

Django チュートリアルで学ぶ Web 開発入門

by DjangoBrothers

2019-11-03 版

はじめに

Web 開発を学ぶメリット

本書は、Python 製の Web フレームワークである Django を用いて、Web サービス開発について解説する本です。私たちは普段の生活で、便利な Web サービスを活用しています。こういった Web サービスを自分で作ったり、既存の Web サービスの機能を作り変えたりできると楽しいと思わないでしょうか？この書籍では、このような Web サービスを自分で作るためのスキルについて解説しています。

Web サービスの開発はそれ自体が楽しいという面もありますが、Web エンジニアという職業を目指す魅力も見逃せません。すでに多くの Web サービスが世の中に生まれていますが、今後も新しいサービスの開発や、既存のシステムのアップデートなどはどんどん増えていき、Web 開発のスキルを持ったエンジニアの需要は高まり続けていきます。企業からの Web エンジニアの求人はまだまだ増えていくと予想されています。需要に対して供給が追いついていない状態なので、今後もエンジニアの給与水準は高くなっていくと想定されます。

また、Web エンジニアは、その自由な働き方も魅力の一つです。IT 系の企業であれば非常にリラックスできる環境が与えられ、福利厚生なども充実していることが多いです。さらに、フリーランスになれば、時間や場所に制約を設げずに働くことさえ可能になります。このように、Web 開発のスキルを身につけるメリットは非常に大きいと言えます。

本書の特徴

本書の特徴は大きく二つあります。

- 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取ります。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、背景にはどのような技術が使われてい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があまり多くは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の書籍は少ないため、皆さんのお学習の一助になれば幸いです。

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

そのため、書籍や Web サービスなどでプログラミングの基礎を学んだ方を対象に、Web サービスの開発と Web フレームワークとしての Django の解説をじっくり進めていきます。

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。

Python 3.6 / Django 2.2.5

実際にチュートリアルを進めていく際にはこちらの環境に合わせるのが良いかと思います。

目次

はじめに	2
Web 開発を学ぶメリット	2
本書の特徴	2
対象読者	3
動作環境	3
第 1 章 Django の概要	7
1.1 Django とは	7
1.2 Web フレームワークの役割	8
第 2 章 Web 開発の前提知識とプロジェクトの初期設定	10
2.1 Web 開発の前提知識	10
2.2 開発環境の設定	15
2.3 Django プロジェクトの初期設定	20
第 3 章 チュートリアル①ブログ	28

3.1 アプリケーションを作ろう	28
3.2 トップページを作ろう	29
3.3 View と URL を紐づける	31
3.4 ブログモデルの作成	36
3.5 Admin ページを作ろう	40
3.6 トップページに記事を表示しよう	44
3.7 記事詳細ページを作ろう	51
3.8 CRUD を理解しよう	56
3.9 記事の削除と更新を実装しよう	65
第4章 チュートリアル②写真投稿サイト	73
4.1 開発準備をしよう	73
4.2 Template 拡張を利用する	75
4.3 Django 標準の User モデルを使う	81
4.4 ImageField で画像をアップロードしよう	86
4.5 ForeignKey でモデル同士を紐づける	95
4.6 ユーザー認証機能を作ろう	104

4.7 ユーザー登録機能を作ろう	110
4.8 写真投稿機能と削除機能の実装	115
4.9 <i>include</i> テレンプレートタグでリファクタリング	123
4.10 カテゴリーで絞り込む	129
第5章 チュートリアル③ECサイト	133
5.1 カスタムユーザー モデルを作ろう	133
5.2 ログイン/サインアップ処理を作ろう	141
5.3 必要なモデルを作ろう	148
5.4 商品をページに表示しよう	154
5.5 セッションを使ってカートに商品を追加しよう	169
5.6 カートページから決済できるようにしよう	175
5.7 注文履歴ページを作ろう	190

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 Web 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「ユーザーの管理機能」や「管理ユーザー用の画面」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なりますが）

さらに、Python ベースの Web フレームワークの中でも最も歴史が長く、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」という点で非常におすすめできる 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で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支持されている 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、各ページで重複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、一つのファイルを修正するだけでよくなるので、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楽にできる

通常、データベースからのデータの取得は「SQL」のようなデータベース側を操作する言語を利用して行い、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、それぞれ操作の方法や実現できることが異なっていました。

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を、実際に手を動かしながら解説し、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は、各チュートリアルの章では割愛することになるので、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、ウェブ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ウェブサイトは基本的に全てHTMLをベースに文字や画像が表示されています。

この本で作るWebサービスは非常に簡単なものですので<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、ある程度の知識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なないものです。

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下さい。

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 で書かれていたりします。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。特別な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、DjangoBrothers ブログの記事(【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)でもまとめているので、事前に復習しておくと良いと思います。

https://djangobrothers.com/blogs/basic_knowledge_of_python/

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。

皆さんの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、マウスなど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。

たとえば、同じ操作を何度も行う場合に、特定のコマンドをもう一度実行するだけで良くなるのです。これはまさにプログラミングでやっていることと同じで、「特定の操作の手順を一度記述すれば、それを繰り返し行うことができる」ようになるのです。

コマンドラインに関しても、全て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ら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 Mac OS 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。

コマンドラインの基礎

Mac OS 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、このマークの後ろに文字列（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！

\$の後ろに **pwd** と入力し、Enter を押してください。

(~/)

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

こ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          Pictures
Documents        Projects
Downloads        Public
Library
Movies
```

このようにたくさんのディレクトリのリストが表示されます。これは個人の環境によって出てくる内容が違うので、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。

```
(~/)
$ 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 ..` というコマンドで一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、一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つてみます。`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 ファイルを作つて見ましょう。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 ファイルを作るときも `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おさらいしました。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、Python3.7.- 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 Python 3.7.-」というボタンから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 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エラーが出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なので、先輩に頼るのが習得への一番の近道です。

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

仮想環境とは、特定のライブラリなどをパッケージ化した開発環境のことです。通常であればパソコンなどの環境ごとに Python の開発環境は一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、そのライブラリが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つ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ことができます。

このコマンドは、仮想環境 (djangobros_venv) がある、一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システムのことです。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。

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

`python` コマンドで Python のコマンドプロンプトを起動し、Django がちゃんとインストールされているか確認してみましょう。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。

```
(djangobos_venv) $ deactivate
$
```

`deactivate` コマンドにより、`(djangobos_venv)` 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。

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、基本的な構成を解説しながら初期設定を行います。なお、ここでの初期設定はこの後の各チュートリアル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。

その前に、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」と呼ばれる二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、「ブログをウェブ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」を一つにまとめたものです。基本的に、Django で Web サービスを一つ作る場合には、全体をまとめる「プロジェクト」が一つと、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。

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ってみましょう。

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を各チュートリアルで指定されたものに変更して利用してください。

```
(~/DjangoBros/)
```

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

これで「django_blog」という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ください。

```
(~/DjangoBros/)
```

```
$ cd django_blog  
$ ls
```

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生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 を設定するファイルです。
- `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（自分のパソコンなど）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て、`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/)
```

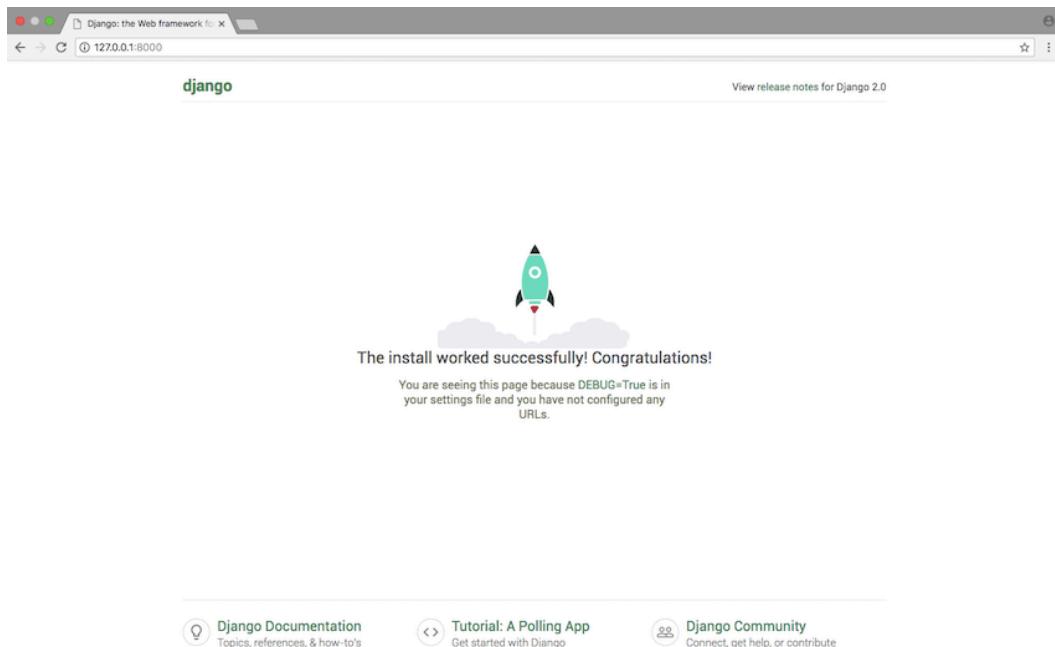
```
$ ls  
django_blog manage.py  
$ python manage.py runserver
```

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 ファイルの一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「マイグレイトしてください」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、いったん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 Welcome 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

初めての Django 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。設定ファイルから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表示する便利な機能があるので、現在の時刻といっても、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、国内向けのサービスであれば、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 ファイルの中の以下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、デフォルトで使うデータベースの指定がされているだけですので、`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。

setting.pyのファイルで import osをしないと無理やった

```
(~/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されますが、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

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いくつかの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

一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单一で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一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、すでにたくさんのファイルがありますが、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` アプリケーションを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` に追加してあげましょう。

```
(~/DjangoBros/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。最終的にこちらのページには、ブログ記事の一覧や新規記事作成ページへのリンクが表示されます。

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
- その「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す。文字が書けたら忘れずに保存してください。

```
(~/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 でアクセスされたときにこのファイルを表示していいのかを設定できません。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いる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」というシンプルな 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。これによって、ユーザーがその URL にアクセスした時にブラウザ上で 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` ファイルを用意し、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をしやすくすることが多いです。例えば、`blogs` アプリケーションに関連する URL は `http://127.0.0.1:8000/blogs/<...>/` のように `blogs` に紐づく部分だけの URL を管理できるようになります。

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。ここでは `blogs` アプリケーションをプロジェクト全体の URL 管理ファイルに紐付けしています。

```
(~/DjangoBros/django_blog/urls.py)
from django.contrib import admin
from django.urls import path, include

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

`path('', include('blogs.urls'))` では何をしているかというと、
`http://127.0.0.1:8000/` (Root URL) にアクセスされた時は、`blogs.urls` ファイルを参照することを意味しています。`blogs.urls` ファイルとは、この後自分で作成する `blogs/urls.py` ファイルのことです。

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ようになります。

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'),  
]
```

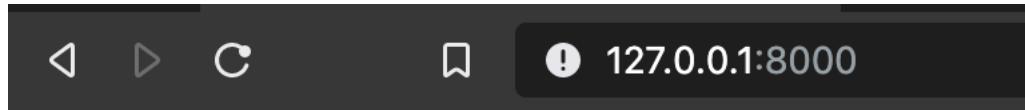
2行目の `from . import views` という部分では、同じ階層（ここでは blogs/ ディレクトリ）にある views.py ファイルをインポートしています。ドットは、同じ階層という意味です。

path 関数では、第一引数で空の文字列を指定し、第二引数で、`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、第一引数では 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。

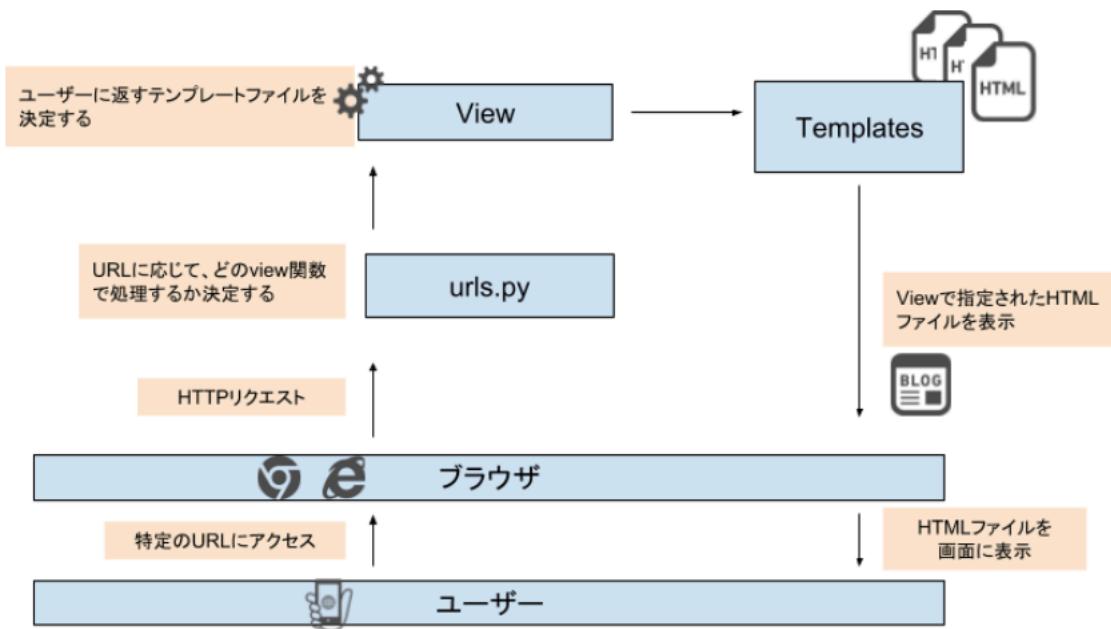
これでページを表示する準備が整いました。ローカルサーバーを立ち上げて、
`http://127.0.0.1:8000/` にアクセスすると、トップページが表示されるはずです。



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で
すので、この流れをしっかりと抑えておきましょう。

=====

Column：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か？

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、`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 のコード内で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。

例えばチュートリアル①では、`views.py` ファイルの `render` 関数の第 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。このとき、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 引数で `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。

チュートリアル①では `blogs` という単一のアプリケーションしか作成しませんでしたが、大きなプロジェクトでは複数のアプリケーションを作ることもあります。

この場合、複数のアプリケーションの間で、同じ名前のテンプレートファイル (`index.html` など) を設定してしまうと、`app1` の `index.html` なのか、`app2` の `index.html` なのかの区別がつかなくなってしまいます。そのため、それぞれのアプリケーション内の `templates` ディレクトリの中に、そのアプリケーションと同名のディレクトリを作成することによって、`app1/index.html` と `app2/index.html` のように区別がつくようにしているのです。

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にということです。

=====

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、データベースについて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、新しくブログが投稿されたときのためにブログ情報を保管しておくデータベースも必要です。

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)				
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（ブログの内容）、作成日時、更新日時の4つの情報を持たせます。

ブログサイトでは、様々なブログ記事が作成されることになりますが、ブログ記事に必要な情報（プロパティ）をブログクラスによって定義してあげます。

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、一つのモデルに対して一つのデータベーステーブルが割り当てられ、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。これは 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 モデルクラスを作成していきます。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。

一つ注意点ですが、モデルを作るときは、「Blogs」の様に複数形で書かずに「Blog」と单数形で書きます。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作成しています。この記述により、以下のようなテーブルを作ることができます。

Blogs テーブル

(例)	<code>id</code>	<code>title</code>	<code>text</code>	<code>created_datetime</code>	<code>updated_datetime</code>
	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。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。（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になることを防ぐことができます。

`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。モデルフィールドにおける `blank` と `null` の使い分けについては DjangoBrothers のブログでも解説しています。

https://djangobrothers.com/blogs/django_null_blank/

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ができますが、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（一つ一つのブログ記事）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。

モデルで使用される型は、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。この設計図を元に実際にデータベースを作成するには、「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。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。

`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、データベースを作るときは、「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 ページを利用するための設定をしましょう。Admin ページとは管理者のみが使用できるページのことと、通常は自分で作成しなければならないものですが、なんと 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 で指定したモデルクラスから自動的に管理ページのインターフェースを提供してくれます。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、最初になん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 で確認できるようにしてみます。また、実際にブログ記事を管理ページ上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。ログインするとこのような画面が表示されます。



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していきましょう。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

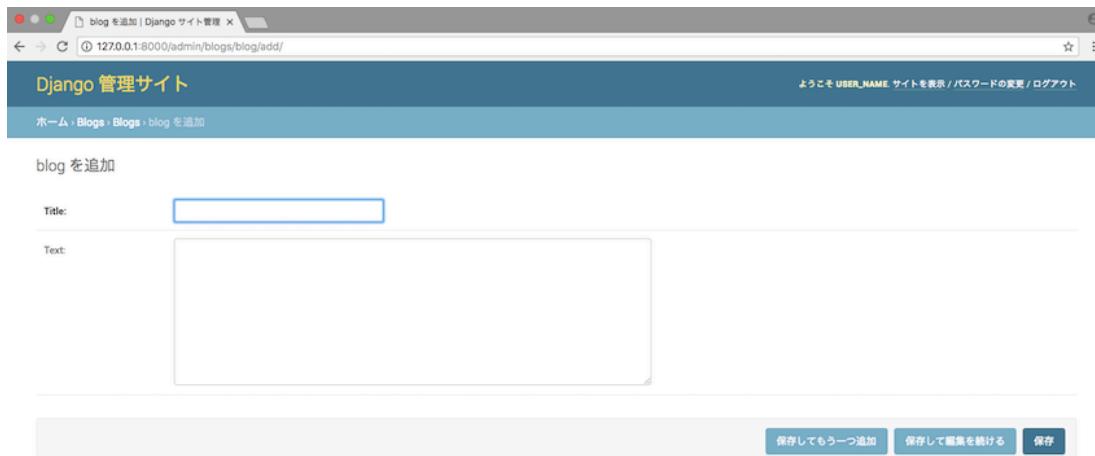
admin.site.register(Blog)
```

`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。



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。



「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、実際に一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。

これで 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。

テスト用にもう一つか二つ程度、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 `def __str__(self)` で、そのオブジェクトを指示する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。

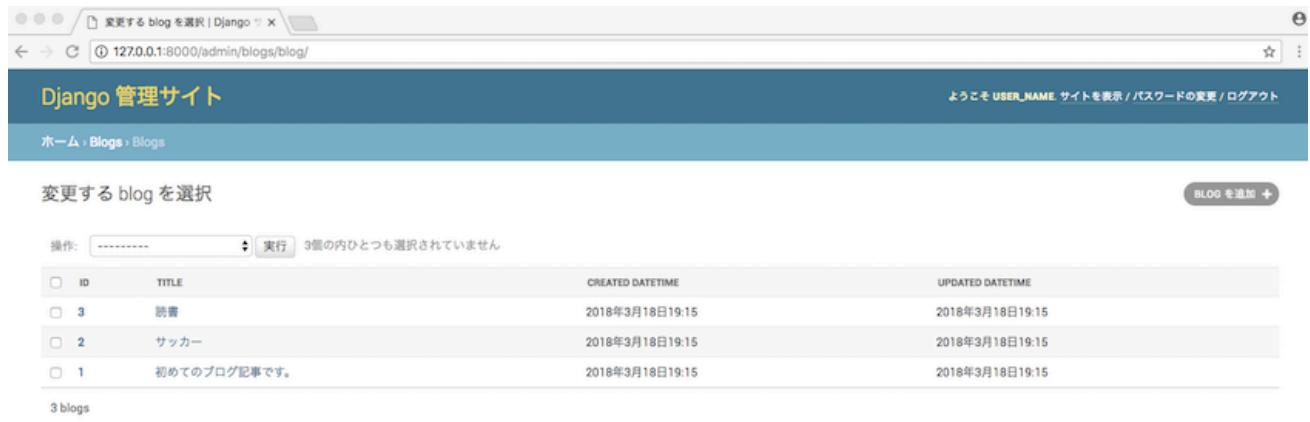
```
(~/DjangoBros/django_blog/blogs/admin.py)
```

```
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存在していませんでしたが、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です。1つのインスタンスに対して1つの番号が割り振られます。



The screenshot shows the Django Admin interface for a 'Blogs' model. The title bar says '変更する blog を選択 | Django'. The URL is '127.0.0.1:8000/admin/blogs/blog/'. The main area is titled 'Django 管理サイト' with a subtitle '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'. Below that is a breadcrumb trail 'ホーム > Blogs > Blogs'. A search bar says '変更する blog を選択'. A toolbar has a dropdown '操作:' with '-----' and a '実行' button, followed by a note '3個の中ひとつも選択されていません'. A table lists three blog entries:

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

At the bottom left is a link '3 blogs' and at the bottom right is a 'BLOG を追加' button.

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 ページから登録した記事を、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、先ほど Admin から作成したブログ記事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、データベースに簡単にアクセス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て、取得するインスタンスを制限することができます。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。`python` 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。

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。以下のコマンドを入力してエンターを押します。

(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)
```

(Django shell)

```
# 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")
```

(Django shell)

```
# 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)
# id の順に取得
# 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を取得する方法はたくさんあり、あらゆる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、少しづ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す。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、プロパティを取得することができるので、こちらの情報を使って様々なデータを Web ページ上に表示することができます。

```
(Django shell)
>>> blog = Blog.objects.get(id=1)
# id=1 の Blog インスタンスのタイトルを出力
>>> blog.title

# 複数の Blog インスタンスを取得
>>> blogs = Blog.objects.all()
# blogs リストの 1 番目のインスタンスの本文を表示
>>> blogs[0].text
```

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、そのクエリセットを 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

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

これまで書いてきたコードと比べて、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 `'blogs'` で、値は変数 `blogs` (つまり、1つ上の行で指定したクエリセット) となります。この記述により、`blogs/index.html` では、`blogs` と書くだけでクエリセットを用いたデータの表示ができるようになります。

今回は、キーと値の名前を同じにしましたが、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なので変数名をそのまま書いているところです。

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 インスタンスのリストを表示させているだけです。リストの中の 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 リスト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、こちらのタグでは、データではなく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。

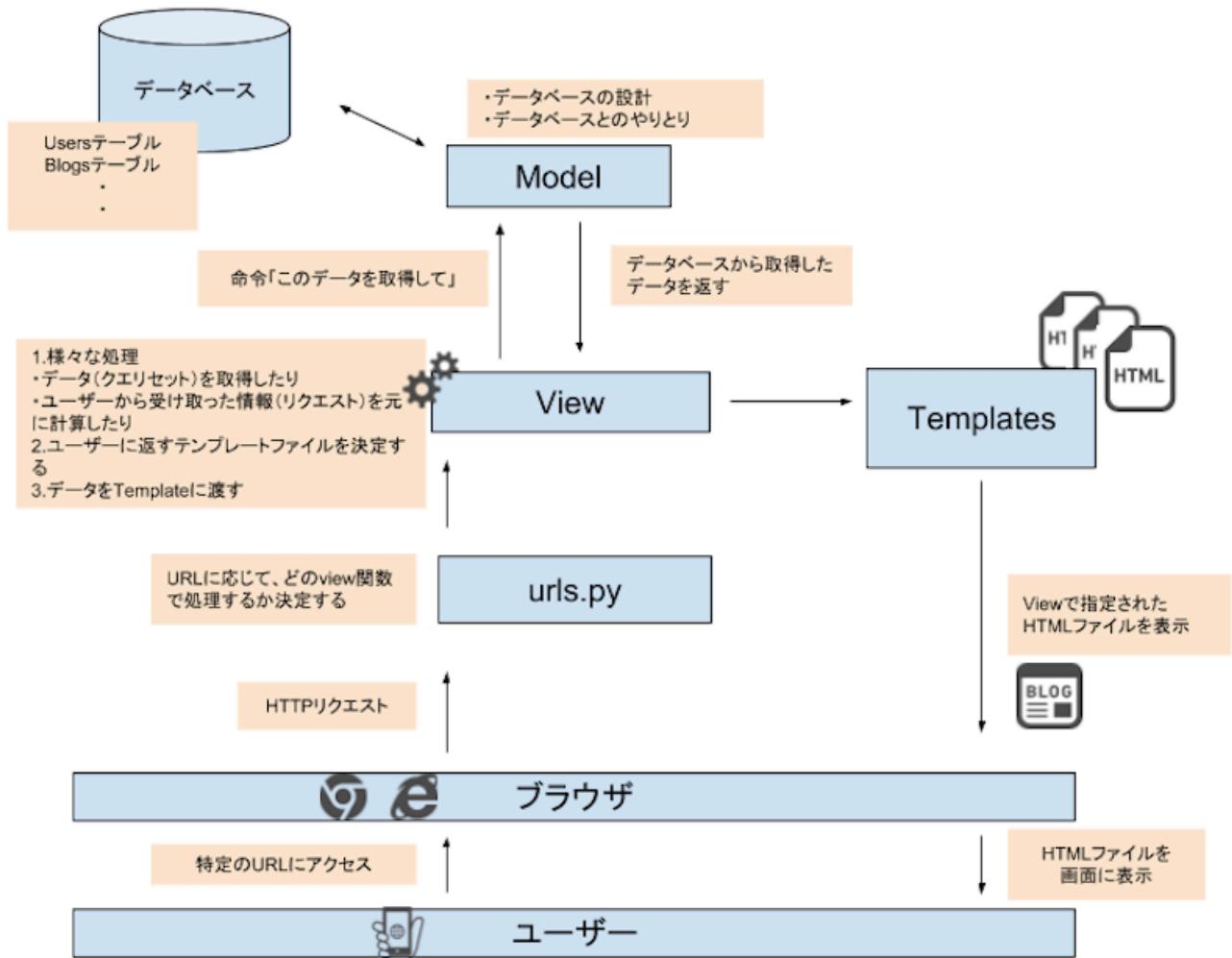
また、通常の Python コードと違い、for 文の終わりを表す`{% endfor %}`というコードが必要になるので、忘れないように注意してください。

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。つまり、`{{ 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 `def __str__(self) return self.title` と指定しているので、タイトルが表示されるようになります。

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 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 }}`と書いてあります。`|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。ここで使っている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>
        <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>
</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、飛び先の 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。これは固定の URL を設定しているのではなく、動的に変化する URL を指定しています。

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 (Integer) を urls.py に渡すことができます。

次に、詳細ページ自体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おきますか。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 ファイルを開いて、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。

第一引数の 'detail/<int:blog_id>' では URL を指定しています。int は、ここに整数型の値が入ることを示しています。トップページからのリンクで、変数 blog_id に数字が代入された状態で情報を受け取っているので、ここにはその数字が入力されることになります。また、ここに入った数字は変数 blog_id に代入されて詳細ページの View にも渡されます。

第二引数は、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 関数が処理されることを意味しています。

第三引数は、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 'blogs:detail' 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。

これで、`http://127.0.0.1:8000/detail/1` のページでは id=1 の記事の詳細ページを表示する URL を作成できました。ただし、まだ View 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 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 の第一引数の、変数 blog_id を受け取っています。

`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 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 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るので、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に 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。この場合は、エラーを

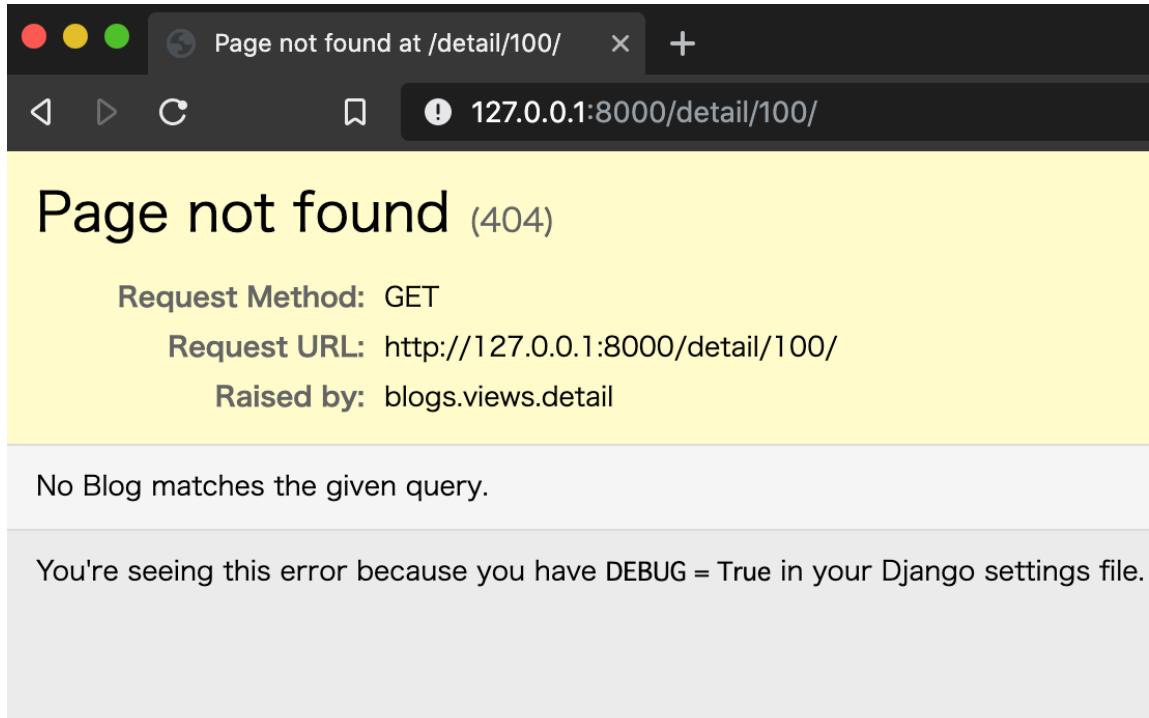
出すのではなく、「指定された ID の記事は存在しませんでした」という内容のページを出してあげるべきです。こういったページを **404 ページ** と呼びます。

Django には特定のモデルインスタンスの取得と、そのインスタンスの存在の確認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) # 追加
    return render(request, 'blogs/detail.html', {'blog': blog})
```

この処理で、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。



基本的な情報の表示の仕方は分かったでしょうか？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>
            <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>
    </div>
</body>
</html>
```

3.8 CRUD を理解しよう

この章では、簡単なブログアプリの作成を通して、Django による基本的な 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。



CRUD とは、**Create**、**Read**、**Update**、**Delete** の 4 つの操作をまとめた概念で、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。

皆さんも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 フレームワークには、この CRUD をベースにした操作をなるべく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 Web サイトのフォームから利用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。

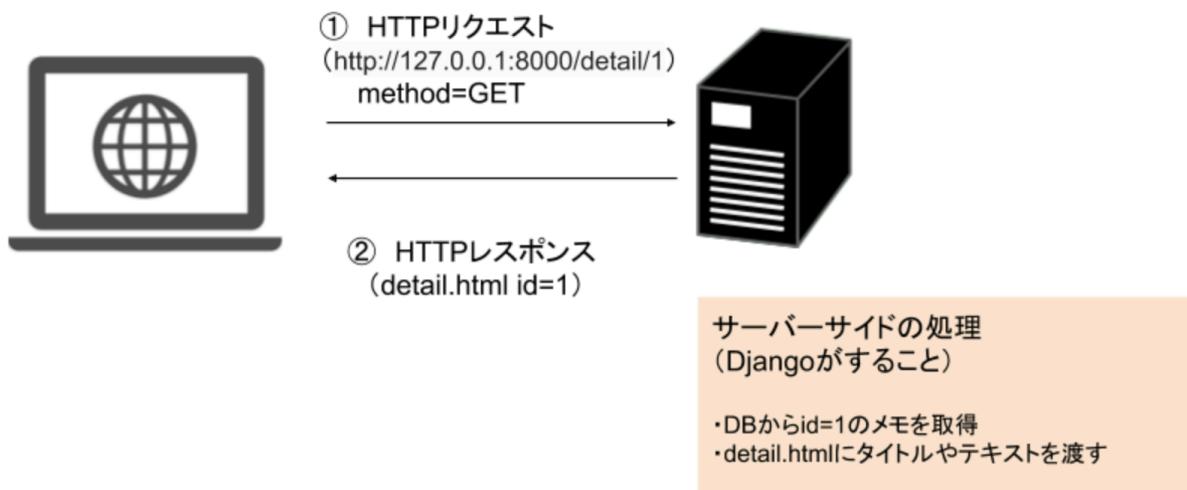
ブラウザとサーバーの通信では、HTTP というプロトコルが使われています。プロトコルとは規約という意味があり、HTTP という決まった方式に従ってブラウザとサーバーがやり取りをしているということです。

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「detail.html を返してほしい」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 のブログ記事を探だし、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 のプロジェクト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。

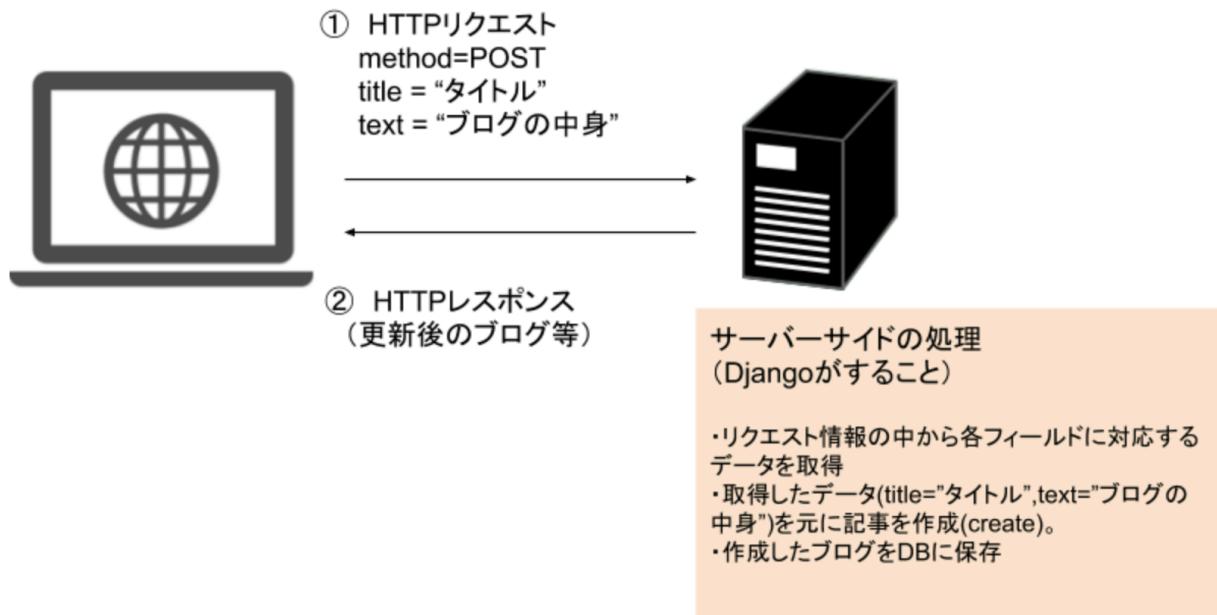
HTTP リクエストには、GET と POST 以外にも種類がありますが、ほとんどの場合このどちらかを利用することになります。

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、新しくブログ記事を Web サービス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` というモジュールをインポートし、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の、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テンプレート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 は以下の内容にしましょう。

```
(~/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>
    <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>
</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。

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、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しょうか。

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、"`{% 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 # 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 の場合を表しています。

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。これはフォームの「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」という内容でブログ記事をフォームから保存した場合、その内容の 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 Django の `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モデルの作成ができます。

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 HTML ファイルのパスを示すのに対して、`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。

今回の例では、新規ブログ記事を保存したらリダクレクトさせて `index` 関数を実行し、トップページを表示するように設計しています。トップページに飛ぶと、フォームから投稿したメモが追加されているのがわかるはずです。

これで、Admin ページだけでなく、Web サービス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 Django における 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;">
    <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

新規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。ここまでで、CRUDについて一通り学べる状態になります。

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、削除するブログの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
]
```

一番最後に path を追加しました。この設定により、`/delete/<id>/` 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 のブログ記事を取得し、それを delete した上で、ブログ記事の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 のブログが削除されるようになりました。

しかし、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、よく考えるとまずいですよね。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、単純に URL にアクセスする処理は、HTTP メソッドでいうと GET になります。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にのみ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ずに追加してください。

```
(~/DjangoBros/django_blog/blogs/views.py)
from django.views.decorators.http import require_POST

...
@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 `request.method=GET` の時は実行されません。

`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。

`request.method=POST` にするためには、`detail.html` に以下のようないし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>
  <div style="margin: 60px 20px;">
    {{ blog.text }}
  </div>
  <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"`としたボタンをフォーム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。

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>
    <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>
</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 タグ内だけ記載しておきます。

```
(~/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>
      <div style="margin: 60px 20px;">
        {{ blog.text }}
      </div>
      <div style="text-align: center; margin-top: 50px">
        <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>
```

```
<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>
```

これで編集ページと、画面遷移を作ることができました。

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での、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。その値に、今回編集する Blog インスタンスを与えます。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走らせます。

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るようにしておいてください。

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。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、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。

こちらのチュートリアルでも、第2章で作成したプロジェクトからのスタートになります。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章で作成したプロジェクトを「PhotoService」という名前にし、Django プロジェクトの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
$ django-admin 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  
  
urlpatterns = [  
    path('admin/', admin.site.urls),  
    path('', include('app.urls')),  
]
```

```
(~/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 get_object_or_404, redirect, 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>
```

ここまでで、トップページに簡単な表示をするだけのアプリケーションができました。このアプリケーションに少しずつ機能を追加していきましょう。

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。

Web サービスを開発するときには、ページを増やすごとにテンプレートファイル (HTML) を作成することになりますが、全てのページで共通して利用したい部分がたくさんあることに気づきます。皆さんが普段使っている 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、ページごとに共通の内容だったり、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、一つの部分を修正するために各ファイルの該当個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、その共通部分を様々な場所で使い回すことができる機能があります。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>
```

index.html に含まれていた<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。

トップページの表示用の 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 を拡張したもの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る所以、確実に基本をおさえておきましょう。

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 を適用していきましょう。

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望ましいです。

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"),
)
```

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。



トップページ

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、Django のプロジェクト内で使うモデルは基本的に自分で要件を定義し、コードを記述する必要があります。しかし Django ではデフォル

トで使える User モデルを提供しており、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。

実を言うと、みなさんはすでにこの User モデルを暗に利用しています。プロジェクトの立ち上げ時に、`python manage.py createsuperuser` コマンドで、管理者アカウント(superuser)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)

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。

The screenshot shows the Django Admin interface at the URL `127.0.0.1:8000/admin/`. The title bar says "Django 管理サイト". On the left, there's a sidebar with "サイト管理" and a red arrow pointing to the "ユーザー" link under the "認証と認可" section. The main content area shows a table with two rows: "グループ" and "ユーザー". Each row has "追加" and "変更" buttons. To the right of the table is a sidebar titled "最近行った操作" with sections for "自分の操作" and "利用不可". At the bottom of the main content area, a red text overlay says "ここから確認する".

ユーザーを変更 | Django サイト x +

← → ⌂ 127.0.0.1:8000/admin/auth/user/1/change/ ☆ ⌂ ⌂

Django 管理サイト

ようこそ JOBS サイトを表示 / パスワードの変更 / ログアウト

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: s2NKBl***** ハッシュ: 4aMnT7*****

生のパスワードは格納されていないため、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して パスワードを変更できます。

個人情報

名:

姓:

メールアドレス:

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。アカウントを消す代わりに選択を解除してください。

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 グループ

選択された グループ



ユーザーを変更 | Django サイト x +

← → ⌂ 127.0.0.1:8000/admin/auth/user/1/change/ ☆ ⌂ ⌂

このユーザーが持つ権限一覧です。ユーザーはすべてのログイン権限を持ちます。複数選択するには Control キーを押しながら選択してください。Mac では Command キーを使ってください。

ユーザー権限:

利用可能 ユーザー権限

Q フィルター

- admin | ログエントリー | Can add log entry
- admin | ログエントリー | Can change log entry
- admin | ログエントリー | Can delete log entry
- admin | ログエントリー | Can view log entry
- auth | グループ | Can add group
- auth | グループ | Can change group
- auth | グループ | Can delete group
- auth | グループ | Can view group
- auth | パーミッション | Can add permission
- auth | パーミッション | Can change permission
- auth | パーミッション | Can delete permission

全て選択

◎すべて削除

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。

選択された ユーザー権限

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

削除

保存してもう一つ追加

保存して編集を続ける

保存

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` のファイルから確認できます。

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。

(~/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@example.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 モデルの扱いをつかんだら、User ページを作っていきましょう。まずは簡易的に、個別の User を識別して、その User のユーザー名と管理者権限を持っているかどうかを画面に表示させてみます。

ここで作成するページの 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'),
]
```

```
(~/PhotoService/app/views.py)
```

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

...
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 %}
```

(管理ユーザーでログインした場合の表示)



@Jobs

True

管理者

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。非常によく利用するモデルなので、一通り確認しておき、実践の中で少しづ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。

Photoモデルによってできるテーブルのイメージ図です。

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						
	.						
	.						
	.						

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、画像ファイルに対応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おい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 サイト管理

Django 管理サイト

ホーム > App > Photos > photo を追加

photo を追加

Title: お散歩

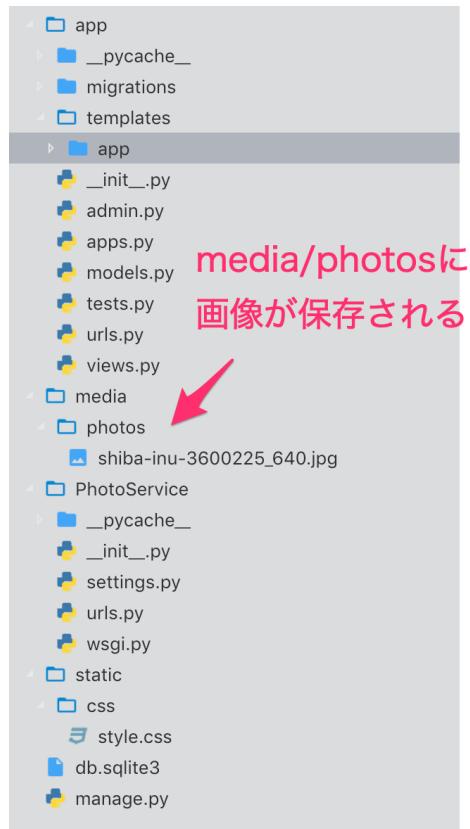
Comment: 寒いけど頑張ってお散歩したよ。

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。



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、アップロードされた画像はそのディレクトリに保存されるような設定をします。

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便利です。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ので、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きます。（今回は、`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。

`MEDIA_URL` は、ユーザーが生成したコンテンツの URL を表すのに対して、`STATIC_URL` は、CSS ファイルやトップページに固定で使われる画像など、サイト開発者が最初から用意している静的コンテンツを保管するディレクトリを表します。合わせて整理しておきましょう。

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。方針としては、「views.py で Photo インスタンスを全て取得して 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

# 修正
def index(request):
    photos = Photo.objects.all().order_by('-created_at')
    return render(request, 'app/index.html', {'photos': photos})
```

```
(~/PhotoService/app/templates/app/index.html)
{% extends 'app/base.html' %}

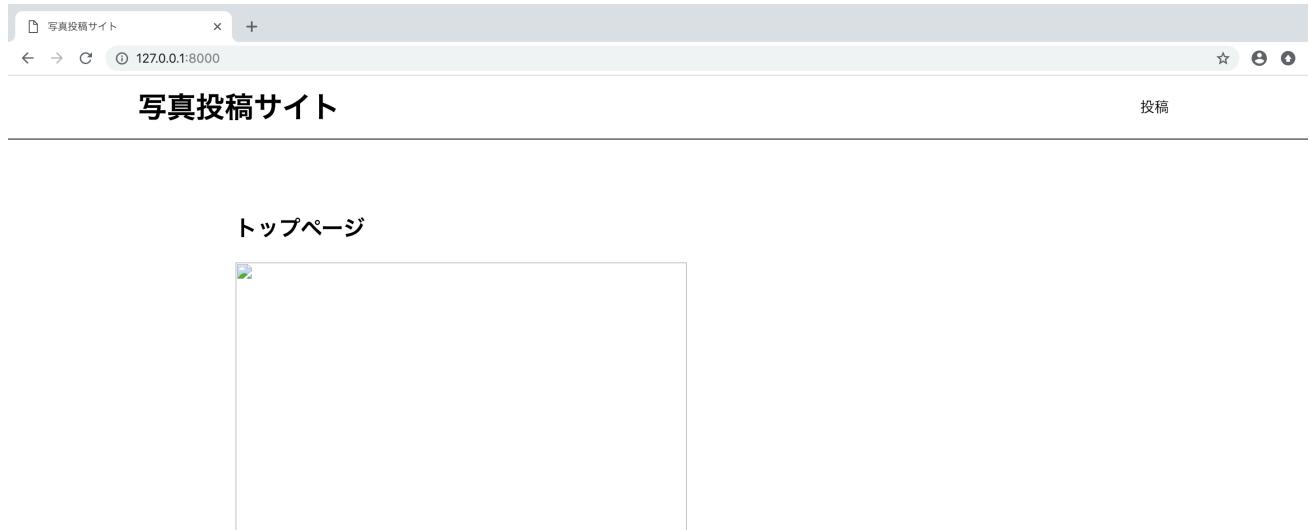
{% block content %}

<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。

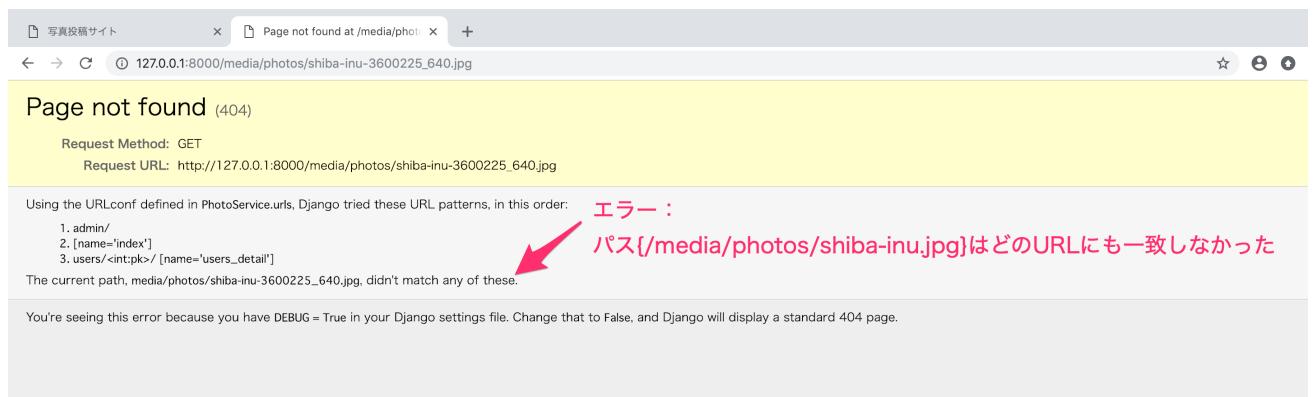


これは、**画像が保管されているディレクトリ**、つまり（**media ディレクトリ**）が一般に公開されていないことが原因です。管理者であれば Admin ページから画像を見ることができますが、一般の人は画像の URL にアクセスしても見ることができません。

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
from django.conf.urls.static import static

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) # 追加
```

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リロードを試してみてください。

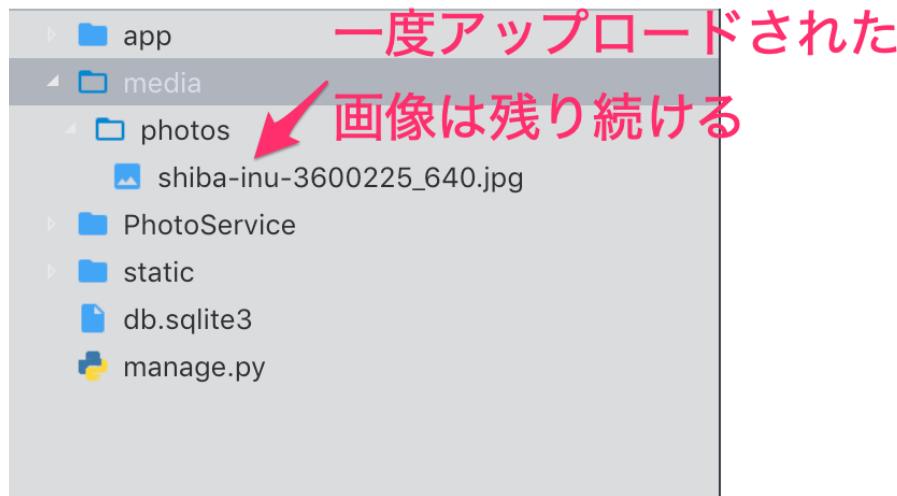
django-cleanup で画像ファイルを削除する

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かない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。

The screenshot shows the Django Admin site's change form for a 'photo' object. The title is 'photo を変更 | Django サイト管理'. The URL is 127.0.0.1:8000/admin/app/photo/1/change/. The page header says 'Django 管理サイト' and 'ようこそ JOBS サイトを表示 / パスワードの変更 / ログアウト'. The main content area has a heading 'photo を変更'. There are fields for 'Title:' (value: お散歩) and 'Comment:' (value: 寒いけど頑張ってお散歩したよ。). Under 'Image:', it shows the current image as 'photos/shiba-inu-3600225_640.jpg' and a file selection dropdown labeled '変更: ファイルを選択 | 選択されていません'. At the bottom, there is a red arrow pointing to a red '削除' (Delete) button. To its right, the text 'この投稿を削除しても、、、' is written in red. On the far right, there are three buttons: '保存してもう一つ追加', '保存して編集を続ける', and '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` というパッケージを使用します。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
```

(~/PhotoService/PhotoService/settings.py)

```
INSTALLED_APPS = (
    ...
    'django_cleanup',
)
```

これで、Photo インスタンスの削除と一緒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なります。

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						
	:						
	:						

Categories テーブル
1. 動物
2. 風景
3. 料理

Users テーブル
1. Jobs
2. Rola

```

    graph TD
      subgraph PhotoTable [Photos テーブル]
        direction LR
        P1[1. お散歩] --- C1[寒いけどがんばってお散歩したよ。]
        P2[2. 初めての富士山] --- C2[富士山の写真を撮ったよ。]
        P3[3. 美味しくできた] --- C3[卵料理つくった。おいしかった。]
        P4[4. ] --- C4[ ]
        P5[ : ] --- C5[ ]
        P6[ : ] --- C6[ ]
      end

      subgraph CategoryTable [Categories テーブル]
        direction TB
        C1 --- A1[1. 動物]
        C2 --- A2[2. 風景]
        C3 --- A3[3. 料理]
      end

      subgraph UserTable [Users テーブル]
        direction TB
        U1[1. Jobs] --- P1
        U2[2. Rola] --- P2
        U3[ ] --- P3
        U4[ ] --- P4
        U5[ ] --- P5
        U6[ ] --- P6
      end

      P1 --> C1
      P2 --> C2
      P3 --> C3
      C1 --> U1
      C2 --> U2
      C3 --> U2
  
```

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一引数に指定します。`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。

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は、`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 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 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、既存の全投稿データの user フィールドには、Jobs が保存されることになります。

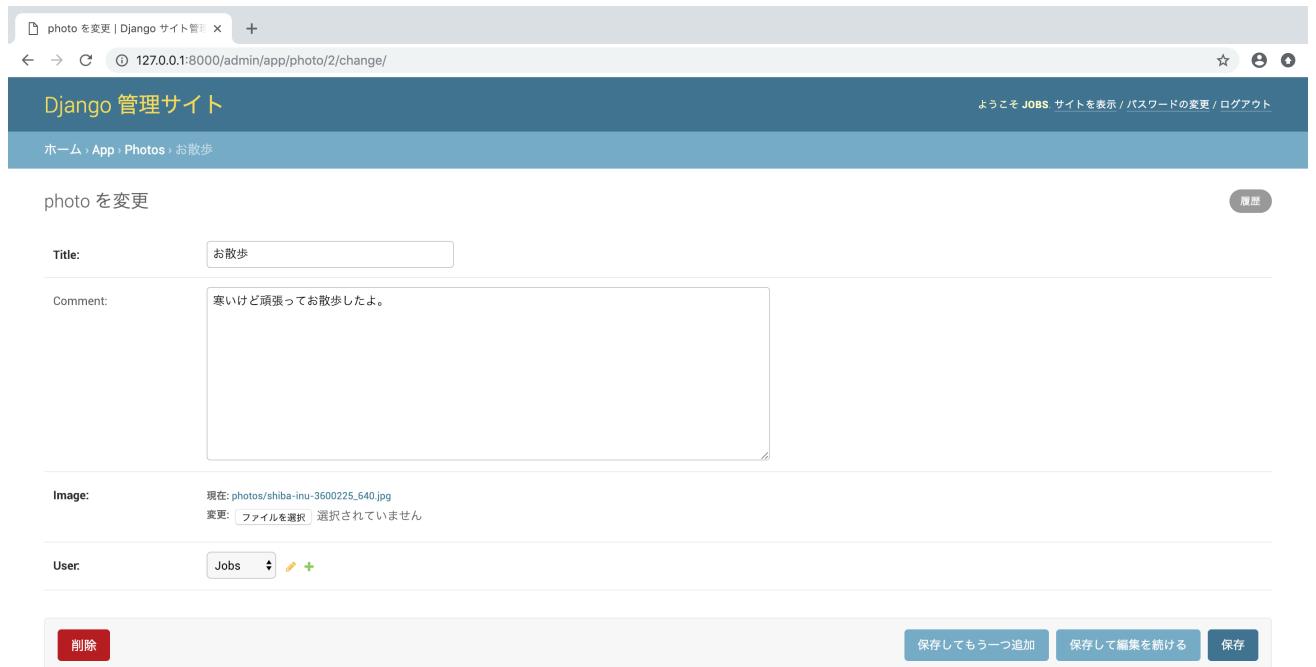
選択肢 2を選んだ場合、コマンドラインの入力モードは終了します。その後、自分で models.py を修正します。具体的には、`user = models.ForeignKey(User, on_delete=models.CASCADE, default=1)` のように、任意のユーザーID を default として指定します。これにより、マイグレートするタイミングで、既存の投稿は全て Jobs と紐づきます。

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 - Add field user to photo 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ユーザーが保存されているはずです。



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 のフィールドにアクセス
>>> photo1.user.id
```

```
1
>>> photo1.user.username
'Jobs'
>>> photo1.user.email
'jobs@sample.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 フィールドを作つてみましょう。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
```

```
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)
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。



ID	TITLE
3	料理
2	風景
1	動物

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):
```

```

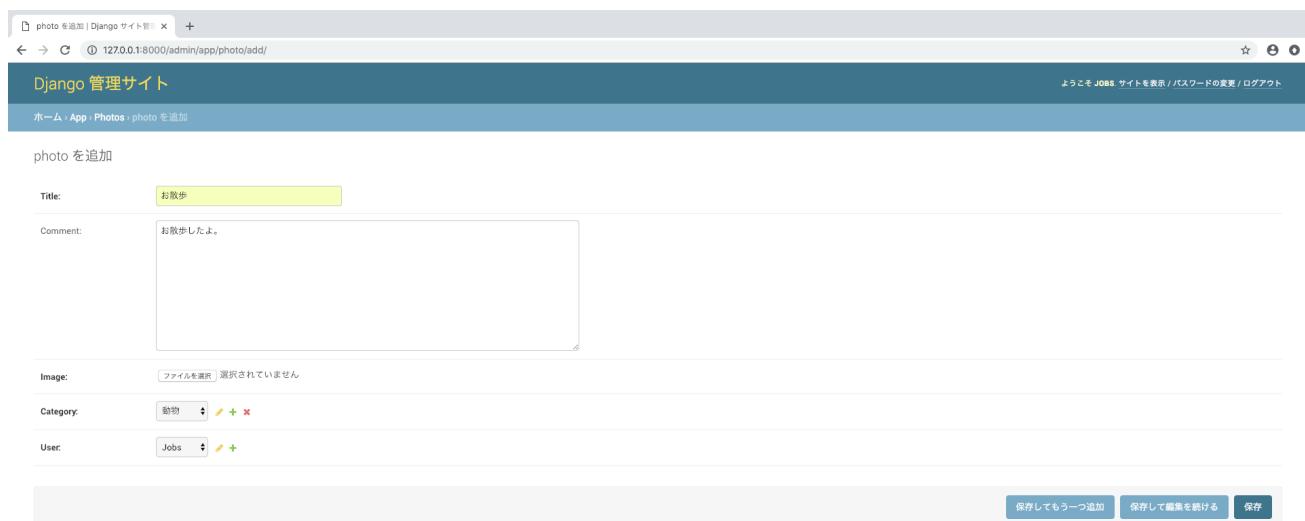
title = models.CharField(max_length=150)
comment = models.TextField(blank=True)
image = models.ImageField(upload_to='photos')
category = models.ForeignKey(Category,
                             on_delete=models.PROTECT) # 追加
user = models.ForeignKey(User, on_delete=models.CASCADE)
created_at = models.DateTimeField(auto_now=True)

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。



ForeignKey は一対多（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、第一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 Photos に参照されていると言うことができます。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 (参照してく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。この関係性を一対多 (OneToMany) の関係といいます。ForeignKey は一対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対一(OneToOne) という関係性もありますので、別のチュートリアルで紹介します。

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)
    photos = user.photo_set.all().order_by('-created_at')
    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、`django.contrib.auth.forms` には、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

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 という関数があるのでこれをそのまま使います。これらの関数が自動的にログイン・ログアウト処理を

行ってくれるのです。LoginView では、`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、`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')` と設定していますので、`http://127.0.0.1:8000/logout/` にアクセスした時点ですでに `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` を作ってないのでエラーになります。では、`login.html` を作っていきましょう。

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` を渡してくれるのと、`{{ 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>
      {% 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 を第一引数に取って 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>
```

```

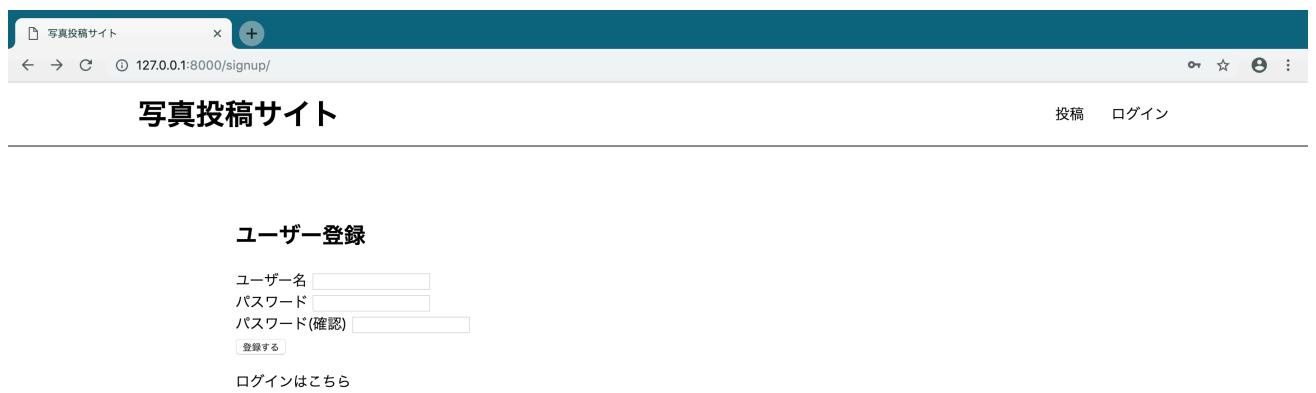
{{ form.password2 }}
{{ form.password2.errors }}
<br>
<input type="submit" value="登録する">
</form>

<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。



前のレッスンで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():
            new_user = form.save() # ユーザー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.contrib.auth.forms import UserCreationForm
from django.contrib.auth import authenticate, login # 追加

def signup(request):
    if request.method == 'POST':
        form = UserCreationForm(request.POST) # ユーザーインスタンスを作成
        if form.is_valid():
            new_user = form.save() # ユーザー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オブジェクト、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 メソッドは、認証がでてなくてもログインさせることができます。(上の authenticate で認証を実行する)
    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` というメソッド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。それぞれのメソッドの役割は以下の通りです。

- `authenticate`: `username` と `password` を引数に取り、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 オブジェクトを引数に取り、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` メソッドは認証に成功した 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` メソッド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 detail ページにリダイレクトさせます。

※`login` メソッド自体は認証機能を備えていないため、その前に `authenticate` メソッドで認証を行なっています。（`login` メソッド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` メソッド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">の中身だけ記述します。

```
(~/PhotoService/app/templates/app/base.html)
<div class="header-menu">
    <a href="{% url 'app:photos_new' %}">投稿</a> # 更新
    {% 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>
```

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。

```
(~/PhotoService/app/views.py)
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` というデコレーターを使って、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。

デコレーターは自作することもできますが、このように Django が提供しているデコレーターを適宜インポートして使うと良いでしょう。

② 入力された情報から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していない）

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` 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)

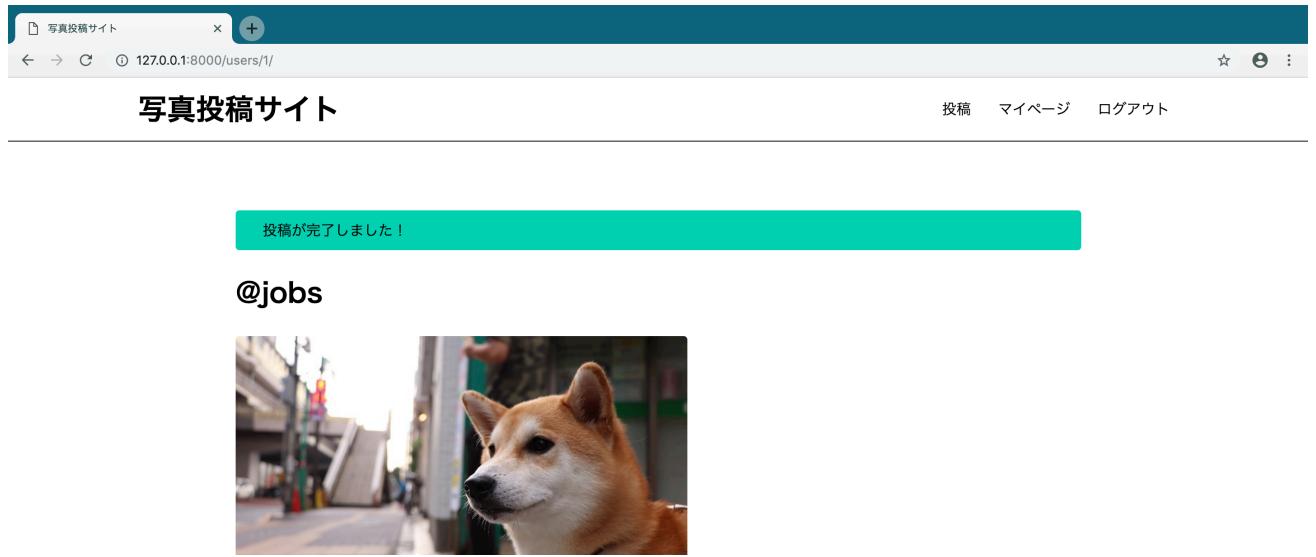
```
from django.contrib import messages

@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！")
            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。



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'),
    path('photos/<int:pk>/', views.photos_detail, name='photos_detail'),
    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)
    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 %}
<a href="{% url 'app:photos_detail' photo.id %}">
    
</a>
{% endfor %}
```

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

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

{% for photo in photos %}
<a href="{% url 'app:photos_detail' photo.id %}">
    
</a>
{% 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">
```

```

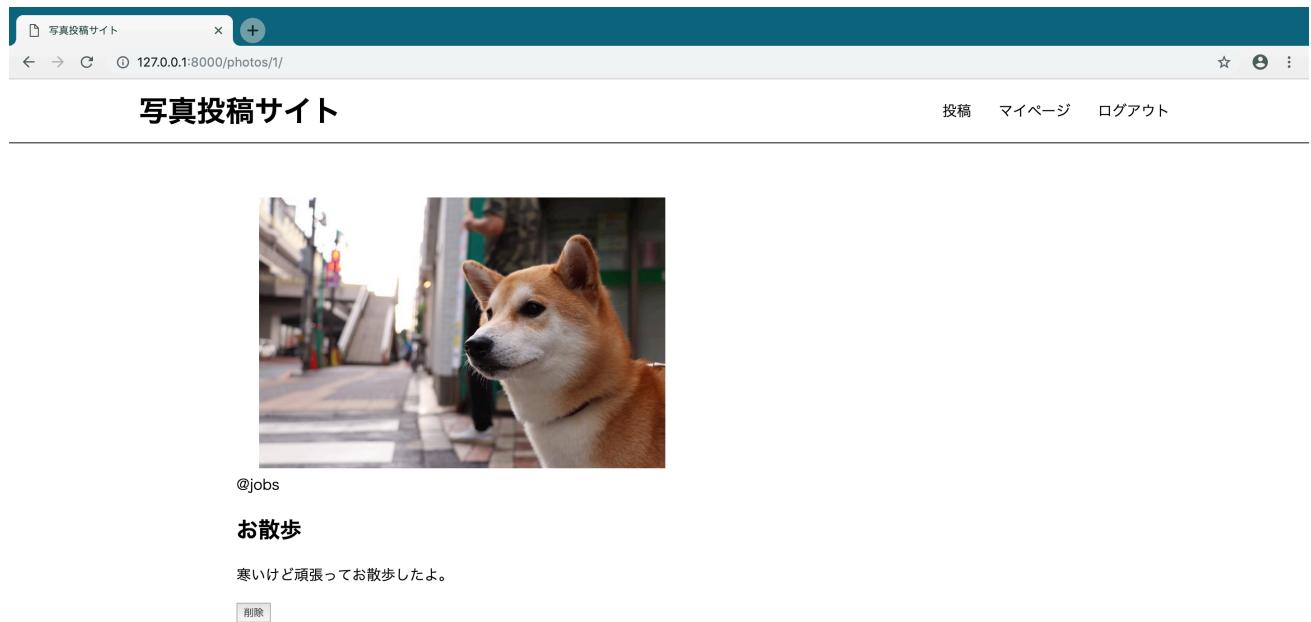

<div class="photo-info">
    <a href="{% url 'app:users_detail'
                  photo.user.id %}">@{{ photo.user }}</a>
</div>

<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 %}

```



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して写真を消去することができてしまいます。

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 テレ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。リファクタリングです。

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)

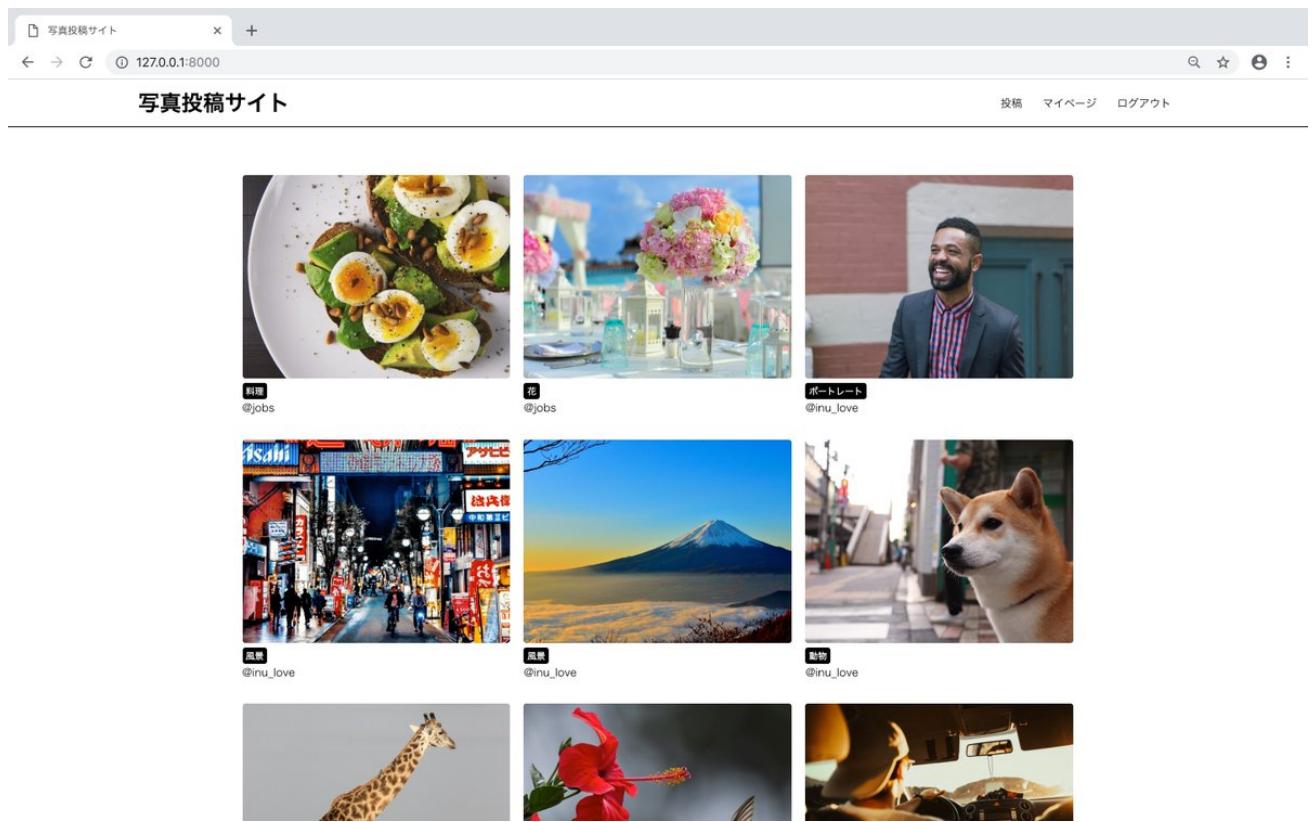
```
{% extends 'app/base.html' %}

{% 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。

- 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
- 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 -->
{% include 'app/photos_list.html' %}

{% endblock %}
```

```
(~/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 -->
{% include 'app/photos_list.html' %}

{% 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' %}
```

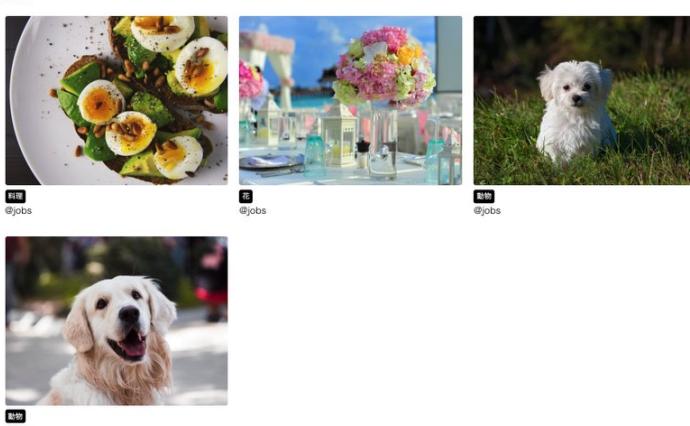
`photos.count != 0` のとき（1つでも投稿があるとき）は、投稿数を表示します。

写真投稿サイト

投稿が完了しました！

@jobs

投稿4件



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



1つも投稿していない他人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。

@rola



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'),
    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 = Category.objects.get(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 %}

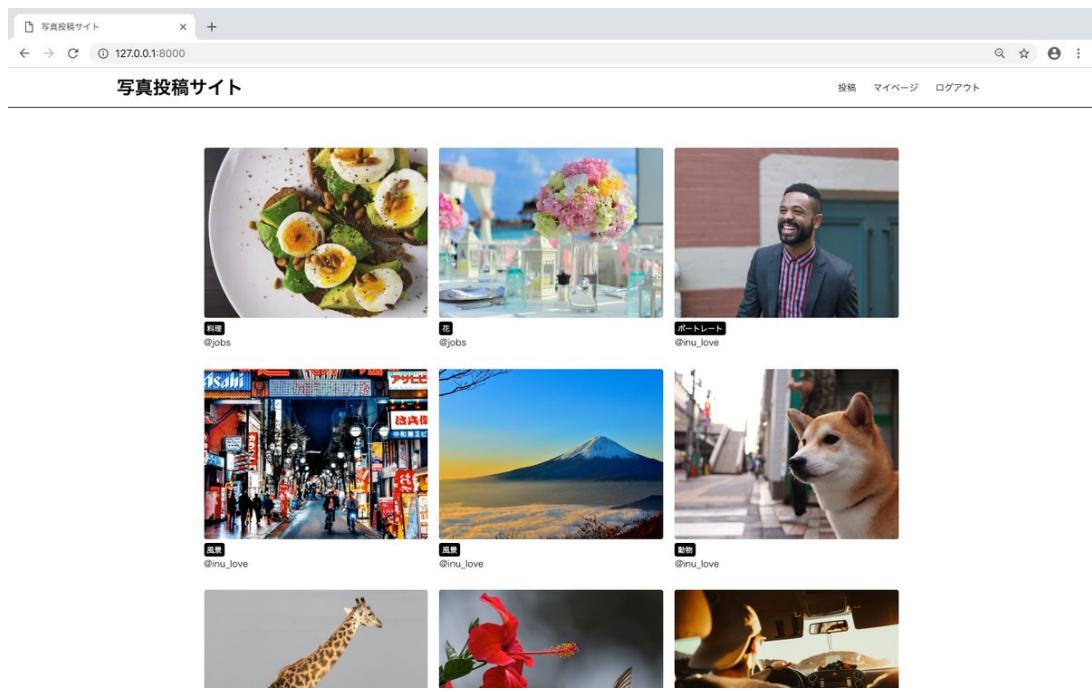
{% if category %}
<h2>カテゴリー：{{ category.title }} の検索結果</h2>
{% endif %}

{% include 'app/photos_list.html' %}

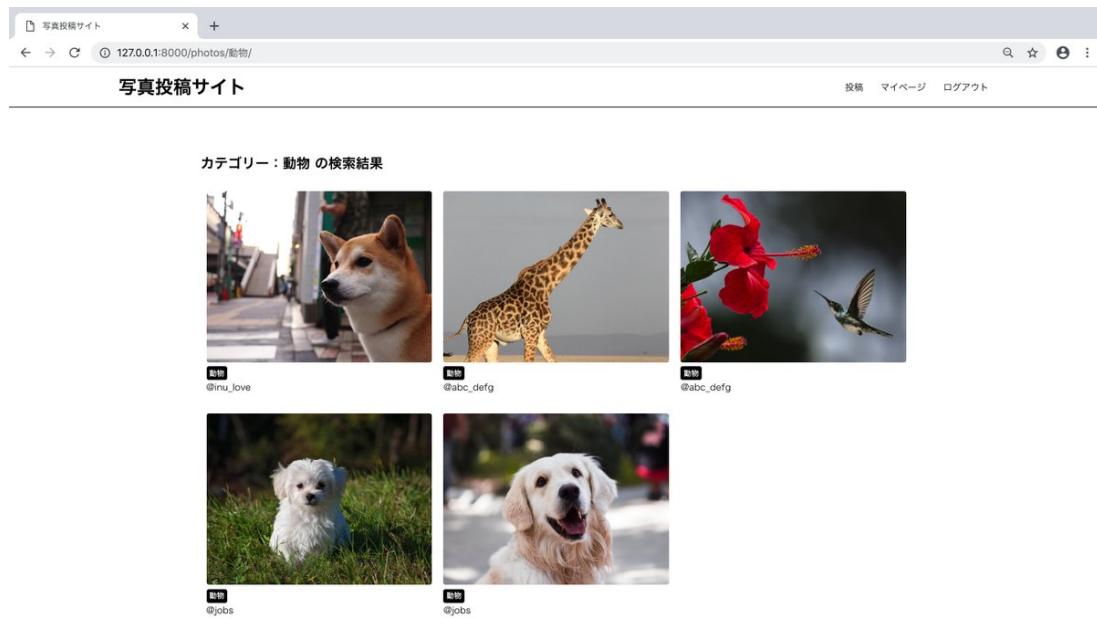
{% endblock %}
```

{% 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">  
      <a href="{% url 'app:photos_category' category=photo.category %}"  
         class="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③EC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。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 処理
- 外部の 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、外部の 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 サイトなどでの決済部分には、外部の 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章のデフォルトのプロジェクトを「ecsitet」という名前に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 に用意されている標準のモデルを利用していました。この User モデルはすぐに利用することができ、簡易的なサイトを作る際には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りますが、昨今の Web サービスでは username の代わりに Email アドレスや SNS アカウントを利用してのログインが主流です。また、デフォルトの属性以外にも、User モデルの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情報を User モデルで保持したい場合などが該当します。今回の場合だと、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。

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。これは、デフォルトの User モデルをアップデートする機能で、Django では `AbstractUser` と `AbstractBaseUser` の 2 種類のクラスを使う方法が用意されています。

`AbstractUser` クラスは、デフォルトの User モデルに含まれているフィールドなどの要素を持ちつつ、新しいフィールドやメソッドを追加する時に利用します。一方、`AbstractBaseUser` クラスは、ユーザーの認証に関わるような部分はデフォルトの User モデルと同様ですが、それ以外は特に何も実装されていない状態です。つまり、これまでデフォルトの User クラスで使われていた `username` や `first_name` などのようなフィールドすら持っていないため、自分が作りたい User モデルを自由に作成することができます。

基本的な User モデル構造を残したまま少しだけ手を加えるのであれば `AbstractUser` の方が良いですが、`AbstractBaseUser` の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りますが、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` によるカスタムユーザー モデルの作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` クラスを継承してカスタムユーザー モデル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 モデルを作成するために、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。もちろん単独のアプリケーションに処理を書くこともできますが、カスタムユーザーを作成する場合には基本的にアプリケーションを分けると管理がしやすくなります。

```
(~/ecssite)
# アプリケーションの作成
$ django-admin 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` で設定しているパラメータは、`users` アプリケーション内で設定する `User` というモデル（カスタムユーザー モデル）をこのプロジェクトの `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 モデルの作成

それでは実際にカスタムユーザー モデルを作っていきましょう。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` モデルの作成
- `BaseUserManager` を継承した `UserManager` クラスの作成
- 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
- EC サイトで使う「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` は以下の通りです。

```
(~/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.core.mail import send_mail
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 を継承したカスタムユーザー モデルを作成しました。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。カスタムユーザーを作成するときには必ず導入するようにしてください。

このカスタムユーザーでは、username の代わりに email をフィールドとして持つようにしました。そのための設定として、email フィールドをユニーク（他のユーザーと被りがない）に指定しています。

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、必ず一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 username がこの役割を担っていましたが、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` と `password` のフィールドについては必ず必須になるので、こちらで指定する必要はありません。

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。

`AbstractBaseUser` モデルの実装はそれほど難しくないので、実際のコードを確認しておくと良いでしょう。

https://github.com/django/django/blob/2.2.5/django/contrib/auth/base_user.py#L47

カスタムユーザーマネージャークラスの作成

続いて、`BaseUserManager` を継承した `UserManager` クラスを作成していきます。以下の内容を `models.py` に追加してください。クラスの参照の関係から、`UserManager` クラスは `User` モデルの記述より上に記載するようにしてください。

(~/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 で User モデルを作成
        email = self.normalize_email(email)
        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)

# User モデルより上に記載する
class User(AbstractBaseUser, PermissionsMixin):
    ...

```

UserManager とは、Django の User モデルの管理に必要なクラスのことです。User を作成するためのメソッドなどを管理しています。これは Django のモデルを扱うクラスも継承しており、User モデルの中の objects という変数で管理しています。`User.objects.all()` のようなメソッドを使う際に objects という記述をこれまでにも何度も目にしていると思います。

こちらのコードは基本的に Django のソースコードの BaseUserManager クラスの実装を利用しており、email によるユーザー認証など、必要な部分だけ書き換えています。

`use_in_migrations = True` と指定することで、こちらのクラスもマイグレーションで管理できるようになります。

BaseUserManager では、`create_user` や `create_superuser` などの関数が定義されており、それらをオーバーライドすることによって、username ではなく、email での登録をするようにしています。

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)
```

Admin ページのカスタマイズ自体は第 4 章でやったことと同じです。ここでは、カスタムした User モデルに合わせて表示するフィールドを変更しています。また、デフォルトの User 作成/変更フォームの内容も変える必要があります。

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、ここで一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このカスタムユーザー 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 モデルを作成することができるのには、基本的に User モデルを最初に作成するときだけなのです。

マイグレーションの init(0001) より後にカスタムユーザーを設定しても、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あとから修正することができません。一度デフォルトの User モデルを使ってマイグレーションをしてしまったら、そのプロジェクトであとから User モデルをカスタマイズすることは大変困難になります。

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ようになります。

今回のチュートリアルの場合、すでに初期状態のプロジェクトでマイグレーションをしていると思いますが、まだ他のモデルなどは作成していないので、一度データベースを削除してからマイグレーションを再度行いましょう。

```
(~/ecssite)
# DB の削除
$ rm db.sqlite3
$ 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 を作ってみてください。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。

(~/ecsite)

```
# 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！

Django 管理サイト ようこそ SAMPLE@GMAIL.COM

ホーム > Users > Users

変更する user を選択

検索

操作: 実行 1個の内ひとつも選択されていません

メールアドレス	IS_STAFF
sample@gmail.com	✓

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。

```
(~/ecsite)
# app アプリケーションの作成
$ python manage.py startapp app
```

```
(~/ecsite/ecsite/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二つです。

- 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。

```
(~/ecsite/ecsite/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')),
]
```

```
(~/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'),
]
```

setting.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。ここでは、`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になります。

ログインページの URL を作成したので、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>

```

(~/ecsite/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"> <!-- 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が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ので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():
            new_user = 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とっています。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。

(~/ecssite/app/urls.py)

```
...
urlpatterns = [
    path('', views.index, name='index'),
    path('signup/', views.signup, name='signup'), # 追加
    ...
]
```

続いては、この Web サービスで使用するモデルを作成しましょう。

5.3 必要なモデルを作ろう

Django での Web 開発の全体像もだいぶわかつってきたと思います。この節では、必要になるモデルを先に書き出していくします。

User モデル以外で必要になるモデルは以下の通りです。

- Product (商品の情報)
- Sale (個別の売上情報)

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。

```
(~/ecsite/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(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。

```
(~/ecsite/app/admin.py)
from django.contrib import admin
from .models import Product, Sale

admin.site.register(Product)
admin.site.register(Sale)
```

```
(~/ecsite)
# 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。

```
(~/ecsite/ecsite/settings.py)
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
MEDIA_URL = '/media/'

(~/ecsite/ecsite/urls.py)
from django.conf import settings
from django.conf.urls.static import static

# 投稿した画像を表示するための設定、解説は 4.3 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。Admin ページからいくつか商品情報を入力してください。

ManyToMany フィールドの追加

このサービスではユーザーごとにお気に入り商品を登録しておくことができますが、カスタムユーザー 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` という名前で、ManyToMany（多対多）のフィールドを付け加えています。先ほど Sale モデルに、Product モデルに対して ForeignKey でフィールドを追加しました。一つの Sale オブジェクトに対して Product は一つだけ紐づきますが、特定の Product は様々な Sale オブジェクトから参照される可能性があります。ForeignKey は 1 対多の関係を表すものでしたね。

User は複数のお気に入り商品を持つ可能性があり、商品は複数のユーザー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 で表現することができます。

ManyToMany によって中間テーブル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` の間の中間テーブルを作成しています。以下のようなイメージです。

(User モデルと Product モデルの個別のテーブル)

Usersテーブル				Productsテーブル			
id	email	point	date_joined	id	name	description	price
	EmailField	PositiveInteger	Datetime		CharField	TextField	PositiveInteger
1	user1@gmail.com	50,000	2019/09/01	1	おせんべい	・・・	3,000
2	user2@gmail.com	25,000	2019/09/02	2	ドリンクセット	・・・	2,500
3	user3@gmail.com	33,000	2019/09/04	3	スマートフォン	・・・	15,000
4	user4@gmail.com	3,000	2019/09/05	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
⋮		
⋮		
⋮		

中間テーブルでは、User と Product のインスタンスの結びつけだけが記録されています。本書では解説しませんが、こちらの中間テーブルを明示的に作成してフィールドもカスタマイズする方法もあります。例えば、その中間テーブルが作られた日時などを記録する場合などに使われます。

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。

(~/ecssite)

```
$ 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 からは、`app.models.Product` をインポートし、app アプリケーションの models.py からは、`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ます。

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。

```
(~/ecsitet/app/models.py)
class Sale(models.Model):
    """売上情報"""
    product = models.ForeignKey(Product, on_delete=models.PROTECT)
    user = models.ForeignKey('users.User',           # 修正
                           on_delete=models.PROTECT)
    ...
```

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ションして、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。

```
(~/ecsitet)
$ 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。

```
(~/ecsite/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')}),
        (_('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。

```
(~/ecsite/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。

```
(~/ecsitet/app/views.py)
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。

```
(~/ecsitet/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` にこのモジュールを入れた後、`{% load humanize %}` をテンプレートファイルに記述し、`{{ price | intcomma }}` という形でテンプレートタグを修飾します。

`settings.py` ファイルに追記をしましょう。

```
(~/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
    '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 の一覧を取得するようにします。

```
(~/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 %}
        {% 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。

```
(~/ecsitetools/ecsitetools/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。

```
(~/ecsitetools/ecsitetools/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;
}

.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 */
.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 */
.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 history */

.order-item {
  padding: 20px;
  display: flex;
}

.order-item-image {
  flex: 1;
}

.order-item-info {
  flex: 1;
  font-size: 16px;
}

.order-item-info .info-value {
  float: right;
}

.order-item-info .info-timestamp {
  float: right;
  font-size: 12px;
  font-style: italic;
  margin-top: 8px;
}

.order-item-amount-form {
  float: right;
  margin: 10px;
}

.order-item-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)

Djamazon

Point: 50,000 お気に入り カート 注文履歴 ログアウト



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。

お気に入り商品ボタン

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、その商品をそのユーザーの 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。

まずは特定のユーザーに対してお気に入り商品の状態を切り替えるための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):
    product = get_object_or_404(Product, pk=request.POST["product_id"])
    user = request.user
    if product in user.fav_products.all():
        user.fav_products.remove(product)
    else:
        user.fav_products.add(product)
    return redirect('app:detail', product_id=product.id)
```

この関数にアクセスすると、特定のプロダクト ID の商品の存在をチェックし、そのユーザーのお気に入り商品に入っているかどうかによって、fav_products に追加か削除を行った後に同じページにリダイレクトする処理が実行されます。

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テンプレートに、お気に入りボタンを追加しましょう。

```
(~/ecsite/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>
    {% 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="
お気に入りに入れる" />
        {% endif %}
      </form>
    {% endif %}
  </div>
  <p>{{ product.description }}</p>
</div>
```

```
...
```

ここでは、ユーザーがログインしている場合にのみ、fav_products の状態に合わせて表示を切り替えてボタンを出し分けています。

続いて、お気に入り商品ページ用の 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を使い回していますが、お気に入り商品ページと分かるように、index.html に少し追記します。

```
(~/ecsite/app/templates/app/index.html)
```

```
...
```

```
{% block content %}
```

```
<!-- 以下を追加 -->
{% if 'fav_products' in request.path %}
<div style="font-weight:bold; font-size: 28px; margin-bottom: 20px;">
    お気に入り商品
</div>
{% endif %}
<!-- 追加ここまで -->
```

```
<div class="product-container">
```

```
...
```

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。

```
(~/ecsite/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'),
]
```

(~/ecsite/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。

5.5 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、今度はセッションを使って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、商品はそのまま決済まで進むのではなく、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なる必要があります。HTTP はブラウザとサーバー間のやり取りの方式を決めた規約のことでしたね。特定の URL にアクセスすることで、そのリクエストに対応したレスポンスをサーバーが判断して情報を返すという一連の仕組みが 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 回のやり取りは全て、完全に独立したやり取りとして認識されるということです。つまり、HTTP ではページを移動するごとにユーザーの情報を引き継ぐことができない仕組みになっているのです。そのため、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、一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。

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。抽象的な説明になってしまいますが、具体例で解説します。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特定のユーザーに対して行われる必要があるため、一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るのか』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。

このセッションを判定するために、Cookie を使っています。最初に Web サービスにアクセスしたときに、一意な Cookie をブラウザに保存しておき、それ以降のサーバーに対するリクエストの際に毎回 Cookie の情報も送信すると、同じ Cookie の値を利用している期間は一つのセッションとして扱うことができるようになります。

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 を送っ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、毎回ユーザー名とパスワードを送信しなくとも、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持して、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、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、状況に応じて対応しましょう。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、ウェブ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などで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ことができるようになります。Web開発の中でも重要な概念なので、大枠を掴んだ後に専門書などで深掘りすることをお勧めします。

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持しておき、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。

<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',
    ...
]
```

DjangoではViewにおけるrequest(view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、特定のユーザー（もしくは特定のブラウザ）をクッキーで判別し、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。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)

```
# セッションの値を取り出す
```

```
cart = request.session['cart']

# セッションにデータを保存する
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
# プロダクト ID と商品の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 個入っている
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。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を入力して処理を行うだけのフォームです。マイナスの数値は入力できないようにするために `min_value = 1` 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。

```
(~/ecsitet/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)
    add_to_cart_form = AddToCartForm(request.POST or None)
    if add_to_cart_form.is_valid():
        num = add_to_cart_form.cleaned_data['num']

        # セッションに cart というキーがあるかどうかで処理を分ける
        if 'cart' in request.session:
            # すでに特定の商品の個数が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:
            # 新しく cart というセッションのキーを追加
            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)
context = {
    'product': product,
    'add_to_cart_form': add_to_cart_form,
}
```

```
return render(request, 'app/detail.html', context)
```

View ではまず、セッションの中に cart というキーがあるかを確認します。まだなければ、cart というキーで、プロダクト ID をキーにして個数を値として持つ辞書型のオブジェクトを格納します。

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。ロジックとしてはシンプルですね。

```
(~/ecsite/app/templates/app/detail.html)
```

```
...
<p>{{ product.description }}</p>

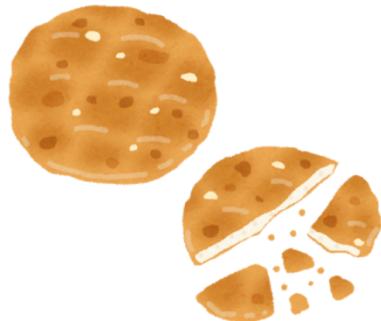
<div>
  {% if request.user.is_authenticated %}
    <form action="{% url 'app:detail' product.id %}"
method="post">{% csrf_token %}
      {{ add_to_cart_form.as_p }}
      <button class="purchase-button" type="submit">カートに追加する
    </button>
  </form>
  {% else %}
    <a href="{% url 'app:login' %}?next={{ request.path }}"><button
class="purchase-button">ログインして購入する</button></a>
  {% endif %}
</div>

</div>
...
```

テンプレートには商品説明の下の部分に以上の内容を追加してください。

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。

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

5.6 カートページから決済できるようにしよう

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。この情報を利用して、カートページで購入する内容を最終確認し、送付先の住所を入力して決済するまでのフローを実装してみましょう。

ここでは、住所の入力を便利にするために、外部の API を使って郵便番号から住所を自動で検索して入力できるような処理も実装してみます。外部の API の使い方は今後より複雑な Web サービスを作る際に必須とも言える知識なので、こちらのチュートリアルで基本を抑えましょう。

セッションから商品情報を取得する

まずはカートのセッションに追加した商品の情報をもとにカートページに商品を並べてみましょう。以下のように実装してください。

```
(~/ecsite/app/urls.py)
urlpatterns = [
    ...
    path('cart/', views.cart, name='cart'), # 追加
]
```

```
(~/ecsite/app/views.py)
```

```
# 追加
@login_required
def cart(request):
    cart = request.session.get('cart', {})
    cart_products = dict()
    total_price = 0
    for product_id, num in cart.items():
        product = Product.objects.get(id=product_id)
        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)
```

```
(~/ecsite/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-item">
    <div class="order-item-image">
        <a href="{% url 'app:detail' product.id %}">
            
        </a>
    </div>
    <div class="order-item-info">
        <h2>{{ product.name }}</h2>
        <div>価格 : <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 %}
```

(~/ecsitet/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、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？実は 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アップデートされないようになっているのです。

今回の場合、セッションに cart というキーの情報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 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)

```
# cart が空の時
session = {}

# 一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}
}

# 二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}  
}
```

このような処理の際にセッションが更新されていることを明示的に伝えるには、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、毎回このような処理を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。

```
(~/ecssite/ecssite/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ちゃんと商品を追加していくことができました！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

おせんべい

面白い本

価格:	1,200
個数:	1

面白い本

自作のテンプレートフィルタ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引き渡すというのも一つのやり方ですが、

今回はテンプレートフィルタを自作して、テンプレート側で小計の計算処理を行うようにしましょう。

テンプレートフィルタは `intcomma` などのような形ですでに利用したことがあります。これらは Django がデフォルトで用意しているテンプレートフィルタなのですが、今回のように価格と個数の掛け算をするフィルタは用意されていません。このフィルタを自分で定義して使っていきます。

app アプリケーション内に `templatetags` というフォルダをつくり、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カスタムしたフィルタを定義した `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。

```
(~/ecssite/)  
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` にカスタムタグの内容を作成していきます。

```
(~/ecssite/app/templatetags/filters.py)  
from django import template  
  
register = template.Library()  
  
@register.simple_tag  
def multiply(value1, value2):  
    return value1 * value2
```

`@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。このように定義することで、テンプレート側で `multiply` という独自定義のテンプレートタグが使えるようになります。

```
(~/ecsite/app/templates/app/cart.html)
{% extends 'app/base.html' %}
{% load humanize %}
{% load filters %} # filters ファイルをロードする

...
<-- 小計を追加する -->


## {{ product.name }}



価格 : <span class="info-value">{{ product.price | intcomma }}</span></div>


個数 : <span class="info-value">{{ num | intcomma }}</span></div>


---



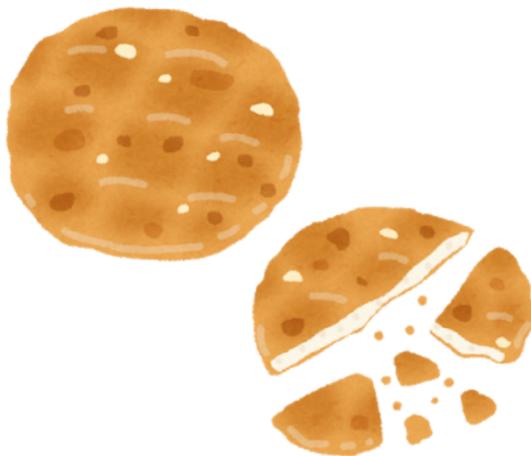
小計 : <span class="info-value">{{% multiply product.price num %}}</span></div>


...


```

独自のテンプレートフィルタ `multiply` には二つの引数が必要なので、`product.price` と `num` を指定します。内部ではこれらの値が掛け算され、その答えがテンプレート上に表示されるようになります。

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

カートに追加した商品の個数を変更できるようにしましょう。こちらは cart セッション内の特定の商品の個数を変更するための View を用意し、テンプレートに紐付けます。

(~/ecsite/app/templates/app/cart.html)

```
...
<!-- 小計の下に form を追加 -->
<div>小計 : <span class="info-value">{%
    multiply product.price
    num %}</span></div>
<div class="order-item-amount-form">
    <form action="{% url 'app:change_item_amount' %}" method="post">{%
        csrf_token %}
        <input type="hidden" name="product_id"
        value="{{ product.id }}>
        <input type="submit" name="action_remove" value="一つ減らす" />
        <input type="submit" name="action_add" value="一つ増やす" />
    </form>
</div>
...
...
```

`change_item_amount` という View 関数に対して POST リクエストを飛ばす form を作ります。ボタンは二つ用意していますが、`action_remove` と `action_add` の名前をつけることで、どちらのボタンからリクエストが送られてきたのかを判別することができます。

(~/ecsite/app/views.py)

```
@login_required
@require_POST
def change_item_amount(request):
    product_id = request.POST["product_id"]
    cart_session = request.session['cart']
    if product_id in cart_session:
        if 'action_remove' in request.POST:
            cart_session[product_id] -= 1
        if 'action_add' in request.POST:
            cart_session[product_id] += 1
        if cart_session[product_id] <= 0:
            del cart_session[product_id]
    return redirect('app:cart')
```

View の方の処理は直感的かと思います。cart セッションから、特定の商品の個数を足し引きし、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います。

最後に URL を紐付けましょう。

(~/ecsite/app/urls.py)

```
urlpatterns = [
    ...
    path('change_item_amount/', views.change_item_amount,
name='change_item_amount'), # 追加
]
```

Web API を利用して住所を取得しよう

商品の情報を表示して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、様々な Web サービスに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、Web API になります。

Web API 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、該当する住所を検索する機能」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。具体的には、**指定の URL に特定の形式で郵便番号を付与してリクエストを送ると、その郵便番号に該当する住所の情報を返してくれる**のです。実際に見てていきましょう。

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 サービスを使いましょう。

<http://zipcloud.ibsnet.co.jp/doc/api>

こちらの Web 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るものです。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。

<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。

<http://zipcloud.ibsnet.co.jp/api/search?zipcode=1000001>

実際にブラウザ上で確認してみると、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": "トウキヨウト",
      "kana2": "チヨダク",
      "kana3": "チヨダ",
      "prefcode": "13",
      "zipcode": "1000001"
    }
  ],
  "status": 200
}
```

Web API 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ので、非常に直感的にデータを扱うことができます。

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適切な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になると思います。

```
{
  "message": null,
  "results": null,
  "status": 200
}
```

result が null になっています。このように、Web API 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ことに気をつけましょう。

それでは郵便番号検索の Web API を利用して情報をコードから取得できるようにしましょう。Python では HTTP ライブラリとして `requests` というものを利用することが多いです。ドキュメントは以下です。

<https://requests-docs-ja.readthedocs.io/en/latest/>

このライブラリを使って View ファイルに郵便番号検索の関数を作りましょう。pip コマンドを使って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
```

関数は以下の内容にします。

```
(~/ecsite/app/views.py)
import json # 追加
import requests # 追加

# 郵便番号検索の API を利用する関数
def get_address(zip_code):
    REQUEST_URL =
f'http://zipcloud.ibsnet.co.jp/api/search?zipcode={zip_code}'
    address = ''
    response = requests.get(REQUEST_URL)
    response = json.loads(response.text)
    result, api_status = response['results'], response['status']
    if api_status == 200:
        result = result[0]
        address = result['address1'] + result['address2'] +
result['address3']
    return address
```

上記の `get_address` 関数では、`zip_code` という名前で郵便番号を入力し、もし適切に情報が返ってきた場合にはその情報をつなぎ合わせて返却するようにしています。

試しに、トップページの View 関数 (index) 内に `print(get_address(1000001))` と記述して、トップページにアクセスしてみてください。うまく API から情報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) に以下のような文字が出力されるはずです。

```
東京都千代田区千代田
```

これで、郵便番号を入力するだけで外部から該当の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、実際にカートの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。

```
(~/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)
```

続いて、View の cart 関数を修正しましょう。PurchaseForm では、テンプレート上で `search_address` と `buy_product` という名前の二つのボタンを用意し、どちらのボタンからリクエストが送信されたかによって処理を分けるようにします。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入れています。

```
(~/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 = dict()
    total_price = 0
    for product_id, num in cart.items():
        product = Product.objects.get(id=product_id)
        cart_products[product] = num
        total_price += product.price * num

    purchase_form = PurchaseForm(request.POST or None)
    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 = get_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に入力してあげる
            purchase_form = PurchaseForm(initial={'zip_code': zip_code,
            'address': address})

        # 購入ボタンが押された場合
        if 'buy_product' in request.POST:
            # 住所が入力済みか確認する
            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確認
            if not bool(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確認
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
for product_id, num in cart.items():
    if not Product.objects.filter(pk=product_id).exists():
        del cart[product_id]
    product = Product.objects.get(pk=product_id)
    sale = Sale(product=product, user=request.user,
amount=num, price=product.price, total_price=num*product.price)
    sale.save()
# ポイントを削減
user.point -= total_price
user.save()
del request.session['cart']
messages.success(request, "商品の購入が完了しました！")
return redirect('app:cart')

else:
    return redirect('app:cart')

context = {
    'purchase_form': purchase_form,
    'cart_products': cart_products,
    'total_price': total_price
}
return render(request, 'app/cart.html', context)

```

テンプレートには以下のように一番上に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>
<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.non_field_errors }}
        {{ purchase_form.zip_code.label_tag }}
        {{ purchase_form.zip_code }}
        <input type="submit" name="search_address" value="住所を検索"
    />
        <span>{{ purchase_form.zip_code.help_text }}</span>
    </div>
    <div>
        {{ purchase_form.address.label_tag }}
        {{ purchase_form.address }}
    </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>
</div>
</form>
</div>

{% for product, num in cart_products.items %}
...

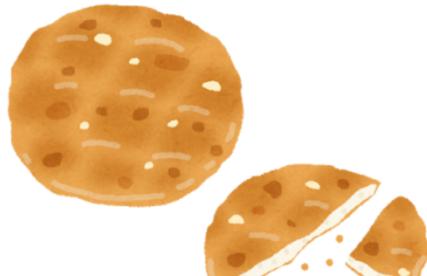
```

カート

郵便番号: 住所を検索
住所:

請求額: 9,000

購入する



おせんべい

価格:	600
個数:	4
小計:	2400

実装できたら、いろいろな入力を試し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。

商品の購入が完了しました！

カート

郵便番号: 住所を検索
住所:

請求額: 0

購入する

5.7 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内容にします。

```
(~/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-item">
        <div class="order-item-image">
            <a href="{% url 'app:detail' sale.product.id %}">
                
            </a>
        </div>
        <div class="order-item-info">
            <h2>{{ sale.product.name }}</h2>
            <div>価格 : <span class="info-value">{{ sale.price | intcomma }}</span></div>
            <div>個数 : <span class="info-value">{{ sale.amount | intcomma }}</span></div>
        </div>
    </div>
{% endfor %}
```

```
<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。

(~/ecsitet/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

これで過去の注文履歴を表示することができました。

お疲れ様です！今回のチュートリアルはこれで終了です。この EC サイトのチュートリアルでは、Web サービス作成のための非常に重要な基本を学習しました。ぜひ、しっかりと内容を復習した上で、本家のドキュメントなどでも知識を深めてみてください。

Django チュートリアルで学ぶ Web 開発入門

2019 年 11 月 3 日 初版発行

2020 年 11 月 6 日 第 2 版発行

By DjangoBrothers

(C)2020 DjangoBrothers