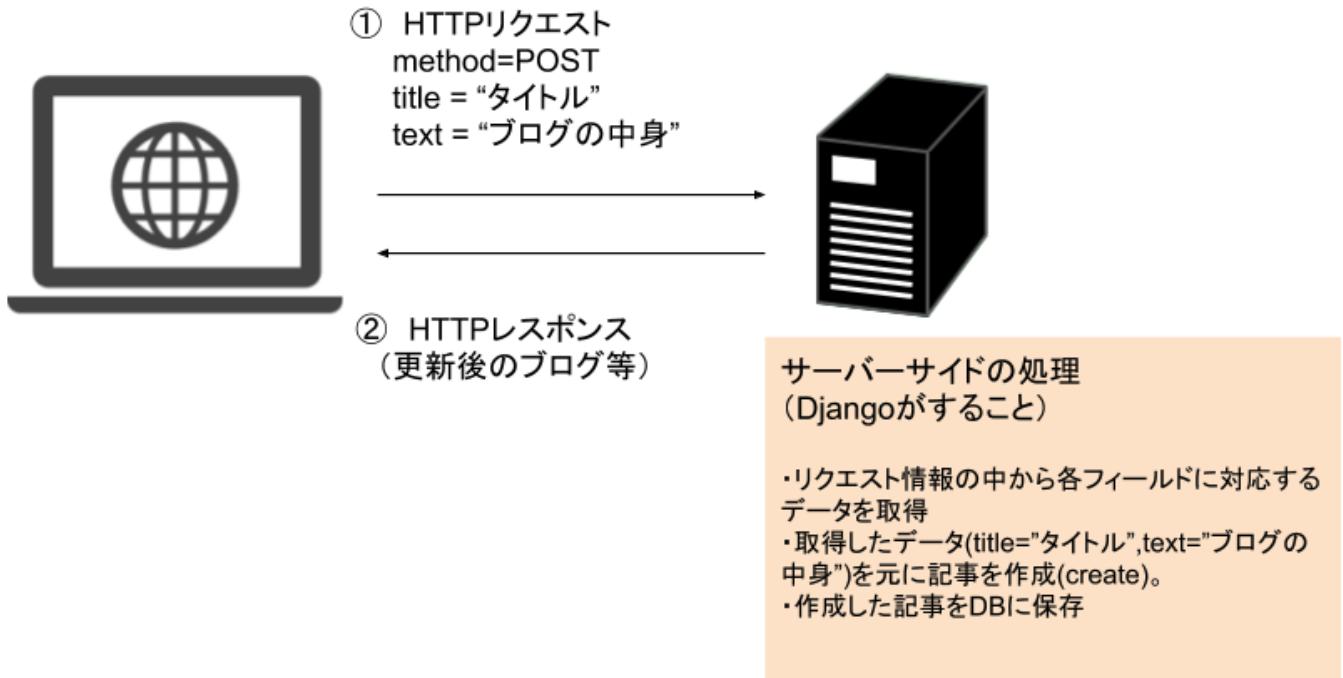
一方、サーバーに対して何かしらのデータを送りたいときは**POST**というメソッドを使います。POSTメソッドはデータを送ることによってデータベースに新しい情報を追加することができます。

HTTPリクエストには、GETとPOST以外にも種類がありますが、ほとんどの場合このどちらかを利用することになります。

- GET: 情報の読み取り、取得
- POST: 情報の送信、投稿

新しくブログ記事を作成するためには、情報をサーバーに送信する必要があるのでPOSTメソッドを使います。イメージとして下の画像のようになります。

Create(生成)



入力フォームから、ブログ記事のタイトルやテキストの情報を受け取り、サーバー側に送信します。サーバー側では、受け取った情報をデータベースに追加・保存します。この一連の流れはCRUDの中のCreateに該当します。

新規ブログ記事投稿フォームを作ろう

新規記事投稿ができるフォームを作成し、新しいブログ記事をブラウザ上から追加できるようにしましょう。

まずは、フォームを作りましょう。今回はフォーム用のファイルを作成します。blogsアプリのディレクトリ内に forms.py というファイルを作成します。

(~/DjangoBros/django_blog/blogs/forms.py)

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

class BlogForm(ModelForm):
    class Meta:
        model = Blog
        fields = ['title', 'text']
```

1行目で、Djangoがデフォルトで用意している**ModelForm**をインポートし、これを継承させて**BlogForm**という名前のフォームを作っています。ModelFormは各々のモデルに対応したフォームを作ってくれます。今回の場合は、`model = Blog` とすることでBlogモデルに対応したフォームを生成しています。

Blogモデルはtitleやtextというフィールドを持っているので、これらに対応する入力欄があるフォームが自動で作成されます。fieldsの部分で表示する入力欄を定義しています。たとえば、`fields = ['title']` とすると、タイトルだけ入力できるフォームとなります。

もちろん、モデルをベースとしないプレーンなフォームを作成することもできます。お問い合わせフォームなど、モデルとは関係のないフォームが必要なときはプレーンなフォームを使用します。今回のようにモデルと関連づいたフォームを作成したいときは、ModelFormを継承するのが便利です。

フォームを表示するページを作成する

続いて、作成したフォームをHTMLで表示させましょう。まずは、views.pyを以下のように編集します。

(~/DjangoBros/django_blog/blogs/views.py)

```
from django.shortcuts import render
from .models import Blog
from .forms import BlogForm # 追加

...
# 追加
def new(request):
    form = BlogForm
    return render(request, 'blogs/new.html', {'form': form})
```

forms.pyファイルからBlogFormをインポートして、変数formに代入しています。そして、render関数の第3引数でそれをテンプレートに渡すように設定しています。これで、new.htmlでは、`{{ form }}` という記述でフォームが表示されるようになります。

URLと紐づけるためにurls.pyも修正します。

(~/DjangoBros/django_blog/blogs/urls.py)

```
from django.urls import path
from . import views

app_name = 'blogs'
urlpatterns = [
    path('', views.index, name='index'),
    path('detail/<int:blog_id>/', views.detail, name='detail'),
    path('new/', views.new, name='new'), # 追加
]
```

new.htmlを作成し、以下のコードを書きます。

(~/DjangoBros/django_blog/blogs/templates/blogs/new.html)

```

<!DOCTYPE html>
<html>
<head>
  <title>ブログ作成</title>
</head>
<body>
  <h1 style="text-align: center;">新規記事作成</h1>
  <div style="width: 70%; margin: 0px auto;">
    <div style="margin: 60px 20px;">
      <form action="{% url 'blogs:new' %}" method="POST">{{ csrf_token }} {{ form.as_p }}
        <button type="submit" class="btn">保存</button>
      </form>
    </div>
    <div style="text-align: center; margin-top: 50px">
      <a href="{% url 'blogs:index' %}">トップページに戻る</a>
    </div>
  </div>
</body>
</html>

```

HTMLの `<form>` タグで `{{ form }}` を囲うことで、BlogFormをフォームとして表示します。`.as_p` をつけることで入力欄ごとにpタグで囲われることになるので、改行されて綺麗に表示してくれるようになります。

フォームの `action` の部分では、このフォームが投稿されたときに実行される処理を定義しています。「投稿されたとき」というのは、`type="submit"` のボタンが押されたときのことです。

上の例だと、保存ボタンが押された時に `"{% url 'blogs:new' %}"` に対してHTTPリクエストが送られることになります。この時、`method="POST"` とあるようにHTTPメソッドはPOSTなので、フォームで入力したデータも一緒にサーバーに送られます。

`{% csrf_token %}` はセキュリティ対策のために必要なものです。これがないとエラーになるので気をつけましょう。

CSRFの詳細については、DjangoBrothersのブログで紹介しています。

- [CSRF\(クロスサイトリクエストフォージェリ\)の意味と対策方法](#)
- [【Django】 csrf_tokenの仕組みとCSRF無効化・画面カスタマイズする方法](#)

ここまで設定で、`http://127.0.0.1:8000/new` でフォームが表示されるようになっているはずです。

新規記事作成

Title: _____

Text:

保存

[トップページに戻る](#)

Titleが空欄の状態で保存ボタンを押すと「このフィールドを入力してください。」といったエラーメッセージが表示されると思います。このBlogFormはBlogモデルに対応したフォームであり、BlogモデルのTitleフィールドはblank=Falseと設定されているので、Blogインスタンスの作成時にTitleフィールドが空欄になっているのを許容しません。

一方、Textは空欄の状態を許容していますので、空欄のままでも投稿することができます。これで、「ModelFormはモデルに対応したフォームを作成する」の意味がわかったのではないですか。

まだ、サーバー側の処理は定義していませんので、保存ボタンを押してもページがリロードされるだけだと思います。views.pyファイルを編集して、保存処理ができるように設定しましょう。

フォームから受け取った情報を保存する

ここまでで、フォームから入力されたデータをサーバーに送ることができました。あとは、サーバー側で受け取った情報を保存する処理ができれば完了です。

保存ボタンが押された時は、actionに指定した `"{% url 'blogs:new' %}"`、つまり blogs/view.py の new 関数が実行されます。new 関数は以下のように編集してください。

(~/DjangoBros/django_blog/blogs/views.py)

```
from django.shortcuts import render, redirect, get_object_or_404 #  
redirect を追加
```

...

```
# 修正
def new(request):
    if request.method == "POST":
        form = BlogForm(request.POST)
        if form.is_valid():
            form.save()
            return redirect('blogs:index')
    else:
        form = BlogForm()
    return render(request, 'blogs/new.html', {'form': form})
```

まず、if文でHTTPリクエストのメソッドがPOSTの時と、それ以外(else)のときで実行する処理を切り分けています。それ以外の時というのは、基本的にmethod=GETの場合を指しています。

method=GET(通常のアクセス時)のときは、特に変更を加えていないのでこれまで定義していた処理をそのまま実行します。つまり、`http://127.0.0.1:8000/new`というURLに直接アクセスされた場合やリンクで飛んできたときは、ただ単にフォームを表示させる処理だけを実行します。

続いて、`request.method=="POST"`のときの処理について説明します。メソッドがPOSTということはつまり、フォームの「保存」ボタンが押された時の処理に当たります。

`form = BlogForm(request.POST)`では、新しいBlogインスタンスを生成して変数formに代入しています。引数に`request.POST`を指定していますが、request.POSTにはユーザーがフォームに入力した情報が含まれていますので、その情報をもとに新しいBlogインスタンスを生成するという処理になります。

例えばユーザーが、タイトル「今日はキャンプ」、テキスト「家族でキャンプに行った」という内容でフォームを保存した場合、その内容でBlogインスタンスが生成されます。**request.POSTを引数にとるだけでこのような処理ができるてしまうのは、BlogFormがModelFormを継承して作られた特別なフォームだからです。**

ModelFormはモデルの設計に合わせたフォームを自動で作成してくれるので、そこから送信されたリクエストのデータを使うだけで楽にインスタンスの作成ができます。

注意点として、この段階ではインスタンスは一時的に生成されただけでまだデータベースに保存されていません。

`form.save()`のように、**save()メソッドが呼び出されたタイミングで初めてデータベースに保存されます。**

`if form.is_valid()`では、生成されたBlogインスタンスが正しい値を持っているかを検証しています。Blogモデルは、タイトルは150文字以内で、空欄ではいけないといった条件を指定していますが、インスタンスがこの条件を満たしているかを判定しているのです。

最後のredirect関数では、**次に実行する処理**を指定しています。render関数と似ていますが、render関数では表示するテンプレートファイルを指定するのに対して、redirect関数ではURLパスを指定します。書き方も、render関数では`'blogs/new.html'`のようにスラッシュを使ってHTMLファイルへのパスを指定するのに対して、redirect関数では、`'blogs:index'`のように指定するため、書き方には注意が必要です。

`'blogs:index'`のURLでは、index関数が実行される設定になっています(urls.pyでの設定)。そのため記事を投稿すると、トップページに遷移されて、投稿した記事の一覧が表示されていることが確認できるはずです。

これで、Webブラウザ側からもブログ記事を作成できるようになりました!これがCRUDのCreateの基本的な処理の流れです。

最後に、トップページに新規記事投稿ページへのリンクを追加しておきましょう。bodyタグの中身だけ記載しておきます。

(~/DjangoBros/django_blog/blogs/templates/blogs/index.html)

```
<body>
  <h1 style="text-align: center;">My Blog</h1>
  <div style="width: 70%; margin: 0px auto;">
    <!-- 追加 -->
    <a href="{% url 'blogs:new' %}">
      <button class="btn">新規記事作成</button>
    </a>
    <hr />
    {% for blog in blogs %}
    <div>
      <h3>{{ blog.title }}</h3>
      <div>{{ blog.text | truncatechars:100 }}</div>
      <div style="text-align: right;">
        <a href="{% url 'blogs:detail' blog_id=blog.id %}">記事を読む</a>
      </div>
    </div>
    <hr />
    {% endfor %}
  </div>
</body>
```

続いて、各記事の編集と削除の方法(Update、Delete)について学んでいきましょう。

3.9 記事の削除と更新を実装しよう

記事を作成できるようになったので、すでに存在している記事の削除と更新の機能を作成しましょう。これが終われば、CRUDについて一通り学べたことになります。

Delete機能を実装しよう

まずは、Delete(削除)機能から作りたいと思います。Delete機能の実装方法は、削除する記事のIDを指定して、`delete()`メソッドを使うだけです。

削除機能を実装するために、`urls.py`と`views.py`を以下のように編集してください。削除するときは、特にページを表示する必要はないので、テンプレートファイルを新しく作る必要はありません。

(~/DjangoBros/django_blog/blogs/urls.py)

```
from django.urls import path
from . import views
```

```
app_name = 'blogs'
urlpatterns = [
    path('', views.index, name='index'),
    path('detail/<int:blog_id>/', views.detail, name='detail'),
    path('new/', views.new, name='new'),
    path('delete/<int:blog_id>/', views.delete, name='delete'), # 追加
]
```

1番最後にpathを追加しました。この設定により、`/delete/<数字>/` というURLにアクセスしたときにdelete関数が実行されるようになります。

(~/DjangoBros/django_blog/blogs/views.py)

```
from django.shortcuts import render, redirect, get_object_or_404
...
# 追加
def delete(request, blog_id):
    blog = get_object_or_404(Blog, id=blog_id)
    blog.delete()
    return redirect('blogs:index')
```

こちらの処理では、該当するIDの記事を取得し、それをdeleteメソッドで削除した上で、トップページにリダイレクトするようになっています。

これで、`http://127.0.0.1:8000/delete/1` にアクセスしたときにid=1の記事が削除されるようになりました。

しかし、ここで注意が必要です!URLを打ち込んだだけで記事が削除されてしまうのは、本当はまずいですよね。URLに直接アクセスした時など、ユーザーが意図しない時に記事が削除されてしまう恐れがあります。また、URLにアクセスする処理は、HTTPメソッドでいうとGETに当たります。GETは情報の読み取りをする時に使われるものなので、削除機能のようにデータベースに変更が加わる場合に使うべきではありません。

正しくは、URLにアクセスした時ではなく、記事詳細ページにある「削除」ボタンが押された時のみに記事が削除されるべきです。別の言い方をすれば、HTTPメソッドがGETの時は何の処理もしないように設定し、POSTメソッドのときにだけ削除機能が実行されるようにする必要があります。

これを実現するには、以下のように関数の上に `@require_POST` を追加します。インポート文も忘れないでください。

(~/DjangoBros/django_blog/blogs/views.py)

```
from django.views.decorators.http import require_POST # 追加
...
# 追加
@require_POST
```

```

def delete(request, blog_id):
    blog = get_object_or_404(Blog, id=blog_id)
    blog.delete()
    return redirect('blogs:index')

```

これにより、delete関数はGETメソッドでは実行されません。`http://127.0.0.1:8000/delete/1`にアクセスしても記事が削除されないので、delete関数が実行されるには、リクエスト送信時のHTTPメソッドがPOSTである必要があります。

リクエスト時のメソッドをPOSTにするためには、detail.htmlに以下のような削除ボタンを追加します。bodyタグの中身だけ記述します。

(~/DjangoBros/django_blog/blogs/templates/blogs/detail.html)

```

<body>
    <h1 style="text-align: center;">{{ blog.title }}</h1>
    <div style="width: 70%; margin: 0px auto;">
        <div style="margin: 60px 20px;">{{ blog.text }}</div>

        <!-- 追加 -->
        <div style="text-align: center; margin-top: 50px">
            <form method="POST" action="{% url 'blogs:delete' blog.id %}">
                {% csrf_token %}
                <button class="btn" type="submit" onclick='return confirm("本当に削除しますか?");'>削除</button>
            </form>
        </div>

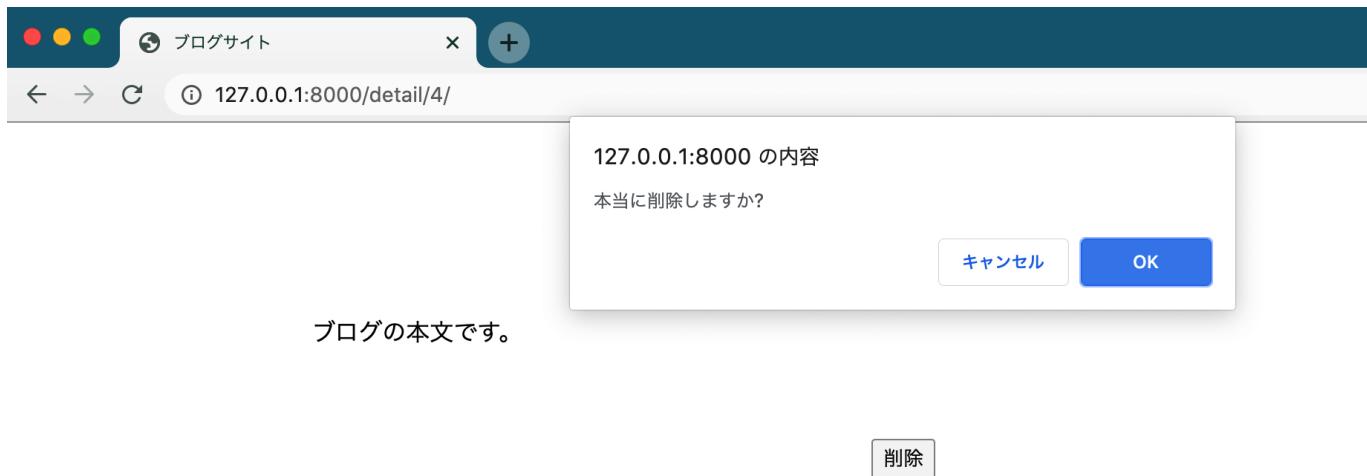
        <div style="text-align: center; margin-top: 50px">
            <a href="{% url 'blogs:index' %}">トップページに戻る</a>
        </div>
    </div>
</body>

```

`type="submit"`としたボタンをformタグで囲むことによって、methodを指定できるようにしています。actionでは、`'blogs:delete'`と指定していますので、削除ボタンが押された時は、HTTPメソッドがPOSTの状態で`http://127.0.0.1:8000/delete/(数字)`にリクエストが送られることになります。POSTの時は、ちゃんとdelete関数が実行されるので記事も正常に削除されるというわけです。

`action="{% url 'blogs:delete' blog.id %}"`のblog.idの部分では、`path('delete/<int:blog_id>', views.delete, name='delete')`の`<int:blog_id>`に渡す数字を指定しています。これがないと、サーバー側はどの記事を削除してよいか判断できないので忘れないようにしましょう。

`onclick`ではボタンがクリックされたときにダイアログを出し、本当に削除して問題ないかをユーザーに確認しています。これがないと、ボタンが1回押されただけで記事が即削除されてしまうことになります。ユーザーが保持するデータを削除するということは、Webサービスの運営上とてもデリケートなアクションですので、このように確認を取れるようにすると良いでしょう。



[トップページに戻る](#)

これで削除機能は完了です。続いて編集機能を追加しましょう。

Update機能を実装しよう

編集機能は、新規投稿機能を作った時とほぼ同じ流れで実装することができます。編集用のページにフォームを用意し、そこから編集内容を投稿できるようにします。

まずは、編集画面を作成しましょう。urls.pyには編集画面用のパスを追加してください。

(~/DjangoBros/django_blog/blogs/urls.py)

```
...
urlpatterns = [
    path('', views.index, name='index'),
    path('detail/<int:blog_id>/', views.detail, name='detail'),
    path('new/', views.new, name='new'),
    path('delete/<int:blog_id>/', views.delete, name='delete'),
    path('edit/<int:blog_id>/', views.edit, name='edit'), # 追加
]
```

views.pyのedit関数は以下のようになります。

(~/DjangoBros/django_blog/blogs/views.py)

```
...
def edit(request, blog_id):
    blog = get_object_or_404(Blog, id=blog_id)
    form = BlogForm
    return render(request, 'blogs/edit.html', {'form': form, 'blog': blog})
```

edit.html は新たに作成してください。

(~/DjangoBros/django_blog/blogs/templates/blogs/edit.html)

```
<!DOCTYPE html>
<html>
<head>
    <title>ブログ編集</title>
</head>
<body>
    <h1 style="text-align: center;">記事編集</h1>
    <div style="width: 70%; margin: 0px auto;">
        <div style="margin: 60px 20px;">
            <form action="{% url 'blogs:edit' blog.id %}" method="POST">
                {% csrf_token %}
                {{ form.as_p }}
                <button class="btn" type="submit">保存</button>
            </form>
        </div>
        <div style="text-align: center; margin-top: 50px">
            <a href="{% url 'blogs:detail' blog.id %}">記事に戻る</a>
        </div>
    </div>
</body>
</html>
```

detail.htmlには編集ページへのリンクも追加しておきます。bodyタグ内だけ記載します。

(~/DjangoBros/django_blog/blogs/templates/blogs/detail.html)

```
<body>
    <h1 style="text-align: center;">{{ blog.title }}</h1>
    <div style="width: 70%; margin: 0px auto;">
        <div style="margin: 60px 20px;">
            {{ blog.text }}
        </div>
        <div style="text-align: center; margin-top: 50px">

            <!-- 追加 -->
            <a href="{% url 'blogs:edit' blog.id %}">
                <button class="btn">編集</button>
            </a>

            <form method="POST" action="{% url 'blogs:delete' blog.pk %}">
                {% csrf_token %}
                <button class="btn" type="submit" onclick='return confirm("本当に削除しますか?");'>
                    削除
                </button>
            </form>
        </div>
    </div>
```

```
<div style="text-align: center; margin-top: 50px">
    <a href="{% url 'blogs:index' %}">トップページに戻る</a>
</div>
</body>
```

これで編集ページと、編集ページへの画面遷移を作ることができました。「編集」ボタンを押すと、

`http://127.0.0.1:8000/edit/1/` のようなURLに遷移しフォームが表示されるはずです。

しかし、編集画面というのは既に保存されてあるブログ記事の内容を修正するページですので、フォームを空欄のまま表示するのではなく、もともと保存されてあった内容が表示されるようにしておくべきです。

views.py を以下のように修正しましょう。

(~/DjangoBros/django_blog/blogs/views.py)

```
def edit(request, blog_id):
    blog = get_object_or_404(Blog, id=blog_id)
    form = BlogForm(instance=blog) # 修正
    return render(request, 'blogs/edit.html', {'form': form, 'blog':
blog})
```

BlogFormに`instance`という引数を加えました。その値として、今回編集するBlogインスタンスを指定しています。このようにすることで、指定したインスタンスに対応したデータが入力された状態でフォームが作成されます。変更を保存したら、編集画面を再度表示してみましょう。

うまく入力欄に元々の値が表示されましたか?このようにModelFormは、`ModelForm(instance=インスタンス)` とすることで、あらかじめ特定のインスタンスの値を持ったフォームを表示させることができます。

編集画面の表示はこれで完成しました。あとは、新規投稿ページと同じように、保存ボタンが押された時(HTTPメソッドがPOSTのとき)の処理を加えましょう。

再度、edit関数を編集します。

(~/DjangoBros/django_blog/blogs/views.py)

```
def edit(request, blog_id):
    blog = get_object_or_404(Blog, id=blog_id)
    if request.method == "POST":
        form = BlogForm(request.POST, instance=blog)
        if form.is_valid():
            form.save()
            return redirect('blogs:detail', blog_id=blog_id)
    else:
        form = BlogForm(instance=blog)
    return render(request, 'blogs/edit.html', {'form': form, 'blog':
blog})
```

新規記事投稿ページを作成する際には、`form = BlogForm(request.POST)` とすることで、フォームから受け取った情報をもとに新しいBlogインスタンスを生成していました。しかし、記事の編集ページの場合は、すでに特定のブログ記事のインスタンスが存在しているため、どのブログ記事インスタンスに対してフォームを作用させるかを指定する必要があります。そのため、`form = BlogForm(request.POST, instance=blog)` のように、`instance`を引数にとって上書きする必要があるのです。

あとは、フォームのフィールドの値のバリデーション(正しいかどうかのチェック)を行い、正常であればデータベースに保存する処理を実行します。

お疲れ様です!これで今回のチュートリアルで必要な機能は全て実装することができました!

このチュートリアルでは、Webサービスの開発の基本の概念と、CRUDの基礎的な実装を学習しました。ここで学んだ内容は今後のチュートリアルや実際のWeb開発プロジェクトでも必須の知識になるので、何度か復習してきちんと身につけておきましょう。

次の章では、今回のチュートリアルからステップアップして、画像データの取り扱いや、ユーザー登録・アカウント管理の基本を学びます。

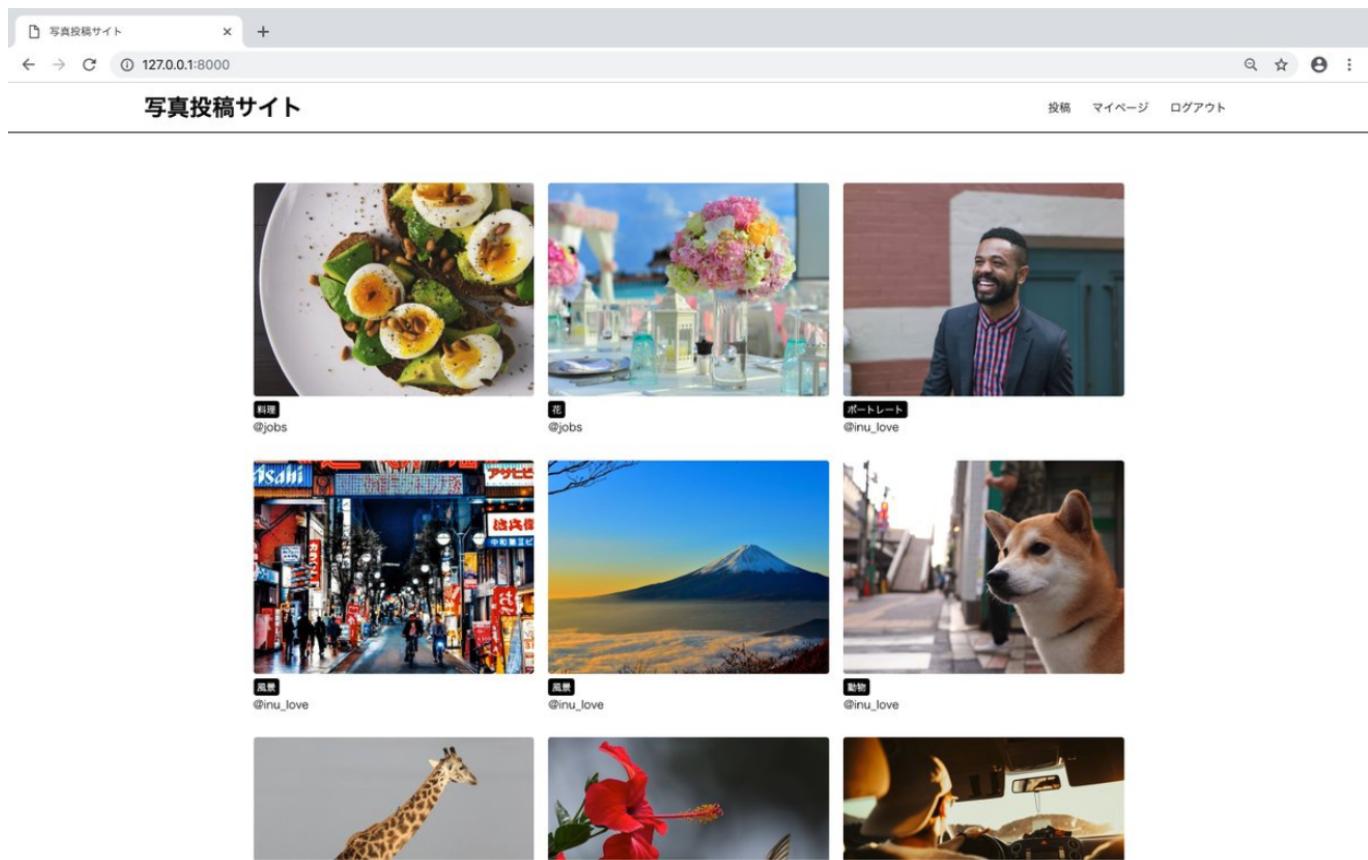
第4章 チュートリアル2 写真投稿サイト

この章では、写真投稿サイトを作成します。このサイトを訪れた人は、他の人が投稿した画像を閲覧することができます。また、カテゴリーごとに写真を絞り込むことも可能です。会員登録・ログインをすれば自分で好きな画像を投稿することもできます。

前章までに学んだDjango開発の基本的な部分を復習しながら、画像データの扱い方やユーザー登録、より複雑なモデルの作成などを身につけましょう。特に、ユーザーの識別ができるようになると、ユーザーごとに表示するコンテンツを変えたり、ユーザー同士での交流が出来たりと、Webサービスに必須とも言える機能の実装ができるようになります。

完成イメージは以下のようになります。

- トップページ



- ユーザーごとのページ

投稿が完了しました！

@jobs

投稿4件

料理
@jobs

花
@jobs

動物
@jobs

動物
@jobs

- カテゴリーで絞り込んだページ

カテゴリー：動物 の検索結果

動物
@inu_love

動物
@abc_defg

動物
@abc_defg

動物
@jobs

動物
@jobs

こちらのチュートリアルでも、第2章と同じ手順でプロジェクトを作成してスタートします。プロジェクト名は「PhotoService」にして開発を進めていきます。

4.1 開発準備をしよう

この節では、第2章と同じ手順で「PhotoService」という名前のプロジェクトを作成し、基本的な設定を行います。

まずは、プロジェクト内に今回の機能を実装するアプリケーションを作成しましょう。今回は簡易的に「app」という名前のアプリケーションにすべての機能を収めることにします。

(~/)

```
# アプリケーションの作成
$ cd PhotoService # PhotoServiceという名前のプロジェクトに移動する
$ python manage.py startapp app
```

(~/PhotoService/PhotoService/settings.py)

```
# アプリケーションの追加
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'app', # 追加
]
```

管理用のユーザーアカウントを作成します。

(~/PhotoService/)

```
# superuserの作成
$ python manage.py migrate
$ python manage.py createsuperuser
```

トップページを以下のように作成しましょう。

URLの設定

(~/PhotoService/PhotoService/urls.py)

```
from django.contrib import admin
from django.urls import path, include # 追加

urlpatterns = [
    path('admin/', admin.site.urls),
```

```
    path('', include('app.urls')), # 追加
]
```

(~/PhotoService/app/urls.py)

```
from django.urls import path
from . import views

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

Viewの設定

(~/PhotoService/app/views.py)

```
from django.shortcuts import render

def index(request):
    return render(request, 'app/index.html')
```

Templateの設定

(~/PhotoService/app/templates/app/index.html)

```
<!DOCTYPE html>
<html>
<head>
    <title>写真投稿サイト</title>
</head>
<body>
    <header>
        <div class="container">
            <h1><a href="{% url 'app:index' %}">写真投稿サイト</a></h1>
            <div class="header-menu">
                <a href="">投稿</a>
            </div>
        </div>
    </header>
    <div class="container">
        <h2>トップページ</h2>
    </div>
</body>
</html>
```

ここまで設定で、トップページにアクセスすると簡易的な画面が表示されていると思います。

4.2 Template拡張を利用する

この節ではTemplate拡張について学習します。

Djangoで開発する上では、テンプレートファイル（HTML）をたくさん作ることになるでしょう。そして、その全てのページで共通して表示させたい部分が出てくるはずです。サービス名（サイト名）やヘッダー、フッターは基本的にどのページにも表示されることが多いので、その例となります。

皆さんおなじみのWebサービスでも、ヘッダーやフッター、サイドバーなどの部分は、各ページで共通していったり、内容が異なっていても同じデザインを使いまわしていたりすることがあります。

<header> タグや <footer> タグをコピーして全てのファイルにペーストしていく、というやり方も考えられますが、何度も同じ作業を繰り返すのはとても面倒ですし、1つの部分を修正するために各ファイルの該当箇所を全て修正する必要があるのはWebサービスの運用上現実的ではありません。

そこで、Djangoでは共通部分を1つのファイルにまとめ、そのファイルを別のファイルで使い回すことができる機能があります。ヘッダーやフッターは、ベースとなるファイルにまとめてしまい、それを各ファイルで利用するのです。これをテンプレート拡張やテンプレートシステムなどと呼びます。

具体的なコードを見ながら理解していきましょう。

先ほど作成したindex.htmlファイルの中には、<body> タグや <header> タグなどの、複数のページに共通で利用できそうな部分が含まれています。この部分をbase.htmlというファイルに書き出して再利用できるようにしましょう。

appアプリケーションのテンプレートディレクトリの中にbase.htmlというファイルを作成してください。

(~/PhotoService/app/templates/app/base.html)

```
{% load static %}

<!DOCTYPE html>
<html>
<head>
    <title>写真投稿サイト</title>
    <link rel="stylesheet" type="text/css" href="{% static 'css/style.css' %}">
</head>
<body>
    <header>
        <div class="container">
            <h1><a href="{% url 'app:index' %}">写真投稿サイト</a></h1>
            <div class="header-menu">
                <a href="">投稿</a>
            </div>
        </div>
    </header>
    <div class="container">
        {% block content %}{% endblock %}
    </div>
</body>
</html>
```

base.htmlファイルは、あらゆるテンプレートファイルのベースとなるものですので、`<html>`、`<head>`、`<body>`タグなどの全ページに共通して必要とされるものを記述します。今回は、「写真投稿サイト」というサイト名も全ページに表示させたいので、`<h1>% url 'app:index' %">写真投稿サイト</h1>`と書いています。

また、index.htmlに記述していた`<h2>トップページ</h2>`という部分を`{% block content %}{% endblock %}`という表示に置き換えていました。この部分で、他のファイルで定義されている中身を取り込みます。

トップページ表示用のindex.htmlを以下のように修正してください。

(~/PhotoService/app/templates/app/index.html)

```
{% extends 'app/base.html' %}

{% block content %}
<h2>トップページ</h2>
{% endblock %}
```

1行目の、`{% extends 'app/base.html' %}`で、このファイルがbase.htmlを拡張したファイルであることを表しています。これにより、base.htmlで記述した`<h1>`タグなどがこのファイルでも表示されるようになります。

次に、index.htmlの中身のコードを、`{% block content %}`と`{% endblock %}`で囲っています。この囲った部分が、base.htmlの`{% block content %}{% endblock %}`の部分に取り込まれることになります。

このように、ベースになるファイルに共通化された部分を記述し、中身の部分はそれぞれのページ用のファイルに任せるというのが、テンプレート拡張の基本です。

テンプレート拡張を利用すると、開発効率も上がり、メンテナンスのやりやすさも格段に上がります。

今後のチュートリアルでも、テンプレート拡張を利用した記述を行います。

CSSを適用しよう

最後に、CSSを適用ていきましょう。

先ほどのテンプレート拡張の処理で、`{% load static %}`という記述について説明しませんでした。これは、CSSや画像などの静的ファイル(staticファイル)を読み込むための記述です。

CSS、Javascript、画像などのサーバー処理を伴わないファイルは静的ファイル(staticファイル)と呼びます。Djangoでは、**静的ファイルはstaticというディレクトリに格納するのが一般的です。**

実際にstaticディレクトリを作成し、CSSファイルを追加してみます。

今回は、`PhotoService/static/css/style.css`のように、プロジェクトディレクトリ直下にstaticディレクトリを作ります。テンプレートファイルのように、1つのアプリの中にstaticファイルをまとめる方法もありますが、プロジェクト全体で利用したいCSSファイルであれば、プロジェクトディレクトリ直下にstaticディレクトリを作成するのが良いでしょう。

ただし、デフォルトの状態では、アプリ内にあるstaticディレクトリしか読み込まれないようになっています。(アプリ内のstaticディレクトリが読み込まれるのは、settings.pyに以下の記述があるからです。)

(~/PhotoService/PhotoService/settings.py)

```
STATIC_URL = '/static/'
```

プロジェクトディレクトリ直下のstaticディレクトリも読み込みたい場合は、以下のように
STATICFILES_DIRS を設定します。

(~/PhotoService/PhotoService/settings.py)

```
STATIC_URL = '/static/'  
STATICFILES_DIRS = (  
    os.path.join(BASE_DIR, "static"),  
)
```

base.htmlでは `<link rel="stylesheet" type="text/css" href="{% static 'css/style.css' %}">` としています。このように static をつかってパスを指定することによって、自動的にstaticディレクトリ配下の静的ファイルを読み込んでくれます。

ここまでできたら、CSS ファイルを追加しましょう。

(~/PhotoService/static/css/style.css)

```
html, body {  
    margin: 0;  
}  
  
div {  
    box-sizing: border-box;  
}  
  
a{  
    text-decoration: none;  
    color: #000;  
}  
  
.container {  
    width: 80%;  
    margin: auto;  
    overflow: auto;  
}  
  
header {  
    border-bottom: solid 1px #000;  
    height: 70px;  
}
```

```
header .container {
  display: flex;
  justify-content: space-between;
}

body > .container {
  width: 65%;
  padding-top: 60px;
}

header h1 {
  margin: 0;
  height: 100%;
  line-height: 70px;
}

.header-menu {
  display: flex;
  align-items: center;
}

.header-menu a {
  padding: 10px;
  margin-left: 10px;
}

.photo {
  width: calc(100%/3);
  float: left;
  padding: 10px;
  margin-bottom: 15px;
}

.photo a {
  display: block;
}

.photo-info a:hover {
  text-decoration: underline;
}

.photo-info .category {
  display: inline-block;
  padding: 2px 5px;
  border-radius: 4px;
  font-size: 12px;
  color: #fff;
  background-color: #000;
}

.user-name {
  font-size: 30px;
}
```

```

.photo-img {
    width: 100%;
    max-width: 500px;
    height: 300px;
    object-fit: cover;
    border-radius: 4px;
}

.photo-img:hover {
    opacity: 0.8;
}

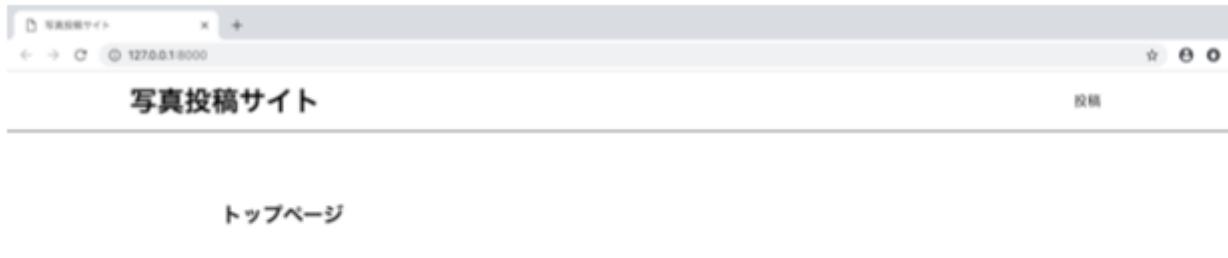
.photo-detail .photo-img{
    object-fit: contain;
}

.photo-detail .photo-img:hover {
    opacity: 1;
}

.message-success {
    background-color: #00d1b2;
    padding: 10px;
    padding-left: 30px;
    border-radius: 4px;
}

```

CSS が適用されるとトップページはこんな感じになります。



4.3 Django標準のUserモデルを使う

この節では、ユーザー登録機能やログイン機能を実装します。

ログイン機能があれば、アクセスしてきたユーザーを識別することができ、各ユーザーに合わせたページ表示などができるようになります。例えば、「アクセスしてきたユーザーが未ログイン状態の場合はログインページにリダイレクトさせる」、「ログイン済みの場合はマイページを表示させる」といった感じです。

ユーザー認証機能は、Webサービスでは頻繁に使われる機能であるため、Djangoではユーザー認証に関する様々な機能がデフォルトで提供されています。

チュートリアル①でも学習したように、普通、モデルは自分で要件を定義してmodels.pyにコードを書く必要があります。しかし、Djangoではデフォルトで使えるUserモデルを提供しています。このUserモデルを活用することで簡単に認証機能が作れるのです。

デフォルトのUserモデルにはいくつかのフィールドが定義されており、「username(ユーザー名)」、「first_name(名)」、「last_name(姓)」、「email(メールアドレス)」、「password(パスワード)」、「date_joined(登録日)」、「last_login(最終ログイン日)」などのフィールドがあります。

実は、みなさんはすでにこのUserモデルを使っています。プロジェクト立ち上げ時に、`python manage.py createsuperuser` コマンドで、管理者アカウントを作成しましたが、この管理者アカウントはUserモデルをベースに作られています。(アカウント作成時にusername、email、passwordを入力しましたよね。)

また、Userモデルについてはデフォルトの設定でAdminページで表示されるようになっていますので、実際に確認してみましょう。自分のアカウントをクリックしてみると、いろんなフィールドがあることがわかると思います。

サイト管理

Groups and Permissions

グループ

ユーザー

ここから確認する

最近行った操作

自分の操作

利用不可

ユーザーを変更 | Django サイト

ユーザー: Jobs

ホーム > 認証と認可 > ユーザー > Jobs

ユーザーを変更

ユーザー名: Jobs

この項目は必須です。半角アルファベット、半角数字、@/./-/./で150文字以下にしてください。

パスワード: アルゴリズム: pbkdf2_sha256 イテレーション: 120000 ソルト: s2NK8h..... ハッシュ: 4aMnT7.....

生のパスワードは暗號化されていません。このユーザーのパスワードを確認する方法はありません。しかしこのフォームをしようとします。パスワードを変更できます。

個人情報

名: [input field]

姓: [input field]

メールアドレス: jobs@example.com

パーミッション

有効

ユーザーがアクティブかどうかを示します。アカウントを削除する代わりに活性化してください。

スタッフ権限

ユーザーが管理サイトにログイン可能かどうかを示します。

スーパーユーザー権限

全ての権限を持っているとみなされます。

グループ: 利用可能なグループ +

The screenshot shows the Django Admin 'User Change' page for a specific user. On the left, there's a sidebar titled 'ユーザー/パーミッション' (User/Permissions) with a search bar and a list of permissions. The permissions listed include various actions on 'admin', 'auth' groups, and 'auth' users. Below this is a '選択されたユーザー/パーミッション' (Selected User/Permissions) section which is currently empty. At the bottom of the sidebar, there are buttons for '全て選択' (Select All) and 'すべて削除' (Delete All). A note at the bottom of the sidebar says: 'このユーザーの持つ権限です。複数選択するときには Control キーを押したまま選択してください。Mac では Command キーを使ってください'.

重要な日程

最終ログイン: 日付: 2019-02-23 今日 時刻: 18:48:22 現在

登録日: 日付: 2019-02-23 今日 時刻: 17:04:08 現在

Userモデルがどのような構成をしているのかをもっと具体的に知りたい場合はソースコードかドキュメントを確認しましょう。DjangoのソースコードはGitHub上に公開されています。

<https://github.com/django/django>

Userモデルは、`django.contrib.auth.models` のファイルで定義されていますので、[django/django/contrib/auth/models.py](https://github.com/django/django/blob/master/django/contrib/auth/models.py)のページから確認できます。

Userクラス自体にはほぼ何も書いてありませんが、UserクラスはAbstractUserを継承しているので、AbstractUserクラスをみると概要がわかります。

Userモデルを使ってみよう

まずは、Userモデルをshellで使ってみましょう。インポートすれば普通のモデルと同様に扱うことができます。Userモデルは `django.contrib.auth.models` で定義されていますので、最初にここからインポートします。

(~/PhotoService)

```
$ python manage.py shell

# Userモデルをインポート
>>> from django.contrib.auth.models import User

# 全てのUserインスタンスを表示
>>> User.objects.all()

# usernameを指定してインスタンスを取得
>>> user1 = User.objects.get(username='Jobs')
```

```

# user1のメールアドレスを表示
>>> user1.email
'jobs@sample.com'

# user1の登録日を表示
>>> user1.date_joined
datetime.datetime(2019, 2, 23, 8, 4, 8, tzinfo=<UTC>)

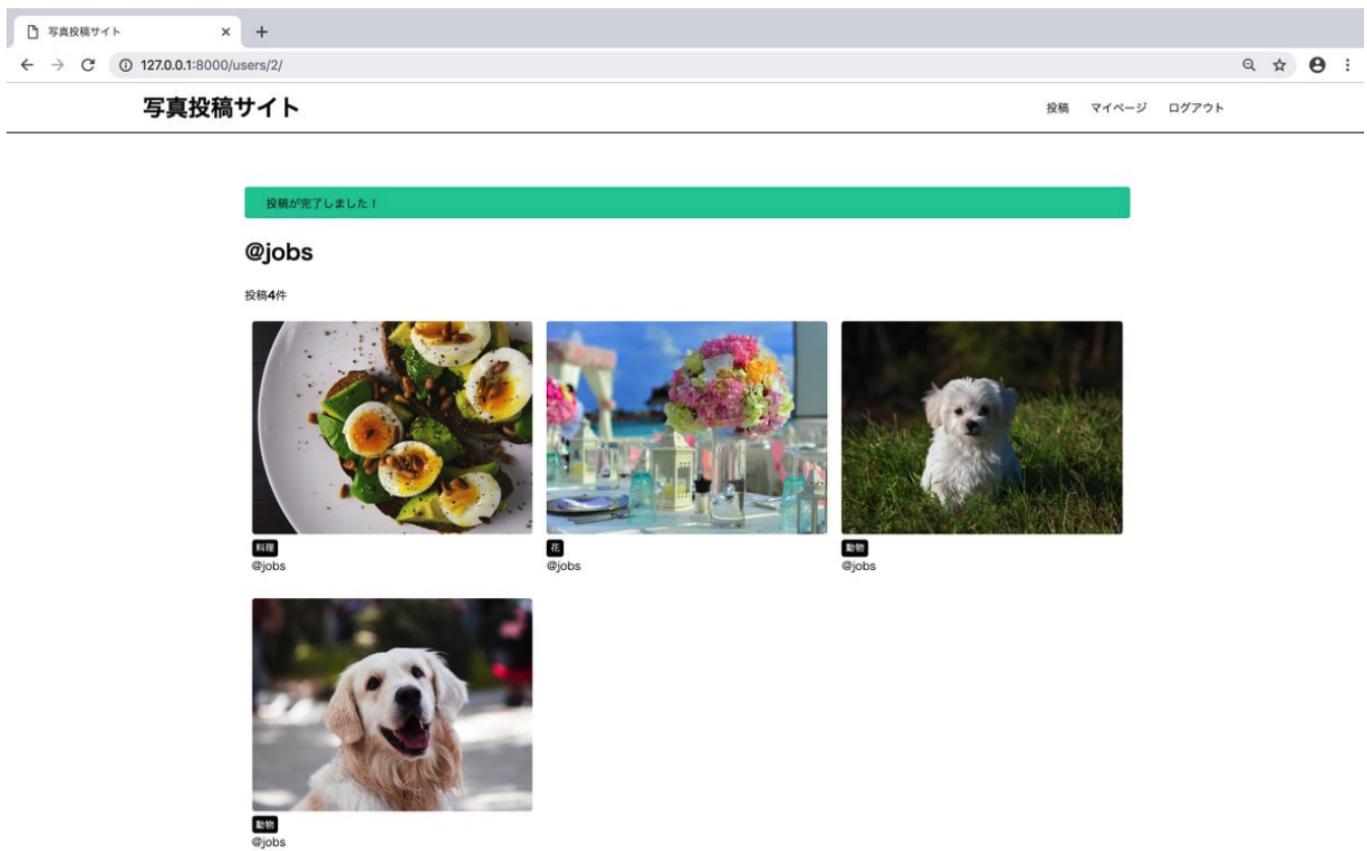
# 取得したUserがスーパーユーザー権限を持っているかの確認(True, False)
>>> user1.is_superuser
True

```

Userページを作ろう

Userモデルの概要を理解できたら、次にユーザーページを作っていきましょう。

ユーザーページの完成イメージは以下のようになります。



このページでは対応するユーザーの、名前、投稿件数、投稿した写真を表示させるようにします。

まずは、対象ユーザーのユーザー名と管理権限を持っているかどうかを画面に表示させてみます。

URLは、`http://127.0.0.1:8000/users/2/` のように、URLの中の数字が、ユーザーのIDに対応するようにします。

(~/PhotoService/app/urls.py)

```
from django.urls import path
from . import views

app_name = 'app'
urlpatterns = [
    path('', views.index, name='index'),
    path('users/<int:pk>', views.users_detail, name='users_detail'), # 追加
]
```

(~/PhotoService/app/views.py)

```
from django.shortcuts import render, get_object_or_404 # 追加
from django.contrib.auth.models import User # 追加

def users_detail(request, pk):
    user = get_object_or_404(User, pk=pk)
    return render(request, 'app/users_detail.html', {'user': user})
```

これまでに説明したように、Userオブジェクトは様々な属性を持っていますので、Templateでもこの属性にアクセスすることでデータを表示させることができます。まずは、ユーザー名を表示させてみましょう。

(~/PhotoService/app/templates/app/users_detail.html)

```
{% extends 'app/base.html' %}

{% block content %}

<h2 class="user-name">@{{ user.username }}</h2>

{% endblock %}
```

Userオブジェクトが持つ属性の中には、Boolean型(TrueかFalse)を返すものもあるので、これを使えば必要に応じて条件分岐ができます。

例えば、`is_superuser` は、そのユーザーがスーパーユーザーかどうかを示します。

(~/PhotoService/app/templates/app/users_detail.html)

```
{% extends 'app/base.html' %}

{% block content %}

<h2 class="user-name">@{{ user.username }}</h2>

{{ user.is_superuser }} <!-- 管理者の場合は True -->
```

```

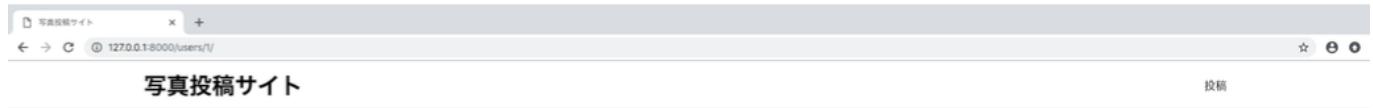
<!-- ユーザーの持つ権限によって表示を切り替え -->
{% if user.is_superuser %}
    <p>管理者</p>
{% else %}
    <p>一般ユーザー</p>
{% endif %}

{% endblock %}

```

<http://127.0.0.1:8000/users/1/> にアクセスすると以下のような表示になります。（ /1/ の部分は実際に存在するユーザーのIDを指定してください。）

(管理ユーザーでログインした場合の表示)



Userモデルがどんな属性を持っているかは、[ドキュメント](#)にも一覧としてまとまっています。よく利用するモデルなので、一通り確認しておき、実践の中で少しずつ身につけていきましょう。

4.4 ImageFieldで画像をアップロードしよう

この節では、「Photo」という名前のモデルを作ります。今回作成するサービスでは、ユーザーがタイトルやコメントを添えて写真を投稿できるようにします。この投稿1つ1つを表すモデルがPhotoモデルとなります。

Photos テーブル

	id	title	comment	image	category	user	created_at
(例)		CharField (文字列型)	TextField (テキスト型)	ImageField(画像ファイル)	ForeignKey (Category型)	ForeignKey (User型)	DateTimeField (日付型)
1	お散歩	寒いけどがんばってお散歩したよ。		dog_walk.jpeg	動物	Jobs	2019/02/24
2	初めての富士山	富士山の写真を撮ったよ。		fujisan.jpeg	風景	Rola	2019/02/25
3	美味しくできた	卵料理つくった。おいしかった。		egg_dish.jpeg	料理	Rola	2019/03/09
4							
.							
.							

新しく、**ImageField**と**ForeignKey**というものが出てきました。この節ではImageFieldについて説明します。

Photoモデルは、models.pyに定義していきます。

(~/PhotoService/app/models.py)

```

from django.db import models

class Photo(models.Model):

```

```
title = models.CharField(max_length=150)
comment = models.TextField(blank=True)
image = models.ImageField(upload_to='photos')
created_at = models.DateTimeField(auto_now=True)

def __str__(self):
    return self.title
```

`models.ImageField`は、画像ファイルを保存するフィールドであることを表しています。この段階で `models.py` を保存するとコンソールに以下のようなエラーがでると思います。

(コンソール)

ERRORS:

```
app.Photo.image: (fields.E210) Cannot use ImageField because Pillow is not
installed.
HINT: Get Pillow at https://pypi.org/project/Pillow/ or run command "pip
install Pillow".
```

エラーメッセージに書いてある通り、`ImageField`を使う場合は、**Pillow**というパッケージが必要となりますので、`pip`コマンドでインストールしておきましょう。

(~/PhotoService)

```
$ pip install Pillow
```

画像の保存先を設定しよう

`ImageField`には画像をアップロードすることができますが、アップロードする画像の保存先を設定しておく必要があります。保存先は、`settings.py`の中に `MEDIA_ROOT` というものを追記して定義します。

(~/PhotoService/PhotoService/settings.py)

```
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
MEDIA_URL = '/media/'
```

`MEDIA_ROOT`は画像の保存先を表すものです。`MEDIA_URL`については後述します。

この記述により、アップロードされた画像は`BASE_DIR`(ルートディレクトリ)直下の `media` というディレクトリに保存されることになります。(`media`ディレクトリは、1つ目の画像がアップロードされたタイミングで自動で生成されます。もしくは、最初から自分で作成しても問題ありません。)

Photoモデルでは、`image = models.ImageField(upload_to='photos')` のようにアップロード先を `photos` に指定していますので、Photoモデルの `image` フィールドからアップロードされた画像は

`PhotoService/media/photos` の中に保存されることになります。

ここまでできたら、マイグレートしてPhotoモデルをデータベースに反映し、Adminページから画像をアップロードしてみましょう。

(~/PhotoService)

```
$ python manage.py makemigrations  
$ python manage.py migrate
```

(~/PhotoService/app/admin.py)

```
from django.contrib import admin  
from .models import Photo  
  
class PhotoAdmin(admin.ModelAdmin):  
    list_display = ('id', 'title')  
    list_display_links = ('id', 'title')  
  
admin.site.register(Photo, PhotoAdmin)
```



photo を追加 | Django サイト [×](#) +

← → ⌂ 127.0.0.1:8000/admin/app/photo/add/ ようこそ JOBS サイトを表示 / パスワードの変更 / ログアウト

Django 管理サイト

ホーム : App : Photos : photo を追加

photo を追加

Title: お散歩

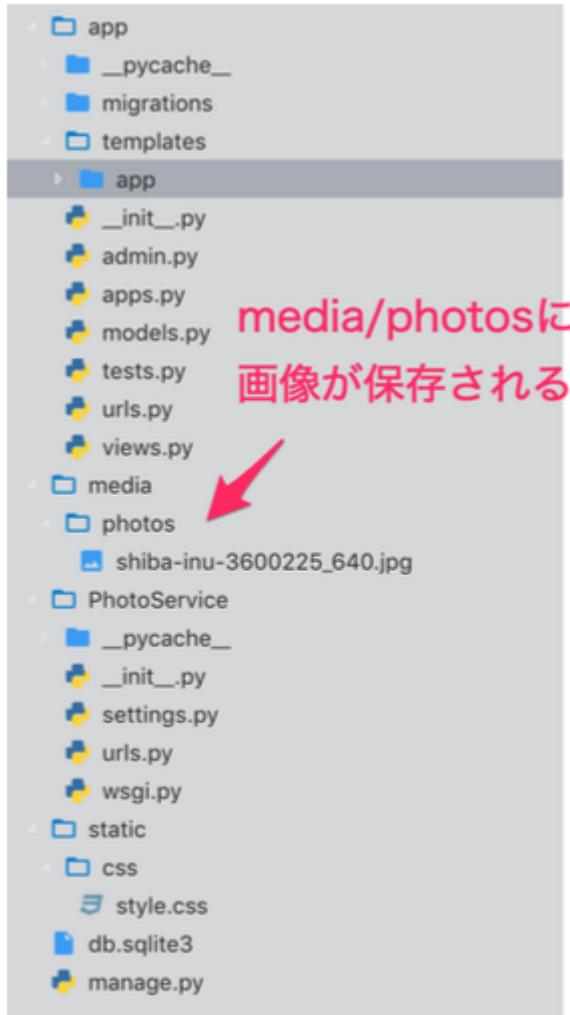
Comment: 寒いけど頑張ってお散歩したよ。|

Image: ファイルを選択 選択されていません

ここから画像を選択する

保存してもう一つ追加 保存して編集を続ける 保存

無事Photoモデルからインスタンスを作成することができたら、`/media/photos` というディレクトリが新しくできていて、その中に画像ファイルが保存されてあることを確認してみましょう。



media/photosに
画像が保存される

上のように、画像が保存されていれば無事にアップロードできています。このように画像のアップロード先となるディレクトリを指定して、画像はそのディレクトリに保存されるように設定します。

しかし、これにはちょっと注意が必要です。本番環境においては、画像保管用のサーバーを別途用意することが一般的です。つまり、開発環境下ではmediaディレクトリに、本番環境下では本番用画像サーバーにアップロードするように、環境によってアップロード先を切り替える設定をしておくと良いでしょう。今回のチュートリアルではとりあえずローカルのディレクトリに画像を保存していく形で解説を進めます。

MEDIA_URLの役割

画像のアップロードができたので次は、ImageFieldからアップロードされた画像にアクセスする方法を説明します。以下のようにフィールド名(今回の場合image)のあとに .url や .path をつけることでアクセスすることができる、実際に試してみましょう。

(~/PhotoService)

```
$ python manage.py shell
>>> from app.models import Photo

# 1つ目のPhotoインスタンスを取得
>>> photo = Photo.objects.all()[0]
```

```
>>> photo.image
<ImageFieldFile: photos/dog.jpeg>

>>> photo.image.url
'/media/photos/dog.jpeg'

>>> photo.image.path
'~/PhotoService/media/photos/dog.jpeg'
```

`photo.image.url` のように、`.url` でその画像のアドレスを取得できるのですが、その際アドレスは `MEDIA_URL/photos/dog.jpeg` のように、アドレスの頭には `MEDIA_URL` に指定した文字列がつきます。(今は、`MEDIA_URL = '/media/'` と設定しているので、画像のアドレスは `'media/photos/dog.jpeg'` となります。)

このように、`MEDIA_URL` はユーザーが生成したコンテンツ(アップロードされた画像、ファイルなど)の URL を表すのに使用されます。

保存されている画像を表示する

ここまでで画像の保存と、画像へのアクセスを学びました。次は保存されている画像をトップページに表示させてみましょう。手順としては、「views.pyでPhotoインスタンスを全件取得してindex.htmlに渡す」→「index.htmlでfor文を回して各Photoインスタンスのimageフィールドにアクセスする」という流れです。

(~/PhotoService/app/views.py)

```
from django.shortcuts import get_object_or_404, redirect, render
from django.contrib.auth.models import User
from .models import Photo # 追加

def index(request):
    # Photoインスタンスを全件取得
    photos = Photo.objects.all().order_by('-created_at')
    # 取得したPhotoインスタンスをテンプレートに渡す
    return render(request, 'app/index.html', {'photos': photos})
```

(~/PhotoService/app/templates/app/index.html)

```
{% extends 'app/base.html' %}

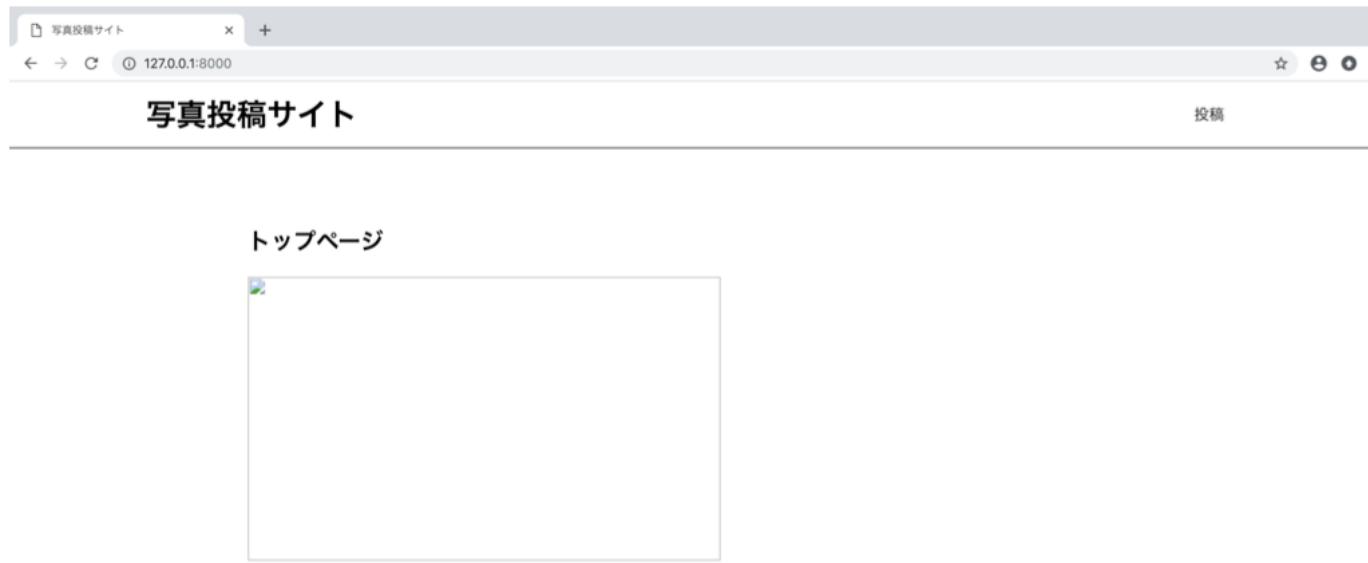
{% block content %}

<h2>トップページ</h2>

{% for photo in photos %}
    
{% endfor %}
```

```
{% endblock %}
```

ここまでできたらトップページ(<http://127.0.0.1:8000/>)を確認してみてください。下図のようになりましたか?画像の枠は表示されていますが、実際の画像はうまく表示されていませんね。



これは、**画像が保管されているディレクトリ**、つまり(**mediaディレクトリ**)が一般に公開されていないことが原因です。管理者であればAdminページから画像を見ることができますが、一般の人は画像のURLにアクセスしても見ることができません。

ディレクトリが公開されていないため、imgタグのsrcで指定したアドレスにアクセスしても画像が取得できないのです。試しに、画像のアドレスをコピー(画像の上で右クリック)して、新しいタブでそのアドレスにアクセスしてみましょう。



このように404ページが表示されます。ディレクトリにアクセスできず画像が取得できていないことがわかりますね。



mediaディレクトリを公開するには、 urls.pyに以下を追加してください。これにより、 mediaディレクトリが公開されてアドレスにアクセスできるようになります。

(~/PhotoService/PhotoService/urls.py)

```
from django.contrib import admin
from django.urls import path, include
from django.conf import settings # 追加
from django.conf.urls.static import static # 追加

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('app.urls')),
]

# MEDIA_ROOTを公開する(アクセス可能にする)
urlpatterns += static(
    settings.MEDIA_URL, document_root=settings.MEDIA_ROOT
)
```

設定が完了したら、もう一度画像にアクセスしてみましょう。うまく表示されれば成功です!トップページにも画像が表示されます。表示されない場合は、キャッシュが影響している可能性もありますのでブラウザのスーパークリードを試してみてください。

django-cleanupで画像ファイルを削除する

画像のアップロードと表示ができました。最後に、投稿が削除された場合の処理について説明します。
今の実装のままだと、Photoインスタンスが削除されても、mediaディレクトリ内にアップロードされた画像は削除されずに残ったままとなります。
つまり、画像ファイルを手動で削除していくかの限りはずっと画像がディレクトリ内に増え続けることになります。

photo を変更

Title: お散歩

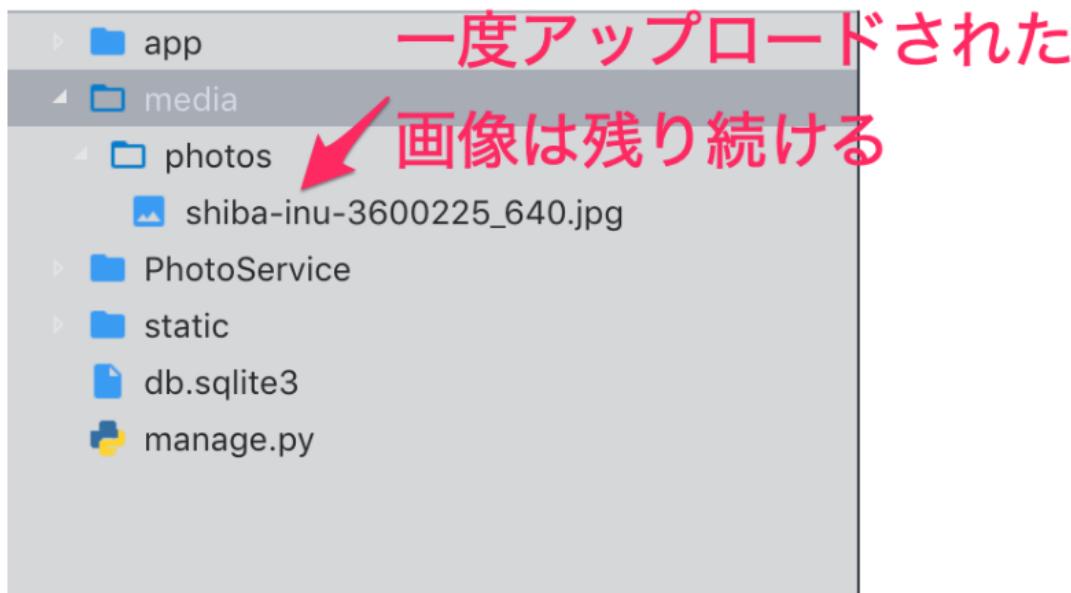
Comment: 寒いけど頑張ってお散歩したよ。

Image: 現在: photos/shiba-inu-3600225_640.jpg
変更: ファイルを選択 選択されていません

この投稿を削除しても、、、

削除 ↑

保存してもう一つ追加 保存して編集を続ける 保存



仮にユーザーが投稿を削除した場合、それに紐づいた画像にアクセスされることもなくなるので、画像を保存しておく必要はありません。そこで、投稿が削除されるのと同時に、それと紐づいた画像ファイルも削除されるような設定をしてみましょう。

この設定には、**django-cleanup**というライブラリを使用します。使い方は簡単でpipコマンドでインストールして、INSTALLED_APPSに追加するだけです。

(~/PhotoService)

```
$ pip install django-cleanup
```

インストールするときは `django-cleanup`、INSTALLED_APPSに追加するのは `django_cleanup` なので注意してください。(ハイフンを使うかアンダーバーを使うかの違い。)

(~/PhotoService/PhotoService/settings.py)

```

INSTALLED_APPS = (
    ...
    'django_cleanup',
)

```

これで、Photoインスタンスの削除と同時に画像ファイルも削除されるようになりました。

もし全てのPhotoインスタンスを削除してしまった場合は、次節のために必ず何枚か画像を上げ直しておいてください。

4.5 ForeignKeyでモデル同士を紐づける

Photoモデルを作ることで画像を表示させることができましたが、Photoモデルにさらに変更を加えて、その投稿は「どのUserが投稿したものか」と「どのカテゴリーに属するものか」をわかるようにしていきましょう。

結論から言うと、**Photoモデルにuserフィールドとcategoryフィールドを追加します**。これらのフィールドには、他のモデルから生成されたインスタンスが保存されることになります。

つまり、**userフィールドにはUserモデルから生成されたユーザーインスタンス、categoryフィールドにはCategoryモデルから生成されたカテゴリーインスタンスが保存されるようにします**。

Photos テーブル

id	title	comment	image	category	user	created_at
	CharField (文字列型)	TextField (テキスト型)	ImageField(画像ファイル)	ForeignKey (Category型)	ForeignKey (User型)	DateTimeField (日付型)
(例) 1	お散歩	寒いけどがんばってお散歩したよ。	dog_walk.jpeg	動物	Jobs	2019/02/24
2	初めての富士山	富士山の写真を撮ったよ。	fujisan.jpeg	風景	Rola	2019/02/25
3	美味しくできた	卵料理つくった。おいしかった。	egg_dish.jpeg	料理	Rola	2019/03/09
4						
.						
.						

Categorys テーブル

- 1. 動物
- 2. 風景
- 3. 料理

The diagram illustrates the relationships between the Photo, Category, and User tables. A curved arrow points from the 'category' column in the Photos table to the 'Categorys' table. Another curved arrow points from the 'user' column in the Photos table to the 'Users' table. The 'Categorys' table contains three entries: '1. 動物', '2. 風景', and '3. 料理'. The 'Users' table contains two entries: '1. Jobs' and '2. Rola'.

Users テーブル

- 1. Jobs
- 2. Rola

ForeignKeyを使ってモデル同士を紐づける

これまでにCharField、TextField、ImageField、DateTimeFieldなどを学習してきましたので、文字列型、画像ファイル、日時型を保存するフィールドはもう作れますよね。

今回は、「モデルから作られたインスタンス」を保存できるフィールドを作ることになります。**userフィールドには、Userテーブルにあるインスタンスのうちの1つが保存されます**。

それでは実際に、userフィールドを作ってみましょう。Userモデルを使うのでimport文を追加することも忘れないでください。

(~/PhotoService/app/models.py)

```

from django.db import models
from django.contrib.auth.models import User # 追加

class Photo(models.Model):
    title = models.CharField(max_length=150)
    comment = models.TextField(blank=True)
    image = models.ImageField(upload_to='photos')
    user = models.ForeignKey(User, on_delete=models.CASCADE) # 追加
    created_at = models.DateTimeField(auto_now=True)

    def __str__(self):
        return self.title

```

インスタンスを保存するフィールドを作るためには**ForeignKey**というものが使えます。**ForeignKey**では、**そのフィールドと紐づけるモデルを第1引数に指定**します。userフィールドには、Userインスタンスを保存したいので、上記のようになります。

`on_delete`では、紐づけられたインスタンスが削除されたときの挙動を定義しています。具体的に上図の例で説明すると、『Rola』というユーザーインスタンスが削除された場合に、それと紐づいたPhotoインスタンス(Rolaの2つの投稿『id:2 初めての富士山』『id:3 美味しくできた』)も一緒に削除するのか、それとも投稿だけは残しておくのか、を定義しています。

`on_delete=models.CASCADE`のようにすると、Userインスタンスを削除すると、それと一緒にそのユーザーと紐づいた投稿も全て削除されます。`on_delete=models.PROTECT`とすれば、特定のUserインスタンスを削除しようとしても、そのユーザーと紐づいた投稿が存在する場合は削除できないようになります。`on_delete`に関しては、DjangoBrothersのブログ記事([Django2.0から必須になったon_deleteの使い方](#))もご参照ください。

null=Falseのフィールドを追加する

ここまでできたら、マイグレーションファイルを作ってみましょう。

すると、こんなメッセージが出るはずです。

```
(photo_venv) ~/Desktop/PhotoService $ python manage.py makemigrations
You are trying to add a non-nullable field 'user' to photo without a default; we can't do that (the database needs
something to populate existing rows).
Please select a fix:
 1) Provide a one-off default now (will be set on all existing rows with a null value for this column)
 2) Quit, and let me add a default in models.py
Select an option: 1
```

userフィールドでは、`null=True`と指定していないので、デフォルトで`null=False`、つまり空欄を許容しない設定になっています。Photoインスタンスのuserフィールドには、何かしらのUserインスタンスが保管されていないといけないということです。

今後新しくPhotoインスタンスを投稿するときは、Userを選んで保存すればいいだけなのですが、前回のレッスンまでに作成した既存のPhotoインスタンスに関しても、userフィールドは空欄にしてはいけないため、何かしらのUserインスタンスを保存する必要があります。

既存のPhotoインスタンスのuserフィールドにUserインスタンスを保存するためには、エラーメッセージにある通り2つやり方があります。

- Provide a one-off default now (will be set on all existing rows with a null value for this column) 「特定の値を今入力する。(全ての既存データにこの値が適用される。)」
- Quit, and let me add a default in models.py 「一旦コマンドラインを閉じて、models.pyでデフォルト値を設定する。」

選択肢1を選んだ場合、コマンドライン上でデフォルト値を指定します。指定の仕方は、紐づくインスタンスのIDを指定します。例えば、User「Jobs」のIDが1だった場合、数字の1をコマンドライン上で入力すれば、既存の全Photoデータのuserフィールドには、Jobsが保存されることになります。

選択肢2を選んだ場合、コマンドラインの入力モードは終了します。その後、自分でmodels.pyを修正します。具体的には、`user = models.ForeignKey(User, on_delete=models.CASCADE, default=1)` のように、任意のユーザーIDをdefaultとして指定します。これにより、マイグレーションするタイミングで、既存の投稿は全てJobsと紐づくことになります。

以下の画像は、選択肢1を選んだ場合の例です。

```
(photo_venv) ~/Desktop/PhotoService $ python manage.py makemigrations
You are trying to add a non-nullable field 'user' to photo without a default; we can't do that (the database needs
something to populate existing rows).
Please select a fix:
1) Provide a one-off default now (will be set on all existing rows with a null value for this column)
2) Quit, and let me add a default in models.py 1を入力して、選択肢1を選択
Select an option: 1
Please enter the default value now, as valid Python default値を入力せよとメッセージが出る
The datetime and django.utils.timezone modules are available, so you can do e.g. timezone.now
Type 'exit' to exit this prompt
>>>
Please enter some code, or 'exit' (with no quotes) to exit. JobsのID「1」を入力
>>> 1
Migrations for 'app':
  app/migrations/0002_photo_user.py - 無事、マイグレーションファイルが作られる
```

マイグレーションファイルができたら、マイグレートしてデータベースに反映させましょう。

管理画面から、Photoインスタンスを確認すると、全てのuserフィールドに自分が設定したIDのユーザーが保存されているはずです。

The screenshot shows the Django Admin interface for the 'photo' model. The URL in the browser is `127.0.0.1:8000/admin/app/photo/2/change/`. The page title is 'photo を変更'. The 'User' field is set to 'Jobs'. The 'Comment' field contains the text '寒いけど頑張ってお散歩したよ。'. At the bottom, there are buttons for '削除' (Delete), '保存してもう一つ追加' (Save and add another), '保存して編集を続ける' (Save and continue editing), and a standard '保存' (Save) button.

ForeignKeyを使ったフィールドへのアクセス

`photo.title` や `photo.created_at` のように書くことでそれぞれの属性にアクセスできますが、これは userフィールドでも同じことです。`photo.user` とすると、Usersテーブルを参照するようになります。

試しに、shellでデータを取得してみましょう。

(~/PhotoService)

```
$ python manage.py shell
>>> from django.contrib.auth.models import User
>>> from app.models import Photo

# 1つ目の Photo インスタンスを取得
>>> photo1 = Photo.objects.all()[0]
>>> photo1
<Photo: お散歩>

>>> photo1.title
'お散歩'

# photo1のuserフィールドにアクセス(Userインスタンスが取得できる)
>>> photo1.user
<User: Jobs>

# photo1に紐づいたUserインスタンスのフィールドにアクセス
>>> photo1.user.id
1
>>> photo1.user.username
'Jobs'
>>> photo1.user.email
'jobs@example.com'
>>> photo1.user.is_superuser
True
```

上記のように、PhotoインスタンスのuserフィールドにアクセスすればUserインスタンスを取得でき、さらにそのUserインスタンスのフィールドにアクセスすることができます。これにより、「お散歩」という投稿をした人のユーザー名やメールアドレス等を取得できるようになりました。

Categoryモデルを作る

次に、categoryフィールドを作ってみましょう。実装の流れは、userフィールドと同じです。ただ、Userモデルは Djangoデフォルトのものを使用していましたが、Categoryモデルは自分で作る必要があります。Categoryモデルは、titleフィールドを持つシンプルなモデルです。「動物」や「風景」といったタイトルをつけてインスタンスを作ることができます。

(~/PhotoService/app/models.py)

```

from django.db import models
from django.contrib.auth.models import User

# Category モデルを作成
class Category(models.Model):
    title = models.CharField(max_length=20)

    def __str__(self):
        return self.title


class Photo(models.Model):
    title = models.CharField(max_length=150)
    comment = models.TextField(blank=True)
    image = models.ImageField(upload_to = 'photos')
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    created_at = models.DateTimeField(auto_now=True)

    def __str__(self):
        return self.title

```

(~/PhotoService/app/admin.py)

```

from django.contrib import admin
from .models import Category, Photo # 追加

# 追加
class CategoryAdmin(admin.ModelAdmin):
    list_display = ('id', 'title')
    list_display_links = ('id', 'title')

class PhotoAdmin(admin.ModelAdmin):
    list_display = ('id', 'title', 'user')
    list_display_links = ('id', 'title')

admin.site.register(Category, CategoryAdmin) # 追加
admin.site.register(Photo, PhotoAdmin)

```

Categoryモデルを作ったら、一度このタイミングでマイグレートしましょう。そして、Admin画面からカテゴリーをいくつか作成します。

変更する category を選択

操作: ID 3 2 1

ID	TITLE
3	料理
2	風景
1	動物

3 categorys

次に、Photoモデルにcategoryフィールドを追加します。

(~/PhotoService/app/models.py)

```
class Photo(models.Model):
    title = models.CharField(max_length=150)
    comment = models.TextField(blank=True)
    image = models.ImageField(upload_to='photos')
    category = models.ForeignKey(Category, on_delete=models.PROTECT) # 追加
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    created_at = models.DateTimeField(auto_now=True)

    def __str__(self):
        return self.title
```

再度マイグレーションファイルを作ります。先ほどと同じようにdefault値を聞かれるので、既存のCategoryインスタンスから任意のものを1つ選び、そのIDを入力して設定しましょう。マイグレーションファイルができたら、マイグレートコマンドを打ってください。

無事マイグレートできたら、Admin画面からいくつかPhotoインスタンスを作成してみましょう。そのとき、ユーザーとカテゴリーが選択できるようになっているはずです。

photo を追加

Title: お散歩

Comment: お散歩したよ。

Image: ファイルを選択 選択されていません

Category: 動物 +

User: Jobs +

ForeignKeyは一对多(OneToMany)の関係を作る

`photo1.user` とすることで、Userインスタンスを取得できると説明しましたが、これを参照するといいます。例えば上記の例では、photo1のuser属性を参照するとJobsが取得できます。

userフィールドのようにForeignKeyを使ったフィールドは、第1引数に指定されたモデルを参照することになります。その際、参照するインスタンスは1つだけです。つまり、Photoのuserフィールドには、1つだけのUserインスタンスが保存されるということです。photo1にはuser1(Jobs)のみが保存され、2つ以上のユーザーを保存することはできません。

一方、Userモデル側からみると、**Userモデルは複数のPhotoに参照されている**と言えます。

user1(Jobs)が、どのPhotoに参照されているかは、`user1.photo_set` とすることで取得できます。これにより、user1を参照している全ての Photoインスタンスを取得できます。このように、参照してきているインスタンスを取得することを**逆参照**といいます。

- 参照(photo1.user): Photo側からUserを取得(参照するインスタンスは1つだけ)
- 逆参照(user1.photo_set): User側からPhotoを取得(参照されているインスタンスは複数ある)

Photoは1つのUserインスタンスを参照するのに対して、Userは多数のPhotoインスタンスに参照されています。この関係性を**一对多(OneToMany)**の関係といいます。**ForeignKeyは一对多の関係性を作ります**。

多数のインスタンス同士を参照させ合う**多対多(ManyToMany)** や、1つのみのインスタンス同士を参照させあう**一对一(OneToOne)** という関係性もあります。

shell画面で逆参照の操作をしてみましょう。

(~/PhotoService)

```
$ python manage.py shell
>>> from django.contrib.auth.models import User
>>> from app.models import Photo, Category
>>> user1 = User.objects.all()[0]
>>> user1
<User: Jobs>

# 逆参照する
>>> user1.photo_set
<django.db.models.fields.related_descriptors.create_reverse_many_to_one_manager.<locals>.RelatedManager object at 0x111016240>

# user1を参照している全てのPhotoインスタンスをクエリセットとして取得する
>>> user1.photo_set.all()
<QuerySet [<Photo: お散歩>, <Photo: 料理作った>]>
```

最後に、views.pyで逆参照してユーザーに紐づいた写真を全て取得してみましょう。

(~/PhotoService/app/views.py)

```
def users_detail(request, pk):
    user = get_object_or_404(User, pk=pk)
    # userに紐づく写真一覧を取得
    photos = user.photo_set.all().order_by('-created_at')
    # photosを追加
    return render(request, 'app/users_detail.html', {'user': user, 'photos': photos})
```

Viewから受け取ったphotosをusers_detail.htmlで表示します。index.htmlと全く同じ方法で表示できますので、for文の部分をコピペしましょう。

(~/PhotoService/app/templates/app/users_detail.html)

```
{% extends 'app/base.html' %}

{% block content %}

<h2 class="user-name">@{{ user.username }}</h2>

{% for photo in photos %}
    
{% endfor %}

{% endblock %}
```

これでユーザーのページでは、その人が投稿した写真の一覧が表示されるようになりました。

4.6 ユーザー認証機能を作ろう

ここまでで、投稿(Photoモデル)の設計や画像を表示させることができるようになりました。ただ、現状ではAdminページからしかデータを操作できないので、ユーザーが投稿できるようにしていきましょう。

まずはユーザーの登録・認証機能を実装します。

登録・認証機能についても非常によく使われる機能ですので、Djangoがデフォルトで認証機能をサポートしてくれています。`django.contrib.auth` でユーザーの登録や認証に関わる機能が定義されているので、それを利用します。

今回使っているUserモデルは、`django.contrib.auth.models` からインポートして使っています。これと同様に、例えば、ユーザー登録用の入力フォームやパスワード変更用フォームなどが用意されています。また、`django.contrib.auth.decorators` には、認証に関わる便利なデコレータがあります。これらを、必要に応じて適宜インポートしながら開発を進めていきます。

ログイン機能の実装

まずは、ログイン機能を実装しましょう。必要なことは主に以下の2つです。

- ログイン画面のURL、ユーザーがログインした後にリダイレクトされるURL、ログアウトした後にリダイレクトされるURLを設定する
- ログイン画面のHTMLファイルを作る

最初に、各URLの設定をします。app内のurls.pyを以下のように編集してください。

(~/PhotoService/app/urls.py)

```
from django.urls import path
from . import views
from django.contrib.auth import views as auth_views # 追加

app_name = 'app'
urlpatterns = [
    path('', views.index, name='index'),
    path('users/<int:pk>', views.users_detail, name='users_detail'),
    path(
        'login/',
        auth_views.LoginView.as_view(template_name='app/login.html'),
        name='login',
    ),
    path('logout/', auth_views.LogoutView.as_view(), name='logout'),
]
```

3行目でインポートしたauth_viewsの中には、LoginView、LogoutViewという機能があるのでこれをそのまま使います。これらが自動的にログイン・ログアウト処理を行ってくれるので、template_nameという引数にログイン画面となるHTMLファイルのパスを指定します。今回はapp/login.htmlというファイルを作り、これをログイン画面のページとします。login.htmlには、ユーザー名とパスワードを入力するログイン用フォームを配置することになります。

ここまでで、login/のURLにアクセスするとログイン画面が表示され、logout/にアクセスするとログアウトするような設定をしています。ログイン画面は後ほど作成します。

次に、settings.pyに以下の3行を追加してください。

(~/PhotoService/PhotoService/settings.py)

```
LOGIN_URL = 'app:login'  
LOGIN_REDIRECT_URL = 'app:index'  
LOGOUT_REDIRECT_URL = 'app:index'
```

LOGIN_URLは、ユーザーがログインする時に使うページを設定します。今回の場合は、login.htmlのページのことです。仮に、「ログイン中のユーザーしか見ることができないページ」に「未ログインのユーザー」がアクセスしてきた場合は、LOGIN_URLに設定したページにユーザーがリダイレクトされることになります。

LOGIN_REDIRECT_URLは、ユーザーがログインした時に、最初にリダイレクトさせるURLを指定します。今回は、'app:index'、つまりトップページを指定します。これにより、ログインしたユーザーは最初にトップページを閲覧することになります。

LOGOUT_REDIRECT_URLは、ログアウトしたユーザーをリダイレクトさせるURLです。こちらも、トップページにリダイレクトさせる設定にしています。

ここで、一度http://127.0.0.1:8000/logout/にアクセスしてみてください。トップページが表示されるはずです。

urls.pyでpath('logout/', auth_views.LogoutView.as_view(), name='logout')と設定しているので、http://127.0.0.1:8000/logout/にアクセスした時点でLogoutViewがユーザーをログアウトさせます。次に、LOGOUT_REDIRECT_URLで指定したトップページにユーザーをリダイレクトさせたのです。

本当にログアウトできているか確かめるために、Adminページにアクセスしてみてください。きっとログインを求められるでしょう。これは正常にログアウトできている証拠です。

ログインページhttp://127.0.0.1:8000/login/にアクセスしても、現状はlogin.htmlを作っていないのでエラーになります。では、login.htmlを作りましょう。

まずは、templates/appディレクトリの中に、login.htmlファイルを作成してください。中身は以下のようになります。

(~/PhotoService/app/templates/app/login.html)

```
{% extends 'app/base.html' %}
```

```

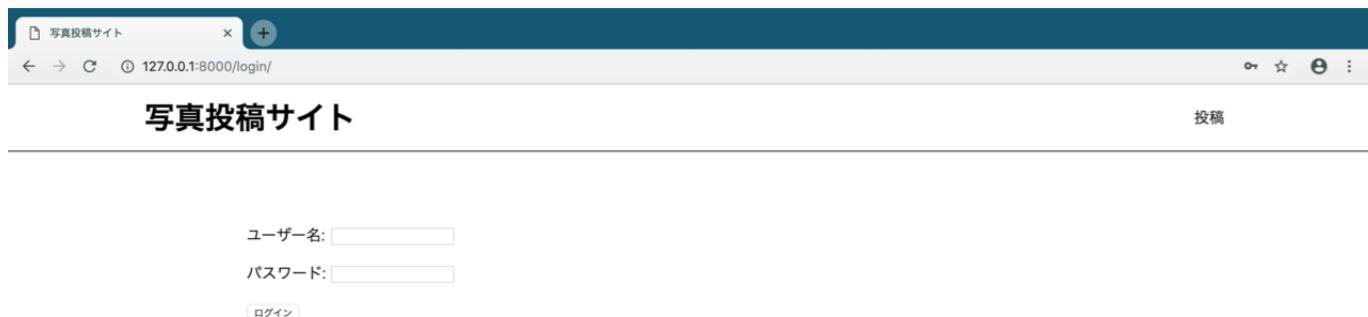
{% block content %}

<form method="POST">{% csrf_token %}
{{ form.as_p }}
<input type="submit" value="ログイン">
</form>

{% endblock %}

```

ファイルを保存して `http://127.0.0.1:8000/login/` にアクセスすると、このようにフォームが表示されるはずです。



`path('login/', auth_views.LoginView.as_view(template_name='app/login.html'), name='login')` の設定により、login.htmlにはLoginViewがformを渡してくれるので、`{{ form }}` と書くことができます。しかも、このformは、**AuthenticationForm**という認証用に作られたフォームですので、ユーザー名とパスワードを入力させることができます。

ラベルなどをカスタマイズしたい場合は、`{{ form.as_p }}` ではなく以下のように書くこともできます。その際、inputタグのname属性は**username**と**password**にしておく必要があります。また、`{{ form.username }}` のように書くことでも、name属性を持つinputタグを生成することができます。

(~/PhotoService/app/templates/app/login.html)

```

{% extends 'app/base.html' %}

{% block content %}

<form class="form-signin" method="POST">{% csrf_token %}
{% if form.errors %}
<p>ユーザー名かパスワードが間違っています。</p>
{% endif %}
<h2>ログイン</h2>
<label>ユーザー名</label>
<input name="username">
<br>
<label>パスワード</label>
<input type="password" name="password">
<br>
<input type="submit" value="ログイン">
</form>

```

```
{% endblock %}
```

それでは、実際にフォームからユーザー名とパスワードを入力してログインしてみましょう。「ログイン」ボタンを押した時に問題なくログインできれば、LOGIN_REDIRECT_URLで設定したトップページにリダイレクトされます。ユーザー名やパスワードが間違っていて認証に失敗した場合は、login.htmlにエラーメッセージが表示されます。

ログインに成功しているかを確認するために、Adminページにアクセスしてみましょう。この時にログインを求められず、問題なく閲覧できればログインは成功しています。

ユーザーのログイン状態に合わせて表示を切り替える

最後に、ユーザーのログイン状態に合わせて画面表示を切り替えてみましょう。base.htmlのheaderに以下のように追記してください。

(~/PhotoService/app/templates/app/base.html)

```
<header>
  <div class="container">
    <h1><a href="{% url 'app:index' %}">写真投稿サイト</a></h1>
    <div class="header-menu">
      <a href="">投稿</a>

      <!-- 追加 -->
      {% if request.user.is_authenticated %}
        <a href="{% url 'app:users_detail' request.user.id %}">マイページ
      </a>
        <a href="{% url 'app:logout' %}">ログアウト</a>
      {% else %}
        <a href="{% url 'app:login' %}">ログイン</a>
      {% endif %}

    </div>
  </div>
</header>
```

{% if request.user.is_authenticated %} というif文を追加しています。

views.pyの各関数で最後に実行されるrenderメソッドでは、requestを第1引数に取ってTemplate側にこのrequestを渡しています。このrequestの中にはサーバーに対してリクエストを送ってきたユーザー情報などが含まれています。そして、Templateでは、`request.user` とすることで、そのUserオブジェクトにアクセスすることができます。

また、Userオブジェクトが**is_superuser**等様々な属性を持っていることは既に紹介しましたが、**is_authenticated**という属性も持っています。これは、ユーザーがログイン状態であればTrue、未ログイン状態であればFalseとなる属性です。

つまりこのif文では、アクセスしてきたユーザーがログイン状態の場合は「マイページ」と「ログアウト」へのリンクを表示し、ログイン状態でない場合には「ログイン画面」へのリンクを表示するようにしています。実際にログインとログアウトをしてみて、ヘッダーの表示が切り替わるか確認してみましょう。

- 未ログイン時



- ログイン時



ログイン機能を実装できましたので、次は新規にアカウントを作成する、ユーザー登録機能をつくりましょう!

4.7 ユーザー登録機能を作ろう

この節では、ユーザー登録機能を実装して、新規ユーザーを作成できるようにします。登録機能の実装に必要なことは以下の通りです。

- 登録画面のURLを設定する。
- 登録画面に入力フォーム(UserCreationForm)を表示する。
- UserCreationFormから受け取った情報でユーザーを新規作成する。
- 作成したユーザーをログイン状態にする。

まずはURLの設定から行いましょう。`http://127.0.0.1:8000/signup/` で登録フォームを表示するようにします。

(~/PhotoService/app/urls.py)

```
from django.urls import path
from . import views
from django.contrib.auth import views as auth_views

app_name = 'app'
urlpatterns = [
    path('', views.index, name='index'),
    path('users/<int:pk>', views.users_detail, name='users_detail'),
    path('signup/', views.signup, name='signup'), # 追加
    path(
        'login/',
        auth_views.LoginView.as_view(template_name='app/login.html'),
        name='login'
```

```
)  
path('logout/', auth_views.LogoutView.as_view(), name='logout'),  
]
```

次に、Viewの設定です。

(~/PhotoService/app/views.py)

```
from django.contrib.auth.forms import UserCreationForm  
  
def signup(request):  
    form = UserCreationForm()  
    return render(request, 'app/signup.html', {'form': form})
```

ユーザー登録用のフォーム(UserCreationForm)をインポートして、signup.htmlに渡します。

signup.htmlを作ります。

(~/PhotoService/app/templates/app/signup.html)

```
{% extends 'app/base.html' %}  
  
{% block content %}  
  
<h2>ユーザー登録</h2>  
  
<form method="POST" action="{% url 'app:signup' %}">{% csrf_token %}  
    <label>ユーザー名</label>  
    {{ form.username }}  
    {{ form.username.errors }}  
    <br>  
    <label>パスワード</label>  
    {{ form.password1 }}  
    {{ form.password1.errors }}  
    <br>  
    <label>パスワード(確認)</label>  
    {{ form.password2 }}  
    {{ form.password2.errors }}  
    <br>  
    <input type="submit" value="登録する">  
</form>  
  
<p><a href="{% url 'app:login' %}">ログインはこちら</a></p>  
  
{% endblock %}
```

これで、 <http://127.0.0.1:8000/signup/> でユーザー登録用のページが表示されるようになりました。

ログイン機能を実装した時は、`auth_views.LoginView`がフォームから情報を受け取って自動的にログイン処理を実行しました。ユーザー登録の場合は、自分でviews.pyに登録処理を実装します。`views.py`を以下のように編集してください。

(~/PhotoService/app/views.py)

```
from django.contrib.auth.forms import UserCreationForm

def signup(request):
    if request.method == 'POST':
        form = UserCreationForm(request.POST) # Userインスタンスを作成
        if form.is_valid():
            form.save() # Userインスタンスを保存
    else:
        form = UserCreationForm()
    return render(request, 'app/signup.html', {'form': form})
```

UserCreationFormは、新しいユーザーを作成するための**ModelForm**ですので、チュートリアル①で扱った**ModelForm**と基本的には同じことをしてくれます。

また、**UserCreationForm**は入力された値に対して以下のようなチェックを行います。もし入力値が不正であればエラーを表示することができます。

- `password1`と`password2`(確認用)の入力値が一致しているかどうか
- パスワードとしてふさわしい値が入力されているか(短すぎたり、ユーザー名と似すぎていたりするとエラーとなる。)
- 同名のユーザーが存在していないか

ここまでできたら、実際に登録フォームに値を入力してみましょう。その際、登録ボタンを押してもページ遷移されませんが、ユーザーは作られているはずですので、Admin画面にアクセスして新規ユーザーが本当にできているかを確かめてみましょう。

登録と同時に、ユーザーをログインさせる

現状の実装では「ユーザーを新規作成する」ことはできていますが、登録してもページ遷移がされませんしログインもできません。実際のWebサービスでは「ユーザー登録完了と同時にログインしてマイページなどに遷移す

る」のが一般的な流れですので、そのように実装してみましょう。

(~/PhotoService/app/views.py)

```
from django.shortcuts import render, redirect, get_object_or_404 # 追加
from django.contrib.auth.models import User
from django.contrib.auth.forms import UserCreationForm
from django.contrib.auth import authenticate, login # 追加
from .models import Photo

def signup(request):
    if request.method == 'POST':
        form = UserCreationForm(request.POST) # Userインスタンスを作成
        if form.is_valid():
            form.save() # Userインスタンスを保存
            input_username = form.cleaned_data['username']
            input_password = form.cleaned_data['password1']
            # フォームの入力値で認証できればユーザーOブジェクト、できなければNoneを返す
            new_user = authenticate(
                username=input_username,
                password=input_password,
            )
            # 認証成功時のみ、ユーザーをログインさせる
            if new_user is not None:
                # login関数は、認証がでなくてなくともログインさせることができる。(認証は上の
                authenticateで実行する)
                login(request, new_user)
                return redirect('app:users_detail', pk=new_user.pk)
            else:
                form = UserCreationForm()
        return render(request, 'app/signup.html', {'form': form})
```

`cleaned_data` という属性で、formに入力された値を取得して、それぞれ変数に代入しています。

また、ファイル冒頭では `authenticate` と `login` という関数をインポートしています。これを使って、ユーザーの認証を行います。それぞれの関数の役割は以下の通りです。

- **authenticate:** usernameとpasswordを取り、その組み合わせで認証に成功すればUserオブジェクトを返す。認証できなければNoneを返す。
- **login:** リクエスト情報とUserオブジェクトを取り、そのユーザーを未ログイン状態からログイン状態にする。

上記Viewの実装では、フォームに入力されたユーザー名(input_username)とパスワード(input_password)の値が、組み合わせとして正しいかを `authenticate(username=input_username, password=input_password)` で検証しています。組み合わせが正しければ、authenticate関数は認証に成功したUserオブジェクトを返すので、変数new_userにはそのオブジェクトが代入されます。認証に失敗した場合は `new_user=None` となります。

そして認証成功時のみ、login関数を実行してユーザーをログイン状態にし、そのユーザーのusers_detailページにリダイレクトさせます。

※ login関数自体は認証機能を備えていないため、その前にauthenticate関数で認証を行なっています。(login関数は認証されていないユーザーに対しても使うことができます。つまり、認証なしでユーザーをログインさせることもできますが、基本的にはauthenticate関数と併用して認証済みのユーザーだけをログインさせるのが一般的でしょう。)

ここまで実装したら、再度ユーザー登録をしてみましょう。登録ボタンを押すと、ログイン状態となり、自分のdetailページに遷移するはずです。

4.8 写真投稿機能と削除機能の実装

この節では、写真の投稿機能と削除機能を作ります。PhotoモデルのModelFormを利用していきます。

投稿機能を実装する

まずは投稿画面を作ります。URLの設定をします。

(~/PhotoService/app/urls.py)

```
app_name = 'app'
urlpatterns = [
    path('', views.index, name='index'),
    path('users/<int:pk>', views.users_detail, name='users_detail'),
    path('photos/new/', views.photos_new, name='photos_new'), # 追加
    path('signup/', views.signup, name='signup'),
    path(
        'login/',
        auth_views.LoginView.as_view(template_name='app/login.html'),
        name='login'
    ),
    path('logout/', auth_views.LogoutView.as_view(), name='logout'),
]
```

投稿画面のHTMLファイルを作ります。投稿フォームのimageフィールドからは、画像ファイルをアップロードすることになります。ファイルをアップロードする場合は、formタグに `enctype="multipart/form-data"` をつけないと正常にアップロードできないので注意しましょう。

(~/PhotoService/app/templates/app/photos_new.html)

```
{% extends 'app/base.html' %}

{% block content %}

<div>
    <a href="{% url 'app:index' %}">ホームに戻る</a>
</div>
```

```

<form action="{% url 'app:photos_new' %}" method="POST"
enctype="multipart/form-data">
  {% csrf_token %}
  <table>
    <tr>
      <th>タイトル</th>
      <td>{{ form.title }}</td>
    </tr>
    <tr>
      <th>コメント</th>
      <td>{{ form.comment }}</td>
    </tr>
    <tr>
      <th>画像</th>
      <td>{{ form.image }}</td>
    </tr>
    <tr>
      <th>カテゴリー</th>
      <td>{{ form.category }}</td>
    </tr>
  </table>
  <button type="submit" class="btn">保存</button>
</form>

{% endblock %}

```

Photoを投稿する用のModelFormを作ります。

(~/PhotoService/app/forms.py)

```

from django.forms import ModelForm
from .models import Photo

class PhotoForm(ModelForm):
    class Meta:
        model = Photo
        fields = ['title', 'comment', 'image', 'category']

```

base.htmlの「投稿」ボタンのリンクを設定し、ヘッダーのボタンから投稿画面に飛べるようにします。 `<div class="header-menu">` の中身だけ記載します。

(~/PhotoService/app/templates/app/base.html)

```

<div class="header-menu">
  <a href="{% url 'app:photos_new' %}">投稿</a>  <!-- 更新 --&gt;
  {% if request.user.is_authenticated %}
    &lt;a href="{% url 'app:users_detail' request.user.id %}"&gt;マイページ&lt;/a&gt;
    &lt;a href="{% url 'app:logout' %}"&gt;ログアウト&lt;/a&gt;
  {% else %}
</pre>

```

```
<a href="{% url 'app:login' %}">ログイン</a>
{% endif %}
</div>
```

views.py で、Photoの新規投稿機能を実装します。以下のように編集してください。

```
from django.contrib.auth.decorators import login_required
from .forms import PhotoForm

@login_required ①
def photos_new(request):
    if request.method == "POST":
        form = PhotoForm(request.POST, request.FILES) ②
        if form.is_valid():
            photo = form.save(commit=False) ③
            photo.user = request.user ④
            photo.save() ⑤
            return redirect('app:users_detail', pk=request.user.pk)
    else:
        form = PhotoForm()
    return render(request, 'app/photos_new.html', {'form': form})
```

各ポイントに番号をつけましたので、それぞれ説明します。

① `@login_required` というものをインポートして、関数の上につけています。`@マーク`がついているものは Python の **デコレータ** と呼ばれる機能です。簡単に説明すると、デコレータは関数の上につけることによってその関数を加工することができます。

今回の場合は、`@login_required` をつけることによって、ユーザーがログイン状態であれば `photos_new` 関数をそのまま実行し、ログインしていない状態であれば `photos_new` 関数を実行せずにログイン画面(`settings.py` で設定した `LOGIN_URL`)にリダイレクトさせるようにしています。ログインしているユーザーだけに限定したい関数には `@login_required` をつけましょう。ログインしていないユーザーが写真を投稿できてしまってはまずいので、`photos_new` 関数にはこのデコレータをつけます。

このあと実装する削除機能では、`@require_POST` というデコレータを使って、HTTPリクエストがPOSTメソッドの時にしかその関数を実行しないという処理をしています。デコレータは自作することができますが、このように Django が提供しているデコレータを適宜インポートして活用すると良いでしょう。

デコレータについてはブログ記事([【Python】初心者向けにデコレータの解説](#))でも説明しています。

② POSTされた情報を元にフォーム情報を生成します。ファイル情報を受け取るときは、`request.FILES` がないと正常にアップロードされないので気をつけてください。今回は `image` フィールドから画像ファイルがアップロードされるので、これが必要です。

③ 入力された情報から、Photoインスタンスを生成します。しかし、`form.save(commit=False)` のように、`save` メソッドの `commit` 引数を `False` にすることで、データベースには保存しないようにしています。なぜなら、この段階では Photo インスタンスの `user` フィールドに入れる値が決まっていないからです。(`PhotoForm` から受け取るのは `['title', 'comment', 'image', 'category']` だけのため、`user` フィールドに入れる値は取得できていません。)

仮に、`form.save()`とした場合、NOT NULL constraint failed: app_photo.user_idというエラーが表示されます。PhotoインスタンスのuserフィールドはNullにできない設定にしているにも関わらず、userフィールドが空の状態で保存しようとしているからです。

④ ③で一時的に生成したPhotoインスタンスのuserフィールドに、`request.user`(写真を投稿したUserのオブジェクト)を代入します。

⑤ この段階で、全てのフィールドに値が入った状態になったので、インスタンスをデータベースに保存します。

ここまでできたら、実際にフォームから投稿してみましょう。投稿後、マイページにリダイレクトされて画像が表示されれば成功です！

messageを表示する

写真が正常にアップロードされた場合は、成功メッセージを表示してあげると親切です。Djangoではメッセージを簡単に表示する機能があるので、以下のように実装してみましょう。

(~/PhotoService/app/views.py)

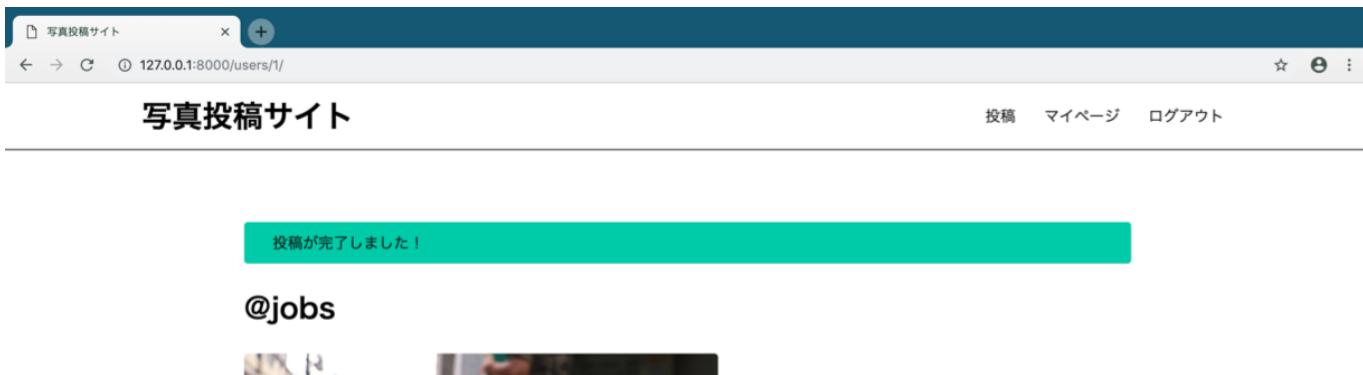
```
from django.contrib import messages

@login_required
def photos_new(request):
    if request.method == "POST":
        form = PhotoForm(request.POST, request.FILES)
        if form.is_valid():
            photo = form.save(commit=False)
            photo.user = request.user
            photo.save()
            messages.success(request, "投稿が完了しました!") # 追加
        return redirect('app:users_detail', pk=request.user.pk)
    else:
        form = PhotoForm()
    return render(request, 'app/photos_new.html', {'form': form})
```

(~/PhotoService/app/templates/app/base.html)

```
<div class="container">
    {% for message in messages %}
        <p class="message-success">{{ message }}</p>
    {% endfor %}
    {% block content %}{% endblock %}
</div>
```

これで、投稿時にメッセージが表示されるようになりました。



message機能の詳細は公式ドキュメントを参照してください。

- <https://docs.djangoproject.com/ja/2.2/ref/contrib/messages/>

削除機能の実装

削除機能を実装します。まずは、各写真の詳細を表示するページを用意して、そのページに削除ボタンを設置するようにします。

(~/PhotoService/app/urls.py)

```
app_name = 'app'
urlpatterns = [
    path('', views.index, name='index'),
    path('users/<int:pk>', views.users_detail, name='users_detail'),
    path('photos/new/', views.photos_new, name='photos_new'),
    # 追加
    path('photos/<int:pk>/', views.photos_detail, name='photos_detail'),
    # 追加
    path(
        'photos/<int:pk>/delete/',
        views.photos_delete,
        name='photos_delete'
    ),
    path('signup/', views.signup, name='signup'),
    path(
        'login/',
        auth_views.LoginView.as_view(template_name='app/login.html'),
        name='login'
    ),
    path('logout/', auth_views.LogoutView.as_view(), name='logout'),
]
```

(~/PhotoService/app/views.py)

```
from django.views.decorators.http import require_POST

def photos_detail(request, pk):
    photo = get_object_or_404(Photo, pk=pk)
    return render(request, 'app/photos_detail.html', {'photo': photo})

@require_POST
def photos_delete(request, pk):
    photo = get_object_or_404(Photo, pk=pk, user=request.user)
    photo.delete()
    return redirect('app:users_detail', request.user.id)
```

写真をクリックすると、詳細ページに飛ぶようにリンクを設定します。

(~/PhotoService/app/templates/app/index.html)

```
<h2>トップページ</h2>

{% for photo in photos %}

{% endfor %}
```

(~/PhotoService/app/templates/app/users_detail.html)

```
<h2 class="user-name">@{{ user.username }}</h2>

{% for photo in photos %}

{% endfor %}
```

最後に、photos_detail.htmlを作り削除ボタンを表示させます。

(~/PhotoService/app/templates/app/photos_detail.html)

```
{% extends 'app/base.html' %}

{% block content %}

<div class="photo-detail">
```

```


<div class="photo-info">
  <a href="{% url 'app:users_detail' photo.user.id %}">
    @{{ photo.user }}
  </a>
</div>

<h2>{{ photo.title }}</h2>
<p>{{ photo.comment }}</p>

<!-- 削除ボタン -->
<form method="POST" action="{% url 'app:photos_delete' photo.id %}">
  {% csrf_token %}
  <button class="btn" type="submit" onclick='return confirm("本当に削除しますか?");'>
    削除
  </button>
</form>

</div>

{% endblock %}

```

写真投稿サイト

投稿 マイページ ログアウト

お散歩

寒いけど頑張ってお散歩したよ。

[削除](#)

削除ボタンを表示させることができました。ただ今のままだと、このページにアクセスしてきたユーザーの誰しもが削除ボタンを押せることになります。

アクセスしてきたユーザー=この写真を投稿したユーザーのときだけ削除ボタンが表示されれば良いので、以下のようにif文を追加します。

(~/PhotoService/app/templates/app/photos_detail.html)

```

<!-- 削除ボタン -->
{% if request.user == photo.user %}

```

```

<form method="POST" action="{% url 'app:photos_delete' photo.id %}">
    {% csrf_token %}
    <button class="btn" type="submit" onclick='return confirm("本当に削除しますか?");'>
        削除
    </button>
</form>
{% endif %}

```

これで、写真の投稿者だけに削除ボタンが表示されるようになりました。

4.9 includeテンプレートタグでリファクタリング

この節では、これまで作ってきた機能をベースに、もっとページやコードの見栄えをよくしていきましょう。重複しているコードを共通化して可読性やメンテナンス性を向上させたり、効率的な処理になるように内部構造を改善することをリファクタリングと言います。

index.htmlとusers_detail.htmlでは、写真の一覧を表示しています。今よりももう少し綺麗な表示にするために、現状のコードを以下のように書き換えてみましょう。

(~/PhotoService/app/templates/app/index.html)

```

{% extends 'app/base.html' %}

{% block content %}

<div class="photo-container">
{% for photo in photos %}
    <div class="photo">
        <a href="{% url 'app:photos_detail' photo.id %}">
            
        </a>
        <div class="photo-info">
            <a href="" class="category">{{ photo.category }}</a>
            <a href="{% url 'app:users_detail' photo.user.id %}">@{{ photo.user }}</a>
        </div>
    </div>
{% endfor %}
</div>

{% endblock %}

```

(~/PhotoService/app/templates/app/users_detail.html)

```

{% extends 'app/base.html' %}

```

```

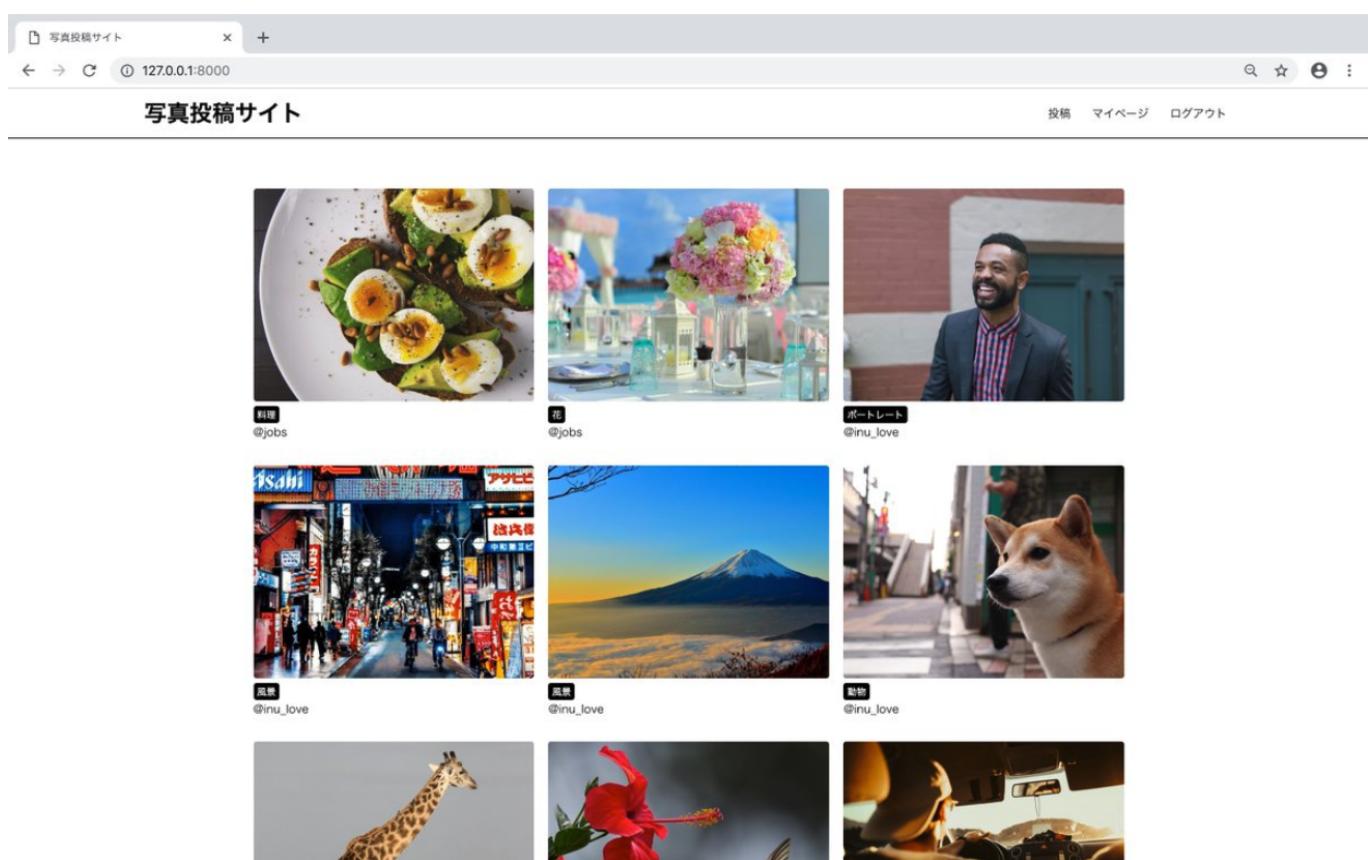
{%
    block content %}

<h2 class="user-name">@{{ user.username }}</h2>

<div class="photo-container">
{%
    for photo in photos %}
        <div class="photo">
            <a href="{% url 'app:photos_detail' photo.id %}">
                
            </a>
            <div class="photo-info">
                <a href="" class="category">{{ photo.category }}</a>
                <a href="{% url 'app:users_detail' photo.user.id %}">@{{ photo.user }}</a>
            </div>
        </div>
    {% endfor %}
</div>

{%
    endblock %
}

```



カテゴリ名や、名前が表示されるようになりました。(今は、カテゴリ名のリンクは何も設定しなくて大丈夫です。)このままのコードでも良いのですが、index.htmlとusers_detail.htmlではどちらも全く同じコードを書いて写真一覧を表示しています。共通部分のコードは、1つのファイルにまとめてしまいましょう。

共通するコードを1つのファイルにまとめる手順は以下の通りです。

1. photos_list.htmlというファイルを新規作成し、共通コードをこのファイルに移す
2. index.htmlとusers_detail.htmlでphotos_list.htmlを読み込む

1. photos_list.htmlを作る

まずは、photos_list.htmlを作り共通部分のコードをこのファイルにコピーします。

(~/PhotoService/app/templates/app/photos_list.html)

```
<div class="photo-container">
{% for photo in photos %}
    <div class="photo">
        <a href="{% url 'app:photos_detail' photo.id %}">
            
        </a>
        <div class="photo-info">
            <a href="" class="category">{{ photo.category }}</a>
            <a href="{% url 'app:users_detail' photo.user.id %}">@{{ photo.user
}}</a>
        </div>
    </div>
{% endfor %}
</div>
```

2. photos_list.htmlを読み込む

photos_list.htmlを他のファイルから読み込むためには、**include**というテンプレートタグが使えます。

(~/PhotoService/app/templates/app/index.html)

```
{% extends 'app/base.html' %}

{% block content %}

<!-- photos_list.htmlを読み込んで写真一覧を表示する --&gt;
{% include 'app/photos_list.html' %}

{% endblock %}</pre>
```

(~/PhotoService/app/templates/app/users_detail.html)

```
{% extends 'app/base.html' %}

{% block content %}

<h2 class="user-name">@{{ user.username }}</h2>

<!-- photos_list.htmlを読み込んで写真一覧を表示する --&gt;
{% include 'app/photos_list.html' %}</pre>
```

```
{% endblock %}
```

これで、共通部分を1つのファイルにまとめることができました。このように、共通しているコードを1つのファイルにまとめて、構造を整理することをリファクタリングと言います。リファクタリングをしておくと、編集範囲が狭くなりメンテナンスなども楽になります。

投稿数を表示する

users_detail.htmlを編集しましたので、ついでにこのページで写真の投稿数がわかるようにしてみましょう。投稿数は、`photos.count` のように、クエリセットのcount属性で取得することができます。

(~/PhotoService/app/templates/app/users_detail.html)

```
<h2 class="user-name">@{{ user.username }}</h2>

{% if photos.count != 0 %}
  <p>投稿<strong>{{ photos.count }}</strong>件</p>
{% else %}
  {% if user == request.user %}
    <p>初めての投稿をしてみましょう!</p>
  {% else %}
    <p>@{{ user.username }}さんはまだ投稿していません。</p>
  {% endif %}
{% endif %}

{% include 'app/photos_list.html' %}
```

`photos.count != 0` のとき(1つでも投稿があるとき)は、投稿数を表示します。

写真投稿サイト

投稿が完了しました！

@jobs

投稿4件

投稿

マイページ

ログアウト

自分のページで、1つも投稿がない場合は「初めての投稿をしてみましょう！」とメッセージを表示します。

写真投稿サイト

初めての投稿をしてみましょう！

@jobs

投稿

マイページ

ログアウト

1つも投稿していない他のページにアクセスしたときは「@{{ user.username }}さんはまだ投稿していません。」と表示されます。

写真投稿サイト

@rolaさんはまだ投稿していません。

投稿

マイページ

ログアウト

@rola

@rolaさんはまだ投稿していません。

最後に、カテゴリーごとのページを作りましょう。

4.10 カテゴリーで絞り込む

Photoモデルは、categoryというフィールドを持っており、写真を投稿するときにカテゴリーを選べるようになっています。

この節では、カテゴリー単位で写真を絞り込み、その結果をページで表示できるようにしていきます。

まずは、URLの設定です。 `http://127.0.0.1:8000/photos/動物/` というURLで、「動物」カテゴリーの写真一覧を表示させたいので、urls.pyを以下のように追加します。

(~/PhotoService/app/urls.py)

```
urlpatterns = [
    path('', views.index, name='index'),
    path('users/<int:pk>/', views.users_detail, name='users_detail'),
    path('photos/<int:pk>/', views.photos_detail, name='photos_detail'),
    path('photos/new/', views.photos_new, name='photos_new'),
    path(
        'photos/<int:pk>/delete/',
        views.photos_delete,
        name='photos_delete'
    ),
    # 追加
    path(
        'photos/<str:category>/',
        views.photos_category,
        name='photos_category'
    ),
    path('signup/', views.signup, name='signup'),
    path(
        'login/',
        auth_views.LoginView.as_view(template_name='app/login.html'),
        name='login'
    ),
    path('logout/', auth_views.LogoutView.as_view(), name='logout'),
]
```

views.pyは以下のようになります。URL内の `<str:category>` に対応する文字列で、filterをかけます。

(~/PhotoService/app/views.py)

```
from .models import Photo, Category

def photos_category(request, category):
    # titleがURLの文字列と一致するCategoryインスタンスを取得
    category = get_object_or_404(Category, title=category)
```

```
# 取得したCategoryに属するPhoto一覧を取得
photos = Photo.objects.filter(category=category).order_by('-created_at')
return render(
    request, 'app/index.html', {'photos': photos, 'category': category}
)
```

テンプレート側は、トップページとほぼ同じ表示になるので、トップページでも使っているindex.htmlを使うように設定しています。ただし、今のままだとトップページなのか、絞り込み後のページなのかがわかりづらいため、以下のように「カテゴリー:動物の検索結果」という文字が表示されるようにします。

(~/PhotoService/app/templates/app/index.html)

```
{% extends 'app/base.html' %}

{% block content %}

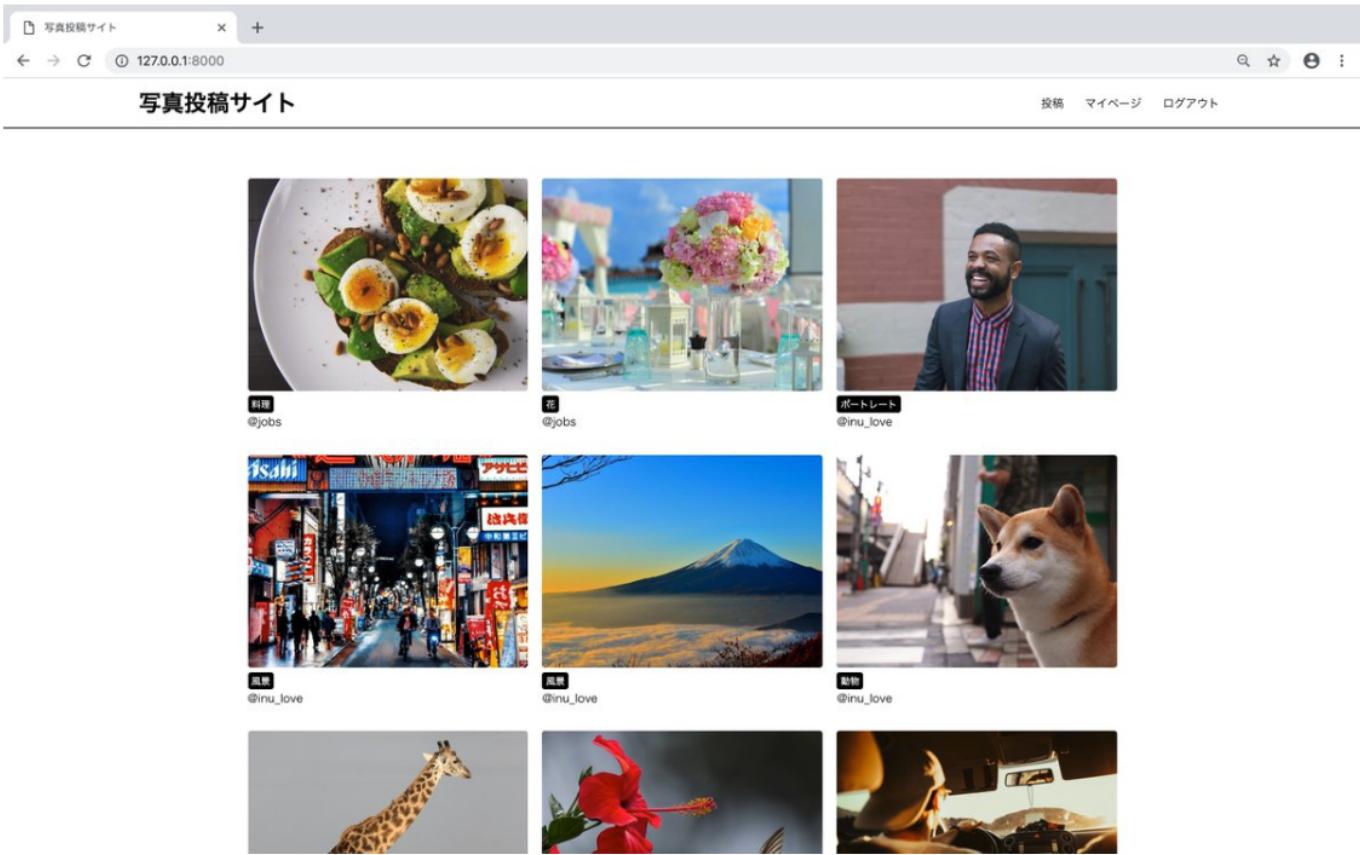
{% if category %}
<h2>カテゴリー:{{ category.title }} の検索結果</h2>
{% endif %}

{% include 'app/photos_list.html' %}

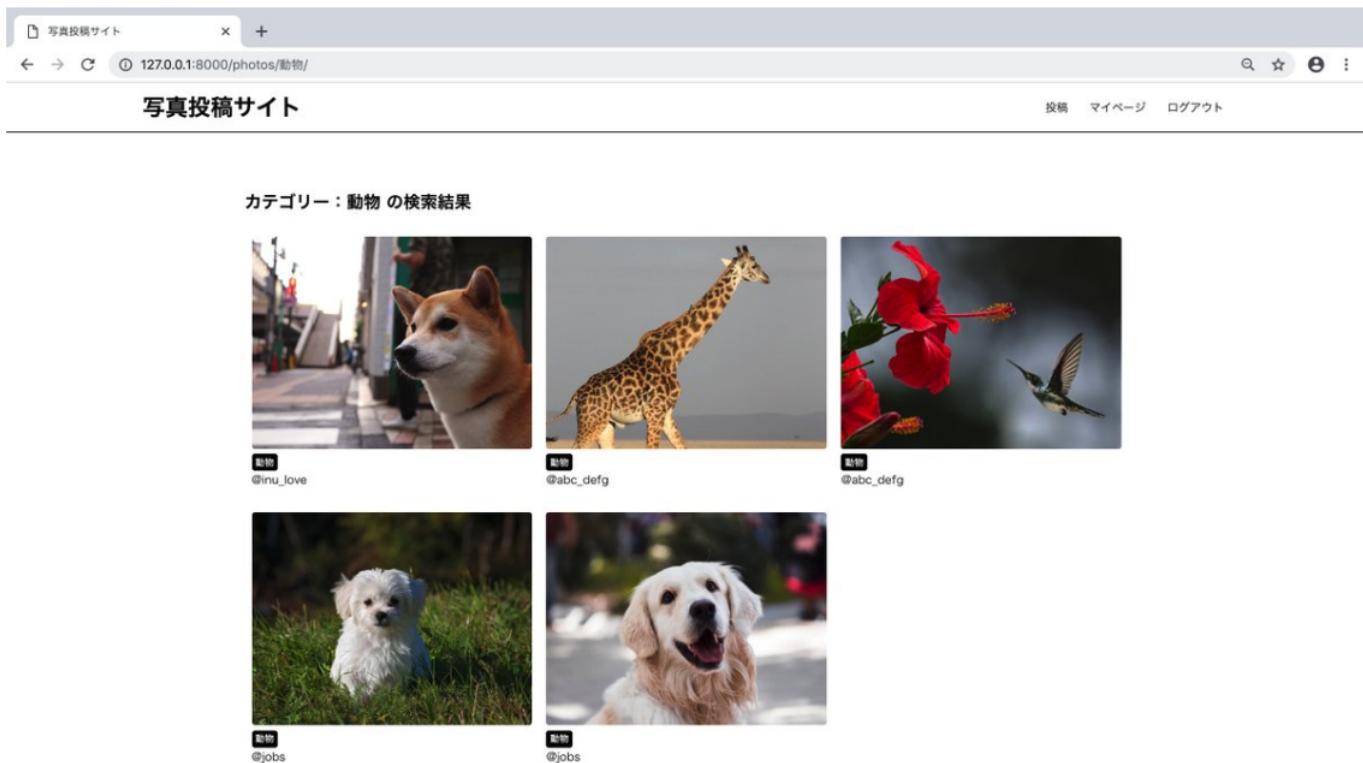
{% endblock %}
```

`{% if category %}` とif文を追加することで、views.pyからcategoryが渡されたとき(つまりphotos_category関数が実行されたとき)のみに、「検索結果」の文字が表示されます。トップページを表示するときは、index関数が実行されるので categoryは渡されず、この文字は表示されません。

- トップページ



- カテゴリーで絞り込んだページ



最後に、photos_list.htmlを更新して、カテゴリーボタンを押すと、絞り込み結果のページに遷移するようにしておきましょう。

(~/PhotoService/app/templates/app/photos_list.html)

```
<div class="photo-container">
{% for photo in photos %}
<div class="photo">
    <a href="{% url 'app:photos_detail' photo.id %}">
        
    </a>
    <div class="photo-info">

        <!-- URLを設定 -->
        <a href="{% url 'app:photos_category' category=photo.category %}">
            {{ photo.category }}
        </a>

        <a href="{% url 'app:users_detail' photo.user.id %}">@{{ photo.user }}</a>
    </div>
</div>
{% endfor %}
</div>
```

これでこの章のチュートリアルは終わりです。お疲れ様でした!

第5章 チュートリアル3 ECサイト

3つ目のチュートリアルでは、簡易的なECサイトを作ります。

ウェブ上に商品が表示してあり、それを一旦カートに入れてから、まとめて決済をするフローを学びます。各ページのイメージは以下のようになります。

- トップページ

Djamazon

Point: 38,600 お気に入り カート 注文履歴 ログアウト



シャンプー
Point: 400



バッグ
Point: 1,500



時計
Point: 1,000

- 商品ページ

Djamazon

Point: 47,400 お気に入り カート 注文履歴 ログアウト



時計

1,000ポイント

お気に入りに入れる

シンプルでかっこいい時計です。

数量:

カートに追加する

- カートページ

Djamazon

Point: 38,600 お気に入り カート 注文履歴 ログアウト

カート

郵便番号: 住所を検索
住所:

請求額: 1,800

[購入する](#)



時計

価格:	1,000
個数:	1
小計:	1,000

[1つ減らす](#) [1つ増やす](#)



シャンプー

価格:	400
個数:	2
小計:	800

[1つ減らす](#) [1つ増やす](#)

- お気に入り商品ページ

Djamazon

Point: 38,600 お気に入り カート 注文履歴 ログアウト

お気に入り商品



時計
Point: 1,000



シャンプー
Point: 400

このチュートリアルで学ぶ主なトピックは以下の通りです。

- カスタムユーザー モデル
- モデルの ManyToMany リレーションシップ
- Session を使った処理
- 外部APIの利用方法

このECサイトでは、これまでのようにusernameではなく、ユーザーのEmailアドレスでログインできるようにします。

各ユーザーは特定の商品をお気に入りすることができ、商品をカートに追加してから自分の住所を入力し、決済処理までをセッション管理できるようにします。住所の入力では、外部APIを使って郵便番号を入力すると住所が自動で入力されるようにします。

また、支払いについては実際のお金ではなく、簡易的に各Userに「ポイント」を振り分ける形をとり、これを支払いに使うようにします。実際のECサイトなどの決済部分には、外部の決済用APIを活用して決済の仕組みを導入することが多いです。

なお、このチュートリアルでは、第2章と同じ手順で「ecssite」という名前のプロジェクトを作成してスタートしてください。

5.1 カスタムユーザー モデルを作ろう

これまでのチュートリアルでは、UserモデルはDjango標準のモデルを利用してきました。もう少し具体的に言うと、`django.contrib.auth.models.User`をインポートして利用していました。

- [ソースコード](#)(357行目)

ソースコードを見るとわかりますが、このUserクラスは**AbstractBaseUser**、**AbstractUser**を継承しているクラスです。(AbstractBaseUser > AbstractUser > User の順で継承しています。)

AbstractBaseUser、AbstractUserのソースコードを見ると、ユーザー関連の様々なフィールドや機能が実装されていることがわかります。そのため、これらのクラスを継承したUserクラスをインポートするだけで、ユーザー周りの一通りの機能を使うことができました。

このUserモデルはインポートすれば一通りの機能を使えるので非常に便利なのですが、**実践的なWebサービスを作る場合にはいくつか不都合な点があります。**

例えば、デフォルトのUserモデルはusernameとパスワードを設定してユーザー認証をすることになっていますが、usernameの代わりにEmailアドレスやSNSアカウントを利用したログイン機能を実装したい場合もあります。また、デフォルトの属性以外にも、独自に属性情報を付け加えたい場合もあります。例えば、ユーザーの**年齢**や**性別**などの属性をUserモデルで保持したい場合などが考えられます。今回のチュートリアルだと、ユーザーごとに保持するポイントをUserモデルの属性として付け加えます。デフォルトのUserモデルだと、こういった独自のカスタマイズが非常に実装しづらいのです。

そんなときに利用するのが**カスタムユーザー モデル**です。`django.contrib.auth.models.User`をインポートして使うのではなく、自分でUserモデルを1から実装するのです。1から実装するといつても、**AbstractBaseUser**や**AbstractUser**を継承したクラスにすることで、デフォルトの機能で使い回したいものがあればその機能は使い回すようにします。

つまり、これまでのチュートリアルでは `AbstractBaseUser > AbstractUser > User` という継承関係のUserクラスを使っていたのに対して、`AbstractBaseUser > AbstractUser > 自作のUserクラス` や `AbstractBaseUser > 自作のUserクラス` という構成で、Userモデルをカスタマイズします。

AbstractUserを継承して `AbstractBaseUser > AbstractUser > 自作のUserクラス` の構成にするか、AbstractBaseUserを直で継承して `AbstractBaseUser > 自作のUserクラス` の構成にするかは場合に応じて決めるところ良いでしょう。

ソースコードをみるとわかりますが、AbstractUserクラスには `username` や `email` などデフォルトのフィールドが定義されています。そのため、デフォルトのUserモデルに含まれているフィールドなどは利用しながらも、

新しいフィールドやメソッドを独自に追加したい時は、 `AbstractBaseUser > AbstractUser > 自作のUserクラス` の構成にすると良いでしょう。

一方、`AbstractBaseUser` クラスは、ユーザー認証に関わるような基本的な部分は実装されていますが、それ以外は特に何も実装されていない状態です。つまり、デフォルトのUserクラスで使われていた `username` や `email` などのようなフィールドすら持っていないため、より柔軟にUserモデルを定義したい場合は `AbstractBaseUser > 自作のUserクラス` の構成を採用すると良いでしょう。

デフォルトのUserモデルの機能をベースにして、少しだけ手を加えるのであれば`AbstractUser`を継承すれば良いですが、`AbstractBaseUser`を直接継承する方が柔軟に変更を加えることができます。どちらを利用してカスタムユーザーを実装するかについては色々な意見があると思いますが、様々なWebサービスの要件に対応できるようになるために、`AbstractBaseUser`を直接継承するやり方を覚えておくのがオススメです。このチュートリアルでも`AbstractBaseUser`クラスを直接継承してカスタムユーザーを作成する方法を取ります。

Usersアプリケーションの作成

カスタムユーザーを作成するために、まずはそれを管理するためのアプリケーションを作りましょう。

(~/ecssite)

```
# アプリケーションの作成
$ python manage.py startapp users
```

これまでと同様に、`settings.py` ファイルにアプリケーションを追加します。さらに、このプロジェクトで使用する User モデルを指定するために、`AUTH_USER_MODEL` も追加する必要があります。

(~/ecssite/ecssite/settings.py)

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'users', # 追加
]

AUTH_USER_MODEL = 'users.User' # 追加
```

`AUTH_USER_MODEL` に `'users.User'` という値を代入していますが、`users` アプリケーションの`models.py` で定義する `User` というモデル(カスタムユーザー)をこのプロジェクトの認証用 User モデルとして利用するという意味になります。

カスタムユーザーの作成

それでは実際にカスタムユーザー モデルを作っていくましょう。カスタムユーザー モデルを作成する際に必要な項目は以下の通りです。

- AbstractBaseUserを継承したUserモデルを作成する
- BaseUserManagerを継承したUserManagerクラスを作成する
- Adminページをカスタムユーザー モデルに対応させる

まずは新しいUserモデルを作成しましょう。今回の要件は以下の通りです。

- 「username」の代わりに、「email」というフィールドを使う(必須かつユニーク)
- ユーザー ログインは「email」と「password」で行う
- 決済時の「お金」代わりに使う「ポイント」のフィールドを追加する
- 「お気に入り商品」のフィールドを追加する

models.py は以下のようになります。

(~/ecsite/users/models.py)

```
from django.db import models
from django.contrib.auth.models import PermissionsMixin
from django.contrib.auth.base_user import AbstractBaseUser
from django.utils import timezone

class User(AbstractBaseUser, PermissionsMixin):
    """カスタムユーザー モデル"""
    initial_point = 50000
    email = models.EmailField("メールアドレス", unique=True)
    point = models.PositiveIntegerField(default=initial_point)
    is_staff = models.BooleanField("is_staff", default=False)
    is_active = models.BooleanField("is_active", default=True)
    date_joined = models.DateTimeField("date_joined", default=timezone.now)

    objects = UserManager()

    USERNAME_FIELD = "email"
    EMAIL_FIELD = "email"
    REQUIRED_FIELDS = []

    class Meta:
        verbose_name = "user"
        verbose_name_plural = "users"
```

AbstractBaseUserを継承したUserモデルを作成しました。これがカスタムユーザー モデルとなります。
is_staffやis_activeフィールドはAbstractUser モデルのソースコードをほぼそのまま流用しています。一方で、
AbstractUserには定義されている `username` や `first_name` フィールド等は、実装していません。今回の
プロジェクトではこれらのフィールドは不要だからです。
また、email フィールドはユーザー認証の際に使うことになるので、ユニーク(他のユーザーと被らない)かつ必須の
フィールドにしています。

- [AbstractUserクラスのソースコード](#)(289行目)

ここでは同時にPermissionsMixinを継承していますが、これはDjangoにおける権限管理周りの機能を追加してくれるクラスです。例えば `is_superuser` のように、管理者権限を持っているかどうかを簡単に確認できるようなメソッドを追加してくれます。

- [PermissomMixinのソースコード](#)(204行目)

また、**USERNAME_FIELD**に `"email"` を指定することによって、emailというフィールドでこのプロジェクトのUserを一意に判別するという設定ができます。つまり、このフィールド(今回はemail)の情報でユーザーを検索すれば、必ず1人のユーザーだけが結果として表示されるという意味です。これまでのチュートリアルではusernameがこの役割を担ってきましたが、今回はemailに変更しました。

REQUIRED_FIELDSでは `createsuperuser` を行うときに必須となるフィールドを指定できます。しかし、**USERNAME_FIELD**に指定したフィールドとpasswordのフィールドについてはデフォルトで必須になります。今回はこの2つのフィールド以外は必要ではないので、空のリストを値として指定しています。

注意点として、現在の段階では「お気に入り商品」についてのフィールドは設定していません。これは、お気に入り商品の対象となるProductクラスを作成したときに紐づけるようにしましょう。

継承しているAbstractBaseUserクラスについても、一度実際のコードを確認しておくと良いでしょう。

- [AbstractBaseUserのソースコード](#)(47行目)

UserモデルはAbstractBaseUserを継承しているので、この中で実装されているメソッドは一通り使えることになります。

カスタムユーザーマネージャークラスの作成

続いて、UserManagerクラスを実装します。

Userクラス内で `objects = UserManager()` と書いていますが、この変数に代入しているクラスを作ります。

デフォルトのUserクラスを使った場合、UserManagerもデフォルトのものが使われます。

- [UserManagerのソースコード](#)(132行目)
- [デフォルトのUserクラスのobjectsにはUserManagerが設定されている](#)(326行目)

UserManagerがあることで、`create_user` メソッドなどが使ってUserオブジェクトを作成できたりします。

しかし、今回は独自にUserクラスを実装したので、UserManagerもそれに合わせて実装する必要があります。

作り方としては、デフォルトのUserManagerと同様にBaseUserManagerを継承したクラスを作れば良いのですが、今回のUserはemailフィールドが必須になっていたりするので、そこは独自に修正する必要があります。

`create_user` や `create_superuser` メソッドでは、usernameではなくemailでの登録をするようにします。

以下のようにUserManagerクラスを実装してください。ソースコードのUserManagerクラスとほぼ同じ実装ですが、`create_user` メソッドの引数でusernameではなくemailを指定している点など、一部カスタマイズしています。

(~/ecsite/users/models.py)

```
from django.contrib.auth.base_user import BaseUserManager # 追加

class UserManager(BaseUserManager):
    """カスタムユーザー マネージャー"""

    use_in_migrations = True

    def _create_user(self, email, password, **extra_fields):
        # emailを必須にする
        if not email:
            raise ValueError('The given email must be set')
        email = self.normalize_email(email)
        # emailを使ってUserデータを作成
        user = self.model(email=email, **extra_fields)
        user.set_password(password)
        user.save(using=self.db)
        return user

    def create_user(self, email, password=None, **extra_fields):
        extra_fields.setdefault('is_staff', False)
        extra_fields.setdefault('is_superuser', False)
        return self._create_user(email, password, **extra_fields)

    def create_superuser(self, email, password, **extra_fields):
        extra_fields.setdefault('is_staff', True)
        extra_fields.setdefault('is_superuser', True)
        if extra_fields.get('is_staff') is not True:
            raise ValueError('Superuser must have is_staff=True')
        if extra_fields.get('is_superuser') is not True:
            raise ValueError('Superuser must have is_superuser=True')
        return self._create_user(email, password, **extra_fields)

class User(AbstractBaseUser, PermissionsMixin):
    ...

```

最後に、Adminページから確認できるようにしましょう。

(~/ecsite/users/admin.py)

```
from django.contrib import admin
from django.contrib.auth.admin import UserAdmin
from django.contrib.auth.forms import UserChangeForm, UserCreationForm
from django.utils.translation import ugettext_lazy as _
from .models import User
```

```

class MyUserChangeForm(UserChangeForm):
    """User情報を変更するフォーム"""
    class Meta:
        model = User
        fields = '__all__' # 全ての情報を変更可能

class MyUserCreationForm(UserCreationForm):
    """Userを作成するフォーム"""
    class Meta:
        model = User
        fields = ('email',) # emailとパスワードが必要

class MyUserAdmin(UserAdmin):
    """カスタムユーザー モデルの Admin"""
    fieldsets = (
        (None, {'fields': ('email', 'password')}),
        (_('Permissions'), {
            'fields': (
                'is_active',
                'is_staff',
                'is_superuser',
                'groups',
                'user_permissions'
            )
        }),
        (_('Important dates'), {'fields': ('last_login', 'date_joined')}),
    )

    add_fieldsets = (
        (None, {
            'classes': ('wide',),
            'fields': ('email', 'password1', 'password2'),
        }),
    )

    form = MyUserChangeForm
    add_form = MyUserCreationForm
    list_display = ('email', 'is_staff')
    list_filter = ('is_staff', 'is_superuser', 'is_active', 'groups')
    search_fields = ('email',)
    ordering = ('email',)

admin.site.register(User, MyUserAdmin)

```

MyUserAdminというクラスを作りました。このクラスは、UserAdminクラスを継承しています。

- [UserAdminのソースコード](#)(41行目)

そのため、UserAdminクラスで定義されているメソッドなどはそのまま流用できます。ただし、UserAdminクラスではusernameを使う実装になっていますが、今回作るUserモデルにはusernameフィールドはないので、そこはemailを使うようにカスタマイズしています。

MyUserChangeFormやMyUserCreationFormも同様に、デフォルトのクラスを継承し必要に応じてオーバーライドしています。これらのフォームは、Admin画面でUserデータを作成・更新するときに使用されるフォームです。

- ソースコード

- UserCreationForm: 74行目
- UserChangeForm: 134行目

カスタムユーザーのマイグレーションは要注意

これで今回の要件を満たすカスタムユーザー モデルの定義ができました。実際にマイグレーションをしていきますが、ここで1つ重大な注意点があります。

実はこのチュートリアルで最初にカスタムユーザー モデルを定義したのには理由があります。

カスタムユーザー モデルは、基本的にプロジェクトの初期段階のタイミングでしか作成することができません。

「最初はデフォルトのUserモデルを使っていて、後からカスタムユーザー モデルに変更する」のようなことが基本的にはできないのです。

一度マイグレートした後にカスタムユーザーを作ろうとすると、データベースの整合性が取れなくなり、他のモデルからUserモデルを参照している場合などは後から修正することが非常に困難になります。

そのため、プロジェクトの初期段階ではUserモデルをカスタマイズするつもりがなくても、常にカスタムユーザーを使っておくというのがベストプラクティスです。そうすることで、あとからモデルの構造を変えたい場合にも柔軟に対応することができるようになります。

今回のプロジェクトを立ち上げたときにマイグレートしてしまっている場合は、デフォルトのUserモデルを使ってマイグレートされてしまっています。ですので、一度データベースを削除してしまいましょう。その後、マイグレートをしなおします。

(~/ecssite)

```
# データベースの削除
$ rm db.sqlite3

# マイグレート
$ python manage.py makemigrations
$ python manage.py migrate
```

ここまでできたらコマンドラインからsuperuserを作成してみてください。これまでのようにusernameではなく、メールアドレスによるユーザー登録になっているかと思います。

(~/ecssite)

```
# usernameではなくemailでsuperuserが作成できる
$ python manage.py createsuperuser
メールアドレス: sample@gmail.com
Password:
Password (again):
```

ローカルサーバーを立ち上げて、<http://127.0.0.1:8000/admin/users/user/> にアクセスすると、以下のようなページが確認できると思います。
これでカスタムユーザーを使い、emailでのユーザー登録をすることができるようになりました!

The screenshot shows the Django Admin interface for the 'Users' model. At the top, it says 'Django 管理サイト' and 'ようこそ SAMPLE@GMAIL.COM'. Below that, the URL 'ホーム > Users > Users' is shown. The main area is titled '変更する user を選択' (Select user to change). There is a search bar with a magnifying glass icon and a '検索' (Search) button. Below the search bar, there is a dropdown menu labeled '操作:' with '---' selected, and a '実行' (Run) button. A message says '1個の内ひとつも選択されていません' (No item has been selected). The user list table has columns: 'メールアドレス' (Email Address) and 'IS_STAFF'. One user is listed: 'sample@gmail.com' with 'IS_STAFF' checked (indicated by a green checkmark). At the bottom left of the table, it says '1 user'.

5.2 ログイン/サインアップ処理を作ろう

カスタムユーザー モデルの作成ができましたので、それに合わせたログイン処理を作っていきましょう。

このチュートリアルでは、カスタムユーザー関連の機能以外は、全て「app」という名前のアプリケーションの中で開発していきます。

まずは app アプリケーションを作り、INSTALLED_APPS にも追記しましょう。

(~/ecssite)

```
# app アプリケーションの作成
$ python manage.py startapp app
```

(~/ecssite/ecssite/settings.py)

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'users',
    'app', # 追加
]
```

ログイン処理については、前回のチュートリアルでも解説したように必要なことは大きく2つです。

- ログイン画面のHTMLファイルを作る
- ログイン画面、ユーザーがログインした後にリダイレクトするページ、ログアウトした後にリダイレクトするページそれぞれのURLを設定する

ページを作るときには先にURLを決めると、表示を確認しながらページを作ることができるのでスムーズに開発することができます。

今回の場合は、ログイン・ログアウト後のリダイレクトページはトップページにするので、簡易的なトップページもまとめて作ります。

以下のようにURLを設定しましょう。

(~/ecssite/ecssite/urls.py)

```
from django.contrib import admin
from django.urls import path, include # 追加

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('app.urls')), # 追加
]
```

(~/ecssite/app/urls.py)

```
from django.urls import path
from . import views
from django.contrib.auth import views as auth_views

app_name = 'app'
urlpatterns = [
    path('', views.index, name='index'),
    path(
        'login/',
        auth_views.LoginView.as_view(template_name='app/login.html'),
        name='login'
    ),
    path('logout/', auth_views.LogoutView.as_view(), name='logout'),
]
```

settings.pyにも以下の内容を追加します。

(~/ecssite/ecssite/settings.py)

```
LOGIN_URL = 'app:login'
LOGIN_REDIRECT_URL = 'app:index'
LOGOUT_REDIRECT_URL = 'app:index'
```

ここまで実装の内容は前回のチュートリアルと同様なので、復習する場合は**4.6**などを参照してください。今回は、`login/`でログイン画面が表示され、`logout/`にアクセスするとログアウトされます。

ログインのURLを設定したので、次はページを作ります。これも前回と同様で、`login.html`ファイルで作成します。

(~/ecssite/app/templates/app/base.html)

```
{% load static %}  
<!DOCTYPE html>  
  
<html>  
  <head>  
    <title>Djamazon</title>  
    <link rel="stylesheet" type="text/css" href="{% static 'css/style.css' %}">  
  </head>  
  <body>  
    <header>  
      <div class="container">  
        <h1><a href="{% url 'app:index' %}">Djamazon</a></h1>  
        <div class="header-menu">  
          {% if request.user.is_authenticated %}  
            <a href="{% url 'app:logout' %}">ログアウト</a>  
          {% else %}  
            <a href="{% url 'app:login' %}">ログイン</a>  
          {% endif %}  
        </div>  
      </div>  
    </header>  
    <div class="container">  
      {% block content %}{% endblock %}  
    </div>  
  </body>  
</html>
```

(~/ecssite/app/templates/app/login.html)

```
{% extends 'app/base.html' %}  
  
{% block content %}  
  
<form class="form-signin" method="POST">{% csrf_token %}  
  {% if form.errors %}  
    <p>ユーザー名かパスワードが間違っています。</p>  
  {% endif %}  
  <h2>ログイン</h2>  
  <label>メールアドレス</label>  
  <input name="username" type="text" /> <!-- emailではない -->  
  <br>  
  <label>パスワード</label>
```

```
<input type="password" name="password">
<br>
<input type="submit" value="ログイン">
</form>

{% endblock %}
```

ここで一点注意が必要です。4.6でも学んだように、このログインページではLoginViewが自動的に生成してくれるログイン用のフォームを使用しています。今回はカスタムユーザーを作成してメールアドレスでの登録をできるようにしていましたが、フォームのメールアドレスのフィールドに指定しているのは `<input name="username">` となっています。

これは、カスタムユーザー モデルを作成したときに、`USERNAME_FIELD="email"` と指定したため、ユーザーを一意に判別するための `username` という属性はメールアドレスに置き換わっているからです。間違えて `<input name="email">` と指定してもうまく動かないで気をつけてください。

最後に、トップページ用のViewとテンプレートを作成します。

(~/ecsite/app/views.py)

```
from django.shortcuts import render

def index(request):
    return render(request, 'app/index.html')
```

(~/ecsite/app/templates/app/index.html)

```
{% extends 'app/base.html' %}

{% block content %}

Djamazonトップページ

{% endblock %}
```

ログインページを確認するとメールアドレスとパスワードでのログインができるようになっています。
そのほか、ログイン・ログアウト時のリダイレクトなどもうまくいっているかを確認してください。

Djamazon

ログイン

Djamazonトップページ

これでログイン処理が完了しました。続いて、サインアップ(新規会員登録)の機能を追加しましょう。

チュートリアル②ではDjangoのデフォルトのUserモデルを利用していたため、付属のUserCreationFormをそのまま使うだけでサインアップ用のフォームを作ることができました。しかし、今回のチュートリアルではカスタムユーザーモデルを作成しているため、新しく設定したフィールドに合わせてユーザー作成のフォームを作る必要があります。

これを実現するために、UserCreationFormをオーバーライドしたCustomUserCreationFormクラスを作成し、サインアップページを作っていきます。

(~/ecsite/app/forms.py)

```
from django.contrib.auth import get_user_model
from django.contrib.auth.forms import UserCreationForm

class CustomUserCreationForm(UserCreationForm):
    class Meta:
        model = get_user_model()
        fields = ('email',)
```

デフォルトのUserモデルの場合、**django.contrib.auth.models.User**をインポートすることになりますが、今回はカスタムユーザーモデルを参照する必要があります。

カスタムユーザーモデルを参照するには**get_user_model**関数を使います。この関数ではsettings.pyの**AUTH_USER_MODEL**に設定したモデルを呼び出します。カスタムユーザーを使っている場合、どのファイルからもこちらの関数を利用することでUserモデルを呼び出すことができる覚えておきましょう。

続いて、サインアップページに対応するViewを作成します。

(~/ecsite/app/views.py)

```
from django.shortcuts import render, redirect # 追加
from django.contrib.auth import authenticate, login # 追加
from .forms import CustomUserCreationForm # 追加

def signup(request):
```

```

if request.method == 'POST':
    form = CustomUserCreationForm(request.POST)
    if form.is_valid():
        form.save()
        input_email = form.cleaned_data['email']
        input_password = form.cleaned_data['password1']
        new_user = authenticate(
            email=input_email,
            password=input_password,
        )
        if new_user is not None:
            login(request, new_user)
            return redirect('app:index')
    else:
        form = CustomUserCreationForm()
return render(request, 'app/signup.html', {'form': form})

```

先ほど作成したCustomUserCreationFormを使って新しいユーザーを作成しています。フォームの入力データからemailとpassword1の値を取り出し、authenticate関数でユーザーの認証を行っています。認証がうまくいった際には、ユーザーをそのままログインさせてトップページにリダイレクトする処理になっています。

サインアップページのテンプレートは以下のようにしてください。

(~/ecsite/app/templates/signup.html)

```

{% extends 'app/base.html' %}

{% block content %}

<h2>ユーザー登録</h2>

<form method="POST" action="{% url 'app:signup' %}">{% csrf_token %}
    <label>メールアドレス</label>
    {{ form.email }}
    {{ form.email.errors }}
    <br>
    <label>パスワード</label>
    {{ form.password1 }}
    {{ form.password1.errors }}
    <br>
    <label>パスワード(確認)</label>
    {{ form.password2 }}
    {{ form.password2.errors }}
    <br>
    <input type="submit" value="登録する">
</form>

{% endblock %}

```

URLも設定すればサインアップページは完成です。

(~/ecsitet/app/urls.py)

```
...
urlpatterns = [
    path('', views.index, name='index'),
    path('signup/', views.signup, name='signup'), # 追加
    ...
]
```

<http://127.0.0.1:8000/signup/> からユーザー登録ができる事を確認してみてください。

続いては、このWebサービスで使用するモデルを作成していきましょう。

5.3 必要なモデルを作ろう

Userモデル以外で必要になるモデルは以下の通りです。

- Product(商品を表すモデル)
- Sale(個別の売上情報を表すモデル)

商品の情報(Product)として、商品名、価格、商品の説明文、商品画像を用意します。売上情報(Sale)では、どのUserが、いつ、どの商品を何個買ったかを記録していきます。そのため、SaleはUserとProductをForeignKeyとして持つことになります。

また、今回のチュートリアルの要件として、Productの価格はいつでも変更できるが、Saleの情報はその商品を買ったタイミングでの価格を保持するようにします。たとえば、200円で売れたりんごが後々300円に値上がりしたとしても、その売上情報では 200円という単価を基準に情報が残るようにします。

(~/ecsitet/app/models.py)

```
from django.db import models
from django.contrib.auth import get_user_model

class Product(models.Model):
    """商品"""
    name = models.CharField(max_length=100)
    description = models.TextField(blank=True)
    price = models.PositiveIntegerField(default=0)
    image = models.ImageField(upload_to='product')

    def __str__(self):
        return self.name

class Sale(models.Model):
    """売上情報"""
    product = models.ForeignKey(Product, on_delete=models.PROTECT)
    user = models.ForeignKey(get_user_model(), on_delete=models.PROTECT)
```

```
amount = models.PositiveIntegerField("購入個数", default=0)
price = models.PositiveIntegerField("商品単価")
total_price = models.PositiveIntegerField("小計")
created_at = models.DateTimeField(auto_now=True)
```

基本的にはこれまで扱ってきた内容と同様です。Saleモデルでは、ForeignKeyのon_delete引数にmodels.PROTECTを設定しています。これは、そこに紐づいているモデルが削除された時に、勝手にSaleの情報まで消えて欲しくないためこのように設定しています。

モデルを作成したらAdminにも登録し、忘れずにマイグレーションしましょう。

(~/ecssite/app/admin.py)

```
from django.contrib import admin
from .models import Product, Sale

admin.site.register(Product)
admin.site.register(Sale)
```

(~/ecssite)

```
# ImageFieldに必要なライブラリをインストール
$ pip install Pillow
$ python manage.py makemigrations
$ python manage.py migrate
```

さらに、ImageFieldのアップロード先を指定し、画像を表示できるようにするため、settings.pyとurls.pyに下記を追加します。

(~/ecssite/ecssite/settings.py)

```
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
MEDIA_URL = '/media/'
```

(~/ecssite/ecssite/urls.py)

```
from django.conf import settings # 追加
from django.conf.urls.static import static # 追加

...
# 投稿した画像を表示するための設定、解説は4.4を参照
urlpatterns += static(
    settings.MEDIA_URL, document_root=settings.MEDIA_ROOT
)
```

これでモデルの作成ができました。動作確認のため、Adminページからいくつか商品情報(Product)を追加しておいてください。

5.5 ManyToManyフィールドで多対多の関係を作る

今回のサービスでは、ユーザーごとにお気に入り商品を登録できるようにしますが、Userモデルを作成した際には商品(Product)モデルが存在していなかったので紐づけすることができませんでした。

Productモデルも作成したので、Userモデルにフィールドを付け加えましょう。

(~/ecsite/users/models.py)

```
from app.models import Product # 追加

...
class User(AbstractBaseUser, PermissionsMixin):
    """カスタムユーザーモデル"""
    initial_point = 10000
    email = models.EmailField("メールアドレス", unique=True)
    point = models.PositiveIntegerField(default=initial_point)
    fav_products = models.ManyToManyField(Product, blank=True) # 追加
    ...

```

ここでは、`fav_products` というManyToMany(多対多)のフィールドを追加しています。

1人のUserは複数のお気に入り商品(Product)を持つ可能性があります。また、1つの商品は複数のUserからお気に入りされる可能性があるため、お互いの関係としては**ManyToMany(多対多)** で表現することができます。

ManyToManyFieldによって**中間テーブル**というものが自動生成されます。中間テーブルでは、お互いのモデルインスタンスの結びつきを記録します。

今回の場合、Userモデル内でProductモデルに紐づくManyToManyフィールドを作成しているため、Djangoが自動的に**User—Product**の中間テーブルを作成しています。次のようなイメージです。

(UsersテーブルとProductsテーブル)

Usersテーブル			
id	email	point	date_joined
	EmailField	PositiveInteger	Datetime
1	user1@gmail.com	50,000	2019/09/01
2	user2@gmail.com	25,000	2019/09/02
3	user3@gmail.com	33,000	2019/09/04
4	user4@gmail.com	3,000	2019/09/05
⋮			
⋮			
⋮			

Productsテーブル			
id	name	description	price
	CharField	TextField	PositiveInteger
1	おせんべい	⋮⋮⋮	3,000
2	ドリンクセット	⋮⋮⋮	2,500
3	スマートフォン	⋮⋮⋮	15,000
4	ふとん	⋮⋮⋮	6,800
⋮			
⋮			
⋮			

(中間テーブルのイメージ)

Users-Productsテーブル

id	User	Product
	Userモデル	Productモデル
1	user1	おせんべい
2	user2	おせんべい
3	user1	ふとん
4	user3	スマートフォン
⋮		
⋮		
⋮		

ManyToManyFieldを使うと中間テーブルは自動生成されます。このテーブルでは、2つのモデル(UserとProduct)のオブジェクトを保存することで多対多の関係を表しています。

上図の例で言うと、user1は「おせんべい」と「ふとん」の2つの商品をお気に入りしていることになり、「おせんべい」と言う商品はuser1、user2の2人にお気に入りされていることになります。

中間テーブルを自動生成せずに、自分で作成してフィールドをカスタマイズする方法もあります。例えば、その中間テーブルのデータが作られた日時(お気に入り登録した日時)などを記録する場合などに使われます。詳細は、ブログ記事で解説しています。

- 【Django】ManyToManyフィールド throughで中間テーブルを自作する

循環Importの解消

ここまで実装した状態でローカルサーバーを立ち上げると、以下のようなエラーが出ると思います。

(~/ecsite)

```
$ django.core.exceptions.ImproperlyConfigured: AUTH_USER_MODEL refers to model 'users.User' that has not been installed
```

これは、**循環Import**が原因のエラーです。

現在のファイルを確認してみると、usersアプリケーションのmodels.pyでは、`app.models.Product` をインポートし、appアプリケーションのmodels.pyでは、`users.models.User` をインポートしています。

もう少し詳しくみてみると、Saleモデルは、Userモデルの情報がなければ作成できないのにもかかわらず、Userモデルの作成にはSaleモデルが定義されているファイル(app/models.py)の情報が必要という状態になっています。

鶏と卵の話ではないですが、どちらのファイルを先に参照するべきなのか、Djangoには判別がつかない状況になっているのです。

このように、お互いのファイルからお互いをインポートし合っている状態を**循環Import**と呼び、どちらのファイルもうまく動かなくなってしまいます。

この場合、DjangoではForeignKeyフィールドの引数をString型にして、参照するモデルの参照タイミングを遅らせることによって解決できます。つまり、Saleモデルの定義時にはUserモデル自体を参照するのではなく、`'users.User'` というString型を参照しておくことで、このエラーを解決できます。app/models.pyのSaleモデルの定義を以下のように修正してください。

(~/ecssite/app/models.py)

```
class Sale(models.Model):
    """売上情報"""
    product = models.ForeignKey(Product, on_delete=models.PROTECT)
    user = models.ForeignKey('users.User', on_delete=models.PROTECT) # 修正
    . . .
```

このようにすることで、問題なくサーバーが立ち上がります。ここまで情報をマイグレートして、ローカルサーバーが立ち上がるかを確認してみてください。

(~/ecssite)

```
$ python manage.py makemigrations
$ python manage.py migrate
```

最後に、Adminページもアップデートしておきましょう。fieldsetsにfav_productsを追加してください。

(~/ecssite/users/admin.py)

```
class MyUserAdmin(UserAdmin):
    """カスタムユーザーモデルのAdmin"""
    fieldsets = (
        (None, {'fields': ('email', 'password', 'fav_products')}), # 修正
        (_('Permissions'), {
            'fields': (
                'is_active',
                'is_staff',
                'is_superuser',
                'groups',
```

```
        'user_permissions'
    )
}),
(_('Important dates'), {'fields': ('last_login', 'date_joined')}),
)
...
.
```

続いて、各ページを作成していきます。

5.4 商品をページに表示しよう

必要な情報を登録できるようになったので、それらを表示するページを作っていきましょう。これから作成するページは、商品一覧が並ぶトップページと、商品詳細ページです。基本的なUIを整えたあと、詳細ページからお気に入り商品を登録・解除できるようにしてみます。

商品詳細ページの作成

トップページは簡易的なものがすでにあるので、先に詳細ページを作りましょう。これまでと同様に、**URLを設定**→**View関数を作成**→**Templateを作成**という手順で進めます。

app/urls.pyに商品詳細ページのURLを追加してください。商品ID毎にページを分けるようにします。

(~/ecssite/app/urls.py)

```
urlpatterns = [
    ...
    path('product/<int:product_id>', views.detail, name='detail'),
]
```

URLを設定したので、対応するView関数を作成します。

(~/ecssite/app/views.py)

```
# 追加
from django.shortcuts import get_object_or_404, render, redirect
from .models import Product
...

def detail(request, product_id):
    product = get_object_or_404(Product, pk=product_id)
    context = {'product': product}
    return render(request, 'app/detail.html', context)
```

特定のProductを取得してきて、データを表示します。

(~/ecsite/app/templates/app/detail.html)

```
{% extends 'app/base.html' %}

{% load humanize %}

{% block content %}

<div class="product-detail">
    <div class="product-detail-image">
        
    </div>

    <div class="product-detail-info">
        <h2>{{ product.name }}</h2>
        <hr>
        <div class="point-fav-section">
            <h4>{{ product.price | intcomma }}ポイント</h4>
        </div>
        <p>{{ product.description }}</p>
    </div>
</div>

{% endblock %}
```

工夫している点として、価格の表示方法があります。桁がある程度大きくなる数値に関しては、3桁ごとにカンマで区切ってあげると可読性が上がります。そのため、Djangoでは `django.contrib.humanize` というモジュールでこれをサポートしています。

settings.pyのINSTALLED_APPSにこのモジュールを追加後、`NUMBER_GROUPING = 3` と記述し3桁ごと区切る設定をします。

`{% load humanize %}` をテンプレートファイルに記述すると、そのファイルでこの機能が使えるようになります。`{{ 数値 | intcomma }}` という形で数値を修飾できます。

(~/ecsite/ecsite/settings.py)

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'django.contrib.humanize', # 追加
    'users',
    'app',
]

NUMBER_GROUPING = 3 # 追加
```

トップページを整える

トップページには、商品一覧を表示しましょう。Viewからは、Productの一覧を渡すようにします。

(~/ecsite/app/views.py)

```
def index(request):
    products = Product.objects.all().order_by('-id')
    return render(request, 'app/index.html', {'products': products})
```

テンプレートは以下のように設定します。CSSファイルはこのチュートリアルを通して使うものなので、現状使っていない属性も含めて記述しておいてください。

(~/ecsite/app/templates/app/base.html)

```
{% load static %}
{% load humanize %}

<!DOCTYPE html>
<html>
<head>
    <title>Djamazon</title>
    <link rel="stylesheet" type="text/css" href="{% static 'css/style.css' %}">
</head>
<body>
    <header>
        <div class="container">
            <h1><a href="{% url 'app:index' %}">Djamazon</a></h1>
            <div class="header-menu">
                {% if request.user.is_authenticated %}
                    <span>Point: {{ user.point | intcomma }}</span>
                    <span><a href="#">お気に入り</a></span>
                    <span><a href="#">カート</a></span>
                    <span><a href="#">注文履歴</a></span>
                    <span><a href="{% url 'app:logout' %}">ログアウト</a></span>
                {% else %}
                    <span><a href="{% url 'app:login' %}">ログイン</a></span>
                    <span><a href="{% url 'app:signup' %}">新規登録</a></span>
                {% endif %}
            </div>
        </div>
    </header>
    <div class="container">
        {% for message in messages %}
            <p class="message-{{ message.tags }}">{{ message }}</p>
        {% endfor %}
    </div>
</body>
</html>
```

```
{% block content %}{% endblock %}
</div>
</body>
</html>
```

(~/ecsite/app/templates/app/index.html)

```
{% extends 'app/base.html' %}
{% load humanize %}

{% block content %}

<div class="product-container">
    {% for product in products %}
        <div class="product">
            <a href="{% url 'app:detail' product.id %}">
                
            </a>
            <div class="product-info">
                <div class="product-name">{{ product.name }}</div>
                <div class="product-price">
                    Point: {{ product.price | intcomma }}
                </div>
            </div>
        </div>
    {% endfor %}
</div>

{% endblock %}
```

また、CSSファイルも追加するため、以下の内容もsettings.pyに追記します。

(~/ecsite/ecssite/settings.py)

```
STATIC_URL = '/static/'
STATICFILES_DIRS = (
    os.path.join(BASE_DIR, "static"),
)
```

CSS ファイルは少し長いですが、コピペしてください。

(~/ecsite/static/css/style.css)

```
html, body {
    margin: 0;
}
```

```
div {
  box-sizing: border-box;
}

a {
  text-decoration: none;
  color: #000;
}

.container {
  width: 80%;
  margin: auto;
  overflow: auto;
}

header {
  border-bottom: solid 1px #000; height: 70px;
}

header .container {
  display: flex;
  justify-content: space-between;
}

body > .container {
  width: 65%;
  padding-top: 60px;
}

header h1 {
  margin: 0;
  height: 100%;
  line-height: 70px;
}

.header-menu {
  display: flex;
  align-items: center;
}

.header-menu a {
  padding: 10px;
  margin-left: 10px;
}

.fav-products-title {
  font-weight:bold;
  font-size: 28px;
  margin-bottom: 20px;
}

.product {
  width: calc(100%/3);
  float: left;
```

```
padding: 10px;
margin-bottom: 15px;
}

.product a {
  display: block;
}

.product-info a:hover {
  text-decoration: underline;
}

.product-info .category {
  display: inline-block;
  padding: 2px 5px;
  border-radius: 4px;
  font-size: 12px;
  color: #fff;
  background-color: #000;
}

.user-name {
  font-size: 30px;
}

.product-img {
  width: 100%;
  max-width: 500px;
  height: 300px;
  object-fit: cover;
  border-radius: 4px;
}

.product-img:hover {
  opacity: 0.8;
}

.product-detail {
  display: flex;
  justify-content: flex-start;
  padding: 10px;
}

.product-detail-image {
  flex: 1;
}

.product-detail-info {
  flex: 1;
}

.product-img {
  object-fit: contain;
}
```

```
.product-detail-info {
  margin-left: 10px;
}

.product-detail-info .point-fav-section {
  display: flex;
  justify-content: space-between;
  align-items: center;
}

.product-detail-info .fav_button {
  text-align: center;
  display: inline-block;
  padding: 8px 16px;
  border: 2px solid blue;
  border-radius: 4px;
  font-size: 16px;
  font-weight: bold;
  color: blue;
}

.product-detail-info .is_faved {
  background-color: blue;
  border-color: blue;
  color: #FFF;
}

.product-detail-info .fav_button:hover {
  background-color: blue;
  border-color: blue;
  color: #FFF;
}

.product-detail-info .purchase-button {
  display: inline-block;
  text-align: center;
  background-color: orange;
  font-size: 16px;
  color: #FFF;
  font-weight: bold;
  padding: 10px 16px;
  border-radius: 4px;
  border-bottom: 4px solid #a26a03;
}

.product-detail-info .purchase-button:active {
  transform: translateY(4px);
  border-bottom: none;
}

.cart {
  margin-bottom: 20px;
}
```

```
.purchase-form {
  width: 100%;
  margin: 10px auto;
}

.purchase-form .purchase-button {
  display: inline-block;
  text-align: center;
  background-color: orange;
  font-size: 16px;
  color: #FFF;
  font-weight: bold;
  padding: 10px 16px;
  float: right;
  margin-right: 5px;
  border-radius: 4px;
  border-bottom: 4px solid #a26a03;
}

.purchase-form-container {
  width: 100%;
  display: flex;
  justify-content: space-between;
}

.purchase-form-address {
  flex: 1;
}

.purchase-form-pay {
  flex: 1;
}

.order-product {
  padding: 20px;
  display: flex;
}

.order-product-image {
  flex: 1;
}

.order-product-info {
  flex: 1;
  font-size: 16px;
}

.order-product-info .info-value {
  float: right;
}

.order-product-info .info-timestamp {
  float: right;
}
```

```
font-size: 12px;
font-style: italic;
margin-top: 8px;
}

.order-product-amount-form {
  float: right;
  margin: 10px;
}

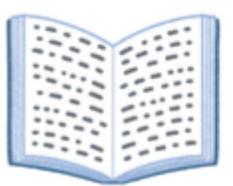
.order-product-amount-form input {
  font-size: 20px;
}

.message-success {
  background-color: #00d1b2;
  color: white;
  padding: 10px;
  padding-left: 30px;
  border-radius: 4px;
}

.message-warning {
  background-color: #ffc107;
  color: white;
  padding: 10px;
  padding-left: 30px;
  border-radius: 4px;
}
```

ここまでで、トップページと商品ページが整ってきたかと思います。

- トップページ

おせんべい
Point: 600ドリンクセット
Point: 2,200スマートフォン
Point: 24,000

- 商品ページ



おせんべい

600ポイント

おいしいおせんべいです。

商品お気に入りボタン

商品詳細ページからは、その商品をお気に入りできるようにしましょう。「お気に入りに追加する」というボタンを押すと、その商品をユーザーのfav_productsに追加します。fav_productsに追加した商品は、ヘッダーの「お気に入り」のリンクから確認できるようにします。

まずは、「お気に入りする・お気に入りから外す」の処理をするView関数を用意しましょう。以下の内容を追加してください。

(~/ecsite/app/views.py)

```

from django.contrib.auth.decorators import login_required
from django.views.decorators.http import require_POST

@login_required
@require_POST
def toggle_fav_product_status(request):
    """お気に入り状態を切り替える関数"""

    product = get_object_or_404(Product, pk=request.POST["product_id"])
    user = request.user

    if product in user.fav_products.all():
        # productがユーザーのfav_productsに既に存在している場合(お気に入り済の場合)
        # → productをfav_productsから除外する(お気に入りを外す)
        user.fav_products.remove(product)
    else:
        # productがユーザーのfav_productsに存在しない場合(お気に入りしていない場合)
        # → productをfav_productsに追加する(お気に入り登録する)
        user.fav_products.add(product)
    return redirect('app:detail', product_id=product.id)

```

この関数は、ユーザーが対象の商品(product)を「お気に入りしているかどうか」によって処理を切り分けます。既にお気に入りしている商品であれば、お気に入りから外します。お気に入りしていない商品であれば、お気に入りに登録します。

ManyToManyFieldにインスタンスを追加するときは `フィールド名.add(インスタンス)`、除外するときは `フィールド名.remove(インスタンス)` が使えます。

`fav_products`に追加するためにはユーザーを識別する必要があるため、`login_required`デコレータを使っています。

先ほど作成した商品ページのHTMLに、お気に入りボタンを追加してみましょう。

(~/ecssite/app/templates/app/detail.html)

```

. . .

<div class="product-detail-info">
    <h2>{{ product.name }}</h2>
    <hr>

    <div class="point-fav-section">
        <h4>{{ product.price | intcomma }}ポイント</h4>

        <!-- 追加 -->
        {% if request.user.is_authenticated %}
            <form action="{% url 'app:toggle_fav_product_status' %}"
method="POST">
                {% csrf_token %}
                <input type="hidden" name="product_id" value="{{ product.id }}">
                {% if product in user.fav_products.all %}
                    <input type="submit" name="submit" class="fav_button" value="お
気に入りから外す">
                {% else %}
                    <input type="submit" name="submit" class="fav_button" value="お
気に入り登録する">
                {% endif %}
            </form>
        {% endif %}
    </div>
</div>

```

```

    {% else %}
        <input type="submit" name="submit" class="fav_button" value="お
    気に入りに入れる">
        {% endif %}
    </form>
    {% endif %}

</div>
<p>{{ product.description }}</p>
</div>

...

```

ユーザーがログインしている場合のみ、「お気に入りボタン」を表示する設定にしました。

ユーザーが既にお気に入りに登録している商品であれば、「お気に入りから外す」と表示し、お気に入りしていない商品であれば、「お気に入りに入る」と表示します。

「ユーザーがお気に入りしているかどうか」は `if product in user.fav_products.all` の部分で判定しています。product変数には、対象商品のインスタンスが代入されています(detail関数から渡ってきてている)。ユーザーのfav_productsフィールドにproductが存在していれば、お気に入り済ということになります。

続いて、お気に入り商品一覧画面用のViewとテンプレートを作成します。

(~/ecsite/app/views.py)

```

@login_required
def fav_products(request):
    user = request.user
    products = user.fav_products.all()
    return render(request, 'app/index.html', {'products': products})

```

テンプレートはトップページと同じもの(`app/index.html`)を使い回していますが、お気に入り商品一覧画面と分かるように、index.htmlに少し追記します。

(~/ecsite/app/templates/app/index.html)

```

...
{% block content %}

<!-- 追加 -->
{% if 'fav_products' in request.path %}
<div class="fav-products-title">
    お気に入り商品
</div>
{% endif %}

<div class="product-container">
...

```

if文を追加しました。ユーザーがアクセスしたURL(`request.path`)に`'fav_products'`という文字列が含まれていれば、「お気に入り商品」を画面に表示する設定にしています。このページのURLは、`http://127.0.0.1:8000/fav_products/`を想定しています。

最後に、お気に入りボタンとお気に入りページのURLを設定し、`base.html`のヘッダーにリンクを追加します。

(~/ecssite/app/urls.py)

```
urlpatterns = [
    ...
    path('fav_products/', views.fav_products, name='fav_products'),
    path(
        'toggle_fav_product_status/',
        views.toggle_fav_product_status,
        name='toggle_fav_product_status'
    ),
]
```

(~/ecssite/app/templates/app/base.html)

```
...
<!--お気に入りのリンクを追加 -->
<span>Point: {{ user.point | intcomma }}</span>
<span><a href="{% url 'app:fav_products' %}">お気に入り</a></span>
<span><a href="">カート</a></span>
...
```

これで商品ページからお気に入り商品の追加・削除ができるようになりました。実際にお気に入りボタンを押してみて、商品が`http://127.0.0.1:8000/fav_products/`に追加されることを確認してみてください。

5.6 セッションを使ってカートに商品を追加しよう

商品ページができたので、今度はカートに商品を追加できる機能をつけましょう。

ECサイトでは、1商品ごとに決済するのではなく、一度カートに商品を入れた後に、まとめて決済するフローが一般的です。このように、一定期間内だけ有効な情報を保持しておくときに便利なのが**Cookie(クッキー)**と**セッション**という概念です。

Cookieとはなにか

Cookieとセッションを理解するためには、3.8でも解説した**HTTP**についてもう少し詳しく知る必要があります。HTTPはブラウザとサーバー間で通信するときの規約のことでしたね。特定のURLにアクセスしたとき(リクエスト

を送ったとき)に、そのリクエストに対応するレスポンスをサーバーが返すという一連の流れが、HTTPという約束事に則って行われています。

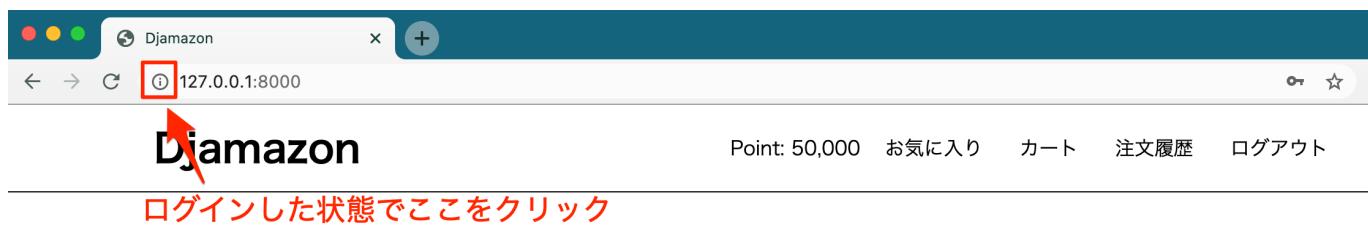
このHTTPによるやりとりの重要な概念として、**ステートレス(stateless、情報を持たないこと)**であることが挙げられます。これはどういうことかというと、ブラウザとサーバー間の1回1回の通信は全て独立した通信であり、1回の通信で処理が完結するということです。「過去にどのような通信が行われたか」のような情報を使うことはできません。つまり、HTTPではページを遷移する時にユーザーの情報を引き継ぐようなことはができないということです。そのためHTTPだけでは、「ユーザーがどの商品をお気に入りしているのか」のような状態を管理することはできないのです。

ここで、特定のユーザーについての情報を保存しておく仕組みが必要になります。ここでCookieが役に立ちます。

Cookie(クッキー)とは、**ブラウザに保存されるテキスト情報**のことです。例えば、ユーザーの認証情報(ログインしているかどうかなど)、最終ログインの日時などの情報を、そのWebサービスにアクセスしてきたブラウザ上に保持します。こうすることで、1度ログインしたユーザーを、次のページに移動した時も同じユーザーであることを継続して認識することができるようになります。

今回開発しているサイトでも、ログイン機能を実装しているので、ブラウザにCookieが保存されているはずです。ログインをした上で、以下の手順で自分のブラウザにCookieが保存されていることを確認してみましょう。

画像はChromeブラウザを使った例です。



「Cookie」を選択すると、以下のようなポップアップが表示されます。

A screenshot of a cookie settings pop-up window titled '使用中の Cookie'. It has two tabs: '許可' (Allow) and 'ブロック中' (Blocked). The '許可' tab is selected. The window displays a list of cookies set by the page. One cookie, 'sessionid', is highlighted with a grey background. Below the list, detailed information for the 'sessionid' cookie is shown:

名前	sessionid
コンテンツ	wonomzhvy4znbl7qc78pzezj96dko506
ドメイン	127.0.0.1
パス	/

At the bottom of the pop-up, there are three buttons: 'ブロック' (Block), '削除' (Delete), and a blue '完了' (Done) button. The background of the pop-up shows a small image of a hand holding a bottle of Maui Moisture shampoo.

このブラウザでは、`csrf_token`と`sessionid`という名前のCookie情報が保存されていることがわかります。「コンテンツ」となっているところが実際のCookieの値です。この`sessionid`の値が、HTTPリクエストする度にサーバーに送られるので、サーバー側ではアクセスしてきた人のユーザー識別ができるのです。

それではここで、ポップアップの削除ボタンを押してCookieを削除してみましょう。削除した上でブラウザをリロードしてみると、先ほどまでログインしてたはずなのにログアウト状態に切り替わるはずです。Cookieを削除したことで、HTTPリクエスト(リロード)時にもCookie情報は送信されず、サーバー側でユーザー識別をできなかったため、ログアウトされたということです。

もう一度ログインしてみると、再度Cookieがブラウザに保存されることが確認できるはずです。

セッションとはなにか

続いて、セッションについても理解しましょう。セッションとは、一連の処理の始まりから終わりまでを表しています。

例えば、今回のチュートリアルのように、サービスにログインして、商品をカートに入れて、決済を行う、といった一連の処理は、1つのセッションと捉えることができます。このセッションの中で、『どのユーザーがどの商品を何個カートに入れていて、合計金額はいくらなのか』などの情報を保持することで、ECサイトとしての機能を開発することができるようになります。

このセッションを管理するために、Cookieを使っています。最初にWebサービスにログインしたときに、一意なCookieをブラウザに保存します。すると、それ以降サーバーにリクエストを送る際には毎回Cookieの情報も送信されることになります。リクエストを受け取ったサーバー側では、「リクエストに含まれるCookieの値を参照する」ことで、「どのユーザーからリクエストが送られてきたのか」や「そのユーザーは何をカートに入れているのか」などを判別できるようになります。同じCookieの値を利用している期間は1つのセッションとして扱うことができるようになるのです。

Djangoの認証機能でもセッションが使われています。一度`username`と`password`を送信して認証が完了してしまえば、Cookie情報がブラウザに保存されるため、それ以降はCookie情報をを利用して認証処理を行います。この機能のおかげで、毎回ユーザーネームとパスワードを送信しなくても、認証が必要なページにアクセスできるのです。

また、特定のセッション内で利用することができる変数をセッション変数と言います。セッション変数の中に、ECサイトのカートの情報などを保存して、決済時に利用するといった使い方ができます。セッション変数は、特定のセッションの有効期限が切れるまで情報が保持されます。Djangoの場合、ユーザーがログアウトした時点でセッションは切れて、保持していた情報は失われます。(カートに商品を入れていたとしてもログアウトした時点でカートは空になります。)

上記の理由から、永続性の必要なデータはちゃんとデータベースに保存するなど、状況に応じて対応する必要があります。例えば、一時的にカートに保存してある商品情報はログアウト時に消去されても大した問題はありませんが、ユーザーの購入履歴はずっと保存しておくことが望ましいため、データベースに保存します。

セッションはDjangoに限らず、Web系のサービスであれば広く利用されている仕組みで、特定のサイトやブラウザ間での状態(state)を引き継ぐ機能を提供します。セッションを利用すると、任意のデータをブラウザごとに保存することができ、その情報を使ってユーザー別に表示を出し分けるようになります。

今回のECサイトの場合、カートに商品を入れた時点ではセッション変数にデータを保存するようにし、決済の時点で初めてSaleモデルのオブジェクトとしてデータベースに保存するというフローを取ります。

Djangoのセッションについてのドキュメントはこちらになります(使い方はこの後解説します)。

<https://docs.djangoproject.com/ja/2.2/topics/http/sessions/>

Djangoのセッションの基本

Djangoでセッション変数を使うためには、`settings.py`で以下の設定が必要になります。こちらの設定はDjangoでプロジェクトを立ち上げた時にデフォルトで設定されているものなので、特に追加で作業が必要なものではありません。

(~/ecsite/ecsite/settings.py)

```
INSTALLED_APPS = [
    ...
    'django.contrib.sessions',
    ...
]

MIDDLEWARE = [
    ...
    'django.contrib.sessions.middleware.SessionMiddleware',
    ...
]
```

実際に、ここまで作ってきたサイトでセッション管理がされていることを確かめてみます。

Djangoでは、セッション情報をデータベースに保存しています。`django.contrib.sessions.models.Session`のモデルを使っているので、他のモデルと同様にshellからデータを参照することができます。

1度サイトでログインをした上で、shellを起動して以下のコードを打ってみてください。

```
$ python manage.py shell
>>> from django.contrib.sessions.models import Session

# データベースに保存されているSession情報を全件取得
>>> Session.objects.all()
<QuerySet [<Session: m19z9cnmmvgijfdv316uzbpx9pqsn3im>, <Session:
9sug6vcwv0cea2iwry52w0ti1nfy38q1>, <Session:
5rvf6de4h9x6lrqwk6ydpc09atv8iai>, <Session:
ey33ovxnfuge653tikitkekw6zx1r20b>]>

# 1つ目のSessionオブジェクトを取得
>>> s = Session.objects.first()

# get_decoded()メソッドでセッション情報を表示
>>> s.get_decoded()
{'_auth_user_id': '1', '_auth_user_backend':
'django.contrib.auth.backends.ModelBackend', '_auth_user_hash':
'70fd97ba9b89faf8171981e4294f6dff070e9fe7'}

# セッションの有効期限を表示
>>> s.expire_date
datetime.datetime(2020, 12, 3, 10, 53, 71841, tzinfo=<UTC>)
```

このように、セッション情報をデータベースから取得することができます。
先ほど、ログアウトした時点でセッションは切れると説明しましたが、セッションには有効期限があり、この期限が過ぎた時にもセッションは切れます。

View関数内では、`request`(View関数の最初の引数)経由でセッション変数にアクセスすることができます。このセッションは、受信したCookieでユーザーを判別し、そのユーザーに合わせたセッションを管理します。
セッション変数は辞書型のオブジェクトのため、通常のPythonでの辞書型の扱いと同じ形で利用することができます。View関数におけるセッションの扱いの基本は以下の通りです。

(セッションの扱い方)

```
# セッションから値を取り出す
cart = request.session['cart']

# セッションに値を保存する
request.session['cart'] = 'value'

# セッションにデータがあるかを確認する
if 'cart' in request.session:
    ...

# セッションのデータを削除する
del request.session['cart']
```

カートに商品を追加する機能では、以下のような内容のセッションを扱っていると考えてください。

(cartセッションのイメージ)

```
# sessionはcartというキーを持ち、cartキーの値は
# キーがプロダクトID、値が商品個数の辞書型になっている
# 以下の例では、id1の商品が5個、id4の商品が3個、id10の商品が6個カートに入っていることを
表す
session = {
    'cart': {
        '1': 5,
        '4': 3,
        '10': 6,
    }
}
```

このセッションの辞書型を扱うことでカートページの情報を更新していきます。

カートに追加する機能を作ろう

それでは商品ページからカートのセッションに商品を追加するフォームを作りましょう。
画面のイメージは以下の様になります。数量を入力し、「カートに追加する」ボタンを押すとセッションに情報が追加されます。

おせんべいを2個カートに入れました！



おせんべい

200ポイント

お気に入りから外す

テスト

数量:

カートに追加する

まずは専用のフォームを作ります。以下の内容をforms.pyに追加してください。

(~/ecsite/app/forms.py)

```
from django import forms

class AddToCartForm(forms.Form):
    num = forms.IntegerField(
        label='数量',
        min_value=1,
        required=True
    )
```

カートに追加する個数(num)を入力できるフォームです。`min_value = 1`で入力できる最小値を1にしています。0やマイナスの数値は、購入個数として入力できないようにするために`required=True`で強制入力をしています。

このフォームを商品ページのViewとテンプレートに追加していきます。

(~/ecsite/app/views.py)

```
from django.contrib import messages # 追加
from .forms import AddToCartForm # 追加

# 内容を修正
def detail(request, product_id):
    product = get_object_or_404(Product, pk=product_id)

    # 「カートに追加する」ボタンが押された時
    if request.method == "POST":
        add_to_cart_form = AddToCartForm(request.POST)
        if add_to_cart_form.is_valid():
            num = add_to_cart_form.cleaned_data['num']
```

```

# セッションに'cart'というキーが存在するかどうか(初めてカート追加するかどうか)で
処理を分ける
if 'cart' in request.session:
    # すでに対象商品がカートにあれば新しい個数を加算、なければ新しくキーを追加する
    if str(product_id) in request.session['cart']:
        request.session['cart'][str(product_id)] += num
    else:
        request.session['cart'][str(product_id)] = num
else:
    # 初めてのカート追加の場合、新しく'cart'というキーをセッションに追加する
    request.session['cart'] = {str(product_id): num}
messages.success(request, f"{product.name}を{num}個カートに入れました!")
return redirect('app:detail', product_id=product_id)

# request.methodがGETのとき(画面にアクセスされたとき)は空のフォームを表示する
add_to_cart_form = AddToCartForm()
context = {
    'product': product,
    'add_to_cart_form': add_to_cart_form,
}
return render(request, 'app/detail.html', context)

```

「カートに追加する」ボタンを押すと、POSTメソッドでリクエストが送られる様に設定します。

セッションの中に「cart」というキーが存在するかを確認し、まだ存在しなければ、「cart」というキーを設定し、その値として「プロダクトIDをキー、個数を値として持つ辞書」を格納します。
もし、すでに「cart」というセッションのキーがあるのであれば、現在追加しようとしている商品のIDがあるかどうかを確認し、個数の値を追加するか新しく指定して格納します。
これによって、先ほど示した以下の様なsession構造が出来上がります。

```

session = {
    'cart': {
        '1': 5,
        '4': 3,
        '10': 6,
    }
}

```

HTMLには、「カートに追加する」ボタンを設定します。

(~/ecsite/app/templates/app/detail.html)

```

. . .

<p>{{ product.description }}</p>

<!-- 追加 -->
<div>
    {% if request.user.is_authenticated %}
        <form action="{% url 'app:detail' product.id %}" method="POST">

```

```
{% csrf_token %}
{{ add_to_cart_form.as_p }}
<button class="purchase-button" type="submit">
    カートに追加する
</button>
</form>
{% else %}
    <a href="{% url 'app:login' %}?next={{ request.path }}">
        <button class="purchase-button">ログインして購入する</button>
    </a>
{% endif %}
</div>

....
```

ボタンは、ログインユーザーにのみ表示されるようにして、ログアウト状態であればログインを促すボタンに変えています。

これで、カートへの追加機能が実装できました。実際に追加ボタンを押してみましょう。

追加ボタンを押したときに、本当にセッションに情報を追加できているか確認します。まずは、ブラウザからsessionidをコピーします。

使用中の Cookie

許可

ブロック中

このページを表示したときに以下の Cookie が設定されました

▼ 127.0.0.1

▼ Cookie

csrftoken

sessionid

名前 sessionid

コンテンツ 9papv1o5eijpmz3rzvlrrbnh3dshnyb1

ドメイン 127.0.0.1

パス /

↑
コピー

ブロック

削除

完了

続いてshellを起動し、セッションの中身を確認します。

(Django shell)

```
$ python manage.py shell
>>> from django.contrib.sessions.models import Session

# 対象のセッションオブジェクトを取得して変数sに代入する
# session_keyにはコピーしたsessionidの値を指定する
>>> s =
Session.objects.get(session_key="9papv1o5eijpmz3rzvlrrbnh3dshnyb1")
```

```
# セッションに'cart'キーが追加されている
>>> s.get_decoded()
{'_auth_user_id': '1', '_auth_user_backend':
'django.contrib.auth.backends.ModelBackend', '_auth_user_hash':
'70fd97ba9b89faf8171981e4294f6dff070e9fe7', 'cart': {'3': 2}}
```

上記の `'cart': {'3': 2}` のように、セッションにcart情報が追加されていることが確認できたでしょうか。上の例だと、ID3の商品が2つカートに入っていることを表しています。複数の商品を追加していたとしても、`{'3': 2}` のように1つの商品についてしか情報が記録されていないと思います。これはDjangoの仕様ですので今は気にしないでください。後ほど解説します。

5.7 カートページから決済する

セッションに商品の情報を一時的に保存できるようになりました。

あとは、この情報を使って決済できるように実装しましょう。決済する際には、送付先住所も入力させるようにします。

より便利にするために、郵便番号を入力すれば住所が自動入力される処理も実装してみます。この機能は、外部APIを使って実装します。

セッションから商品情報を取得する

まずは、「カートに追加した商品一覧」を画面に並べて表示してみましょう。カートに追加した商品情報は、セッションで管理しているので、セッションから情報を取得します。

(~/ecssite/app/urls.py)

```
urlpatterns = [
    ...
    path('cart/', views.cart, name='cart'), # 追加
]
```

(~/ecssite/app/views.py)

```
@login_required
def cart(request):
    # セッションから'cart'キーに対応する辞書を取得する。
    # セッションに'cart'キーが存在しない場合は{}(空の辞書)がcart変数に代入される。
    cart = request.session.get('cart', {})

    # cart_products → Productオブジェクトをキー、購入個数を値として持つ辞書
    # (初期値は空の辞書)
    cart_products = {}

    # total_price → カート内商品の合計金額を表す変数(初期値は0)
```

```

total_price = 0

# cart_productsとtotal_priceを更新する
for product_id, num in cart.items():
    product = Product.objects.filter(id=product_id).first()
    if product is None:
        # productがNoneのとき(対象商品がデータベースから削除されている場合等)は画面に表示しない
        continue
    cart_products[product] = num
    total_price += product.price * num

context = {
    'cart_products': cart_products,
    'total_price': total_price,
}
return render(request, 'app/cart.html', context)

```

(~/ecssite/app/templates/app/cart.html)

```

{% extends 'app/base.html' %}

{% load humanize %}

{% block content %}

<div class="cart">

    <h2>カート</h2>
    {% for product, num in cart_products.items %}
        <div class="order-product">
            <div class="order-product-image">
                <a href="{% url 'app:detail' product.id %}">
                    
                </a>
            </div>
            <div class="order-product-info">
                <h2>{{ product.name }}</h2>
                <div>
                    値格:
                    <span class="info-value">
                        {{ product.price | intcomma }}
                    </span>
                </div>
                <div>
                    個数:<span class="info-value">{{ num | intcomma }}</span>
                </div>
            </div>
        </div>
    {% endfor %}

</div>

```

```
{% endblock %}
```

(~/ecsite/app/templates/app/base.html)

```
<!-- カートのリンクを追加 -->
<span><a href="{% url 'app:fav_products' %}">お気に入り</a></span>
<span><a href="{% url 'app:cart' %}">カート</a></span>
<span><a href="">注文履歴</a></span>
```

ここまで実装で、カートに追加した商品がカートページに並ぶようになっているはずです。しかし、先ほど少し触れたように最初に追加した商品以外はセッションに反映されないことに気づいたでしょうか? Djangoのセッションは、デフォルトの状態だと、セッション情報を追加したり削除したりする場合にしかセッションが更新されないようになっていることが原因です。

今回の場合、セッションに `'cart'` というキーを新しく追加した際は、正しくセッションが更新されます。しかし、この `cart` の値に対して新しい情報を追加しようとしても、変更は反映されません。これは、Djangoのデフォルトの仕様がそのようになっているからです。

(Djangoのセッションの挙動イメージ)

```
# cartが空の時
session = {}

# 1つ目の商品を登録した時
# 新しく 'cart' というキーが追加される(セッション情報が正しく書き換わる)
session = {
    'cart': {'1': 5}
}

# 2つ目の商品を追加した時、以下のようにはならず、正しく書き換わらない
# session['cart'] = {'1': 5}のままになる
session = {
    'cart': {
        '1': 5,
        '2': 3
    }
}
```

このような処理でセッションの更新を反映させるためには、一連の処理の後に `request.session.modified = True` と書けばきちんと更新されます。ただ、毎回このように書くのは面倒なので、`settings.py` に以下のように書くことで、セッションに対する全ての操作で更新のリクエストを行うようになります。セッションを使う場合にはこちらを設定しておくと良いでしょう。

(~/ecsite/ecsite/settings.py)

```
# セッションを毎回更新する  
SESSION_SAVE_EVERY_REQUEST = True
```

これでセッションに複数の商品を追加できるようになりました。カートページは以下のようになっているはずです。

Djamazon

Point: 50,000 お気に入り カート 注文履歴 ログアウト

カート

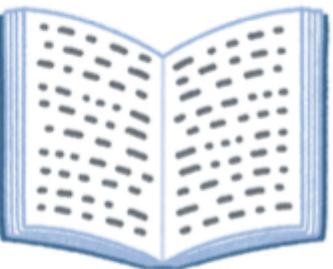
おせんべい

価格: 600
個数: 2



面白い本

価格: 1,200
個数: 1



自作のテンプレートタグで小計を表示しよう

カートの中身を表示するときには、各商品ごとに小計を出すとわかりやすくなります。

「View側で商品ごとの小計を計算してテンプレートに渡す」というのも1つのやり方ですが、**今回はテンプレートタグを自作して、テンプレート側で小計の計算処理を行うようにしてみましょう。**

テンプレートタグは `intcomma` や `truncatechars` などのような形ですでに利用することができます。これらはDjangoがデフォルトで用意しているテンプレートタグなのですが、今回のように価格と個数の掛け算をするタグはデフォルトでは用意されていませんので、このタグを自分で定義して使っていきます。

アプリ内に **templatetags** という名前のフォルダを作ります。このフォルダの中で定義されたものは、テンプレートタグとして使えるようになります。

appアプリケーション内に **templatetags** というフォルダを作ってください。また、Pythonのパッケージであることを示すために、その中に `_init_.py` という空のファイルも作りましょう。このフォルダの中に `filters.py` ファイルを作ります。フォルダの構成は以下のようになっています。

(~/ecsite/)

```
app/
    __init__.py
    admin.py
    apps.py
    forms.py
    migrations/
    models.py
    templates/
    templatetags/ # 追加
        __init__.py # 追加
        filters.py # 追加
    tests.py
    urls.py
    views.py
```

あとは、filters.pyにカスタムテンプレートタグの内容を定義していきます。

(~/ecssite/app/templatetags/filters.py)

```
from django import template

register = template.Library()

@register.simple_tag
def multiply(value1, value2):
    result = value1 * value2
    return f"{result:,}"
```

multiplyという名前の関数を作りました。これで、テンプレート側では `multiply` というタグが利用できることになります。

multiply関数は2つの引数を掛け算した結果を返します。`f"{{ 数値: , }}"` という書き方は、数値を3行ごとにカンマ区切りするPythonの書き方です。

`@register.simple_tag` というデコレータは、シンプルな構成のテンプレートタグを作成するときに利用します。テンプレート内でタグを利用するときのインプットの数だけ引数を取り、値を返します。

(~/ecssite/app/templates/app/cart.html)

```
{% extends 'app/base.html' %}
{% load humanize %}
{% load filters %} <!-- filters.pyファイルをロードする -->
. . .


## {{ product.name }}



價格:


```

```

    {{ product.price | intcomma }}  

    </span>  

</div>  

<div>  

  個数:<span class="info-value">{{ num | intcomma }}</span>  

</div>  

<!-- 小計を追加する -->  

<hr>  

<div>  

  小計:  

  <span class="info-value">  

    {% multiply product.price num %}  

  </span>  

</div>  

</div>  

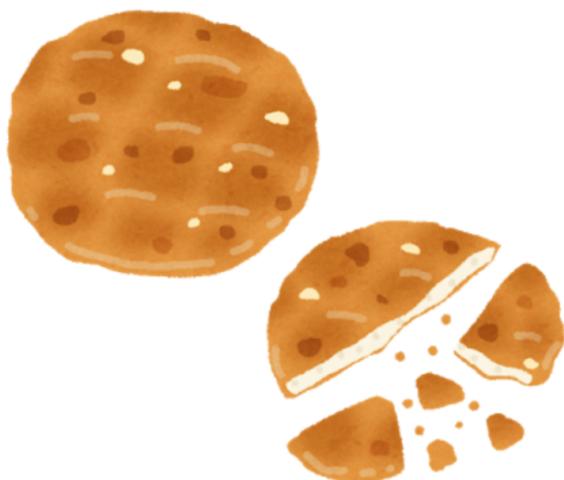
...

```

テンプレートタグmultiplyには2つの引数が必要なので、product.price(商品単価)とnum(購入個数)を指定します。つまり、multiply関数のvalue1引数にproduct.price、value2引数にnumが代入されます。multiply関数では、これらの値が掛け算されて値が返されます。その値がテンプレート上に表示されることになります。

再度サーバーを起動しなおして、画面を確認してみてください。以下のようになります。

カート



おせんべい

価格 :	200
個数 :	6
小計 :	1,200

カート内の商品を追加・削除できるようにしよう

続いて、カートに追加した商品の購入個数を変更できるようにしましょう。

cartセッション内に保存されている「商品の購入個数情報」を更新するためのView関数を作ることになります。

その前にまずは、HTMLを編集して「1つ減らすボタン」と「1つ増やすボタン」を用意します。

(~/ecsite/app/templates/app/cart.html)

```
<div>
    小計:
    <span class="info-value">
        {% multiply product.price num %}
    </span>
</div>

<!-- formを追加 -->
<div class="order-product-amount-form">
    <form action="{% url 'app:change_product_amount' %}" method="POST">
        {% csrf_token %}
        <input type="hidden" name="product_id" value="{{ product.id }}">
        <input type="submit" name="action_remove" value="1つ減らす">
        <input type="submit" name="action_add" value="1つ増やす">
    </form>
</div>
```

このformでは、`type="submit"` のボタンが2つあります。`type="submit"` のボタンが押された時にはフォームが送信されることになりますが、どちらのボタンが押された場合にも、フォームは送信されます。ただし、name属性の値はそれぞれのボタンによって異なる(action_remove、action_add)ので、サーバー側ではどちらのボタンが押されたのかを判別することができます。

また、`type="hidden"` なので画面上には表示されませんが、`name="product_id"` のフィールドもあります。このフィールドの値はvalueに指定している`{{ product.id }}`です。「1つ減らす」または「1つ増やす」ボタンを押したときは、この情報も送信されることになるので、サーバー側ではどの商品の個数を増減させるのかを判別することができます。

続いて、フォームの送信先となるView関数を作ります。

(~/ecsite/app/views.py)

```
@login_required
@require_POST
def change_product_amount(request):

    # name="product_id"のフィールドの値を取得(どの商品を増減させるか)
    product_id = request.POST["product_id"]
    # セッションから"cart"情報を取得
    cart_session = request.session['cart']

    # セッションの更新
    if product_id in cart_session:
        # 1つ減らすボタンが押された時
        if "action_remove" in request.POST:
            cart_session[product_id] -= 1
        # 1つ増やすボタンが押された時
        if "action_add" in request.POST:
```

```

    cart_session[product_id] += 1
    # 商品個数が0以下になった場合は、カートから対象商品を削除
    if cart_session[product_id] <= 0:
        del cart_session[product_id]
    return redirect('app:cart')

```

フォームがPOSTされた時、各inputタグの情報は `request.POST` の中に格納されています。そのため、`if "action_remove" in request.POST` のようにすることで `name="action_remove"` のボタンが押されたかどうかを判定することができます。

押されたボタンに応じて、購入個数を加算・減算してcartセッションを更新します。購入個数がゼロになればセッション上からキー自体も削除して、カートからその商品を削除します。

最後にURLを紐付けましょう。

(~/ecsite/app/urls.py)

```

urlpatterns = [
    ...
    # 追加
    path(
        'change_product_amount/',
        views.change_product_amount,
        name='change_product_amount'
    ),
]

```

これで完了です。カート画面で商品の個数を増減できるか確認してみてください。

WebAPIを活用して住所を取得しよう

商品の情報を表示・修正できるようになったので、決済のフローに進みましょう。

今回のチュートリアルでは、決済の時に送付先の住所を入力して、決済のボタンを押すことができるようになります。

住所の入力をすべてユーザーに任せることもできますが、多くのWebサービスで実装されているように郵便番号を入力すれば途中まで住所を自動で取得してくれるような設計にしましょう。ここで利用するのが、**WebAPI**です。

WebAPIとは、インターネットでHTTPを通して外部のシステムを利用する仕組みのことです。この仕組みによって、第三者から提供されている様々な便利な機能を簡単に自分のサービスに組み込むことができるようになります。

今回の場合だと、「郵便番号を入力すると、その番号に対応する住所を取得する機能」を自分で実装するのではなく、外部すでに提供されているサービスを利用することで簡単に実装することができます。

それではここからは実際にAPIを使っていきます。

このチュートリアルでは、無料で利用できるこちらの郵便番号検索APIサービスを使うこととします。

- <http://zipcloud.ibsnet.co.jp/doc/api>

APIにはそれぞれ仕様があるので、その仕様に則った使い方をする必要があります。大抵の場合、APIの使い方をまとめたページを提供してくれているので、まずはそのページの説明を読んで使い方の概要を理解するのが良いでしょう。

今回のAPIの場合、トップページ(<http://zipcloud.ibsnet.co.jp/doc/api>)に使い方の概要が書いてあります。

このAPIでは、以下のようなリクエストURLに対して、郵便番号をパラメータとして付与することで、その郵便番号に紐づいたレスポンスを返してくれます。実装するときにはこのURLをベースにして、ユーザーが指定した郵便番号に対してリクエストを送るようにします。

リクエストURL: <http://zipcloud.ibsnet.co.jp/api/search>

たとえば、郵便番号「100-0001」の情報は以下のURLで取得することができます。リクエストURLの末尾に `?zipcode=1000001` をつけています。

- <http://zipcloud.ibsnet.co.jp/api/search?zipcode=1000001>

実際にブラウザ上でアクセスしてみると、以下のような内容のレスポンスを取得できることがわかります。

```
{  
    "message": null,  
    "results": [  
        {  
            "address1": "東京都",  
            "address2": "千代田区",  
            "address3": "千代田",  
            "kana1": "トキヨウ",  
            "kana2": "チヨダ",  
            "kana3": "チヨダ",  
            "prefcode": "13",  
            "zipcode": "1000001"  
        }  
    ],  
    "status": 200  
}
```

WebAPIのレスポンスにはいくつかタイプがありますが、今回のようなJSON形式のものが多いです。これはPythonの辞書型と同様に扱うことができるため、直感的にデータを扱うことができます。

message、results、statusなどがそれぞれ何を表しているのかは、[トップページ](#)に記載されている仕様を確認するとわかりますので、「レスポンスフィールド」の所を参照してください。

また、不正な値を入力した場合のレスポンスも確認してみましょう。以下のURLにアクセスしてみてください。

<http://zipcloud.ibsnet.co.jp/api/search?zipcode=5555555>

「555-5555」という郵便番号には住所が割り振られていないため、以下のようなレスポンスが返ってきます。

```
{  
    "message": null,  
    "results": null,
```

```
        "status": 200
    }
```

resultsがnullになっています。このように、WebAPIを利用する場合には適切な答えが返ってこないことも想定して実装する必要があるので気をつけましょう。

それでは、Pythonコードを使ってWebAPIから住所情報を取得してみましょう。

Pythonでは、HTTP関連の機能を提供する**requests**というライブラリがあるのでこれを利用します。このライブラリの使い方の詳細はドキュメント(<https://requests-docs-ja.readthedocs.io/en/latest/>)に記載されています。

このライブラリを利用して、郵便番号検索をする関数を作りましょう。pipコマンドでライブラリをインストールしてください。

(コンソール)

```
$ pip install requests
```

まずは、コンソール上でこのライブラリを使ってみましょう。

(コンソール)

```
$ python3

# requests、jsonモジュールをインポート
>>> import requests, json

# リクエストURLを変数に代入
>>> REQUEST_URL = 'http://zipcloud.ibsnet.co.jp/api/search?
zipcode=1000001'

# requestsモジュールの利用: REQUEST_URLにGETメソッドでリクエストを送る。結果を
response変数に代入する
>>> response = requests.get(REQUEST_URL)

# レスポンス内容はtext属性で取得可能
>>> response.text
'{
  "message": null,
  "results": [
    {
      "address1": "東京都",
      "address2": "千代田区",
      "address3": "千代田",
      "kana1": "トカヨウ",
      "kana2": "チヨダ",
      "kana3": "チヨダ",
      "prefcode": "13",
      "zipcode": "1000001"
    }
  ],
  "status": 200
}

# text属性は文字列型
>>> type(response.text)
<class 'str'>
```

`requests.get(REQUEST_URL)`のようにすることで、指定のURLにGETメソッドでHTTPリクエストを送ることができます。レスポンスが返ってくるのでその値は変数に代入することができます。今回の場合 `response`

という変数に代入しました。

また、レスポンスの内容は `.text` で取得可能ですが、上に示したようにこれは文字列型なので少々扱いづらいです。以下のように `json.loads()` を使うことで辞書型に変換すると良いでしょう。

```
# response.text(文字列型)を解析して辞書型として扱えるようにする
>>> dict_response = json.loads(response.text)

>>> dict_response
{'message': None, 'results': [{['address1': '東京都', 'address2': '千代田区',
'address3': '千代田', 'kana1': 'トウヨウト', 'kana2': 'チヨダク', 'kana3': 'チヨダ',
'prefcode': '13', 'zipcode': '1000001'}], 'status': 200}

# dict_responseは辞書型
>>> type(dict_response)
<class 'dict'>

# 辞書型なのでキー名で各値を取得できる
>>> dict_response['status']
200

>>> dict_response['results'][0]
{'address1': '東京都', 'address2': '千代田区', 'address3': '千代田', 'kana1':
'トウヨウト', 'kana2': 'チヨダク', 'kana3': 'チヨダ', 'prefcode': '13', 'zipcode':
'1000001'}

>>> dict_response['results'][0]['address1']
'東京都'

>>> dict_response['results'][0]['address2']
'千代田区'

# 不正な郵便番号の場合
>>> REQUEST_URL = 'http://zipcloud.ibsnet.co.jp/api/search?
 zipcode=5555555'
>>> response = requests.get(REQUEST_URL)
>>> response.text
'{\n\t"message": null,\n\t"results": null,\n\t"status": 200\n}'
>>> dict_response = json.loads(response.text)
# 'message'と'results'の値(null)はNoneに変換される
>>> dict_response
{'message': None, 'results': None, 'status': 200}
```

ここまでで、`requests`モジュールの基礎的な使い方の説明はしましたので、これをView関数に組み込んでみましょう。

関数は以下のように実装します。

(~/ecssite/app/views.py)

```

import json # 追加
import requests # 追加

def fetch_address(zip_code):
    """
    郵便番号検索APIを利用する関数
    引数に指定された郵便番号に対応する住所を返す
    住所取得に失敗した場合は空文字を返す
    """

    REQUEST_URL = f'http://zipcloud.ibsnet.co.jp/api/search?zipcode={zip_code}'
    response = requests.get(REQUEST_URL)
    response = json.loads(response.text)
    results, api_status = response['results'], response['status']

    address = ''
    # レスポンスステータスが200かつ'results'が存在する場合
    # → address変数に取得した住所を代入する
    if api_status == 200 and results is not None:
        result = results[0]
        address = result['address1'] + result['address2'] + result['address3']
    return address

```

fetch_address関数では、zip_codeという引数名で郵便番号を受け取ります。zip_codeをREQUEST_URLに埋め込んでリクエストURLを生成し、GETメソッドでリクエストを送ります。もし正常にレスポンスが返ってきた(レスポンスステータスが200)場合、address変数に取得した住所を代入しています。住所を取得できなかった場合は、if文の条件式はFalseになるのでaddress変数の値は空文字のままでです。

試しに、トップページのView関数(index関数)内に `print(fetch_address('1000001'))` と書いた上で、トップページ(`http://127.0.0.1:8000/`)にアクセスしてみてください。

(~/ecsite/app/views.py)

```

def index(request):
    print(fetch_address('1000001')) # 追加
    products = Product.objects.all().order_by('-id')
    return render(request, 'app/index.html', {'products': products})

```

fetch_address関数が正常に実行されてAPIから住所を取得できていれば、コンソール(`python manage.py runserver`を実行したタブ)に住所が出力されるはずです。

(コンソール)

```

...
Django version 2.2.5, using settings 'ecsite.settings'
Starting development server at http://127.0.0.1:8000/

```

```
Quit the server with CONTROL-C.  
東京都千代田区千代田 ← 住所が出力される  
[27/Dec/2020 16:21:05] "GET / HTTP/1.1" 200 2488
```

これで、郵便番号を入力するだけで住所を取得する機能を追加することができました。この関数を、決済時に利用できるようにしていきます。

今回の実装では、PurchaseFormというフォームを作り、郵便番号と住所を入力できるようにします。そして、その入力内容に問題がなければ、実際に決済の処理を行い、Saleモデルに決済情報を追加するという形にします。

まずはPurchaseFormを作りますが、これはモデルに紐づいたフォームではないため、ModelFormではなく普通のFormを継承して作成します。

(~/ecsite/app/forms.py)

```
# 追加  
class PurchaseForm(forms.Form):  
    zip_code = forms.CharField(  
        label='郵便番号',  
        max_length=7,  
        required=False,  
        widget=forms.TextInput(  
            attrs={'placeholder': '数字7桁(ハイフンなし')})  
    )  
  
    address = forms.CharField(  
        label='住所', max_length=100, required=False
```

zip_codeとaddressという2つのフィールドを定義しました。どちらも CharField を使っているので、type="text"のinputタグが生成されることになります。(つまり、文字列を入力できるフィールドが2つ画面に表示されることになります。)

続いて、Viewのcart関数を修正してPurchaseFormを画面に表示しましょう。

PurchaseFormを表示する際は、search_address(住所自動取得ボタン)とbuy_product(購入処理ボタン)という名前の2つのボタンを用意します。

前者のボタンが押されたときは住所取得処理、後者のボタンが押されたときは購入処理をするように実装します。また、住所の入力は必須にするべきなので、その部分のバリデーションも実装します。

(~/ecsite/app/views.py)

```
from .models import Sale # 追加  
from .forms import PurchaseForm # 追加  
  
@login_required  
def cart(request):  
    user = request.user
```

```

cart = request.session.get('cart', {})
cart_products = {}
total_price = 0

# 合計金額の計算
for product_id, num in cart.items():
    product = Product.objects.filter(id=product_id).first()
    if product is None:
        # productがNoneのとき(対象商品がデータベースから削除されている場合等)は画面に表示しない
        continue
    cart_products[product] = num
    total_price += product.price * num

if request.method == 'POST':
    purchase_form = PurchaseForm(request.POST)
    if purchase_form.is_valid():
        # 住所検索ボタンが押された場合
        if 'search_address' in request.POST:
            zip_code = request.POST['zip_code']
            address = fetch_address(zip_code)
            # 住所が取得できなかった場合はメッセージを出してリダイレクト
            if not address:
                messages.warning(request, "住所を取得できませんでした。")
                return redirect('app:cart')
            # 住所が取得できたらフォームの値として入力する
            purchase_form = PurchaseForm(
                initial={'zip_code': zip_code, 'address': address}
            )

# 購入処理ボタンが押された場合
if 'buy_product' in request.POST:
    # 住所が入力済みかを確認する。未入力の場合はリダイレクトする。
    if not purchase_form.cleaned_data['address']:
        messages.warning(request, "住所の入力は必須です。")
        return redirect('app:cart')
    # カートが空じゃないかを確認する。空の場合はリダイレクトする。
    if not cart:
        messages.warning(request, "カートは空です。")
        return redirect('app:cart')
    # 所持ポイントが十分にあるかを確認する。不足して場合はリダイレクトする。
    if total_price > user.point:
        messages.warning(request, "所持ポイントが足りません。")
        return redirect('app:cart')

# 各プロダクトのSale情報を保存(売上記録の登録)
for product, num in cart_products.items():
    sale = Sale(
        product=product,
        user=request.user,
        amount=num,
        price=product.price,
        total_price=num * product.price,
    )

```

```

    sale.save()

    # 購入した分だけユーザーの保有ポイントを減らす。
    user.point -= total_price
    user.save()
    # セッションから'cart'を削除してカートを空にする。
    del request.session['cart']
    messages.success(request, "商品の購入が完了しました！")
    return redirect('app:cart')

else:
    purchase_form = PurchaseForm()
    context = {
        'purchase_form': purchase_form,
        'cart_products': cart_products,
        'total_price': total_price,
    }
    return render(request, 'app/cart.html', context)

```

これで住所検索と購入処理をする関数は完成です。if文が多くネストが深くなってしまっているので、本当は部分的に別関数に切り出したりした方が綺麗ですが、今回はあくまで練習なので全体像がわかりやすいように1つのView関数に全ての処理をまとめています。

cart.htmlには、フォームを追加します。住所検索のボタンと購入ボタンは分けて表示します。

(~/ecsite/app/templates/app/cart.html)

```

. . .

<h2>カート</h2>

<!-- 追加 -->
<div class="purchase-form">
    <form action="{% url 'app:cart' %}" method="POST">
        {% csrf_token %}
        <div class="purchase-form-container">
            <div class="purchase-form-address">
                <div>
                    {{ purchase_form.zip_code.label_tag }}
                    {{ purchase_form.zip_code }}
                    <input type="submit" name="search_address" value="住所を検索">
                </div>
                <div>
                    {{ purchase_form.address.label_tag }}
                    {{ purchase_form.address }}
                </div>
            </div>
        </div>
        <div class="purchase-form-pay">
            請求額: {{ total_price | intcomma }}
            <input type="submit" name="buy_product" class="purchase-
button" value="購入する">
        </div>
    </form>
</div>

```

```

        </div>
    </form>
</div>

{%
    for product, num in cart_products.items %}
    ...

```

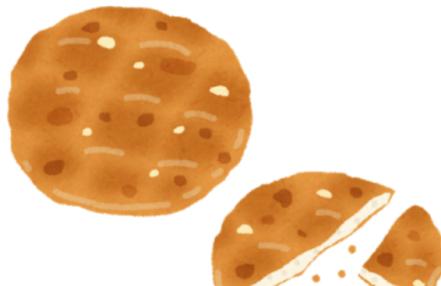
Djamazon

Point: 50,000 お気に入り カート 注文履歴 ログアウト

カート

郵便番号:
 住所:

請求額: 9,000



おせんべい

価格 :	600
個数 :	4
<hr/>	
小計 :	2,400

実装できたら、色々と入力を試してみてエラーハンドリングがうまくいっているか確認してみてください。

購入が完了すると、以下のように所持ポイントが減っています。また、Admin画面ではSaleのデータが追加されていることが確認できるはずです。

Djamazon

Point: 41,000 お気に入り カート 注文履歴 ログアウト

商品の購入が完了しました！

カート

郵便番号:
 住所:

請求額: 0

5.8 注文履歴ページを作ろう

最後に、注文履歴ページを作りましょう。特定のユーザーの決済データを集めて表示します。まずはURLを追加しましょう。

(~/ecsite/app/urls.py)

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

Viewとテンプレートは以下のようにします。

(~/ecsite/app/views.py)

```
# 追加
@login_required
def order_history(request):
    user = request.user
    sales = Sale.objects.filter(user=user).order_by('-created_at')
    return render(request, 'app/order_history.html', {'sales': sales})
```

(~/ecsite/app/templates/app/order_history.html)

```
{% extends 'app/base.html' %}

{% load humanize%}

{% block content%}

<div class="order-history">
<h2>注文履歴</h2>

{% for sale in sales %}
<div class="order-product">
    <div class="order-product-image">
        <a href="{% url 'app:detail' sale.product.id %}">
            
        </a>
    </div>
    <div class="order-product-info">
        <h2>{{ sale.product.name }}</h2>
        <div>
            價格：
            <span class="info-value">{{ sale.price | intcomma }}</span>
        </div>
        <div>
            個数：
            <span class="info-value">{{ sale.amount | intcomma }}</span>
        </div>
        <hr>
        <div>
```

```
    小計：  
    <span class="info-value">{{ sale.total_price | intcomma }}  
</span>  
        </div>  
        <span class="info-timestamp">{{ sale.created_at | date:"Y/m/d" }}  
</span>  
            </div>  
        </div>  
    {%- endfor %}  
  
</div>  
  
{% endblock %}
```

ヘッダーのリンクも修正しておきます。

(~/ecsite/app/templates/app/base.html)

```
    . . .  
  
<!-- 注文履歴のリンクを追加 -->  
<span><a href="{% url 'app:cart' %}">カート</a></span>  
<span><a href="{% url 'app:order_history' %}">注文履歴</a></span>  
  
    . . .
```

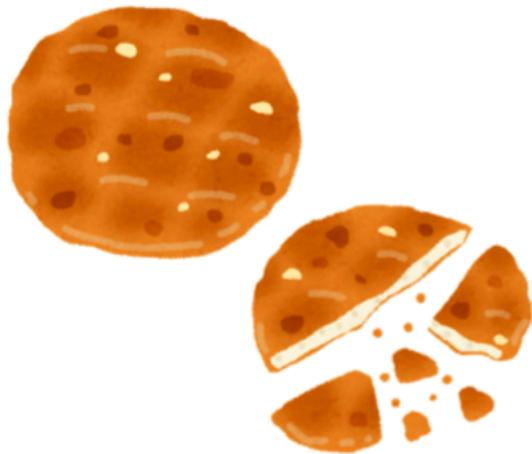
注文履歴



ドリンクセット

価格：	2,200
個数：	3
小計：	6,600

2019/11/02



おせんべい

価格：	600
個数：	4
小計：	2,400

2019/11/02

過去の注文履歴を表示することができるようになります。

これでチュートリアルは全て終了です。お疲れ様でした！

最後に、アンケートにご協力いただける方は以下リンクからご回答をお願いいたします。今後のサービス開発に活かしてまいります。

<https://form.run/@djangobrothers--1609573904>

2019年11月3日 初版発行

2020年11月9日 第2版発行

2021年1月1日 第3版発行

(C)2019 DjangoBrothers