

現場で使える **Django** の教科書

《基礎編》

横瀬 明仁 (**akiyoko**) 著

2018-08-21 版 発行

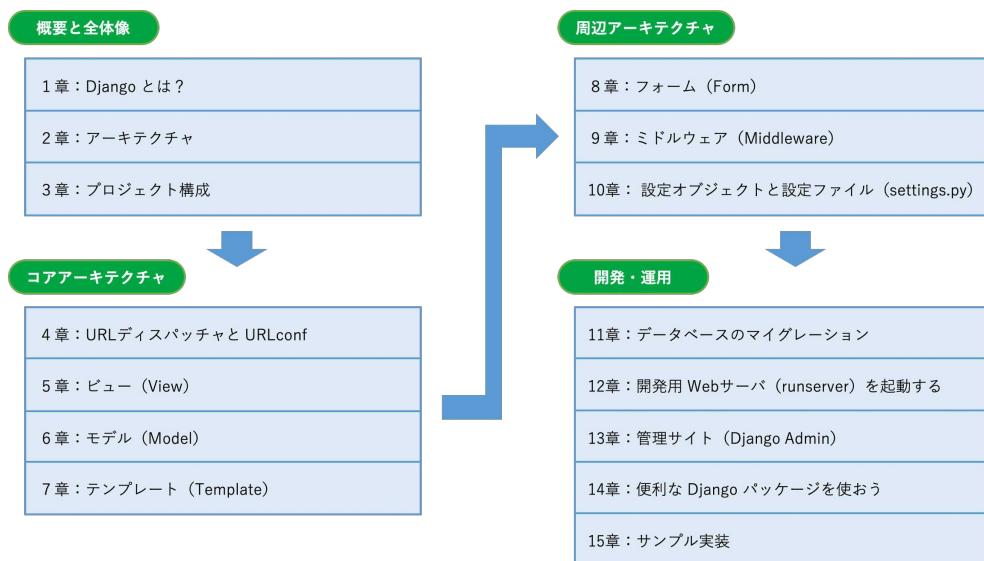
はじめに

本書は、Python で Web アプリケーションを作成するためのフレームワークである「Django（ジャンゴ）」を現場で使う際に必要となる基礎知識についてまとめた解説書です。

データ分析や機械学習のライブラリが充実しているなどの理由から、Python を学び始める人が最近急増しています。Python の得意分野である科学技術計算やデータ分析、機械学習、画像処理などと Web アプリケーションを連携させようとした場合に、同じ Python が使える Django が選択されるケースが今後ますます増えていくことは想像に難くありません。本書で Django の基礎をマスターすれば、Python を活用する幅がさらに広がっていくことでしょう。

本書の構成

本書では、第 3 章までで Django の全体像や重要な概念の説明をおこない、第 4 章以降でそれぞれの構成要素（コンポーネント）の説明をしながら、筆者の知見や経験から得たベストプラクティスについても紹介していきます。



この本は基本的に座学中心で、手を動かさずに読めるようになっています。しかしながら、いくら頭で泳ぎ方を覚えてても水泳をマスターすることができないのと同じで、Django は現場で使ってはじめてその真価を発揮するものです。本書で全体像を把握し、基礎的な知識を手に入れたあとは、実際に Django を動かしてトライ・アンド・エラーをしながらコツを掴んでいってください。

対象読者

本書の読者としては、

- Django を始めてみたけど今いちコツが掴めないという初級者の方
- Django のベストプラクティスを学びたいと考えている中級者の方

を想定しています。また特に、

- Django の日本語書籍が無くて困っている方
- Django で一度挫折したことがある方

にピッタリな再入門書にもなっています。筆者が仕事で Django を使い始めた数年前に「こんな本が現場にあったらよかったのに」という本を目指して執筆したので、そういうニーズには十分応えられているはずです。

Web アプリケーション開発の経験がまるっきり無いという方には多少難しい内容になっているかもしれません、分かるところから読んでいただき、少し慣れた頃にディープな部分を読み進めるようにすればよいでしょう。

対象 OS

著者が動作確認をしている環境は次の通りです。

- macOS 10.14, 10.15
- Windows 10 Home
- Ubuntu 18.04 (Docker Desktop for Mac 上で動作確認)

なお、本書で掲載しているコマンドは、注意書きがなければ基本的に macOS のターミナルで実行させることを前提としています。Windows など他の環境で実行する場合は適宜読み替えてください。ここで、

```
(venv) $
```

というプロンプトの表記がされている場合は、「venv」という名前の仮想環境にアクティ

べートしている状態を表すものとします。また、

>>>

という表記は、付録「E：覚えておきたい Django 管理コマンド 10 選」で解説する Django 管理コマンドの shell (Django シェル)

```
(venv) $ python manage.py shell
```

で起動された REPL (対話型評価環境) での実行内容を指すものとします。

巻末の付録「D：Docker でラクラク開発」では、Docker を使って Ubuntu 上で Django のサンプルプロジェクトを動かす方法を紹介しています。Ubuntu は AWS やくらのクラウド、DigitalOcean などのクラウドサービスで仮想 OS として利用できるため、開発時から利用することで本番移行時のトラブルが少なくなるといったメリットが期待できます。ぜひ参考にしてください。

対象バージョン

本書が対象としている Python および Django のバージョンはそれぞれ次の通りです。

- Python 3.7
- Django 2.2 LTS

Django がサポートしている Python のバージョンは、下表 1 に示す通り Django のバージョンごとに異なります。Django 2.2 は Python 3.7 以外にも 3.5 および 3.6 をサポートしています。また、2.2.8 以降は Python 3.8 にも対応しています。

▼表 1 Django バージョンごとの Python バージョン対応状況

	2.7	3.4	3.5	3.6	3.7	3.8
Django 1.11	○	○	○	○	△ (1.11.17 以降)	-
Django 2.0	-	○	○	○	○	-
Django 2.1	-	-	○	○	○	-
Django 2.2	-	-	○	○	○	△ (2.2.8 以降)
Django 3.0	-	-	-	○	○	○

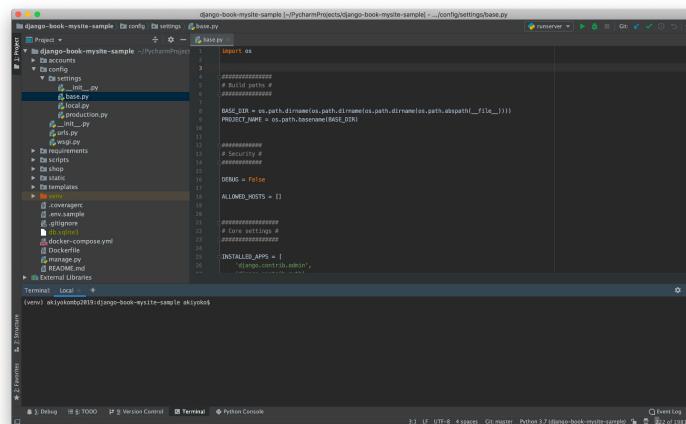
著者

横瀬 明仁 (akiyoko)

Python / Django 推しのサーバサイドエンジニア。「技術書典 4」で人生初の技術同人誌を発表し、これまでに本作を含めて 3 冊の Django 関連の技術同人誌を執筆。趣味で「akiyoko blog」^{*1} という技術ブログを執筆中。

筆者オススメの開発環境

筆者は Python を使った開発には「PyCharm」^{*2} という JetBrains 社製の IDE（統合開発環境）を愛用しています。PyCharm を使っている理由は、コードジャンプやコードの自動整形などの機能が開発効率を著しく向上させる点と、インストールしたそのままの状態で快適に使える点が気に入っているからです。有償の Professional Edition を利用していますが、無償の Community Edition (CE) でも本書で説明する操作は問題なく動作させることができますのでご安心ください。



▲図 1 PyCharm CE の画面

ローカル環境への Python のインストール手順については巻末の付録「A : Python 3 のインストール手順」を、PyCharm のインストールと初期設定については「B : PyCharm のインストールと初期設定」を、PyCharm を使った Django 開発環境の構築手順については「C : PyCharm による Django 開発環境の構築手順」を参考にしてください。

^{*1} <https://akiyoko.hatenablog.jp/>

^{*2} <https://www.jetbrains.com/pycharm/>

サンプルコードの著作権

本書に掲載しているサンプルコードの著作権については原則的に著者に帰属しますが、(IT の現場で利用する限りにおいては) 著作権上の複製権・翻案権、および著作者人格権は行使しませんので、コピーして商用プロジェクトで使うなり改変するなりして自由に使っていただいて構いません。著作者の表記も不要です。

免責事項

本書に掲載したサンプルコードや情報、本書を利用することで発生したトラブルや損失、損害に対して、著者は一切責任を負いません。

目次

はじめに	2
第 1 章 Django とは？	11
1.1 概要	11
1.2 Django のバージョンとリリースサイクル	12
1.3 まとめ	13
第 2 章 アーキテクチャ	14
2.1 全体像	14
2.2 MTV フレームワークとは？	16
2.3 まとめ	16
第 3 章 プロジェクト構成	17
3.1 プロジェクトとアプリケーション	17
3.2 django-admin と manage.py	18
3.3 よくあるプロジェクト構成	19
3.4 ベストプラクティス 1：分かりやすいプロジェクト構成	21
3.5 まとめ	24
第 4 章 URL ディスパッチャと URLconf	25
4.1 概要	25
4.2 URLconf の書き方	26
4.3 エラーハンドリング	30
4.4 ベストプラクティス 2：アプリケーションごとに urls.py を配置する	31
4.5 まとめ	32
第 5 章 ビュー (View)	33
5.1 概要	33
5.2 ビュー関数の書き方（関数ベース vs クラスベース）	35
5.3 すべての基本となる基本汎用ビュー	36
5.4 シンプルでよく使う基本汎用ビュー	38
5.5 さまざまな用途に特化した汎用ビュー	40

目次

5.6	ログイン・ログアウトについて	42
5.7	まとめ	45
第6章	モデル (Model)	46
6.1	概要	46
6.2	モデルクラスの書き方	46
6.3	「一対一」「多対一」「多対多」リレーションはどう定義するか?	48
6.4	よく使われる User モデル	53
6.5	モデルマネージャとクエリセット	54
6.6	単体のオブジェクトを取得する	56
6.7	複数のオブジェクトを取得する	57
6.8	単体のオブジェクトを保存・更新・削除する	62
6.9	ベストプラクティス 3: User モデルを拡張する	63
6.10	ベストプラクティス 4: 発行されるクエリを確認する	65
6.11	ベストプラクティス 5: select_related / prefetch_related でクエリ本数を減らす	66
6.12	まとめ	68
第7章	テンプレート (Template)	69
7.1	概要	69
7.2	変数表示	70
7.3	フィルタ	71
7.4	テンプレートタグ	74
7.5	ベストプラクティス 6: ベーステンプレートを用意する	79
7.6	まとめ	80
第8章	フォーム (Form)	81
8.1	概要	81
8.2	バリデーションの仕組み	82
8.3	フォームクラスの書き方	84
8.4	ビューやテンプレートからフォームを利用する方法	86
8.5	CSRF 対策について	89
8.6	ベストプラクティス 7: こんなときは ModelForm を継承しよう	91
8.7	まとめ	96
第9章	ミドルウェア (Middleware)	97
9.1	概要	97
9.2	主なミドルウェアの役割	98
9.3	ミドルウェアの書き方	101
9.4	ベストプラクティス 8: メッセージフレームワークを使う	103
9.5	まとめ	105

第 10 章 設定オブジェクトと設定ファイル (<code>settings.py</code>)	106
10.1 概要	106
10.2 インストールするアプリケーション一覧	107
10.3 デバッグ設定	108
10.4 静的ファイル関連の設定	109
10.5 メディアファイル関連の設定	113
10.6 データベースの設定	115
10.7 ロギングの設定	118
10.8 その他の重要な設定	122
10.9 ベストプラクティス 9：個人の開発環境の設定は <code>local_settings.py</code> に書く	126
10.10 ベストプラクティス 10：シークレットな変数は <code>.env</code> ファイルに書く .	127
10.11 まとめ	129
第 11 章 データベースのマイグレーション	130
11.1 概要	130
11.2 <code>makemigrations</code> (マイグレーションファイルの作成)	131
11.3 <code>migrate</code> (マイグレーションの実行)	132
11.4 マイグレーション履歴	133
11.5 まとめ	134
第 12 章 開発用 Web サーバ (<code>runserver</code>) を起動する	135
12.1 概要	135
12.2 <code>runserver</code> コマンドについて	136
12.3 まとめ	137
第 13 章 管理サイト (Django Admin)	138
13.1 概要	138
13.2 モデルの登録方法	139
13.3 一部機能のカスタマイズ方法	139
13.4 利用条件	141
13.5 使い方	142
13.6 まとめ	144
第 14 章 便利な Django パッケージを使おう	145
14.1 概要	145
14.2 DRY 系パッケージ	146
14.3 開発補助系パッケージ	148
14.4 まとめ	149
第 15 章 サンプルコード	150
15.1 概要	150

目次

15.2	サンプルプロジェクトを動かすまでの最短ステップ	151
15.3	まとめ	153
A	: Python 3 のインストール手順	154
B	: PyCharm のインストールと初期設定	162
C	: PyCharm による Django 開発環境の構築手順	169
D	: Docker でラクラク開発	173
E	: 覚えておきたい Django 管理コマンド 10 選	177

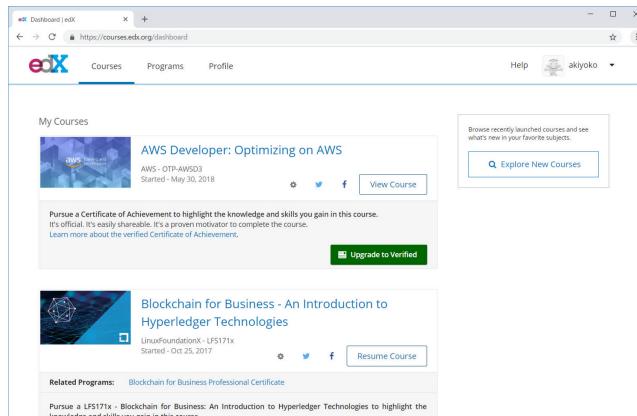
第1章

Django とは？

1.1 概要

Django（ジャンゴ）は Python で Web アプリケーションを作成するためのフレームワークです。Web アプリケーションを作成するために必要な機能が何でも揃っているというのが最大の特徴で、その特徴は一般に「フルスタック」と呼ばれます。

ビジネス領域での利用実績が多いのも Django の大きな特徴のひとつです。海外では「Instagram」や世界最大級のオンライン学習プラットフォーム「edX」^{*1}、国内では「日経電子版」や「connpass」^{*2}、「PyQ」^{*3}などの事例があります。中～大規模な Web アプリケーションを作成することを主眼としていますが、小規模な Web アプリケーションや REST API バックエンド（^{*4}）に利用することも可能です。



▲図 1.1 edX のダッシュボード画面

*1 <https://www.edx.org/>

*2 <https://connpass.com/>

*3 <https://pyq.jp/>

*4 Django REST Framework という Django パッケージがよく利用されます。

<https://www.djangoproject-rest-framework.org/>

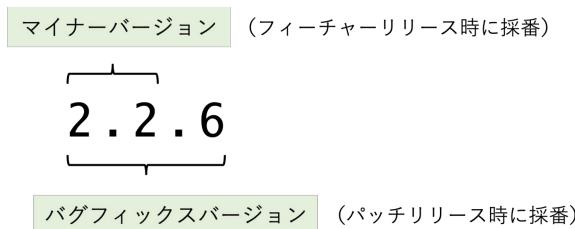
Django は 2005 年にオープンソース（BSD ライセンス）で公開されて以来、数ある Python 製の Web アプリケーションフレームワークの中で長らく人気トップの座に君臨し続けていますが、おそらく「フルスタックで機能が豊富」「ビジネス領域での実績が多い」というのが大きな要因になっていると考えられます。「マイクロフレームワーク」と呼ばれるシンプルな Web フレームワークの「Flask」^{*5} と比較されることが多い Django ですが、Django はユーザー認証まわりの機能（ユーザーモデル、パーミッション、ユーザーセッションなど）が標準で提供されているため、ログイン機能が必要なシステムを作りたい場合は Django を選ぶといよいよでしょう。Django を使うことで得られるメリットとしてはそのほかにも次のようなものが挙げられます。

- 高速開発できる（ひな形作成コマンドやよく使う汎用クラスの提供）
- 機能拡張が容易（豊富な Django パッケージ）
- データベースの構築とスキーマ変更が容易（マイグレーション機能）
- データベースのレコードの CRUD が GUI で操作できる（管理サイト機能）
- ユニットテストが効率よく書ける（テストクラスやテストクライアントの提供）
- さまざまな運用管理系コマンドが提供されている
- セキュリティ対策が充実（XSS、CSRF、SQL インジェクション、クリックジャッキングなどの対策、TLS/HTTPS サポート、パスワード暗号化など）
- 國際化対応（多言語翻訳が可能）
- キャッシュの利用が容易（アプリケーションの高速化が図れる）
- 地理空間データを扱える（GeoDjango 機能）

このように、Django はこの一冊で紹介しきれないほど幅広い分野をカバーしています。そこでネックになるのが学習コストですが、もちろんすべての機能について深く理解する必要はありませんし、使う機能を絞ってミニマムに利用することもできます。

1.2 Django のバージョンとリリースサイクル

Django には次のようなバージョニングルールが定められています。^{*6}



左から 1 番目はメジャーバージョンです。2 番目は細かな機能追加のためのフィー

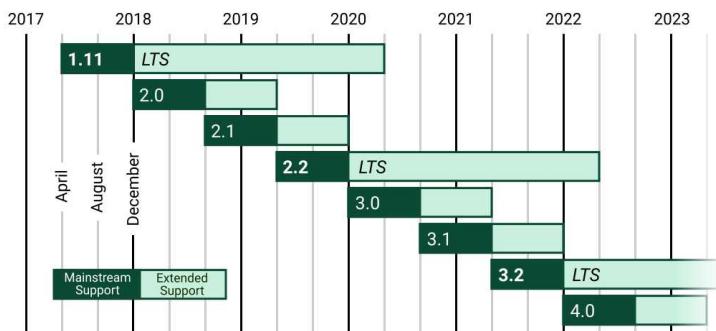
^{*5} <https://palletsprojects.com/p/flask/>

^{*6} <https://docs.djangoproject.com/ja/2.2/internals/release-process/>

チャーリリース時に付けられるバージョンで、フィーチャーリリースはおよそ 8 ヶ月ごとに予定されています。3 番目はバグフィックスやセキュリティパッチをともなうリリース時に付けられるバージョンで、必要に応じて不定期におこなわれます。

2017 年末に実に 9 年ぶりのメジャーバージョンアップがおこなわれ、Django 2.0 がリリースされました。その後、2 度のフィーチャーリリースと数回のパッチリリースがおこなわれ、現時点（2019 年 10 月）の最新バージョンは 2.2.6 となっています。

「LTS (Long-Term Support)」とは、Django の管理団体である「Django Software Foundation (DSF)」からのセキュリティパッチなどが 3 年もの長期間サポートされる特別バージョンのことです。Django 2 系で唯一の LTS バージョンとなる「2.2」は 2019 年 4 月にリリースされました。ひとつ前の LTS である「1.11」は 2020 年 4 月まで公式セキュリティサポートが継続されます。このようにリリースサイクルやセキュリティサポートが明確になっていることも、ビジネス領域で Django を採用しやすい一因となっています。



▲図 1.2 Django のリリースサイクル (<https://www.djangoproject.com/download/> より)

1.3 まとめ

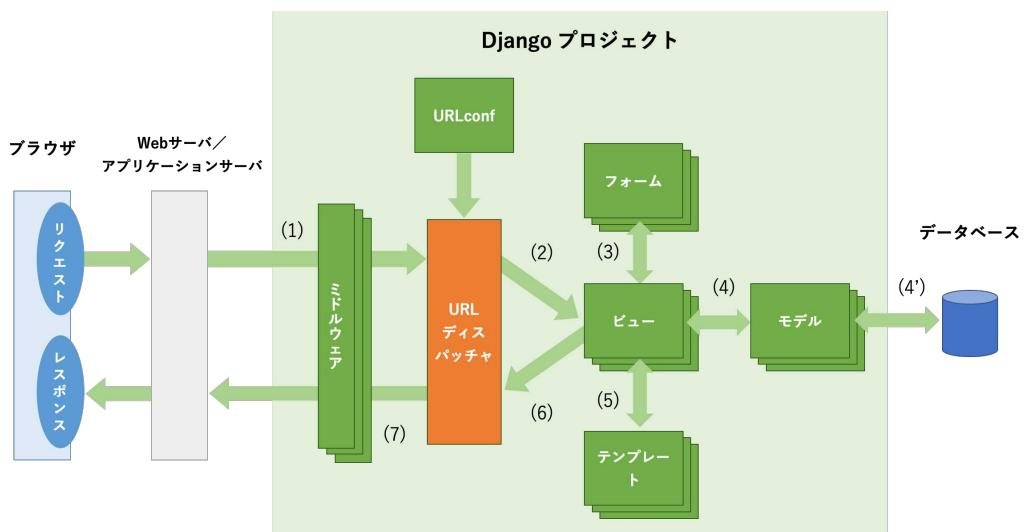
- Django は Python で Web アプリケーションを作成するためのフレームワーク
 - 最大の特徴は「フルスタック」（必要な機能が全部入り）
- 2019 年 10 月時点の Django の最新バージョンは 2.2.6
 - 2.2 は LTS バージョン。公式セキュリティサポートが 2022 年 4 月まで継続
- Django 3 系で唯一の LTS は 3.2 で、2021 年 4 月に正式リリース予定

第2章

アーキテクチャ

2.1 全体像

Django には Web アプリケーションを構成するためのさまざまな要素があり、それらが連携し合っています。Django のアーキテクチャの全体像はこのようなイメージになります。



▲図 2.1 Django のアーキテクチャの全体像

図に示した構成要素の概要と役割を簡単に説明すると表 2.1 のようになります。ちなみに Django では、「URLconf」は「urls.py」、「ビュー」は「views.py」、「モデル」は「models.py」、「フォーム」は「forms.py」、「ミドルウェア」は「middlewares.py」といったように、それぞれの役割ごとに対応するモジュールの名前が慣例的に決められています。そのあたりを含めた詳細については上第3章「プロジェクト構成」および下表の「章」の列に書かれた各章で解説をしています。なお、「URL ディスパッチャ」は図では分かり

やすいうように簡易的に書いていますが、その実体は Django 内部のコアモジュールで、開発時に触る必要はありません。

▼表 2.1 Django の構成要素

構成要素	概要	章
URLconf	URL パターンとビューのマッチング情報などを保持したモジュール	第 4 章
URL ディスパッチャ	URLconf にしたがってリクエスト URL に応じたビューを呼び出す	第 4 章
ビュー	モデルやフォーム、テンプレートと連携してレスポンスを作成する。 ユーザーにどのデータを提示するかを記述する	第 5 章
モデル	モデルクラスとテーブルの定義を紐付ける。 アプリケーションが扱う領域のデータとビジネスロジックを記述	第 6 章
テンプレート	特別な記法で表示内容を動的に変更できる HTML コンテンツ。 ユーザーにどのようにデータを提示するかを記述する	第 7 章
フォーム	ユーザーからの入力をオブジェクトとして扱うための入れ物。 入力データのバリデーション（妥当性チェック）もおこなう	第 8 章
ミドルウェア	リクエストの前処理とレスポンスの後処理をおこなう	第 9 章
設定オブジェクト	Django プロジェクト全体の設定を保持するオブジェクト (図には非掲載)	第 10 章

図 2.1においてブラウザがリクエストを送信してレスポンスを受け取るまでの一連の処理の流れは、次のようにになります。

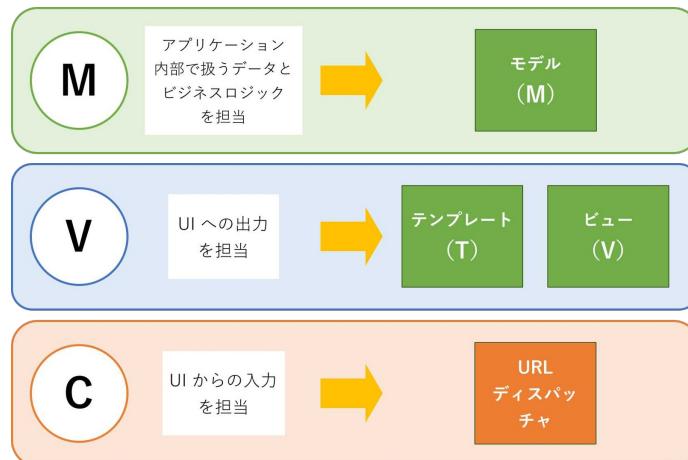
- (1) リクエストに対して「ミドルウェア」が前処理をおこなう
- (2) 「URL ディスパッチャ」が「URLconf」に登録された URL パターンにマッチする
「ビュー」を探し出して呼び出す
- (3) ビューがリクエストから取得した入力値を「フォーム」オブジェクトに変換し、
バリデーションをおこなう
- (4) ビューが「モデル」オブジェクトを取得してビジネスロジックを実行する
- (4') 適切なタイミングでデータベースにクエリが実行される
- (5) 取得したモデルオブジェクトやフォームオブジェクト、その他の変数の内容を
「テンプレート」にレンダリングしてレスポンスを作成する
- (6) URL ディスパッチャがレスポンスや例外をハンドリングする
- (7) ミドルウェアがレスポンスに後処理をおこない、ブラウザに返す

リクエストが処理されてレスポンスが返されるまでの一連の流れ、およびそれぞれの構成要素の大まかな役割を把握しておきましょう。

2.2 MTV フレームワークとは？

Django を構成する要素の中で、特に重要なものは「モデル」「テンプレート」「ビュー」の3つです。これら3つを合わせて、Django は「MTV (Model - Template - View) フレームワーク」とあると説明されることがあります^{*1}。

「MTV」は Web アプリケーションフレームワークの標準的な「MVC (Model - View - Controller)」モデルとは何が違うのでしょうか？ Django は「MVC」の「V」の部分を「テンプレート（データの見せ方）」と「ビュー（データの選び方）」に明示的に役割分担して定義ただけで、アーキテクチャの設計上は「MVC」と何ら変わりありません。なお「MVC」の「C」に相当するものは、URL ディスパッチャと考えればよいでしょう。



▲図 2.2 MVC と MTV

2.3 まとめ

- アーキテクチャの全体像を理解しよう
- 主に開発するのは「モデル」「テンプレート」「ビュー」の3つ

^{*1} <https://docs.djangoproject.com/ja/2.2/faq/general/>

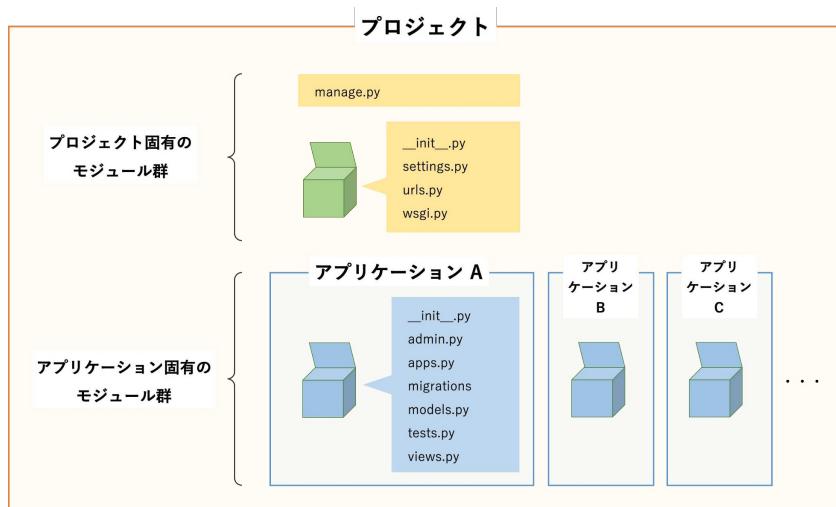
第3章

プロジェクト構成

3.1 プロジェクトとアプリケーション

Django プロジェクトのディレクトリおよびファイル構成にはちょっとしたクセがありますが、慣れてくると非常に分かりやすい構成をしています。

まず、構成を理解する上で非常に重要な概念があります。それは「プロジェクト」と「アプリケーション」です。次に示したものは、プロジェクトとアプリケーションの関係を示したイメージ図です。



▲図 3.1 プロジェクトとアプリケーションの関係（イメージ）

プロジェクトは一番外側の大きな「箱」です。それに対して、アプリケーションは機能ごとに分割された Python パッケージです（Python パッケージも「箱」ですが、内部の Python モジュールを外部から呼び出せるような名前空間を備えた箱で、Python パッケージとして識別するためのマーカーとして `__init__.py` を持ちます。^{*1} 本書では

^{*1} ちなみに Python 3.3 以降、`__init__.py` は必須ではなくなりました。ただしその場合は、「名前空間

「パッケージ」と「ディレクトリ」をほぼ同じ意味で使っています)。ある程度まとまった機能を実現するアプリケーションを、プロジェクト固有のモジュールが束ねているというイメージです。アプリケーションは丸ごと入れ替えできるように作るのがコツで、他のアプリケーションとなるべく依存しないように適度に分離させます。

Django を使った開発では、プロジェクトにアプリケーションを次々と追加する形で機能を組みしていくのが基本スタイルになります。なお、追加したアプリケーションを有効化するには、プロジェクト固有の設定を記述する特別なモジュール（「設定ファイル」と呼ばれます）にアプリケーションを「インストール」するための設定が必要です（詳細については「10.2 インストールするアプリケーション一覧」を参照）。

本書では「モジュール」という用語を何度も使いますが、モジュールとは簡単にいうと Python のコードが書かれたファイルのことです。他の Python プログラムから呼び出すことができる再利用可能なファイルを指すとされますが、あまり深く考えず、「Python のファイル」を格好良く呼び替えたものと考えればよいでしょう。

3.2 django-admin と manage.py

Django には、「django-admin」と「manage.py」の2種類の管理コマンドユーティリティが存在します。

「django-admin」は Django をインストールすると使えるようになる管理コマンドユーティリティで、基本的にどこからでも利用することができます。Django プロジェクトを新しく作成するときに利用しますが、後述する manage.py の方がより簡単に利用できるため、プロジェクトの新規作成以外の用途で利用することはほとんどないと考えてよいでしょう。

django-admin を使ってプロジェクトのひな形を作成するコマンドは次のようになります。コマンドの第一引数には「プロジェクト名」を指定しますが、この名前が図 3.1 の「プロジェクト固有のモジュール群」を格納する Python パッケージ名になるため、「-」などの一部の文字は使えません。第二引数にはプロジェクトを作成するディレクトリを指定します。これを省略すると、現在のディレクトリ直下にプロジェクト用のディレクトリが作成されます。また「.（ドット）」を指定すると、プロジェクト用のディレクトリが作成されずに現在のディレクトリ直下にひな形となるモジュール群が作成されます。

```
$ django-admin startproject <プロジェクト名> [<ディレクトリ>]
```

「パッケージ」と呼ばれる特殊なパッケージとして扱われます。

もう一方の「`manage.py`」は、`startproject` でプロジェクトを作成した際に自動で作成されるモジュールです。Django が提供する便利な管理コマンドが使えるほか、自作した Django 管理コマンドを `manage.py` 経由で実行することができます。`django-admin` をより使いやすくしたバージョンと考えてよいでしょう。

`manage.py` は開発時や運用時に頻繁に使いますが、`django-admin` は初回のプロジェクト作成時に一回使うだけで利用頻度は高くないでしょう。巻末の付録「E：覚えておきたい Django 管理コマンド 10 選」によく使う `manage.py` コマンドを掲載しましたので、ぜひひご一読ください。

ちなみに先に述べたように、独自の Django 管理コマンドを「`django.core.management.base.BaseCommand`」を継承して自作することもできます。その場合は、各アプリケーションディレクトリ内の「`management/commands/`」配下にモジュールを配置する必要がありますのでご注意ください。

3.3 よくあるプロジェクト構成

Django のチュートリアルでよく見かける構成のプロジェクトを作成してみます。たとえばプロジェクト名を「`mysite`」として、

```
$ django-admin startproject mysite
```

と実行すると、現在のディレクトリ直下にプロジェクト名と同じ「`mysite`」というディレクトリ（以降、起点となるこのディレクトリを「ベースディレクトリ」と呼ぶことにします）が作成されます。

ベースディレクトリ配下のファイル構成は、次のようにになります。

```
$ tree mysite
mysite           (< ベースディレクトリ)
|-- manage.py
`-- mysite        (< 設定ディレクトリ)
    |-- __init__.py
    |-- settings.py
    |-- urls.py
    `-- wsgi.py
```

ベースディレクトリの下にプロジェクト名と同名のディレクトリがもう一段作られ、その中に「`__init__.py`」「`settings.py`」「`urls.py`」「`wsgi.py`」が自動で作成されています。`settings.py` や `urls.py` といった特別な設定ファイルが置かれるため、このディレクトリを「設定ディレクトリ」と呼ぶことにします。一部の公式ドキュメント ^{*2} では「ベース

^{*2} <https://docs.djangoproject.com/ja/2.2/ref/django-admin/#startproject>

「ディレクトリ」を「プロジェクトディレクトリ」、「設定ディレクトリ」を「プロジェクトパッケージ」と呼んでいることがあるようですが、ややこしいので本書ではこのように呼び替えることにしました。またベースディレクトリ名と設定ディレクトリ名が同じになつていてこちらもややこしいのですが、これについては後述の「3.4 ベストプラクティス1：分かりやすいプロジェクト構成」で解決します。

次に、「accounts」という名前のアプリケーションを作成します。

```
$ cd mysite/
$ python3 manage.py startapp accounts
```

このコマンドを実行すると「accounts」というアプリケーション用のディレクトリ（以降「アプリケーションディレクトリ」）が作成され、その直下にいくつかのファイルが生成されます。コマンド実行後のファイル構成は次のようにになります。

```
$ tree
.
|-- accounts      (< アプリケーションディレクトリ)
|   |-- __init__.py
|   |-- admin.py
|   |-- apps.py
|   |-- migrations
|   |   `-- __init__.py
|   |-- models.py
|   |-- tests.py
|   `-- views.py
|-- manage.py
`-- mysite       (< 設定ディレクトリ)
    |-- __init__.py
    |-- settings.py
    |-- urls.py
    `-- wsgi.py
```

よくあるチュートリアルでは、ここでアプリケーションディレクトリごとにテンプレートファイルや静的ファイルを配置するためのディレクトリを追加していきます。たとえばこのような感じです。

```
$ tree
.
|-- accounts      (< アプリケーションディレクトリ)
|   |-- __init__.py
|   |-- admin.py
|   |-- apps.py
|   |-- migrations
|   |   `-- __init__.py
|   |-- models.py
|   |-- static      (< 静的ファイル用のディレクトリ)
|   |   `-- accounts
|   |       `-- images
|   |           `-- no-image.png
|   |-- templates   (< テンプレートファイル用のディレクトリ)
```

```

|   |   '-- accounts
|   |   '-- login.html
|   |-- tests.py
|   '-- views.py
|-- manage.py
`-- mysite           (< 設定ディレクトリ)
    |-- __init__.py
    |-- settings.py
    |-- urls.py
    '-- wsgi.py

```

アプリケーションが追加されるごとに、それぞれのアプリケーションディレクトリの下にテンプレートファイルや静的ファイルが配置される形になります。Django の設計思想として、アプリケーションを再利用しやすいうようにということでこのように凝集した構成になっています。しかしながら、テンプレートファイルや静的ファイルがアプリケーションごとにバラバラになってしまっているのは分かりづらかったり、デザイナーと分業する際に不便だったりするのではないかでしょうか。こちらの問題についてもベストプラクティスで解決します。

3.4 ベストプラクティス 1：分かりやすいプロジェクト構成

先に紹介したプロジェクト構成の例では次のような問題がありました。

- ベースディレクトリ名と設定ディレクトリ名が同じでややこしい
- テンプレートと静的ファイルがアプリケーションごとにバラバラに配置されてしまう

これらを解決したベストプラクティスを紹介します。

まずプロジェクト作成時に、ベースディレクトリと設定ディレクトリの名前を別々のものにするために、ベースディレクトリを任意の名前で作成した後にベースディレクトリの下に移動し、第一引数に任意の設定ディレクトリ名、第二引数に「.」を指定して startproject を実行します。

```

$ mkdir mysite
$ cd mysite/
$ django-admin startproject config .

```

これでベースディレクトリと設定ディレクトリの名前を変えることができました。

```

$ tree mysite
mysite           (< ベースディレクトリ)
|-- manage.py
`-- config       (< 設定ディレクトリ)
    |-- __init__.py

```

```
|-- settings.py  
|-- urls.py  
`-- wsgi.py
```

筆者はいつも設定ディレクトリ名を「config」という名前にしていますが^{*3}、この名前は「default」でも「root」でも何でも構いません。

次に、テンプレートと静的ファイルをベースディレクトリ直下にまとめて配置します。この配置をするためには「10.4 静的ファイル関連の設定」および「10.8.1 TEMPLATES（テンプレートに関する設定）」の設定をおこなう必要があります。

```
$ tree  
. . .  
|-- accounts          (<-- アプリケーションディレクトリ)  
|   |-- __init__.py  
|   |-- admin.py  
|   |-- apps.py  
|   |-- migrations  
|   |   `-- __init__.py  
|   |-- models.py  
|   |-- tests.py  
|   `-- views.py  
|-- config           (<-- 設定ディレクトリ)  
|   |-- __init__.py  
|   |-- settings.py  
|   |-- urls.py  
|   `-- wsgi.py  
|-- manage.py  
|-- static            (<-- 静的ファイル用のディレクトリ)  
|   |-- accounts  
|   |   `-- images  
|   |       `-- no-image.png  
|   |-- js  
|   |   `-- jquery-3.4.1.min.js  
|   |-- images  
|   |   `-- logo.png  
`-- templates         (<-- テンプレートファイル用のディレクトリ)  
    |-- accounts  
    |   `-- login.html  
    `-- base.html
```

これにより、テンプレートファイルをベースディレクトリ直下の「templates」で、静的ファイルを同じくベースディレクトリ直下の「static」で一元管理することができるようになります。

最後にプロジェクト内のモジュールおよびパッケージの簡単な説明を次の表3.1に示します。

^{*3} 設定ディレクトリ名の「config」は「Two Scoops of Django」(<https://www.twoscoopspress.com/products/two-scoops-of-django-1-11>)という書籍を参考にしています。

3.4 ベストプラクティス 1：分かりやすいプロジェクト構成

▼表 3.1 startproject で作成されるモジュール

モジュール	説明
manage.py	コマンドラインからさまざまな操作をおこなうための管理コマンドユーティリティ
<設定ディレクトリ>/settings.py	プロジェクト固有の設定を記述する設定ファイル
<設定ディレクトリ>/urls.py	URL パターンとビューのマッチング情報を記述するためのモジュール（「URLconf」と呼ばれる）
<設定ディレクトリ>/wsgi.py	WSGI（ウィズギー）インターフェースに対応したアプリケーションサーバから Django プロジェクトを起動するためのエントリーポイントとなるモジュール

▼表 3.2 startapp で作成されるモジュール・ディレクトリ

モジュール・ディレクトリ	説明
<アプリケーションディレクトリ>/admin.py	管理サイトに関する記述をするためのモジュール
<アプリケーションディレクトリ>/apps.py	アプリケーションを識別するための設定を記述
<アプリケーションディレクトリ>/migrations/	マイグレーションファイルが作成されるディレクトリ
<アプリケーションディレクトリ>/models.py	モデルの定義やビジネスロジックを記述
<アプリケーションディレクトリ>/tests.py	テストを記述するモジュール
<アプリケーションディレクトリ>/views.py	ビューを記述するモジュール

必要に応じて、フォームを記述する「forms.py」や、アプリケーションごとの URLconf を記述する「urls.py」、ミドルウェアを記述する「middlewares.py」を各アプリケーションディレクトリに作成することもあります。

なお、このベストプラクティスのプロジェクト構成は一例です。自分がしっくりくるものをぜひ見つけてみてください。さらに実用的なプロジェクト構成のアイデアについては、高機能な Django プロジェクトのひな形を作成してくれる「cookiecutter-django」^{*4} というツールで作成したサンプルプロジェクトが参考になるかと思います。以下のコマンドを実行後、どんなプロジェクトを作成するかについて 20 個ほどの質問に答えていくだけで（すべてデフォルトでも可）、それに合わせたプロジェクトのひな形を作成してくれます。

```
$ pip3 install cookiecutter
$ cookiecutter https://github.com/pydanny/cookiecutter-django
```

次に示したプロジェクト構成は、「cookiecutter-django」で作成したサンプルプロジェクトのものです（すべてデフォルトで回答。深さ 2 まで）。

^{*4} <https://github.com/pydanny/cookiecutter-django>

```
$ cd my_awesome_project/
$ tree -L 2
.
|-- CONTRIBUTORS.txt
|-- LICENSE
|-- README.rst
|-- config
|   |-- __init__.py
|   |-- settings
|   |-- urls.py
|   `-- wsgi.py
|-- docs
|   |-- Makefile
|   |-- __init__.py
|   |-- conf.py
|   |-- index.rst
|   `-- make.bat
|-- locale
|   `-- README.rst
|-- manage.py
|-- my_awesome_project
|   |-- __init__.py
|   |-- conftest.py
|   |-- contrib
|   |-- static
|   |-- templates
|   |-- users
|   `-- utils
|-- pytest.ini
|-- requirements
|   |-- base.txt
|   |-- local.txt
|   `-- production.txt
|-- setup.cfg
`-- utility
    |-- install_os_dependencies.sh
    |-- install_python_dependencies.sh
    |-- requirements-bionic.apt
    |-- requirements-jessie.apt
    |-- requirements-stretch.apt
    |-- requirements-trusty.apt
    `-- requirements-xenial.apt
```

3.5 まとめ

- プロジェクトは大きな箱
 - django-admin の「startproject」コマンドでひな形を作成する
- アプリケーションはまとめた機能を実現するための Python パッケージ
 - manage.py の「startapp」コマンドでひな形を作成する
- プロジェクト内のモジュールは役割ごとに名前が慣例的に決められている
 - プロジェクトやアプリケーション作成時に自動生成されるモジュールの名前はなるべく変更しないこと

第4章

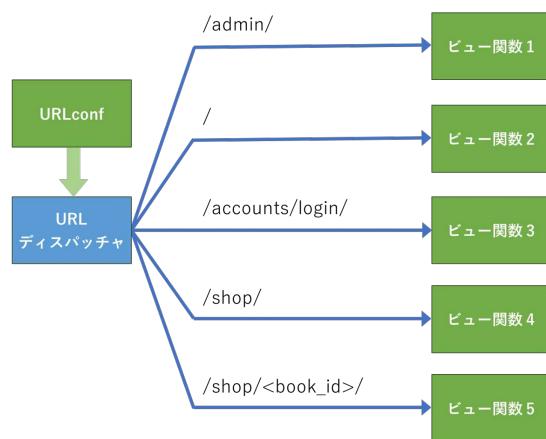
URL ディスパッチャと URLconf

4.1 概要

プロジェクト作成時に自動作成される urls.py は「URLconf」(URL 設定) と呼ばれ、URL のパターンとそれに紐付くビュー関数の設定をリスト形式で保持しています。

「URL ディスパッチャ」はその URLconf に登録された設定に基づいて、リクエストされた URL にマッチしたビュー関数を探し出して呼び出す役割を持っています。 *1

URL ディスパッチャの振り分け例を図にすると、次のようにになります (URL から「http(s)://<ホスト名>」の部分は省略)。図では、たとえば「/shop/」という URL にアクセスした場合は「ビュー関数 4」が、「/shop/123/」にアクセスした場合は「ビュー関数 5」が呼び出されることを示しています。



▲図 4.1 URL ディスパッチャの振り分け例

このように URL からそれに対応するビュー関数を見つけ出すことを「正引き」と呼ぶ

*1 「ディスパッチ」とは、処理を振り分けるという意味です。URL ディスパッチャの実体は「django.core.handlers.wsgi.WSGIHandler」を中心とした Django のコアモジュールです。

ことがあります。それに対して URL パターンに付けた名前から URL を取得することを「逆引き」と呼びますが、逆引きを利用するケースとしては、ビューでリダイレクト先の URL を取得したい場合やテンプレートで <a> タグのリンク先を取得したい場合が挙げられるでしょう。逆引きを利用することにより、ビューやテンプレートに URL をハードコーディングしなくて済むというメリットが得られます。

4.2 URLconf の書き方

プロジェクト作成時に設定ディレクトリ配下に urls.py が生成されますが、このモジュールの「urlpatterns」という名前のリスト型の変数に、URL パターンとビュー関数のマッピングを追加していきます。ちなみになぜこのモジュールが URL ディスパッチャに読み込まれるかというと、リスト 4.1 に示すように設定ファイルの「ROOT_URLCONF」に URLconf のパスが設定されているからです。

▼リスト 4.1 config/settings.py

```
ROOT_URLCONF = 'config.urls'
```

Django 2.0 以降で URL パターンの書き方が若干変わり、「パスコンバータ」という記法を利用するようになりました。なお、パスコンバータ記法以外にも Django 1.11 以前で利用されていた「正規表現」を使った方法でも書くことができますが、パスコンバータを使った方が簡単に URL パターンを定義することができます。

4.2.1 パスコンバータを使った URL パターンの書き方

Django 2.0 以降で django.urls.path 関数が追加され、「パスコンバータ」という記法を使って URL パターンをより簡単に記述することができるようになりました。Django が提供するコンバータには表 4.1 に示すように「int」「str」「path」「slug」「uuid」の 5 種類があり、自作することも可能です。

▼表 4.1 コンバータの例

コンバータ	意味	等価な正規表現の例
int	0 または正の整数にマッチ	[0-9]+
str	空でない文字列にマッチ（ただし「/」を除く）	[^/]+
path	空でない文字列にマッチ（ただし「/」を含む）	.+
slug	半角英数字・ハイフン・アンダースコアのみで構成された文字列にマッチ	[a-zA-Z0-9-_]+
uuid	UUID 型（8 桁-4 桁-4 桁-4 桁-12 桁の小文字の 16 進数値）にマッチ	[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12}

urls.py は次のように書くことができます。

▼リスト 4.2 config/urls.py

```
from django.contrib import admin
from django.urls import path

from . import views
from accounts import views as accounts_views
from shop import views as shop_views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', views.index, name='index'),
    path('accounts/login/', accounts_views.login, name='accounts_login'),
    path('shop/', shop_views.index, name='shop_index'),
    path('shop/<int:book_id>/', shop_views.detail, name='shop_detail'),
]
```

`path()` の第一引数には「パスコンバータ」を使った URL パターンを指定し、第二引数には URL パターンにマッチするビュー関数を指定します。URL からビュー関数を正引きする場合は、`urlpatterns` に追加された URL のパターンのリストから一番先に見つかったビュー関数を呼び出します（先勝ち）。

リスト 4.2 の例では、「`'shop/'`」のパターンには「`shop/`」という URL のみがマッチし、「`'shop/<int:book_id>/'`」のパターンには「`shop/1/`」や「`shop/123/`」のように「`/`」の間に 1 文字以上の数字が含まれる URL がマッチします。この数字は、「`book_id`」という変数名でキャプチャされ、URL ディスパッチャがビュー関数を呼び出す際に引数として追加してくれるのです。そこで対応するビュー側には、リスト 4.3 のような形で引数を追加しておくようにします（変数名を URL パターンのものと合わせる必要がありますのでご注意を）。ビューの書き方については次章で詳しく説明しますので、今のところは雰囲気だけを掴んでおいてください。

▼リスト 4.3 `book_id` を引数として受け取る例 (shop/views.py)

```
class BookDetailView(View):
    def get(self, request, book_id, *args, **kwargs):
        ...
```

4.2.2 正規表現を使った URL パターンの書き方

正規表現の URL パターンを使ってビュー関数をマッチングするには、`django.urls.re_path` 関数を利用します。^{*2} たとえば次のリスト 4.4 のように書くことができます。

^{*2} Django 1.11 以前と同じように `django.conf.urls.url()` を使うこともできますが、これは互換性のために残されているもので利用は推奨されません。`django.urls.re_path()` を使いましょう。

▼リスト 4.4 config/urls.py

```

from django.contrib import admin
from django.urls import re_path

from . import views
from accounts import views as accounts_views
from shop import views as shop_views

urlpatterns = [
    re_path(r'^admin/', admin.site.urls),
    re_path(r'^$', views.index, name='index'),
    re_path(r'^accounts/login/$', accounts_views.login, name='accounts_login'),
    re_path(r'^shop/$', shop_views.index, name='shop_index'),
    re_path(r'^shop/^(?P<book_id>\d+)/$', shop_views.detail, name='shop_detail'),
]

```

`re_path()` の第一引数は URL の正規表現で、URL のパターンを柔軟に指定できるようになっています。正規表現には、Python 組み込みの `re` モジュールの正規表現シンタックス^{*3} を利用することができ、次に示す「特殊文字」と「特殊シーケンス」、それ以外の通常文字を組み合わせてパターンを表記します。なお、それぞれの正規表現パターン文字列の前に付けられた「`r`」は「raw 文字列」であることを示すプレフィックスで、特殊文字の「\ (バックスラッシュ)」自体を表す際に「\\」と重ねて書かなくても済むようにするためのおまじないです。

特殊文字 (^ \$. [] | * + ? { } () \)

いくつかの記号は「特殊文字」と呼ばれ、表 4.2 に示すような特別な解釈がされます。

▼表 4.2 正規表現の特殊文字の例（一部）

特殊文字	意味
^	文字列の先頭を表し、前方一致のパターンを構成する。「^abc」など
\$	文字列の末尾を表し、後方一致のパターンを構成する。「abc\$」など
.	改行以外の任意の文字を表す
[]	文字集合。「-」を使用して指定した範囲に含まれる任意の 1 文字を表すこともできる。「[abc]」「[a-Z]」「[0-9]」など
*	直前にあるパターンを 0 回以上できるだけ多く繰り返すための量指定子
+	直前にあるパターンを 1 回以上繰り返すための量指定子
?	直前にあるパターンを 0 回か 1 回繰り返すための量指定子
{ }	直前にあるパターンの繰り返し回数を指定する。「[0-9]{4}」など
()	グループを表す。マッチした場合にキャプチャ（回収）されてあとで参照できるようになる
\	特殊文字の直前に置くとエスケープとして働き、特定の文字の直前に置くと特殊シーケンスを構成する

URLconf 内では、特殊文字を用いた次のような URL パターンがよく使われます。

1. ^パターン\$

^{*3} <https://docs.python.org/ja/3.7/library/re.html#regular-expression-syntax>

2. [^/]
 3. (?P<グループ名>パターン)
1. のように先頭に「^」、末尾に「\$」を置いた場合は、パターンとの完全一致を表します。したがって「^\$」は、空文字との完全一致を表すということになります。2. は、「/ (スラッシュ)」以外の任意の 1 文字を表します。「[]」内の最初に使われた「^」は前方一致ではなく「補集合」として扱われるのです。3. では、「()」で囲まれたパターンにマッチした文字がキャプチャ（回収）されてあとで参照できるようになりますが、「(?P<グループ名>パターン)」という表記をすることでマッチした文字列に正規表現グループ名をつけて参照できるようになります。

なお、特殊文字の直前に「\ (バックスラッシュ)」を置くことで特殊文字としての解釈をエスケープ（迂回）することができ、そのままの文字として認識させることができます。

特殊シーケンス (\d \s \w など)

「\」と特定の文字から構成されたパターンは「特殊シーケンス」と呼ばれ、表 4.3 に示すような特別な解釈がされます。

▼表 4.3 正規表現の特殊シーケンスの例（一部）

特殊シーケンス	意味
\d	任意の数字。「[0-9]」と同じ
\w	任意の英数字およびアンダーバー。「[a-zA-Z0-9_]」と同じ
\s	空白文字。「[\t\n\r\f\v]」と同じ

以上を踏まえてリスト 4.4 を見返すと、「r'^shop/\$」のパターンには「shop/」という URL のみがマッチし、「r'^shop/(?P<book_id>\d+)/\$」のパターンには「shop/1/」や「shop/123/」のように「/」の間に 1 文字以上の数字が含まれる URL がマッチすることが理解できるでしょう。またこの数字は、「book_id」という名前の正規表現グループとしてキャプチャされ、URL ディスパッチャがビュー関数を呼び出す際に引数として追加してくれます。

`re_path()` の第二引数には、第一引数の URL パターンにマッチする、`get` メソッドや `post` メソッドを持ったビュー関数を指定します。

4.2.3 その他の注意点

ここで、URLconf の URL パターンの書き方についての注意点を 3 つほど挙げておきます。

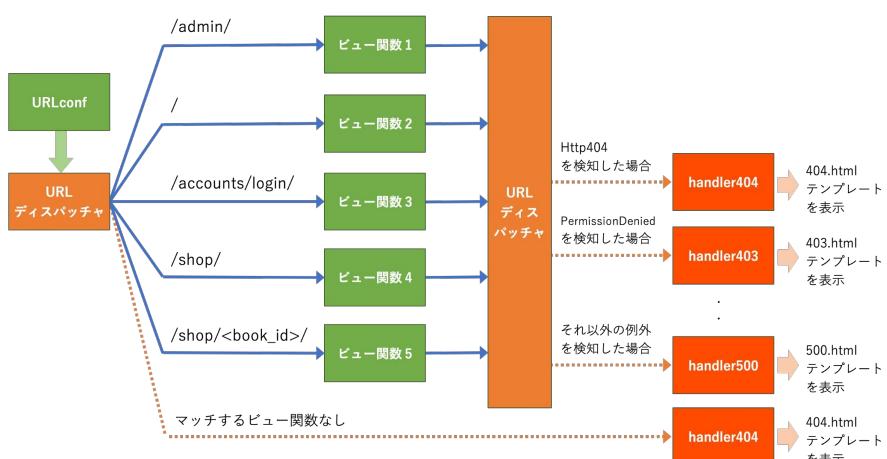
一点目は「/ (スラッシュ)」について。URL パターンの左端には「/」を付けない、右端には付けるといった慣例があるようです（ただし空文字の場合を除く）。

二点目は「クエリ文字列」^{*4}について。たとえば「shop/?keyword=Book」や「shop/?keyword=Django&page=3」などはすべて、「shop/」のパターンにマッチします。URLconf の URL パターンにはクエリ文字列に関する記述は不要です。クエリ文字列で指定されたパラメータについては、ビュー側で取得します。

三点目は「name」について。path() や re_path() の引数として指定している name は、URL を逆引きするために URL のパターンに付ける任意の名前です。逆引きをするために、この name の値は同じ urls.py の中でユニークであることが望ましいです（もし重複して登録している場合には後に登録しているものが優先されます。つまり後勝ち）。URL を逆引きする際は、ビューでは django.urls.reverse 関数を、テンプレートでは url タグを利用します（それぞれ後の章で説明します）。

4.3 エラーハンドリング

URL ディスパッチャは、ビュー関数を呼び出すだけではなく、マッチするビュー関数が見つからなかった場合やビュー関数から伝播された例外を受け取った場合のエラーハンドリングもしてくれます。たとえば、アクセスされた URL がどのパターンにも合致しない場合は「handler404」というビュー（正確には「handler404」という変数に設定されたビュー関数）を介して「404.html」という名前のテンプレートを表示する仕組みになっています（「DEBUG」が False の場合）。同じように、ビュー関数内で「django.http.response.Http404」や「django.core.exceptions.PermissionDenied」などといった特別な例外をハンドリングしてくれるようになっています。



▲図 4.2 例外発生時のディスパッチ先

^{*4} 「shop/?keyword=Django&page=3」のように URL の「?」以降に「key=value」形式でパラメータを埋め込んだ文字列を「クエリ文字列」と呼びます。

4.4 ベストプラクティス 2：アプリケーションごとに urls.py を配置する

startproject 実行時に生成される urls.py のみに URL パターンの設定を追加していくと、設定がどんどん肥大化して管理が大変になってしまいます。そこで、urls.py を分割し、アプリケーションディレクトリごとに 1 つずつ urls.py を配置するというベストプラクティスを適用します。

まず、urls.py を分割するために、config/urls.py 内で django.conf.urls.include 関数を利用します。次の例を見てください。

▼リスト 4.5 config/urls.py

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

from . import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', views.index, name='index'),
    path('accounts/', include('accounts.urls')),
    path('shop/', include('shop.urls')),
]
```

include 関数を使ってアプリケーションごとの urls.py（この例では accounts/urls.py や shop/urls.py）を読み込んでいます。これにより、「/accounts/」で始まる URL パターンのすべてを「accounts」アプリケーションの accounts/urls.py に、「/shop/」で始まる URL パターンのすべてを「shop」アプリケーションの shop/urls.py に委譲しているのです。

そして各アプリケーション側の urls.py では、リスト 4.6 で示すようにアプリケーション内部の URL パターンの設定のみに集中できるようになるのです。

▼リスト 4.6 accounts/urls.py

```
from django.urls import path

from . import views

app_name = 'accounts'
urlpatterns = [
    path('login/', views.login, name='login'),
    path('logout/', views.logout, name='logout'),
    path('register/', views.register, name='register'),
]
```

ここで気を付けてほしいポイントが 2 つほどあります。

まず、各アプリケーション側の urls.py は、startapp コマンドでは作成されないので適宜作成するようにしてください。

次に、リスト 4.6 の例でも記述しているように、変数「`app_name`」は必ず設定するようにしてください（設定する値はアプリケーション名と同じにしておけばよいでしょう）。これはアプリケーションごとの urls.py に付けられる名前空間で、`reverse` メソッドや `url` タグから「<名前空間名>:<名前空間内の name>」という文字列で URL を逆引きできるようにするためのものです。

4.5 まとめ

- URL ディスパッチャは URL の仕分け係
- URLconf (URL 設定) は urls.py に書く
 - 変数「`urlpatterns`」に URL パターンとそれに紐付けるビュー関数を追加する
 - パスコンバータを使った URL パターンには `django.urls.path` 関数を使う
 - 正規表現を使った URL パターンには `django.urls.re_path` 関数を使う
- URLconf はアプリケーションごとに分割して配置する
- URL を逆引きするために「`name`」や「`app_name`」は必ず設定しておく

第 5 章

ビュー (View)

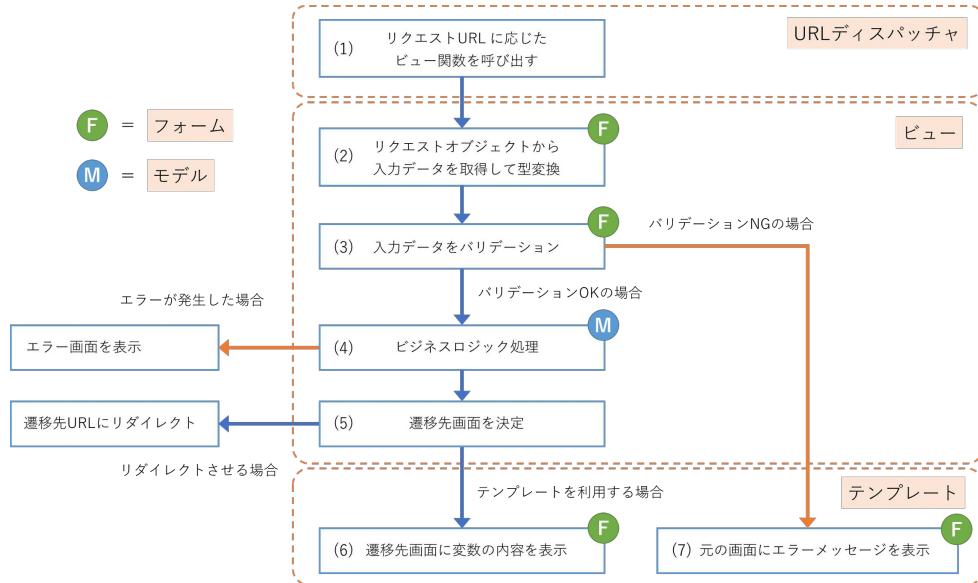
5.1 概要

ビューの役割を大雑把にいうと、リクエストオブジェクトを受け取ってレスポンスオブジェクトを返すことです。ビューでは状況に応じて、

1. HTML ページを表示するためのコンテンツを保持したレスポンス
2. リダイレクトするためのレスポンス
3. エラーを通知するためのレスポンス

などさまざまな種類のものを作成して返します。特に 1. の「HTML ページを表示するためのコンテンツを保持したレスポンス」を生成する場合には、第 7 章で解説するテンプレートシステムが利用されます。

さて、Django プロジェクトにおけるサーバ側処理のよくあるパターンは、概ねこのような感じになるのではないかでしょうか。



▲図 5.1 Django プロジェクトにおけるよくあるサーバ側処理のフロー

正常パターンでは、

- (1) URL ディスパッチャからビュー関数が呼び出される
- (2) ビュー内で、リクエストオブジェクトから入力データを取得して型変換
- (3) 入力データのバリデーション（妥当性チェック）をおこなう
- (4) バリデーション OK の場合は、ビジネスロジック処理を続行し、画面に表示するための変数を用意
- (5) 遷移先画面（表示するテンプレートやリダイレクト先）を決定
- (6) 遷移先画面に変数の内容をレンダリング

という流れになります。レンダリングした HTML コンテンツは最終的に、レスポンスオブジェクトの内部に含められてビューから送り出されることになります。

そのほか、

- バリデーション NG の場合に元の画面にエラーメッセージを表示する
- エラーを通知するためのレスポンスオブジェクトを返す
- 遷移先 URL にリダイレクトする

というパターンもありますが、結局はビューがそれぞれに応じたレスポンスオブジェクトを作成して返すのは正常パターンと同じです。`django.shortcuts` パッケージにはいろいろなタイプのレスポンスを簡単に作成するための関数が揃っているので積極的に利用することをお勧めします。なおこれらのパターン以外にビューから例外が漏れてしまう場合

には、URL ディスパッチャが例外をハンドリングすることになります。

5.2 ビュー関数の書き方（関数ベース vs クラスベース）

ビュー関数とは、次のような特徴を備えた関数です。

- 第一引数に `django.http.request.HttpRequest` オブジェクトを受け取る
- 戻り値として `django.http.response.HttpResponse` オブジェクトを返す

ビュー関数の書き方には大きく分けて、

- 関数で書く方法（関数ベース）
- クラスで書いて（クラスベース）、ビュー関数に変換する方法

の 2 通りがあります。同じ内容の処理を、関数で書いた場合とクラスで書いた場合とでそれぞれ挙げてみます。

まずは関数ベースで書いたビューの例です。

▼リスト 5.1 関数ベースのビューの例

```
from django.shortcuts import render

def hello(request):
    if request.method == 'GET':
        context = {
            'message': "Hello World!",
        }
    return render(request, 'hello.html', context)
```

ここで `django.shortcuts.render()` はテンプレートシステムを使って `HttpResponse` オブジェクトを作成してくれる便利メソッドです。作成したオブジェクトの内部には、第二引数に指定したテンプレートと第三引数に `dict` 型で指定したコンテキスト（変数と値のマッピング）を使ってレンダリングした HTML コンテンツを保持しています。このメソッドは頻繁に利用するので要チェックです。

次はクラスベースで書いたビューの例です。

▼リスト 5.2 クラスベースのビューの例

```
from django.shortcuts import render
from django.views import View

class HelloView(View):
    def get(self, request, *args, **kwargs):
        context = {
            'message': "Hello World!",
```

```
    }
    return render(request, 'hello.html', context)

hello = HelloView.as_view()
```

ここで最終行の `as_view()` は、クラスベースのビューをビュー関数化するためのメソッドです。これにより、URL ディスパッチャからだけでなく他のビューから呼び出すこともできるようになります。こちらも要チェックです。

一般に、関数ベースのビューは理解しやすいが再利用しにくいのが難点で、クラスベースのビューはクラスを再利用することで短く書けるケースがある一方でコードの見通しが悪くなってしまうという欠点があります。

筆者は断然、クラスベースのビューをお勧めします。それは、

- 汎用ビュー (Generic View) としてさまざまな用途のクラスが提供されているのでその恩恵を受けられる
- すべての基本となる基本汎用ビュー 「`django.views.generic.base.View`」 を利用すれば見通しのよいビューが書けるし、応用も効く
- 汎用ビューのほかにも便利な Mixin クラスを多重継承してお決まりの処理を再利用できる

という理由からです。本書では関数ベースのビューについては扱わず、クラスベースのビューについてのみ説明をおこなうこととします。

次節以降で、基本汎用ビューと呼ばれる

- `django.views.generic.base.View`
- `django.views.generic.base.TemplateView`
- `django.views.generic.base.RedirectView`

と、その他の汎用ビューについて詳しく説明します。

5.3 すべての基本となる基本汎用ビュー

Django ではさまざまな用途に応じた汎用ビュークラスが提供されていますが、中でも一番基本となるクラスは「`django.views.generic.base.View`」です。コードの見通しが良く、どのような用途で使うにしても応用が効きます。ですので、まずは `django.views.generic.base.View` を継承したクラスベースのビューに慣れることをお勧めします。

`django.views.generic.base.View` を継承したクラスベースのビューは、たとえば次のように書きます。

▼リスト 5.3 `accounts/views.py`

```
from django.contrib.auth import login as auth_login
from django.shortcuts import render, redirect
from django.urls import reverse
from django.views import View

from .forms import LoginForm


class LoginView(View):
    def get(self, request, *args, **kwargs):
        """GET リクエスト用のメソッド"""
        context = {
            'form': LoginForm(),
        }
        # ログイン画面用のテンプレートに値が空のフォームをレンダリング
        return render(request, 'accounts/login.html', context)

    def post(self, request, *args, **kwargs):
        """POST リクエスト用のメソッド"""
        # リクエストからフォームを作成
        form = LoginForm(request.POST)
        # バリデーション（ユーザーの認証も合わせて実施）
        if not form.is_valid():
            # バリデーション NG の場合はログイン画面のテンプレートを再表示
            return render(request, 'accounts/login.html', {'form': form})

        # User オブジェクトをフォームから取得
        user = form.get_user()

        # ログイン処理（取得した User オブジェクトをセッションに保存 & User データを更新）
        auth_login(request, user)

        # ショップ画面にリダイレクト
        return redirect(reverse('shop:index'))

login = LoginView.as_view()
```

なお、この基本汎用ビュークラスは次のいずれの方法でも `import` することができます。後者の方法は 1.10 から利用できるようになりました。

```
from django.views.generic import View
from django.views import View
```

クラスベースのビューを書く際のポイントとしては、次のものが挙げられます。

- GET リクエストには `get()`、POST リクエストには `post()` メソッドを用意する
- テンプレートをレンダリングする場合は `django.shortcuts.render()` を使う
- リダイレクトする場合は `django.shortcuts.redirect()` を使う
 - リダイレクト先の URL は `django.urls.reverse()` を使って取得し、ハードコーディングしないようにする

- URL ディスパッチャにエラーをハンドリングさせたい場合（たとえば 404 や 403 エラーページを表示したいとき）は、`django.http.response.Http404` や `django.core.exceptions.PermissionDenied` などの特別な例外を `raise` して伝播させる
- `as_view()` メソッドを使ってビュー関数化する

5.4 シンプルでよく使う基本汎用ビュー

`django.views.generic.base.View` 以外に基本汎用ビューと呼ばれるベーシックなビュークラスがあと 2つあります。この節では残りの

- `django.views.generic.base.TemplateView` (テンプレートの表示に特化)
- `django.views.generic.base.RedirectView` (リダイレクトに特化)

について説明します。

5.4.1 テンプレートの表示に特化した基本汎用ビュー (TemplateView)

「TemplateView」は `django.views.generic.base.View` をベースにして、テンプレートを表示することに特化した基本汎用ビュークラスです。トップ画面やヘルプ画面などの単純なテンプレートを表示するのによく利用されます。

TemplateView を使ったビューのコード例を次に示します。

▼リスト 5.4 TemplateView を継承したビューの例 (config/views.py)

```
from django.views.generic import TemplateView

class IndexView(TemplateView):
    template_name = 'index.html'

index = IndexView.as_view()
```

コードはたったこれだけです。省けるところを限りなく省いてしまうのが汎用ビューの特徴です。でもこれだと「任意の変数をテンプレートに表示することができないのでちょっと使えないな」と思われるかもしれません。ご心配なく。汎用ビューにはオーバーライドできる変数やメソッドがいくつか用意されていて、デフォルトの挙動を「ある程度」自由に変更することができるのです。たとえばこの `TemplateView` であれば、「`get_context_data`」メソッドをオーバーライドして、テンプレートに渡すコンテキストに任意の変数を追加することができます。

▼リスト 5.5 `get_context_data` メソッドをオーバーライドした例 (config/views.py)

```
from django.contrib.auth.models import User
from django.views.generic import TemplateView

class IndexView(TemplateView):
    template_name = 'index.html'

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        # テンプレートに渡すコンテキストに「user_count」という変数を追加
        context['user_count'] = User.objects.all().count()
        return context

index = IndexView.as_view()
```

TemplateView を使ったビューは非常に簡潔に書けるので、URLconf 内に直接書いてしまう場合もあります。

▼リスト 5.6 config/urls.py

```
from django.views.generic import TemplateView

urlpatterns = [
    ...
    path('', TemplateView.as_view(template_name='index.html'), name='index'),
    ...
]
```

5.4.2 リダイレクトに特化した基本汎用ビュー (RedirectView)

「RedirectView」は任意の URL にリダイレクトすることに特化した基本汎用ビュークラスです。

▼リスト 5.7 RedirectView を使ったビューの例 (config/views.py)

```
from django.views.generic import RedirectView

class IndexView(RedirectView):
    url = '/accounts/login/'

index = IndexView.as_view()
```

またしてもコードはこれだけです。この汎用ビューカラスでは、クラス変数「url」の代わりに URLconf に登録されたパターン名を指定できる変数「pattern_name」を使ったり、リダイレクトする URL を動的に組み立てるための「`get_redirect_url`」メソッドをオーバーライドしたりすることができます。

RedirectView も TemplateView と同様に URLconf 内に直接書いてしまう場合があります。

▼リスト 5.8 urls.py

```
from django.views.generic import RedirectView

urlpatterns = [
    ...
    path('', RedirectView.as_view(url='/accounts/login/'), name='index'),
    ...
]
```

5.5 さまざまな用途に特化した汎用ビュー

先に紹介した基本汎用ビュークラス以外にも、さまざまな用途の汎用ビュークラスが提供されています。その中から代表的なものを次に挙げます。

▼表 5.1 代表的な汎用ビュークラス

汎用ビュークラス	説明
django.views.generic.list.ListView	モデルオブジェクトの一覧を表示するための View
django.views.generic.detail.DetailView	モデルオブジェクトの詳細を表示するための View
django.views.generic.edit.FormView	フォームを利用するための View
django.views.generic.edit.CreateView	モデルオブジェクトを登録するための View
django.views.generic.edit.UpdateView	モデルオブジェクトを更新するための View
django.views.generic.edit.DeleteView	モデルオブジェクトを削除するための View

一例として、ListView を継承したビュークラスのコード例を示します。

▼リスト 5.9 ListView を使ったビューの例 (shop/views.py)

```
from django.views.generic import ListView

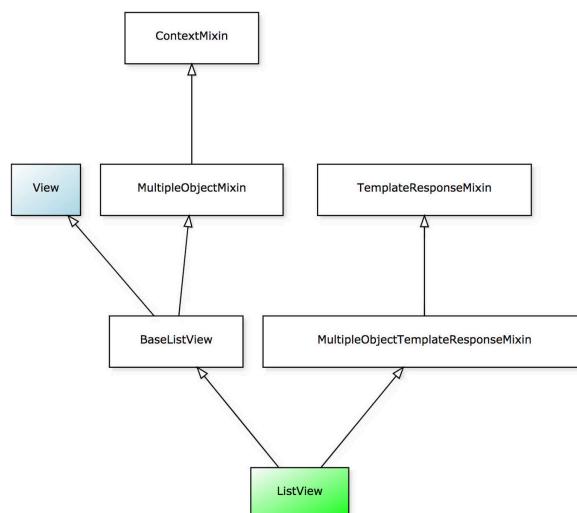
class BookListView(ListView):
    model = Book
```

これだけで、「shop/book_list.html」という名前のテンプレートに「object_list」という変数名で（オマケで「book_list」という変数名でも）Book モデルのすべてのレコードの一覧を渡してくれるのです。しかしながらこの ListView のデフォルトの挙動を知らないと、このビューが何をしてくれるのかパッと見ても全然分からぬですよね。挙動を変えたいときにオーバーライドできる変数やメソッドについても、主要なものだけでもこんなにたくさんあります。

- template_name (利用するテンプレート)
- model (取り扱うモデルクラス)

- queryset (オブジェクトの一覧を取得するためのクエリセット)
- context_object_name (オブジェクトの一覧を格納するための変数名)
- paginate_by (ページングする場合の 1 ページあたりの最大件数)
- get_template_names (テンプレート名を動的に変更する)
- get_queryset (取得するオブジェクトの一覧を動的に変更する)
- get_context_data (テンプレートに渡すコンテキストを動的に変更する)

ややこしいですよね。しかもそれぞれの汎用ビューカラスでオーバーライドできる変数やメソッドがそれぞれ異なるので厄介です。現状の公式ドキュメントでもどんな変数やメソッドがオーバーライドできるかについて整理されていなかったり日本語訳されていない部分があったりして、あまり使い物になりません。ここで筆者がお薦めしたいのは、「Classy Class-Based Views (CCBV)」^{*1} というサイトです。Django のバージョンや汎用ビューカラスごとにどんな変数やメソッドが使われているかが見やすく整理されていて、非常に有用です。それだけでなく、ページの「Hierarchy diagram」ボタンを押せば、図 5.2 で示したような継承関係を示した図を表示することもできます。



▲図 5.2 ListView の継承関係を示した図 (Classy Class-Based Views (CCBV) より引用)

これらの汎用ビューは想定される利用シーンに合致している場合にはシンプルに使って非常に有用ですが、複雑なことをしたい場合や、そもそもやりたいことと汎用ビューの種類が合っていないのが原因でオーバーライドを乱用してしまうと、クラスの可読性が悪くなってしまいます。まずは基本汎用ビューの `django.views.generic.base.View` ベースで書いておき、あとでリファクタリングしていくのがよいでしょう。

^{*1} <https://ccbv.co.uk/>

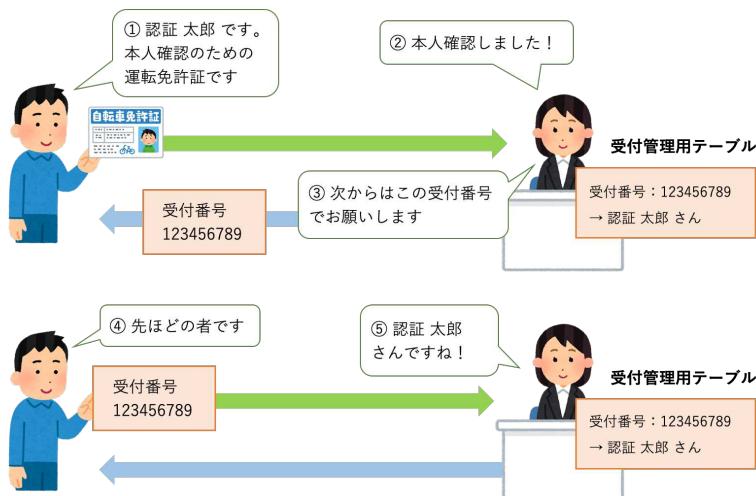
5.6 ログイン・ログアウトについて

Web アプリケーションにおける「ログイン済み」の状態とは一般に、

1. ユーザー固有の情報を照合することでユーザーが特定されており、
2. リクエストごとにユーザーの特定作業をやり直さなくともユーザーの情報を引き継げる

状態であると説明できます。1. は「ユーザー認証」と呼ばれ、パスワード方式をはじめとするさまざまな方法でユーザーの本人確認がおこなわれます。2. については多くの場合、ブラウザ側の Cookie とサーバ側のセッションを利用してリクエストをまたいだデータの引き継ぎを実現します。これは一般に「ユーザーセッション管理」と呼ばれます。

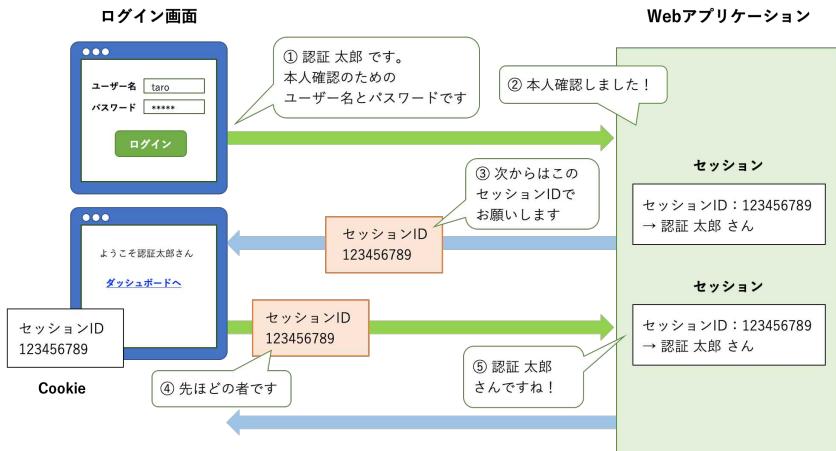
このような Web アプリケーションのログインの仕組みはよく、銀行や役所の受付窓口でのやり取りに例えられます。次の図を見てください。



▲図 5.3 ログインの仕組みは銀行や役所での受付のフローに例えられる

上図のように、受付窓口では、はじめに運転免許証などを使って本人確認をおこないます。いったん本人確認の手続きを済ませてしまえば、あとは本人を特定するための「受付番号」でやり取りすれば、面倒な本人確認を何度もやり直す必要はありません。

多くの Web アプリケーションで利用されている「セッション認証」あるいは「Cookie 認証」と呼ばれる方式のログインでは、次のような処理がおこなわれます。



▲図 5.4 セッション認証（Cookie 認証）を利用したログインの仕組み

①で送信されたユーザー名・パスワードと、事前に登録されたユーザーテーブルのユーザー名・パスワードを突合することで本人確認をおこないます（②）。本人確認が済んだら、ログイン情報を「セッション」というサーバ側の保存領域に格納し、キーとなる「セッション ID」と呼ばれるランダムな文字列を発行します。この「セッション ID」はログイン時のレスポンス（③）でブラウザに返され、ブラウザ側の「Cookie」という保存領域に格納されます。ブラウザの仕様により、Cookie に保存されたセッション ID はリクエストのたびに送信される仕組みになっているため、④以降では、逐一本人確認をしなくともログイン済みユーザーの一連の操作として扱われるようになります。

ログインの仕組みを理解したところで、Django におけるログイン・ログアウトが実際にどうなっているのかを見てみましょう。

5.6.1 ログイン処理について

ユーザー認証はさまざまな方法で実現できますが、たとえばユーザー名とパスワードを使った認証であれば「`django.contrib.auth.authenticate()`」を利用することができます。

一方、ユーザー SESSION の管理をするには「`django.contrib.auth.login()`」が使えます。このメソッドを実行すると、

- ランダムに生成された「セッション ID」をキーにしてログイン情報（ユーザー IDなどの情報）がセッション（デフォルト設定ではデータベース）に保存される
- シグナル^{*2}が実行されて User オブジェクトに対応するレコードの最終更新日時が更新される

*2 特定のアクションに対するイベントです。本書では詳しく解説しません。

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

という処理がおこなわれ、その後に「SessionMiddleware」(第9章「ミドルウェア(Middleware)」にて説明)によってCookieの「sessionid」キーにセッションIDが保存されます。

ログイン後は、「AuthenticationMiddleware」(同じく第9章にて説明)により、リクエストのたびにCookieからセッションIDを取り出し、それを元にしてセッションからユーザーの情報を取り出し、そしてそれを元にしてUserオブジェクトを取得して、リクエストオブジェクトのuser属性にセットしてくれます。ちなみにログインしていないければ、Userオブジェクトの代わりに「AnonymousUser」という特別なクラスのオブジェクトがセットされます。ところでユーザーがログイン済みかどうかを判定するには「is_authenticated」という属性を使います。これは、Userオブジェクトのis_authenticated属性が常にTrueになるのに対し、AnonymousUserオブジェクトのis_authenticatedが常にFalseになることを利用したものです。

以上の処理を経て、ログイン後はビューやテンプレートから「request.user」にアクセスすることで、ログイン済みのUserオブジェクトを参照することができるようになります。

5.6.2 ログアウト処理について

ログアウト処理の実体は「django.contrib.auth.logout()」です。ログアウトすると、サーバ側のセッションがクリアされ、リクエストオブジェクトのuser属性にAnonymousUserオブジェクトがセットされます。

5.6.3 LoginView / LogoutViewについて

Djangoでは、ログインおよびログアウトにそれぞれ特化した

- django.contrib.auth.views.LoginView
- django.contrib.auth.views.LogoutView

というビュークラスも提供されています。これらを利用すると大幅にログインおよびログアウトの実装負荷を減らすことができます。

たとえばログインのビューは次のように実装します。

▼リスト 5.10 LoginViewを使ったビューの例 (accounts/views.py)

```
from django.contrib.auth.views import LoginView as AuthLoginView

class LoginView(AuthLoginView):
    template_name = 'accounts/login.html'

login = LoginView.as_view()
```

なおこれらを利用する際は、次の設定値の調整が必要になる場合もあるので注意しましょう。

▼表 5.2 LoginView および LogoutView で利用される変数一覧

変数名	説明	デフォルト値
LOGIN_URL	ログインページの URL	/accounts/login/
LOGIN_REDIRECT_URL	ログイン後のリダイレクト先	/accounts/profile/
LOGOUT_REDIRECT_URL	ログアウト後のリダイレクト先	None

5.6.4 LoginRequiredMixin について

「django.contrib.auth.mixins.LoginRequiredMixin」は、未ログインのユーザーがアクセスしようとしたときに何らかのペナルティを課すための Mixin です。Django 1.9 から導入されました。クラスベースのビューであれば、これを継承するだけで未ログインのユーザーがアクセスした場合に「LOGIN_URL」で定義した URL にリダイレクトしてくれるようになります。「raise_exception = True」の記述をすれば 403 エラーの画面を表示することも可能です。

次のコードは、LoginRequiredMixin を利用したビューの実装例です。

▼リスト 5.11 LoginRequiredMixin を利用したビューの例 (shop/views.py)

```
from django.contrib.auth.mixins import LoginRequiredMixin
from django.views.generic import ListView

class BookListView(LoginRequiredMixin, ListView):
    model = Book
    # 403 エラー画面を表示する場合は次のコメントアウトを外す
    # raise_exception = True
```

5.7 まとめ

- ビューは views.py に書く
- レスポンスオブジェクトを作成して返すのがビューの基本
 - テンプレートを使う場合は django.shortcuts.render() を使うと便利
 - リダイレクトする場合は django.shortcuts.redirect() を使うと便利
- 関数ベースではなく、クラスベースのビューを使おう
 - まずはすべての基本となる django.views.generic.base.View を継承したビューの書き方に慣れよう
- ビューには複雑なロジックは書かず、見通しをよくしよう

第6章

モデル (Model)

6.1 概要

Django は「Django ORM」と呼ばれる OR マッパー (Object-Relational Mapper) の機能を提供しています。OR マッパーとは、データベースのテーブルとカラムの定義と、「モデル」と呼ばれるクラスとそのクラス属性の定義を対応させ、データベースのレコード 1 件 1 件をモデルクラスのオブジェクトとして扱えるようにする仕組みです。これにより INSERT / UPDATE / DELETE などの SQL 文を一切使わずにレコードを操作することができるようになります。

OR マッパーが内部で発行されるクエリを最適化してくれたり、データベースの種類 (SQLite、MySQL、PostgreSQL など) による違いを可能な限り吸収してくれたりするので、Django ではなるべく生の SQL 文を使わないようにするのがポイントになっています。

ちなみに Django には、CREATE 文や ALTER 文を書かずにモデルの定義からテーブルを作成・変更するための「マイグレーション」という機能が備わっています（マイグレーションについては「第 11 章」で説明します）。そのため、Django を使った開発では、「テーブル作成」→「モデル作成」とするのではなく、「モデル作成」→「テーブル作成」を繰り返して進めていくのが基本になります。

なお本書で使用している「クエリ」という用語は、データベースにアクセスするときに発行される「SQL 文」を指すものとします。

6.2 モデルクラスの書き方

モデルクラスは、アプリケーションディレクトリ内の「models.py」に記述します。また、「`django.db.models.Model`」クラスを継承する必要があります。通常、1 つのモデルクラスはデータベースの 1 つのテーブルに対応します（抽象クラスを継承する場合はその限りではありません）。

サンプルとして、「本」テーブルに対応する Book クラスの実装例を次に示します。

▼リスト 6.1 models.py

```
from django.db import models

class Book(models.Model):
    """本モデル"""
    class Meta:
        # テーブル名を定義
        db_table = 'book'

    # テーブルのカラムに対応するフィールドを定義
    title = models.CharField(verbose_name='タイトル', max_length=255)
    price = models.IntegerField(verbose_name='価格', null=True, blank=True)

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

モデルクラス内部の Meta クラスの属性に、対応するテーブル名や、複数カラムに対するインデックスやユニーク制約などのモデルクラス全体に対する付加情報を記述します。ちなみに「`Meta.db_table`」でテーブル名を定義していない場合は、デフォルトのテーブル名「<アプリケーション名>_<モデルのクラス名をスネークケースにした文字列>」がマイグレーション時に付けられます。

テーブルのカラムは、モデルクラスのクラス属性として定義します。クラス属性には「`django.db.models.fields.Field`」のサブクラスを利用します。Field クラスは用途に応じてさまざまな種類のものが用意されていますが、それに対応するテーブルのカラムの型はデータベースの種類によって異なります。次に、SQLite（バージョン 3）および MySQL（バージョン 5.7）の場合の主な Field クラスとデータベースのカラムの型の対応表を示します。

▼表 6.1 Field クラスに対応するカラムの型（一部）

Field クラス	カラムの型 (SQLite)	カラムの型 (MySQL)
<code>django.db.models.fields.BooleanField</code>	<code>bool</code>	<code>tinyint(1)</code>
<code>django.db.models.fields.CharField</code>	<code>varchar</code>	<code>varchar</code>
<code>django.db.models.fields.TextField</code>	<code>text</code>	<code>longtext</code>
<code>django.db.models.fields.IntegerField</code>	<code>integer</code>	<code>int(11)</code>
<code>django.db.models.fields.FloatField</code>	<code>real</code>	<code>double</code>
<code>django.db.models.fields.DateField</code>	<code>date</code>	<code>date</code>
<code>django.db.models.fields.DateTimeField</code>	<code>datetime</code>	<code>datetime(6)</code>

Field クラスには「フィールドオプション」と呼ばれる属性を指定することができます。これにより、データベース上の制約や、第 8 章で説明する `ModelForm` を利用したときの画面上の表示名やバリデーションを定義することができます。

▼表 6.2 Field クラスに指定できるフィールドオプション（一部）

フィールドオプション	説明
verbose_name	フィールド名
null	データベースの NOT NULL 制約（デフォルト値は False：許可しない）
unique	データベースのユニーク制約
db_index	データベースのインデックスを設定するかどうか
default	レコード登録時に値が指定されなかったときのデフォルト値
blank	フォーム利用時に入力必須にするかどうか（デフォルト値は False）
max_length	文字列の最大文字数。CharField では指定必須
choices	フォーム利用時にセレクトボックスに表示する選択肢
validators	文字種チェックなどのバリデーションを指定
error_messages	バリデーション NG の場合のエラーメッセージ

モデルクラスを書くときのその他の注意点としては、次のようなものがあります。

- 主キー（id）は自動で生成されるため明示的に定義する必要なし
- 主キー（id）はレコードが登録されるたびに自動採番（1 からインクリメント）
- 複合キーを主キーにできない（「Meta.unique_together」で代替するなど）
- 他テーブルへの一対一、多対一、多対多のリレーションも定義可能（次節を参照）
- `__str__` メソッドで管理サイト（第13章参照）などに表示される文字列を定義

6.3 「一対一」「多対一」「多対多」リレーションはどう定義するか？

リレーションナル・データベースの世界では、「一対一」「多対一」「多対多」といった3つのパターンのリレーション（関連）が存在します。結論からいえば、Django のモデルではそれらを「`OneToOneField`」「`ForeignKey`」「`ManyToManyField`」でそれぞれ定義することができます。以降で詳しく見ていきます。

6.3.1 「一対一」のリレーション

ある種類のデータに対して他の種類のデータが必ずひとつ（あるいはゼロ）しか関連付かないという関係性が、「一対一」の関係です。

たとえば、

- 本と本の在庫情報
- システム利用者と（アカウント登録時に発行する）アクティベーションキー情報
- システム利用者とショッピングカート

などが挙げられるでしょうか。アプリケーションの設計においては、あるエンティティとそれに付随する情報のエンティティが一对一で関連しているというのがよく見られるパターンです。別のパターンとして、正規化すればひとつのテーブルで済むようなテーブルを敢えて分割しなければいけないケースもあります。User モデルに付け加える追加情報を別テーブルに持たせるというのがその代表的なパターンなのですが、これについては章末の「6.9 ベストプラクティス 3：User モデルを拡張する」で説明します。

一对一のリレーションは、「`django.db.models.fields.related.OneToOneField`」をどちらか一方のモデルに持たせることで実現できます。次のコードは、「本」モデルと「本の在庫を管理するエンティティ」モデルの実装例です。

▼リスト 6.2 「一对一」のリレーションの実装例（shop/models.py）

```
from django.db import models

class Book(models.Model):
    """本モデル"""
    title = models.CharField(verbose_name='タイトル', max_length=255)

class BookStock(models.Model):
    """本の在庫モデル"""
    book = models.OneToOneField(Book, verbose_name='本',
                               on_delete=models.CASCADE)
    quantity = models.IntegerField(verbose_name='在庫数', default=0)
```

6.3.2 「多対一」のリレーション

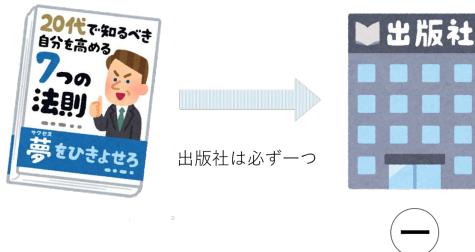
「一对一」の関係はやや特殊な状況と考えてもよいので、エンティティ間に関連がある場合はほとんどの場合が「多対一」か「多対多」になるでしょう。その場合にややこしいのが、それが「多対一」なのか「多対多」なのかという点ですが、一方から見たときに相手を「同時に 2 つ以上」紐付けることができるかどうかで相手が「多」かどうかが決まります。その観点で双方のエンティティの立場から確認してみると分かりやすいです。

例として「本」と「出版社」を考えてみましょう。ひとつの「出版社」から見たときに、複数の本を出版するということは当然あり得るので（すなわちひとつの出版社に複数の本が関連するので）、「本」は「多」になります。



▲図 6.1 出版社から見ると本は「多」

一方、一冊の本から見たときに、出版社はひとつしか関連付かないはずなので出版社は「一」となります。したがって、本と出版社の関係は「多対一」ということになります。



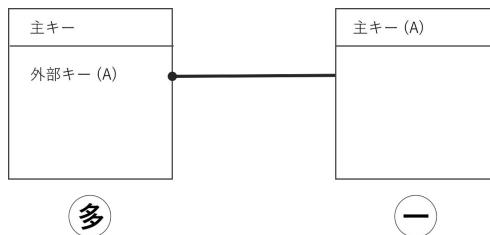
▲図 6.2 本から見ると出版社は「一」

- 「注文」(多) と 「システム利用者」(一)
- 「注文明細」(多) と 「注文」(一)
- 「従業員」(多) と 「部署」(一)
- 「ブログ記事」(多) と 「ブログの投稿者」(一)

なども同様に考えることができるでしょう。

一般に、「多対一」のリレーションでは、「一」側のテーブルの主キーを参照する外部キーを「多」側のテーブルに設けることで実現する場合が多いでしょう。

6.3 「一对一」「多対一」「多対多」リレーションはどう定義するか？



▲図 6.3 多側に外部キーを設ける

Django のモデルでは、「多」側のモデルに「`django.db.models.ForeignKey`」のフィールドを持たせることで実現できます。次のコードは、多対一で関連する「本」モデルと「出版社」モデルを定義した実装サンプルです。

▼リスト 6.3 「多対一」のリレーションの実装例（shop/models.py）

```
from django.db import models

class Publisher(models.Model):
    """出版社モデル"""
    name = models.CharField(verbose_name='出版社名', max_length=255)

class Book(models.Model):
    """本モデル"""
    ...
    publisher = models.ForeignKey(Publisher, verbose_name='出版社',
                                   on_delete=models.PROTECT)
```

リスト 6.3 の最終行で `ForeignKey` に「`on_delete`」というフィールドオプションを付けていますが、`ForeignKey` と `OneToOneField` には `on_delete` 属性を指定するクセを付けるようにしておきましょう。Django 2 系では `on_delete` 属性の設定が必須になっているからです。この設定により、参照先のオブジェクトが削除された際に自身のオブジェクトがどのような挙動をするかが変わってきます。

▼表 6.3 `on_delete` オプションの設定値

設定値	説明
<code>django.db.models.CASCADE</code>	自身のレコードも削除される
<code>django.db.models.PROTECT</code>	自身のレコードは削除されない
<code>django.db.models.SET_NULL</code>	このフィールドに NULL がセットされる。 ただし、null オプションが True の場合のみ
<code>django.db.models.SET_DEFAULT</code>	このフィールドにデフォルト値がセットされる。 ただし、default オプションが設定されている場合のみ

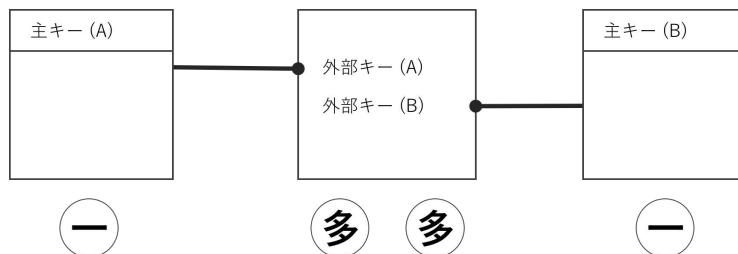
6.3.3 「多対多」のリレーション

次に、「本」と「著者」^{*1} を考えてみましょう。この場合には、「多対多」の関係になります。本側から見ても著者側から見ても、相手との関連が複数となり得るからです。



▲図 6.4 本から見ても著者から見ても相手は複数

実際のテーブル設計では、中間テーブルを作ることで多対多のリレーションを実現するのが一般的です。



▲図 6.5 複数の外部キーを使って中間テーブルを作る

Django のモデルでも同じように中間テーブルを作成して多対多の関連を実現しますが、実装上は一方のモデルクラスに「`django.db.models.fields.related.ManyToManyField`」を付けるだけで対応が可能です。後はマイグレーションを実行したときに Django が自動的に中間テーブルを作成してくれるのです。便利ですね。

*1 一人で一冊全部を書く単著だけでなく、複数人で共同して一冊の本を書く共著の場合も考慮します。

▼リスト 6.4 「多対多」のリレーションの実装例（shop/models.py）

```
from django.db import models

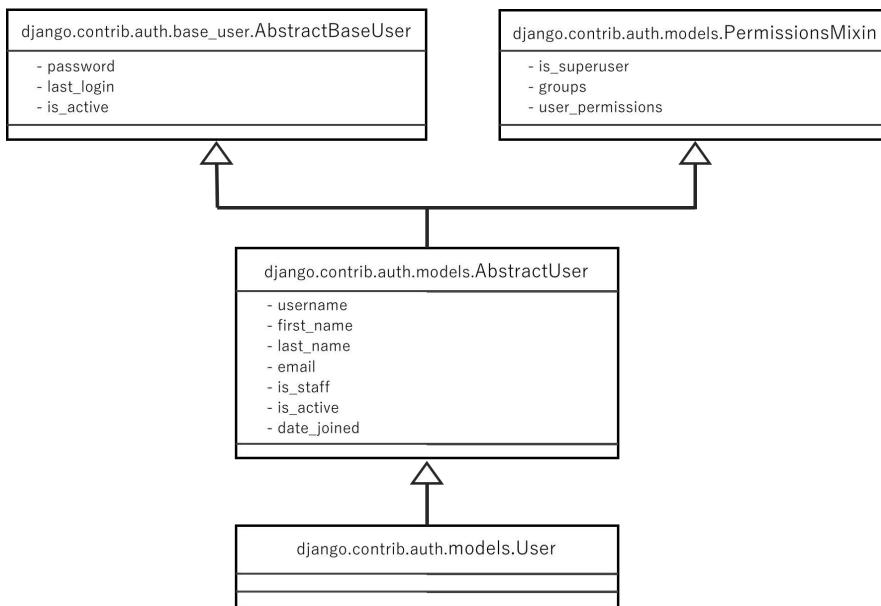
class Author(models.Model):
    """著者モデル"""
    name = models.CharField(verbose_name='著者名', max_length=255)

class Book(models.Model):
    """本モデル"""
    ...
    authors = models.ManyToManyField(Author, verbose_name='著者')
```

6.4 よく使われる User モデル

Django で最もよく使われるモデルが User モデルです。User モデルは Django がデフォルトで提供しているクラスで、システム利用者（ユーザー）として利用するのが一般的です。

User モデルのクラス構造は少し複雑で、次に示したような継承関係があります。



▲図 6.6 User モデルクラスの継承関係

フィールドだけを抜き出すと、次のようになっています（若干並べ替えてあります）。

▼リスト 6.5 User モデルのフィールド（属性など一部省略）

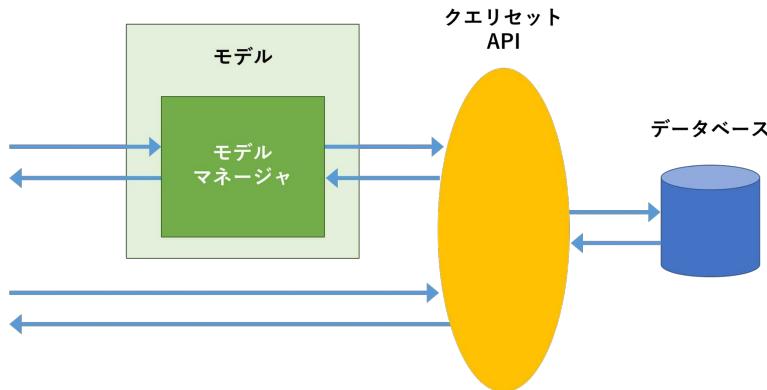
```
username = models.CharField(
    _('username'),
    max_length=150,
    unique=True,
    help_text=_('Required. 150 characters or fewer. Letters, digits and @./+/-/_ only.'),
    validators=[username_validator],
    error_messages={
        'unique': _("A user with that username already exists."),
    },
)
first_name = models.CharField(_('first name'), max_length=30, blank=True)
last_name = models.CharField(_('last name'), max_length=150, blank=True)
email = models.EmailField(_('email address'), blank=True)
password = models.CharField(_('password'), max_length=128)
is_staff = models.BooleanField(
    _('staff status'),
    default=False,
    help_text=_('Designates whether the user can log into this admin site.'),
)
is_superuser = models.BooleanField(
    _('superuser status'),
    default=False,
    help_text=_(
        'Designates that this user has all permissions without '
        'explicitly assigning them.'
    ),
)
is_active = models.BooleanField(
    _('active'),
    default=True,
    help_text=_(
        'Designates whether this user should be treated as active. '
        'Unselect this instead of deleting accounts.'
    ),
)
date_joined = models.DateTimeField(_('date joined'), default=timezone.now)
last_login = models.DateTimeField(_('last login'), blank=True, null=True)
```

細かい意味は分からなくても、「こんなフィールドがあるんだな」くらいが理解できれば問題ないでしょう。以降に User モデルを使ったコード例が何度も出てきますので、そのときのために心に留めておいてください。

6.5 モデルマネージャとクエリセット

ここから、データベースのレコードをオブジェクトとして取得する仕組みについて見ていきます。まずは、モデルマネージャとクエリセットについて説明します。

次の図を見てください。



▲図 6.7 モデルマネージャとクエリセット API

モデルは「モデルマネージャ」と呼ばれるデータベースのテーブルレベルのクエリ操作を提供するインターフェースを持っています。しかしながらモデルマネージャは、直接データベースとはクエリのやり取りはおこないません。データベースと直接クエリをやり取りするのは「クエリセット API」と呼ばれるものです。

テーブルを検索する際はまず、モデルのモデルマネージャを経由してクエリセット API のメソッドを呼び出します。するとクエリセット API は、データベースにアクセスしてモデルのオブジェクトやその他の結果を返す場合もあれば、データベースにはアクセスせずにクエリセットオブジェクトを返す場合もあります。クエリセットオブジェクトを受け取った場合は引き続きクエリセット API を利用することができる、という仕組みになっています。

なおモデルマネージャは通常、モデルクラスに「`objects`」という名前で保持されています。

少し抽象的な話になってしまったので、以降で具体的な説明をしていきます。なおモデルの例として、先述の User モデル、および次に示す出版社 (Publisher) モデル、著者 (Author) モデル、本 (Book) モデルを利用します。また、サンプルコード中の import 文は割愛します。

▼リスト 6.6 本章で利用するモデルクラスの例 (shop/models.py)

```

from django.db import models

class Publisher(models.Model):
    """出版社モデル"""
    class Meta:
        db_table = 'publisher'

    name = models.CharField(verbose_name='出版社名', max_length=255)

    def __str__(self):
        return self.name

```

```

class Author(models.Model):
    """著者モデル"""
    class Meta:
        db_table = 'author'

    name = models.CharField(verbose_name='著者名', max_length=255)

    def __str__(self):
        return self.name


class Book(models.Model):
    """本モデル"""
    class Meta:
        db_table = 'book'

    title = models.CharField(verbose_name='タイトル', max_length=255)
    publisher = models.ForeignKey(Publisher, verbose_name='出版社',
                                  on_delete=models.PROTECT)
    authors = models.ManyToManyField(Author, verbose_name='著者')
    price = models.IntegerField(verbose_name='価格', null=True, blank=True)
    description = models.TextField(verbose_name='概要', null=True, blank=True)
    publish_date = models.DateField(verbose_name='出版日')

    def __str__(self):
        return self.title

```

6.6 単体のオブジェクトを取得する

オブジェクトを1件だけ取得したい場合は、モデルマネージャを経由してクエリセットAPIの「`get`」メソッドを使います。メソッドのキーワード引数としてモデルクラスのフィールド名に任意の値を指定することで、条件検索することができます。

```
>>> User.objects.get(username='admin')
<User: admin>
```

また「`pk`」（「`id`」でもOK）というキーワード引数を使うと、モデルに自動的に追加された主キーで検索することができます。

```
>>> User.objects.get(pk=2)
<User: akiyoko>
```

`get()`はデータベースにアクセスしてモデルのオブジェクトを返します。なお1件も見つからない場合は、モデルクラスの `DoesNotExist`例外が発生するので注意が必要です。

```
>>> User.objects.get(username='wakuwaku-san')
Traceback (most recent call last):
... (略) ...
line 380, in get
```

```
self.model._meta.object_name
django.contrib.auth.models.DoesNotExist: User matching query does not exist.
```

6.7 複数のオブジェクトを取得する

複数のオブジェクトを取得するにはモデルマネージャを経由してクエリセット API の「`all`」あるいは「`filter`」メソッドを使います。順に説明します。

6.7.1 `all()` メソッド

レコード全件に対応するオブジェクトのリストを取得するには、クエリセット API の「`all`」メソッドを利用します。

```
>>> User.objects.all()
<QuerySet [<User: admin>, <User: akiyoko>]>
```

このメソッドは即座にデータベースにはアクセスせず^{*2}、クエリセットオブジェクトを返します。このクエリセットオブジェクトは、「しかるべき」タイミングでデータベースにアクセスされてオブジェクトのリストが取得できるようになります。これを「遅延評価」といいます（後述します）。

そして `all()` は、`get()` とは異なり結果が 0 件の場合でも例外は発生しません。結果として空のリストを取得することができます。

6.7.2 `filter()` メソッド

条件を利用しない検索をする際に「`all`」を利用するのに対し、検索条件を付けてオブジェクトのリストを取得したい場合は、クエリセット API の「`filter`」メソッドを利用します。メソッドのキーワード引数としてモデルクラスのフィールド名と任意の値を指定することで、条件検索することができます。

```
>>> User.objects.filter(is_active=True)
<QuerySet [<User: admin>, <User: akiyoko>]>
```

`filter()` も `all()` 同様、クエリセットオブジェクトが返されます。必要に迫られるまではクエリが発行されないという性質を利用して、`filter()` を何度も繋げて書くことができます。

^{*2} 厳密にはこのタイミングでクエリが発生してしまいます。REPL でオブジェクトを表示すると `__repr__()` が実行されるのですが、それが「しかるべき」タイミングのひとつになっているからです。

▼リスト 6.7 filter() メソッドを繋げる例 (shop/views.py)

```
from django.views.generic import View
from .models import Book

class IndexView(View):
    def get(self, request, *args, **kwargs):
        keyword = request.GET.get('keyword')

        queryset = Book.objects.filter()
        if keyword:
            queryset = queryset.filter(title=keyword)

        print(queryset)
```

リスト 6.7 の例で発行されるクエリの総数は 1 本です（最後の print で発行されます）。プログラムがしやすい仕組みになっているともいえますね。

6.7.3 AND 条件

検索条件を次のように列挙すれば、AND 条件になります。次の例は、タイトルが「Django Book」で且つ価格が 1,000 円の本を検索する場合の書き方です。

```
>>> Book.objects.filter(title='Django Book', price=1000)
<QuerySet [<Book: Django Book>]>
```

6.7.4 OR 条件

OR 条件はちょっとややこしいですが、「django.db.models.Q」と「| (パイプ)」を使います。次の例は、タイトルが「Django Book」または価格が 1,000 円という条件の本を検索する場合の書き方です。

```
>>> from django.db.models import Q
>>> Book.objects.filter(Q(title='Django Book') | Q(price=1000))
<QuerySet [<Book: Django Book>, <Book: Django Book 2>]>
```

6.7.5 不等号 (>,<,>=,<=) / IN 句 / LIKE 句を使った検索

不等号 (>,<,>=,<=) や IN 句、LIKE 句を使った検索をするには、「__」(アンダーバー 2 連続) でフィールド名と特別なキーワード (gt, lt, in など) を繋ぎます。

1,000 円より高い本を検索する場合 (>1000) は「`gt`」(Greater Than) を使います。

```
>>> Book.objects.filter(price__gt=1000)
```

1,000 円より安い本を検索する場合 (<1000) は「`lt`」(Less Than) を使います。

```
>>> Book.objects.filter(price__lt=1000)
```

1,000 円以上の本を検索する場合 (≥ 1000) は「`gte`」(Greater Than or Equal) を使います。

```
>>> Book.objects.filter(price__gte=1000)
```

1,000 円以下の本を検索する場合 (≤ 1000) は「`lte`」(Less Than or Equal) を使います。

```
>>> Book.objects.filter(price__lte=1000)
```

価格が 900 円または 1,000 円の本を検索する場合 (IN 句) は「`in`」を使います。

```
>>> Book.objects.filter(price__in=[900, 1000])
```

タイトルに「Django」を含む本を検索する場合 (LIKE 句) は「`icontains`」を使います。LIKE 検索には `icontains` と `contains` が使えますが、`contains` は大文字と小文字を区別する点が異なります。

```
>>> Book.objects.filter(title__icontains='Django')
```

6.7.6 リレーション先のモデルを使った条件検索

出版社の名前で本のレコードを抽出する場合など、リレーション先のモデルを使った条件検索はどうすればよいでしょうか？通常の SQL 文を書く場合は JOIN 句を使わなければいけませんが、Django のモデルで `OneToOneField`、`ForeignKey`、`ManyToManyField` を使ってリレーションの定義をしている場合は簡単です。検索条件のキーワード引数として「`__`」でフィールド名とリレーション先のモデルのフィールド名を繋ぐだけで、モデルの定義にしたがって自動的に JOIN をおこなってくれます。

```
>>> Book.objects.filter(publisher__name='自費出版社')
```

これは、JOIN 句を使って次のような SQL 文 を実行したのと同じです。

```
SELECT * FROM book b INNER JOIN publisher p ON b.publisher_id = p.id  
WHERE p.name = '自費出版社';
```

6.7.7 クエリセットオブジェクトの遅延評価のタイミング

先に、all() や filter() はクエリセットオブジェクトを返し、しかるべきタイミングでクエリが発行されてデータベースへのアクセスがおこなわれる書きました。これは「遅延評価」と呼ばれます。ここで「しかるべき」タイミングとは具体的にどういったタイミングなのでしょうか。

クエリが発行されるタイミングは、次のように明確に決められています。詳しくは、公式ドキュメント^{*3} を参照してください。

1. for ループなどのイテレーションが開始されたタイミング
2. [] を使ってスライスしたタイミング^{*4}
3. オブジェクトを直列化したタイミング
4. オブジェクトを REPL (対話型評価環境) や print で表示したタイミング
5. len() でサイズを取得したタイミング
6. list() で強制的にリストに変換したタイミング
7. bool() で強制的に bool に変換したタイミング

クエリセットオブジェクトにこれらの操作をおこなった場合、次のように即座にクエリが発行されます。

```
>>> book_list = list(Book.objects.all())  
2019-04-01 00:02:53,738 [DEBUG] /usr/local/lib/python3.7/site-packages/django/db/  
backends/utils.py:111 (0.005) SELECT "book"."id", "book"."title", "book"."image",  
"book"."publisher_id", "book"."price", "book"."description", "book"."publish_dat  
e" FROM "book"; args=()
```

遅延評価に伴う弊害もあります。モデルのリレーションに気を付けないと想定外にクエリの数が増えてしまう可能性があるのです。この問題については章末の「6.11 ベストプラクティス 5 : select_related / prefetch_related でクエリ本数を減らす」を参照してください。

^{*3} <https://docs.djangoproject.com/ja/2.2/ref/models/querysets/#when-querysets-are-evaluated>

^{*4} ただし、[0:5] のように範囲を指定した場合にはクエリは即時発行されません。

6.7.8 その他のクエリセット API メソッド

クエリセット API には先に紹介したメソッド以外に多数のメソッドが用意されています。そのうちの一部を紹介します。

exists

レコードが存在するかどうかを True / False で返します。

```
>>> Book.objects.all().exists()
True
>>> Book.objects.filter(title__icontains='Flask').exists()
False
```

count

レコードの件数を返します。

```
>>> Book.objects.all().count()
5
>>> Book.objects.filter(title__icontains='Django').count()
2
```

order_by

検索結果をソートします。デフォルトは昇順です。

```
>>> Book.objects.all().order_by('price')
```

降順でソートする場合は、指定するフィールド名の前に「- (マイナス)」を付けます。

```
>>> Book.objects.all().order_by('-price')
```

複数フィールドでソートする場合は、次のようにカンマ区切りで列挙します。

```
>>> Book.objects.all().order_by('price', 'publish_date')
```

6.8 単体のオブジェクトを保存・更新・削除する

データベースのテーブルレベルのクエリ操作はモデルの「モデルマネージャ」を利用しましたが、単体のオブジェクトを保存、更新あるいは削除するような行レベルのクエリ操作は、モデルオブジェクトのメソッドを利用します。具体的にはモデルの `save()` メソッドと `delete()` メソッドを使います。

単体のオブジェクトをデータベースに保存するには、モデルクラスのコンストラクタにキーワード引数として値をセットするかインスタンスを作成してから属性に値をセットし、`save()` メソッドを実行します。

```
>>> user = User(username='admin', is_active=True)
>>> user.email = 'admin@example.com'
>>> user.save()
```

オブジェクトの更新は、データベースから取得したオブジェクトの属性値を変更して `save()` を実行します。

```
>>> user = User.objects.get(username='admin')
>>> user.username = 'root'
>>> user.save()
```

オブジェクトを削除する場合は `delete()` メソッドを実行します。

```
>>> user = User.objects.get(username='root')
>>> user.delete()
```

操作はこのように簡単です。

Django のデフォルト設定では、オブジェクトの保存・更新・削除は `save()` や `delete()` が実行された時点で即時にデータベースに反映されます（「オートコミットモード」と呼ばれます）。すなわち「トランザクション」（データの一貫性を確保するために複数のレコード操作をひとまとめにするためのコミットやロールバックの仕組み）は考慮されません。

それでは困る、という場合のために Django ORM では次のようにトランザクションの有効範囲を規定することができます。この例では、同じ `username` で 2 人のユーザーを登録しようとして 2 人目の登録時点での一意制約違反の例外が発生するのですが、1 人目のユーザーの保存はロールバックされてユーザーは結局 1 人も作成されません。

```
>>> from django.db import transaction
>>> with transaction.atomic():
...     User(username='aki').save()
...     User(username='aki').save()
...
...     (略)
django.db.utils.IntegrityError: UNIQUE constraint failed: auth_user.username
>>> User.objects.filter(username='aki').exists()
False
```

しかしながら、実際のビューやモデルではモデルオブジェクトの保存・更新・削除が入り組んだ実装になるかと思います。そのすべてに対して不測の事態を考慮してロールバックやコミットの処理を入れておくのは非常に面倒です。そこでもっと簡単にトランザクションの問題を解決できるアイデアとして、トランザクションのデフォルト設定を変更し、トランザクションの有効範囲をリクエストの開始から終了までにする方法があります。

手順は簡単です。設定ファイルの「`DATABASES.default`」に次のように「`ATOMIC_REQUESTS`」の記述を追加するだけです。

▼リスト 6.8 `settings.py`

```
DATABASES = [
    'default': {
        ...
        'ATOMIC_REQUESTS': True,
    }
]
```

この設定をしておくことで、ビューから例外が漏れてしまった場合にビュー内のデータベースの保存・更新・削除がすべてロールバックされることになります。非常に便利です。

6.9 ベストプラクティス 3：User モデルを拡張する

デフォルトで提供されている User モデルには リスト 6.5 で示したようなフィールドが定義されていますが、

- 性別
- 年齢
- 生年月日
- 住所
- ログイン回数

などの情報を格納するフィールドが無くて不便を感じるかもしれません。そのような場合には User モデルの拡張（フィールドやメソッドの追加）を検討することになりますが、組み込みの User モデルを変更する際の方法としては次の 3 つがよく利用されます。

1. 抽象クラス「AbstractBaseUser」を継承する
2. 抽象クラス「AbstractUser」を継承する
3. 別モデルを作って「OneToOneField」で関連させる

まだ本番リリースしていない開発段階であれば 1. や 2. の方法を推奨します。

その場合はまず、任意のアプリケーションに AbstractBaseUser または AbstractUser を継承したカスタムユーザークラスを作成します。どちらのクラスを継承するかはケースバイケースですが、ガラッと変えたいときは AbstractBaseUser を、User モデルに少しだけ手を加えたいだけなら AbstractUser を選べばよいでしょう。

▼リスト 6.9 AbstractUser を継承したクラスの例 (accounts/models.py)

```
from django.contrib.auth.models import AbstractUser
from django.db import models

class CustomUser(AbstractUser):
    """拡張ユーザークラス"""
    class Meta:
        db_table = 'custom_user'

    login_count = models.IntegerField(verbose_name='ログイン回数', default=0)
```

1. や 2. の方法を利用した場合に忘れてはいけないのが、「AUTH_USER_MODEL」の設定です。設定ファイル (settings.py) に次の設定を追加して、作成したモデルクラスを定義します。ここで、「AUTH_USER_MODEL」の値は「accounts.models.CustomUser」ではなく「accounts.CustomUser」(<アプリケーション名>.<モデルクラス名>) としなければならない点に注意しましょう。

▼リスト 6.10 config/settings.py

```
AUTH_USER_MODEL = 'accounts.CustomUser'
```

拡張 User クラスを利用する場合は次のように django.contrib.auth.get_user_model() 関数を使って、拡張した User モデルのクラスを取得します。

```
>>> from django.contrib.auth import get_user_model
>>> get_user_model()
<class 'accounts.models.CustomUser'>
>>> get_user_model().objects.all()
<QuerySet [<CustomUser: admin>, <CustomUser: akiyoko>]>
```

すでに本番稼働している Django プロジェクトの User モデルを拡張したい場合には、3. の方法もよく利用されます。1. や 2. の方法で User モデルのクラスを入れ替えしまうと、関連しているモデルのスキーマもすべて変わってしまい、影響範囲が大きくなってしまうからです。

次のクラスは、edX (edx-platform) で利用されている User モデル拡張の例^{*5} です。

▼リスト 6.11 common/djangoapps/student/models.py

```
class UserProfile(models.Model):
    ... (略) ...
    user = models.OneToOneField(User, unique=True, db_index=True,
                               related_name='profile', on_delete=models.CASCADE)
    ... (略) ...
    year_of_birth = models.IntegerField(blank=True, null=True, db_index=True)
    GENDER_CHOICES = (
        ('m', ugettext_noop('Male')),
        ('f', ugettext_noop('Female')),
        # Translators: 'Other' refers to the student's gender
        ('o', ugettext_noop('Other/Prefer Not to Say'))
    )
    gender = models.CharField(
        blank=True, null=True, max_length=6, db_index=True, choices=GENDER_CHOICES
    )
    ... (略) ...
```

6.10 ベストプラクティス 4：発行されるクエリを確認する

オブジェクトの検索が実行される際にどのようなクエリが発行されるかを確認することは、モデルの使い方が合っているか、あるいはパフォーマンスに影響を与えていなさそうかをチェックする上で非常に有用です。次のいずれかの手順を使って実際に発行されるクエリを見てみるとよいでしょう。

(その 1) Django シェルを使う

クエリセットオブジェクトの `query` 属性を `print` することで、発行されるクエリを出力することができます。Django シェルを使って、実際に発行されるクエリを確認してみましょう。

```
>>> print(Book.objects.filter(title__icontains='Django').query)
SELECT "book"."id", "book"."title", "book"."image_path", "book"."publisher_id",
"book"."price", "book"."description", "book"."publish_date", "book"."created",
"book"."modified" FROM "book" WHERE "book"."title" LIKE %Django% ESCAPE \'
```

本書では説明しませんが `pdb`^{*6} を利用したデバッグ中でも、この方法を使ってクエリを確認することができます。

^{*5} <https://github.com/edx/edx-platform/blob/master/common/djangoapps/student/models.py>

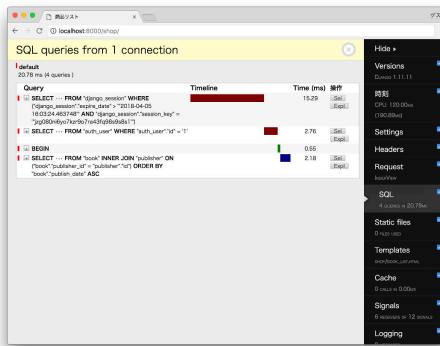
^{*6} <https://docs.python.org/ja/3.7/library/pdb.html>

(その2) ロギングの設定を変更する

設定ファイルのロギング設定（LOGGING）をうまく設定することで、発行されるクエリをコンソールやログにリアルタイムで出力することができます。詳しくは、第10章の「10.7 ロギングの設定」を参照してください。

(その3) django-debug-toolbar の SQL パネルを使う

「django-debug-toolbar」という Django パッケージのインストールとそれに伴う設定が必要なもの、一番手軽に発行されるクエリの内容を確認することができる方法です。実際に画面を操作しながら、右側に表示される SQL パネルを開いてクエリの内容を確認することができます。



▲図 6.8 django-debug-toolbar の SQL パネル

django-debug-toolbar の設定方法については、「14.3.1 django-debug-toolbar (GUIによるデバッグ)」をご参照ください。

6.11 ベストプラクティス 5：select_related / prefetch_related でクエリ本数を減らす

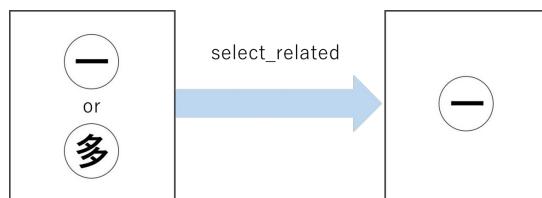
突然ですが、次の場合は何本のクエリが発行されると思いますか？（Book モデルに対するレコードの数は 10 とします。）

```
>>> for book in Book.objects.all():
...     print(book.publisher.name)
```

6.11 ベストプラクティス 5: `select_related` / `prefetch_related` でクエリ本数を減らす

1 本と答えた方、残念でした。正解は 11 本です。イテレーションが開始された時点でクエリが 1 本発行され、10 回ループする中でリレーション先のオブジェクト（この例では publisher）を取得しようとしてそれぞれ 1 回ずつクエリが発行されてしまうのです。これは「1+N 問題」（あるいは「N+1 問題」）と呼ばれ、特にリレーションを持ったモデルの検索結果（クエリセットオブジェクト）をループ処理する場合に起こりがちです。モデルを使うとクエリをあまり意識せずに使えて便利な半面、何も分からずに使っているとクエリの本数が想定以上に多くなり、簡単な一覧画面のはずがレコードのサイズが大きくなるとなぜか急激に重くなってしまうといった落とし穴にハマってしまいかねません。そのような場合は、前述のベストプラクティスを使って発行されているクエリを確認した上で、「`select_related`」や「`prefetch_related`」を使ってクエリの本数を減らすことを検討すべきです。

まず、`select_related` について説明します。`select_related` を使うと、リレーション先のオブジェクトを取得するために JOIN を使ったクエリを発行することができます。「一」や「多」側から「一」のリレーションのモデルオブジェクトを JOIN で取得したい場合に利用すると考えればよいでしょう。



▲図 6.9 `select_related` は「一」のオブジェクトを JOIN で取得

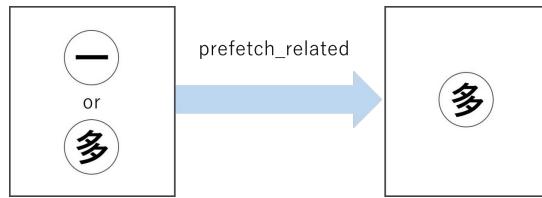
Book モデルを検索する際に `select_related` を次のように利用すると、

```
>>> Book.objects.all().select_related('publisher')
```

次のような JOIN を含むクエリが発行されます。

```
SELECT * FROM book INNER JOIN publisher ON book.publisher_id = publisher.id;
```

他方の `prefetch_related` は、「一」や「多」側のモデルから「多」のリレーションのモデルを参照する場合に利用します。こちらは JOIN を使うわけではなく、必要になった時点でのリレーション先のオブジェクトを取得するためのクエリを発行するのは通常の場合と同じなのですが、取得したオブジェクト群をオブジェクト内部のキャッシュに保持し、それを使い回すことで同じクエリが何度も発行されないような仕組みが施されます。



▲図 6.10 prefetch_related は「多」のオブジェクトを取得してキャッシュに保持

同じように prefetch_related を使って Book モデルを 次のように検索すると、一度取得したオブジェクト群に再度アクセスしようとしたときにキャッシュのデータが利用されるようになります。

```
>>> Book.objects.all().prefetch_related('authors')
```

`select_related` と `prefetch_related` はいずれも、後続の処理で何度もアクセスされそうなオブジェクトに対して前もって処理を施しておくことでクエリの本数を減らすという目的で利用されます。

6.12 まとめ

- モデルは `models.py` に書く
- モデルクラスは `django.db.models.Model` クラスを継承して作る
- モデルのフィールドには用途に合わせた `django.db.models.fields.Field` のサブクラスを定義する
- テーブルレベルのクエリ操作はモデルマネージャを経由してクエリセット API のメソッドを利用する
 - オブジェクトを 1 件取得する場合は `get()` を使う
 - 複数件取得する場合は `all()` または `filter()` を使う
- 行レベルのクエリ操作はモデルのメソッドを利用する
 - オブジェクトの保存・更新には `save()` を使う
 - オブジェクトの削除には `delete()` を使う
- クエリの発行回数がボトルネックにならないよう注意し、必要に応じて `select_related` や `prefetch_related` を使う

第7章

テンプレート (Template)

7.1 概要

Django には、HTML ファイルに変数の中身を表示したり条件分岐などの少し複雑なロジックを埋め込んだりすることができる「テンプレート」システムが備わっています。Django ではバックエンドのテンプレートエンジンとして「DTL (Django Template Language)」を利用することができます。^{*1}

DTL では、特別な記法を使って

- 変数表示：変数の内容を表示する
- フィルタ：変数の内容を表示する際の表示形式を変更する
- テンプレートタグ：テンプレートの機能を拡張する

を実現することができます。そして DTL 記法には

- 「{ }」「%」「|（パイプ）」「:」「#」を使う
- 「()」や「[]」は使えない
- Python 風の記法（Python そのままの記法ではない）
- 複雑なロジックは書けない（たとえば変数を定義したりすることができない）
- Python モジュールを直接 import して使うことはできない（独自の関数を使うにはテンプレートタグやフィルタを自作する）

といった特徴があります。変数定義ができないため、表示する変数は基本的にビュー側で事前にすべて用意しておく必要があります。筆者は、ちょっとしたロジックはテンプレートタグを駆使してなんとかする、それでもダメなら関数を独自テンプレートタグとして作るという使い方をしています。しかしながら、Django が提供しているフィルタやテンプレートタグはさまざまな用途を想定して数多く取り揃えられているので、大抵のものはすでにあるはずです。「こんなのが無いかな」と思ったら、まずは公式ドキュメント^{*2}を

^{*1} DTL 以外にも「Jinja2」というテンプレートエンジンを利用するこも可能ですが、本書ではデフォルトで使うことのできる DTL について説明をおこないます。

^{*2} <https://docs.djangoproject.com/ja/2.2/ref/templates/builtins/>

チェックすることをお勧めします。

7.2 変数表示

テンプレートには「コンテキスト」と呼ばれる変数と値がマッピングされたオブジェクトが渡されるため、テンプレートからはこのコンテキストに格納されている変数を参照することができます。コンテキストには、

- ビューから渡された変数（たとえば `django.shortcuts.render()` で渡される）
- 設定ファイル (`settings.py`) の `TEMPLATES.OPTIONS.context_processors` に定義された関数でセットされた変数

が含まれています。Django 2.2 の `context_processors` の初期設定では、たとえば次の変数をテンプレートからデフォルトで利用することができます。

- 「`request`」・・・リクエストオブジェクト
- 「`user`」・・・サイトにアクセスしているユーザー
- 「`perms`」・・・サイトにアクセスしているユーザーのパーミッション
- 「`messages`」・・・フラッシュメッセージ（第9章の「9.4 ベストプラクティス 8：メッセージフレームワークを使う」の節を参照）

変数を表示するための記法は、次のように二重の中括弧 `«{ { <変数名> }}»` で囲みます。

```
«{ { <変数名> }}»
```

たとえば、変数 `user` の内容を表示するのであれば次のように使います (`__str__()` の結果が表示されます)。

```
«{ { user }}»
```

ここで、変数の後ろに

```
«{ { user.username }}»
```

のようにして「. (ドット)」を付けると

- `user['username']` (辞書としての照合)
- `user.username` (属性値の照合)
- `user.username()` (メソッド呼び出し)
- `user[username]` (リストやタプルの添字指定)

の順番に値を取得しようとして、一番先に取得できた値を表示します。これを活用して、「.0」や「.1」などと記述することでリストの0番目、1番目の値を参照することができます。次の例では、`book_list` の0番目の要素の `title` という属性値が表示されます。

```
{{ book_list.0.title }}
```

ところで DTL で変数表示をする際は、特に何もしなくても XSS（クロスサイトスクリプティング）対策として「<」「>」「'」「"」「&」の文字列が自動でエスケープされます。逆に HTML タグをそのまま出力したいときは、後述する「`safe`」フィルタや「`autoescape`」タグを利用します。

7.3 フィルタ

変数表示の `{{ ~ }}` の内側にはロジックを書くことができませんが、その代わりとして、変数の表示内容を加工するための「フィルタ」という仕組みが用意されています。

フィルタは、次のように変数名の直後に「|（パイプ）」を使って繋げて書きます。フィルタによっては引数を取ることもできます。その場合はフィルタの直後に「:」で繋げて書きます。

```
{{ <変数名>|<フィルタ名> }}  
{{ <変数名>|<フィルタ名>:<引数> }}
```

フィルタは「|」で次々と連結することができます。

```
{{ <変数名>|<フィルタ名1>:<引数>|<フィルタ名2>:<引数> }}
```

「|」の前後にはスペースを入れることができますが、実際には入れない方がよいでしょう。なおフィルタの引数を指定するための「:」の前後にはスペースは入れられません。

DTL で利用できるフィルタは多数ありますが、その中でもよく利用される次のものを以降で紹介します。

- `default`（デフォルト表示）
- `length`（文字列長）
- `safe`（エスケープの無効化）
- `date`（日時フォーマット）

- `linebreaksbr` (改行タグの変換)
- `urlize` (リンクの変換)
- `truncatechars_html` (文字の切り詰め)

7.3.1 default (デフォルト表示)

変数が存在しない場合、あるいは変数の値が `None`, `''`, `0`, `False`, `[]` などの場合に、指定した文字列を表示することができます。特に、値が `None` の場合に「`None`」という文字列がそのまま出てしまうのを回避するためによく利用します。

```
 {{ user.first_name|default:"" }} {{ user.last_name|default:"" }}
```

また、`None` のときだけ指定した文字列に変換してくれる、`default_if_none` というフィルタもありますので、状況に合わせて使い分けるとよいでしょう。

7.3.2 length (文字列長)

変数の値の文字列長を表示します。

```
 {{ user.username|length }}
```

7.3.3 safe (エスケープの無効化)

変数表示の際は XSS 対策として「<」「>」「,」「"」「&」の文字列が自動でエスケープされますが、そのエスケープを無効にしたいときに利用します。変数の内容が安全だと分かっている場合にのみ利用するようにしてください。

```
 {{ book.description|safe }}
```

7.3.4 date (日時フォーマット)

日時オブジェクトを引数に指定した形式に変換して表示します。

```
{{ user.last_login|date:"Y-m-d H:i:s" }}
```

はたとえば、次のようにレンダリングされます。

```
2019-04-01 10:22:30
```

7.3.5 linebreaksbr (改行タグの変換)

「\n」を「
」に変換してくれます。このフィルタは、テキストエリアに「\n」が含まれる文字列を表示する場合によく利用されます。

```
{{ book.description|linebreaksbr }}
```

7.3.6 urlize (リンク変換)

文字列内に URL とメールアドレスが含まれる場合に、その部分だけをアンカータグで囲んでクリック可能なリンクに変換してくれます。

たとえば「book.description」の値が「<https://akiyoko.hatenablog.jp/> も見てね。」であれば、

```
{{ book.description|urlize }}
```

は「<https://akiyoko.hatenablog.jp/> も見てね。」に変換され、画面上ではリンク表示がおこなわれます。

7.3.7 truncatechars_html (文字の切り詰め)

文字列が長い場合に、決められた文字数まで切り詰めたあとで「...」という文字を付与してくれるフィルタです。「truncatechars」という似たようなフィルタがありますが、「truncatechars_html」の方は HTML タグを考慮して省略後の文字にきちんと閉じタグを付けてくれるのでさらに便利です。具体的には、「book.description」の値が「絶賛頒布中です。」であれば、

```
{{ book.description|truncatechars_html:5 }}
```

は「絶賛...」となります。

7.4 テンプレートタグ

テンプレートタグはテンプレートの機能を拡張するための記法です。テンプレートタグごとに異なるさまざまな機能を利用することができます。

テンプレートタグは「`{% ... %}`」で囲んで記述します。

```
{% <タグ名> "<引数>" %}
```

のように単行で書く場合もあれば、

```
{% <タグ名> "<引数>" %}
タグが適用される行 その 1
タグが適用される行 その 2
{% end<タグ名> %}
```

のようにタグの開始と終了を指定して、間に複数の行を挟むこともできます。

条件分岐やループなどのほか、便利なテンプレートタグが多数提供されていますが、その中でも特に頻繁に利用されるものを紹介します。

- `if` (条件分岐)
- `for` (ループ)
- `extends` (テンプレートの継承)
- `block` (オーバーライド対象ブロック)
- `include` (外部テンプレートの読み込み)
- `static` (静的ファイルの URL 取得)
- `url` (URL の逆引き)
- `autoescape` (自動エスケープ制御)

7.4.1 if (条件分岐)

「`{% if ... %}`」「`{% elif ... %}`」「`{% else ... %}`」「`{% endif %}`」というタグを利用して条件に合わせた表示の切り替えをおこなうことができます。「if」「elif」「else」を使うのは通常の Python の構文と同じですが、最後に「endif」を付ける必要がありますので注意しましょう。またそれぞれのタグ内部の末尾には「: (コロン)」は必要ありませんので合わせてご注意を。

```
{% if user.is_superuser %}
システム管理者です。
{% elif user.is_staff %}
スタッフです。
{% else %}
一般ユーザーです。
{% endif %}
```

if タグにはフィルタを併用することもできます。

```
{% if user.username|length < 3 %}
ユーザー名が短すぎます。
{% endif %}
```

しかしながら、次の例のように「!」の前後にスペースを空けるとシンタックスエラーが出てしますので注意しましょう。

```
{% if user.username | length < 3 %}
```

```
{# シンタックスエラー! #}
```

7.4.2 for (ループ)

「{% for ... %}」と「{% endfor %}」で囲まれた範囲をループさせることができます。

```
{% for book in book_list %}
{{ book.title }}
{% endfor %}
```

変数が存在しない、あるいは変数の値が空のリストでループが回せない場合に表示する内容を記述するための「empty」タグを併用すると、さらに便利に使える場合があります。

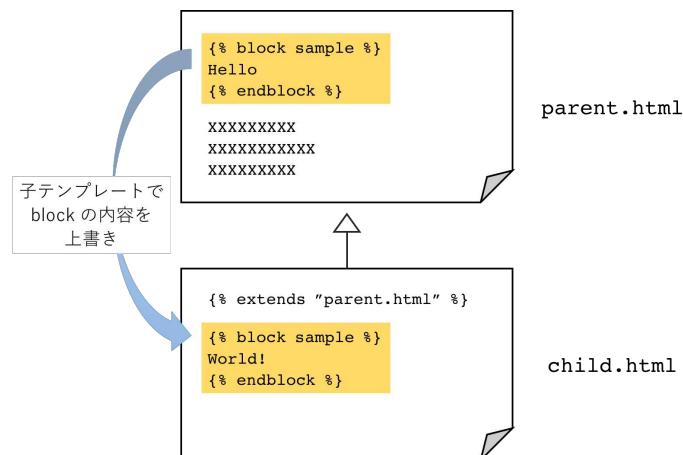
```
{% for book in book_list %}
{{ book.title }}
{% empty %}
  本がありません。
{% endfor %}
```

また for ループには、ループ内で使える特別な変数が用意されています。

- forloop.counter … ループ回数 (1 から開始)
- forloop.counter0 … ループ回数 (0 から開始)
- forloop.first … 最初のループかどうか

7.4.3 extends (テンプレートの継承)

「extends」タグを利用してすることで、別のテンプレートをベースにして必要な部分だけを書き換えるといったクラスの継承のようなことが実現できます。後述する「block」タグと組み合わせて利用されます。



▲図 7.1 extends タグでテンプレートを継承

extends タグは子テンプレート側のファイルの冒頭に記述します。

```
{% extends "parent.html" %}
```

7.4.4 block (オーバーライド対象ブロック)

「{% block <任意の名前> %}」と「{% endblock %}」で囲まれた範囲（ブロック）に任意の名前を付けて、継承関係のある子テンプレートから内容の上書きができるようになるためのタグです。

たとえば次の例では、「sample」ブロックが子テンプレートによって書き換えられるため、子テンプレートには「World!」と出力されます。

▼リスト 7.1 親テンプレートの例 (parent.html)

```
{% block sample %}Hello {% endblock %}
```

▼リスト 7.2 子テンプレートの例 (child.html)

```
{% extends "parent.html" %}
{% block sample %}World!{% endblock %}
```

block タグの内側では「`block.super`」という特別な変数を使うことができ、これは継承元となる親テンプレートのブロック内部の値をそのまま保持した変数になります。これにより、block タグをすべて入れ替えるのではなく、親テンプレートの内容に子テンプレートの内容を付け加えることが可能になります。

▼リスト 7.3 親テンプレートの例 (parent.html)

```
{% block sample %}Hello {% endblock %}
```

▼リスト 7.4 子テンプレートの例 (child.html)

```
{% extends "parent.html" %}
{% block sample %}{% block.super %}World!{% endblock %}
```

この場合は子テンプレートには「Hello World!」と出力されます。

7.4.5 include (外部テンプレートの読み込み)

「`include`」はテンプレートから他のテンプレートを読み込むためのタグです。ヘッダナビゲーションやフッタなどの部品化したテンプレートを別のテンプレートから読み込むような場合に用いられます。

```
{% include "_message.html" %}
```

7.4.6 static (静的ファイルの配信 URL 取得)

「`static`」は静的ファイルの配信 URL を取得するためのタグです。静的ファイルのパスについては、第 10 章の「10.4 静的ファイル関連の設定」で詳しく説明しています。

```
{% static "images/logo.png" %}
{% static "shop/images/no-image.png" %}
```

static タグは公式ドキュメント^{*3}にも書かれている通り、Django デフォルトで使える組み込みタグではありません。したがって static タグを利用するためには、利用する前に次の記述をして機能をロードしておかなければいけません。

```
{% load static %}
```

大抵の場合、テンプレートファイルの先頭で書いておけばよいでしょう。注意したいのは、extends タグを使って他のテンプレートを継承している場合に継承元のテンプレートで「{% load static %}」を記述していても自身のテンプレートでは有効にならない点です。static タグを利用する側のテンプレートで改めて「{% load static %}」を記述する必要があります。

7.4.7 url (URL の逆引き)

「url」タグは、URLconf に登録された URL のパターン名から URL を逆引きするために利用します。これにより、テンプレートに URL をハードコーディングしないようにすることができます。

1 つ目の引数には URL のパターン名（たとえば「index」「accounts:login」など）を指定します。2 つ目以降の引数には、URL パターンに正規表現グループが含まれている場合にそのパターンにマッチするように固定引数やキーワード引数で値を指定することができます。

```
{% url "index" %}
{% url "accounts:login" %}
{% url "shop:detail" book.id %}
{% url "shop:detail" book_id=book.id %}
```

7.4.8 autoescape (自動エスケープ制御)

「{% autoescape ... %}」と「{% endautoescape %}」で囲まれた範囲の自動エスケープ機能を制御します。autoescape タグは引数に on または off を取り、「off」の場合に自動エスケープ機能が無効化されます。

^{*3} <https://docs.djangoproject.com/ja/2.2/ref/templates/builtins/#other-tags-and-filters-libraries>

```
{% autoescape off %}  
自動エスケープがオフになる範囲  
{% endautoescape %}
```

7.5 ベストプラクティス 6：ベーステンプレートを用意する

どのテンプレートにも書かなければいけないような共通の内容、たとえば <head> タグの記述や <body> タグ終了の前に JavaScript を読み込んだりするといった記述をすべてのテンプレートにいちいち書くというのは、その内容に変更が入ったときのメンテナンスを考えるとさすがにやりたくないですよね。そこで、すべてのテンプレートのベースとなる「base.html」（テンプレート名は任意）をトップの位置に用意し、各テンプレートが extends タグで base.html を継承してテンプレートの一部だけを書き換えるという方法がよく取られます。

たとえば次のようなテンプレートファイル構成で base.html を配置し、

```
-- templates           (← テンプレートファイル用のディレクトリ)
  |-- accounts
  |   |-- login.html
  `-- base.html
```

base.html にはリスト 7.5 に示すように、共通となるコンテンツに加えて子テンプレートで拡張できるような block タグを埋め込んでおきます。

▼リスト 7.5 base.html

```
<!doctype html>
{% load static %}
<html lang="ja">
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>{% block page_title %}{% endblock %}</title>

    {% --- css --- %}
    <link rel="stylesheet" href="{% static 'css/semantic.min.css' %}">
    <style type="text/css">
        body {
            background-color: #dadada;
        }
    </style>
    {% block extra_css %}{% endblock %}
</head>
<body>
<div class="ui container">
    {% block content %}{% endblock %}
</div>

{% --- js --- %}
```

```
<script src="{% static 'js/jquery-3.3.1.min.js' %}"></script>
<script src="{% static 'js/jquery-ui.min.js' %}"></script>
<script src="{% static 'js/semantic.min.js' %}"></script>
{% block extra_js %}{% endblock %}
</body>
</html>
```

各テンプレートからはそれぞれ extends タグで base.html を継承し、上書きしたい部分や追加したい部分だけを書き換えるようにします。

▼リスト 7.6 accounts/login.html

```
{% extends "base.html" %}

{% block page_title %}ログイン{% endblock %}

{% block extra_css %}
<style type="text/css">
    .container > .grid {
        height: 100%;
    }
</style>
{% endblock %}

{% block content %}
<div class="ui middle aligned center aligned grid">
    <div class="column">
        <form action="{% url 'accounts:login' %}" method="post" class="ui form">
            (略)
        </form>
    </div>
</div>
{% endblock %}
```

これで無駄な記述をテンプレートに毎回書かなくて済むようになります。

7.6 まとめ

- ・ テンプレートは、表示するデータの見た目を整えるスタイルリスト
- ・ テンプレートエンジンは「DTL (Django Template Language)」と「Jinja2」が利用できる
- ・ DTL では「変数表示」「フィルタ」「テンプレートタグ」の記法を使って表示内容を動的に変更することができる
- ・ 変数表示の際は、何もしなくても自動的に危険な HTML タグがエスケープされる
- ・ フィルタで変数の表示形式を変更する
- ・ テンプレートタグでテンプレートの機能を拡張する
- ・ 組み込みのフィルタやテンプレートタグ^{*4} を駆使すれば大抵のことはできる
- ・ すべてのテンプレートのベースとなる base.html を用意しよう

^{*4} <https://docs.djangoproject.com/ja/2.2/ref/templates/builtins/>

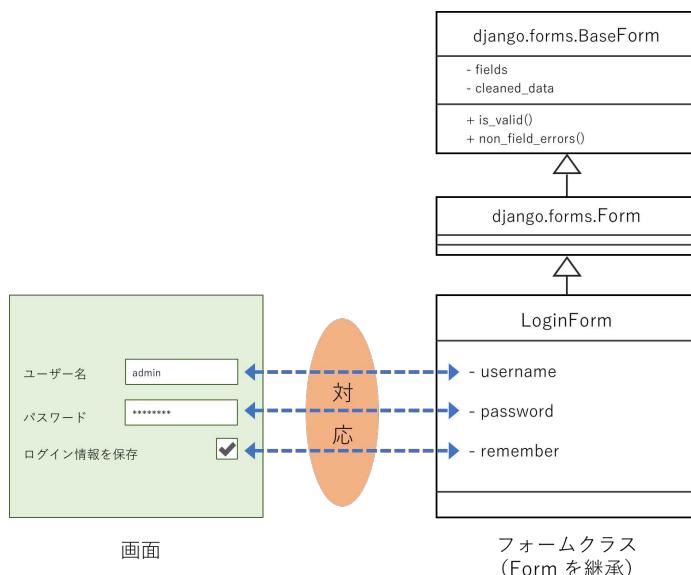
第8章

フォーム (Form)

8.1 概要

フォームは、次の2つの大きな役割を持っています。

1. ユーザーの入力データを保持
 2. 入力データのバリデーション（妥当性チェック）をおこない、妥当性検証済みのデータやエラーメッセージを保持
1. については、図8.1のように画面の入力フィールドのname属性とフォームクラスのクラス変数名を対応させて定義しておくことで、画面の入力内容からフォームオブジェクト、フォームオブジェクトから画面の入力内容へのデータ変換を容易におこなうことができます。



▲図8.1 画面の入力フィールドとフォームクラスの属性を対応させる

2. については、フォームに適切な名前のバリデーションメソッドを用意しておくことで、バリデーション実行時にフォームが決まったルールにしたがって次々とバリデーションメソッドを呼び出してくれます。フォームには、バリデーション OK の場合は妥当性検証済みのデータを、バリデーション NG の場合はエラーメッセージをフォームオブジェクトの内部に保持しておくための仕組みが備わっています。

正直なところフォームの利用は必須ではありません。しかしながら次に挙げたようなメリットが得られるため、利用価値は高いでしょう。

- ユーザーの入力データを（定義にしたがって）型変換してくれる
- バリデーションをビューから分離できる
- 入力データやエラーメッセージをテンプレートに簡単に表示できる
- ユーザー認証系のよくあるフォームは Django がデフォルトで用意してくれている

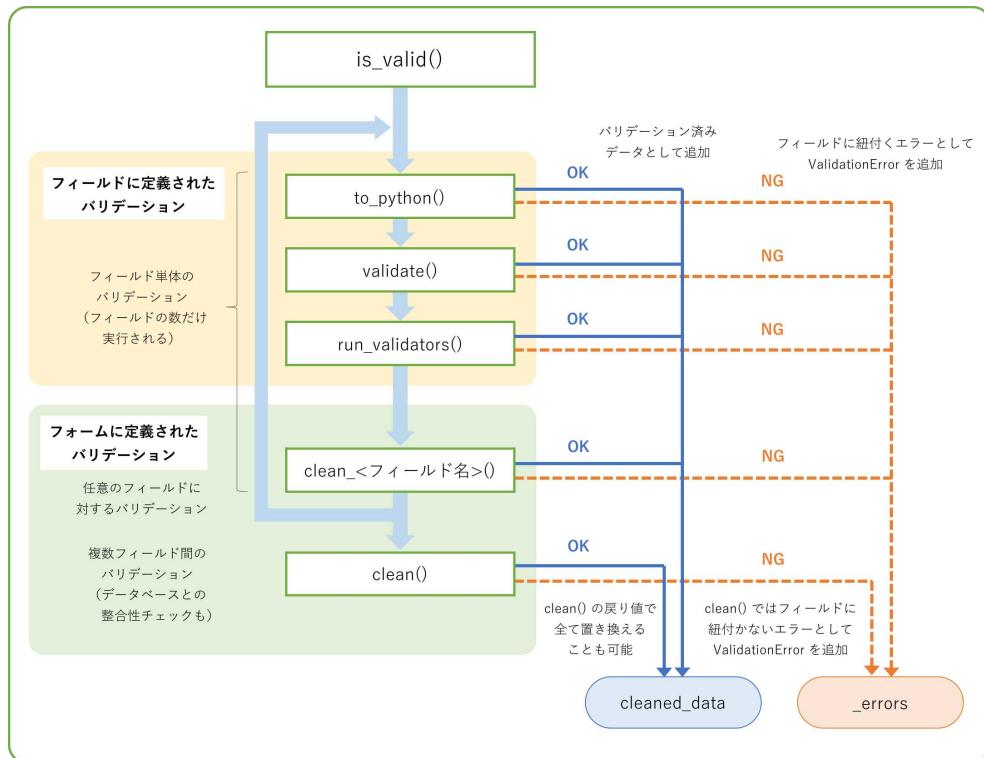
8.2 バリデーションの仕組み

前述のように、フォームはバリデーション実行時に決まったルールにしたがって次々とバリデーションメソッドを呼び出してくれますが、そのルールは少し複雑です。

まず、バリデーションのエントリーポイントはフォームの「`is_valid`」メソッドです。`is_valid()` が実行されると、フォームは大まかに次の順にバリデーションをおこないます。

1. フィールドに定義されたフィールド単体のバリデーション（文字種などの最低限の形式チェック）
2. フォームに定義されたフィールド単体のバリデーション（値の妥当性チェック）
3. フォームに定義された複数フィールド間のバリデーション

図 8.2 に、フォームのバリデーション実行時に処理されるメソッドのフローを示します。



▲図 8.2 Form のバリデーションフロー

まず、フィールドに定義されたバリデーションが実行されます。ここで「`to_python()`」や「`validate()`」は、フォームに定義するフィールドに自作したクラスを使わない限り意識する必要はないでしょう。「`run_validators()`」は文字種チェックなどのバリデーションをフィールドの `validators` 属性に定義している場合に実行されますが、こちらについても、フォームのフィールドとして後述する `django.forms.fields.Field` クラスのサブクラスを適切に使っている限りあまり意識しなくともよいでしょう。

続いて、フォームに定義されたバリデーションのうち、フィールド単体に対するバリデーションが実行されます。フィールド単体のバリデーションメソッドは「`clean_<フィールド名>`」という名前で任意に用意します(バリデーションが必要なければ用意しなくても構いません)。

ここまででのバリデーションが、フォームのフィールドの数だけ実行されます。実行されるフィールドの順番は、クラスに定義した順(上から順)です。フィールド名のアルファベット順などではありませんので注意しましょう。

そして最後に、相関チェックやデータベースとの整合性チェックをおこなうための「`clean()`」メソッドが実行されます。こちらも任意で用意します。

各バリデーションメソッドでは、バリデーション OK の場合に「`cleaned_data`」という dict 型の変数に妥当性検証済みデータとして値がセットされ、バリデーション NG の場合

にエラーメッセージが list 型の変数に追加されます。図 8.2 で示したように、`is_valid()` が実行されてはじめてフォームオブジェクトの内部に `cleaned_data` という変数が保持されるようになる仕組みには注意が必要です。`is_valid()` 実行前にフォームオブジェクトの `cleaned_data` を参照しようとして「`AttributeError: 'LoginForm' object has no attribute 'cleaned_data'`」などというエラーに遭遇してしまうというのは、フォームあるあるですので気を付けましょう。

8.3 フォームクラスの書き方

慣例的に、フォームは各アプリケーションディレクトリの `forms.py` という名前のモジュールに記述します。フォームクラスは「`django.forms.Form`」を継承し、内部に画面の入力フィールドに対応するクラス変数と任意のバリデーションを定義します。まずは入力フィールドの定義から見ていきます。

8.3.1 入力フィールドの定義

入力フィールドに対応する変数には、表 8.1 に示すような「`django.forms.fields.Field`」クラスのサブクラスを利用します。利用できる組み込みの Field クラスは表に示したもの以外にもありますので、公式ドキュメント^{*1}をご確認ください。

▼表 8.1 利用可能な Field クラス（一部）

Field クラス	適用されるウィジエット	画面に表示される入力フィールドの型
CharField	TextInput	テキストフィールド
EmailField	EmailInput	メールアドレスフィールド (type="email")
BooleanField	CheckboxInput	チェックボックス
ChoiceField	Select	セレクトボックス
IntegerField	NumberInput	数値フィールド (type="number")
DateField	DateInput	テキストフィールド
TimeField	TimeInput	テキストフィールド
DateTimeField	DateTimeInput	テキストフィールド
FileField	ClearableFileInput	ファイル選択フィールド

フォームクラスとフィールド定義の実装例を次に示します。クラス変数名（この例では「`username`」と「`password`」）がそのまま、画面に表示したときの入力フィールドの `name` 属性になります。

^{*1} <https://docs.djangoproject.com/ja/2.2/ref/forms/fields/#built-in-field-classes>

▼リスト 8.1 フォームクラスの例 (accounts/forms.py)

```

from django import forms
from django.contrib.auth.forms import UsernameField

class LoginForm(forms.Form):
    """ログイン画面用のフォーム"""
    username = UsernameField(
        label='ユーザー名',
        max_length=255,
    )

    password = forms.CharField(
        label='パスワード',
        strip=False,
        widget=forms.PasswordInput(render_value=True),
    )

```

フィールドクラスには「ウィジェット (widget)」と呼ばれる画面に表示される際の入力フィールドの型やプレースホルダ・CSS クラスなどを定義するオブジェクトを指定することができますが、利用する Field クラスによってデフォルトで適用されるウィジェットがそれぞれ決まっています（表 8.1 の「適用されるウィジェット」列に記載）。もしこれをデフォルトのものから変更したい場合には、フィールドクラスの `widget` 属性を変更します。リスト 8.1 の `password` フィールドは、パスワードの入力フィールドを表示するためにデフォルトの「TextInput」から「PasswordInput」というウィジェットクラスに変更した例です。

8.3.2 バリデーションメソッドの定義

ここでバリデーションメソッドの書き方について説明します。
前に述べたように、フォームクラスには

- `clean_<フィールド名>` … フィールド単体のバリデーション
- `clean` … 複数フィールド間の相關チェックやデータベースとの整合性チェック

という 2 種類のメソッドを記述します。

まず、`clean_<フィールド名>` から説明します。入力値は `cleaned_data` というフォーム内部で保持している dict オブジェクトから取得します。バリデーション OK の場合は妥当性チェック済みの値を `return` することで、検証済みデータとして `cleaned_data` に値を再セットすることができます。ここで何も値を `return` しなければ、`cleaned_data` から値が消えてしまう仕組みになっています。一方でバリデーション NG の場合には、「`django.forms.ValidationError`」を `raise` することでフォームオブジェクト内部の変数にエラーメッセージを追加することができます。

フィールド `username` の単体バリデーションメソッド `clean_username` の実装例は次のようになります。

```
def clean_username(self):
    # 入力値は cleaned_data から取得
    username = self.cleaned_data['username']
    # 入力値が 3 衝より短ければバリデーションエラー
    if len(username) < 3:
        raise forms.ValidationError(
            '%(min_length)s 文字以上で入力してください', params={'min_length': 3})
    return username
```

ValidationError にはこのようにエラーメッセージを設定することができます。

次は、clean メソッドの書き方です。

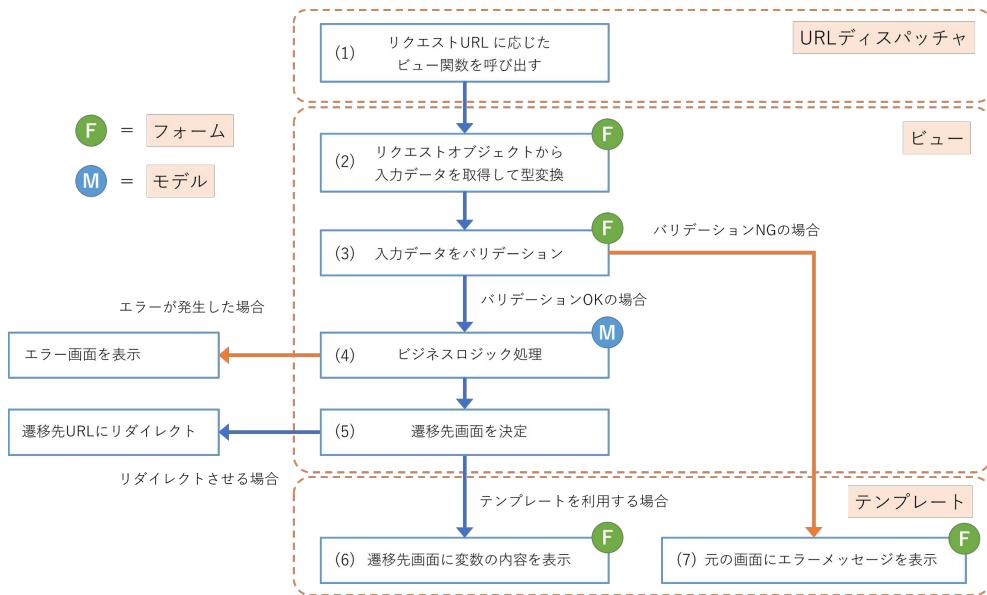
clean メソッドは clean_<フィールド名> メソッドとは違い、バリデーション OK の場合に検証済みデータを return する必要はありません。バリデーション NG の場合に django.forms.ValidationError を raise することで、エラーメッセージを内部の変数に追加することができます。次の例では、ユーザー名とパスワードの組み合わせが登録済みのものと合致するかどうかをチェックしています。 *2

```
def clean(self):
    username = self.cleaned_data.get('username')
    password = self.cleaned_data.get('password')
    try:
        user = User.objects.get(username=username)
    except ObjectDoesNotExist:
        raise forms.ValidationError("正しいユーザー名を入力してください")
    # パスワードはハッシュ化されて保存されているので平文での検索はできない
    if not user.check_password(password):
        raise forms.ValidationError("正しいパスワードを入力してください")
```

8.4 ビューやテンプレートからフォームを利用する方法

フォームは基本的に、ビューおよびテンプレート内で利用します。Django プロジェクトのよくあるサーバ側処理のフローを示した次の図においては、(2) (3) (6) (7) でフォームを利用するケースが多いでしょう。

*2 この例はあくまでサンプルです。セキュリティ上、攻撃者にヒントを与えるようなエラーメッセージを出すべきではありません。



▲図 8.3 フォームはビューやテンプレート内で利用する

8.4.1 ビューでの利用例

- (2) リクエストオブジェクトから入力データを取得して型変換
- (3) 入力データをバリデーション

リクエストオブジェクトからユーザーの入力値を取り出すには、もしフォームを使わなければ次のように実装します（たとえば POST リクエストの場合）。

```

username = request.POST.get('username')
password = request.POST.get('password')
...
  
```

このように取り出した値はすべて str 型になっているため、場合によっては型変換が必要になります。フォームを使わなければここからバリデーションをダラダラと書いていくことになりますが、フォームを使うことで、リクエストオブジェクトから入力データを取り出して型変換してすべてのバリデーションを完了させるまで、次のようにわずか 2 行で済みます。

```

form = LoginForm(request.POST)
is_valid = form.is_valid()
  
```

1 行目でフォームオブジェクトを作成しています。この時点では型変換もバリデーションもまだ実行されていません。2 行目でフォームの is_valid() メソッドを呼び出すことで、

バリデーションをまとめて実行します。ビューからバリデーションのロジックが分離されてスッキリとしていますね。

`is_valid()` はバリデーションの成否を `bool` 型で返します。その結果に応じて表示するテンプレートを決定し、フォームの入力データやエラーメッセージを表示するためにテンプレートにフォームオブジェクトを渡します。次の例では、バリデーション NG の場合に、遷移元画面のテンプレートにフォームオブジェクトを `'form'` という変数名で渡しています。

```
if not is_valid:  
    return render(request, 'accounts/login.html', {'form': form})
```

8.4.2 テンプレートでの利用例

(6)(7) テンプレート内でフォームの入力データやエラーメッセージを表示

テンプレート内では、ビューから渡されたフォームオブジェクトを使って内部に保持している入力データやエラーメッセージをレンダリングします。その際、

```
{{ form }}
```

とフォームオブジェクトを変数表示するだけで、次のように入力データがセットされた入力フィールドをすべて出力することができます。

```
<label for="id_username">ユーザー名:</label><input type="text" name="username"  
value="admin" id="id_username" required="">  
<label for="id_password">パスワード:</label><input type="password" name="password"  
id="id_password" required="">
```

なお、フィールドの出力順序はフォームクラスにフィールドを定義した順（上から順）になります。もしこの順序を変更したければ、フォームに `「field_order」` というクラス変数を定義して、出力したいフィールドの順序を次のように指定します。

```
class LoginForm(forms.Form):  
    field_order = ['password', 'username']
```

「`{{ form }}`」という出力方式は大変便利ですが、CSS のスタイルなどの細かい装飾が施せないという欠点もあります。ですので、この方式は簡易的にフォームオブジェクトをレンダリングするために開発時に使うものと考えた方がよさそうです。

実際には、次のようにフォームオブジェクトの要素を for タグでループさせてそれぞれのフィールドのスタイルを調整することが多いでしょう。

```
{# --- フィールドの分だけループさせる --- #}
{% for field in form %}
    <div class="field">
        {# --- 入力フィールド --- #}
        <div class="ui input">
            {{ field }}
        </div>
        {# --- 入力フィールドごとのエラーメッセージ（最初のエラーのみを表示） --- #}
        {% if field.errors %}
            <p class="red message">{{ field.errors.0 }}</p>
        {% endif %}
    </div>
{% endfor %}

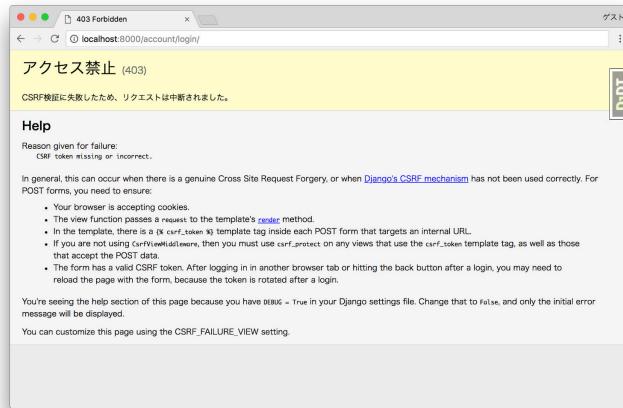
{# --- 全体エラーメッセージ --- #}
{% if form.non_field_errors %}
<div class="ui red message">
    <ul class="list">
        {% for non_field_error in form.non_field_errors %}
            <li>{{ non_field_error }}</li>
        {% endfor %}
    </ul>
</div>
{% endif %}
```

フォームオブジェクトをイテレートするとフォームで定義したフィールドオブジェクト (BoundField) が順番に取り出せるというのはよく利用するので覚えておきましょう。ここで取り出せるフィールドの順番も、クラスに定義した順番になります。

最後に、フォームオブジェクトからエラーメッセージを取り出す方法について説明します。フィールドに関連するエラーメッセージはそれぞれのフィールドの「errors」属性にリスト形式で保持されています。一方の clean() メソッドで追加した単体のフィールドに関連しない全体エラーメッセージは、フォームオブジェクトの「non_field_errors」という属性からアクセスすることができます。

8.5 CSRF 対策について

フォームを使っていると（使っていなくても）、次のような「CSRF 検証に失敗・・」などというエラーに遭遇してしまうことはないでしょうか。



▲図 8.4 CSRF エラー画面

これは、POST リクエストの際に「`csrfmiddlewaretoken`」というパラメータがリクエストに含まれていない場合に発生するエラーです。・・といきなりといわれても、意味が分かりませんよね？

CSRF（「シーサーフ」と読むことも）は「クロスサイトリクエストフォージェリ」と呼ばれるセキュリティ攻撃の一種ですが、Django ではその対策として POST リクエストに CSRF 対策を施すことが強く推奨されています。そして、POST リクエストに CSRF 対策のためのトークン文字列が含まれていない場合にこのような 403 エラーを返す仕組みがデフォルトで入っていて、そのトークンが「`csrfmiddlewaretoken`」というわけなのです。

しかしながら、CSRF 対策といっても何も難しく考える必要はありません。テンプレートの `<form>` タグの内部に「`{% csrf_token %}`」という特別なテンプレートタグを単に含めるだけです。たとえば次のようにします。

```
<form action="" method="post" class="ui form">
  ...
  {% csrf_token %}
</form>
```

これが実際には次のように name が「`csrfmiddlewaretoken`」で value が複雑なハッシュ文字列になった `<input>` タグが output されます (type が "hidden" なので画面上には表示されません)。

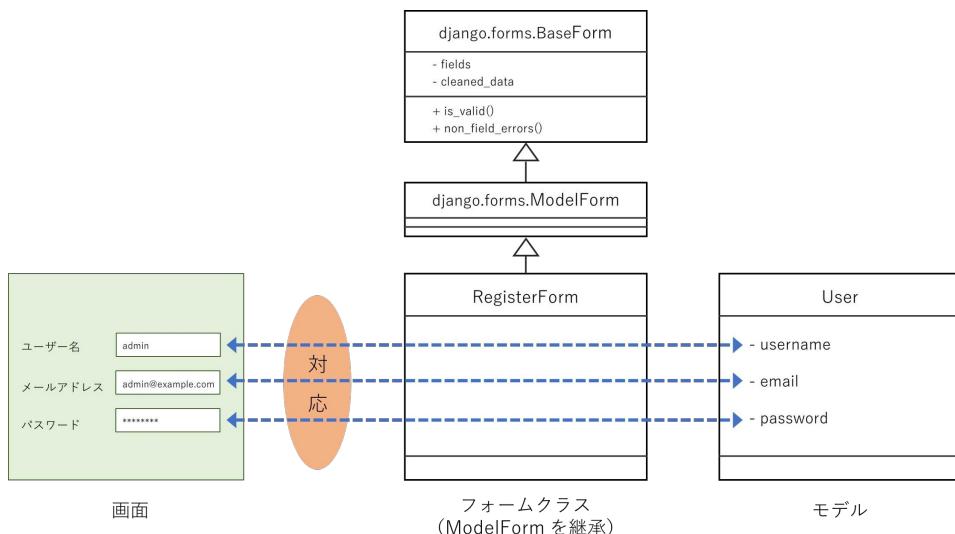
```
<form action="" method="post" class="ui form">
  ...
  <input type="hidden" name="csrfmiddlewaretoken" value="58N817xaDD5iULLWKy09JS
WPmKfwo03NIcG6AQF4oVwevH6dUDXoNPTeXnrJkoA3">
</form>
```

なお、第 9 章で説明する「CsrfViewMiddleware」というミドルウェアの設定を無効化すると CSRF トークンの検証をしなくなるためこのエラーは出なくなりますが、セキュリティ上のリスクが発生するので注意が必要です。

また、Ajax による POST リクエストの場合も CSRF 対策が求められます（HTTP ヘッダに「X-CSRFToken」を設定することでも対応が可能です）。フォームを利用した場合に限りませんが、POST リクエストをする際には気を付けましょう。

8.6 ベストプラクティス 7：こんなときは ModelForm を継承しよう

さて、モデルを登録・変更する画面のフォームを作っているときなどで、モデルとフォームのフィールドの定義が似たようなものになることがあるかと思います。同じようなフィールドをモデルにもフォームにも定義するのは億劫ですよね。そんなときは「ModelForm」の出番です。



▲図 8.5 ModelForm を継承したクラス

通常のフォームが継承している「django.forms.Form」の代わりに「django.forms.models.ModelForm」を継承することで特定のモデルのフィールドの定義を再利用することができ、フォームの記述量を減らすことができます。

ModelForm を継承したフォームは、次のように Meta クラスに利用するモデルクラスとフィールド名を定義しておくことで、モデルのフィールド定義を参照してフォームのフィールドへの変換がおこなわれます。

▼リスト 8.2 ModelForm を利用したユーザー登録画面のフォームクラスの例

```

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

class RegisterForm(forms.ModelForm):
    """ユーザー登録画面用のフォーム"""
    class Meta:
        # 利用するモデルクラスを指定
        model = User
        # 利用するモデルのフィールドを指定
        fields = ('username', 'email', 'password',)

...

```

モデルとフォームではフィールドを定義するクラスに違いがありますが、ModelForm は、モデルの django.db.models.fields.Field のサブクラスを自動判別してそれに対応するフォームの django.forms.fields.Field のサブクラスに変換してくれます。その対応表は次のようにになります。

▼表 8.2 モデルとフォームの Field クラスの対応表（一部）

モデルの Field クラス	フォームの Field クラス
django.db.models.fields.CharField	django.forms.fields.CharField
django.db.models.fields.EmailField	django.forms.fields.EmailField
django.db.models.fields.BooleanField	django.forms.fields.BooleanField
django.db.models.fields.related.ForeignKey	django.forms.models.ModelChoiceField
django.db.models.fields.IntegerField	django.forms.fields.IntegerField
django.db.models.fields.DateField	django.forms.fields.DateInput
django.db.models.fields.TimeField	django.forms.fields.TimeInput
django.db.models.fields.DateTimeField	django.forms.fields.DateTimeInput
django.db.models.fields.FileField	django.forms.fields.FileField

もし変換されたフォームの Field クラスそのままでは使えないという場合には、Meta の `widget` でウィジェットを上書きしたり、`__init__()` で `self.fields[<フィールド名>]` の属性を書き換えたりすることも可能です。また、そもそもモデルに足りないフィールドについては ModelForm クラスのフィールドを追加することで対応可能です。リスト 8.3 の例は、リスト 8.2 を少し修正して、以下の修正を加えています。

- モデルの password フィールドが変換された CharField に適用される TextInput ウィジェットを PasswordInput に変更
- User モデルには無い「確認用パスワード (password2)」フィールドを追加
- email フィールドに必須設定を付与
- 各フィールドにプレースホルダを付与

▼リスト 8.3 ModelForm を利用したフォームクラスの例（その 2）

```

class RegisterForm(forms.ModelForm):
    """ユーザー登録画面用のフォーム"""
    class Meta:
        model = User
        fields = ('username', 'email', 'password',)
        widgets = {
            'password': forms.PasswordInput(attrs={'placeholder': 'パスワード'}),
        }

        password2 = forms.CharField(
            label='確認用パスワード',
            required=True,
            strip=False,
            widget=forms.PasswordInput(attrs={'placeholder': '確認用パスワード'}),
        )

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.fields['username'].widget.attrs = {'placeholder': 'ユーザー名'}
        self.fields['email'].required = True
        self.fields['email'].widget.attrs = {'placeholder': 'メールアドレス'}

```

ModelForm は具体的には次の条件を満たすような場合に利用価値が高いでしょう。

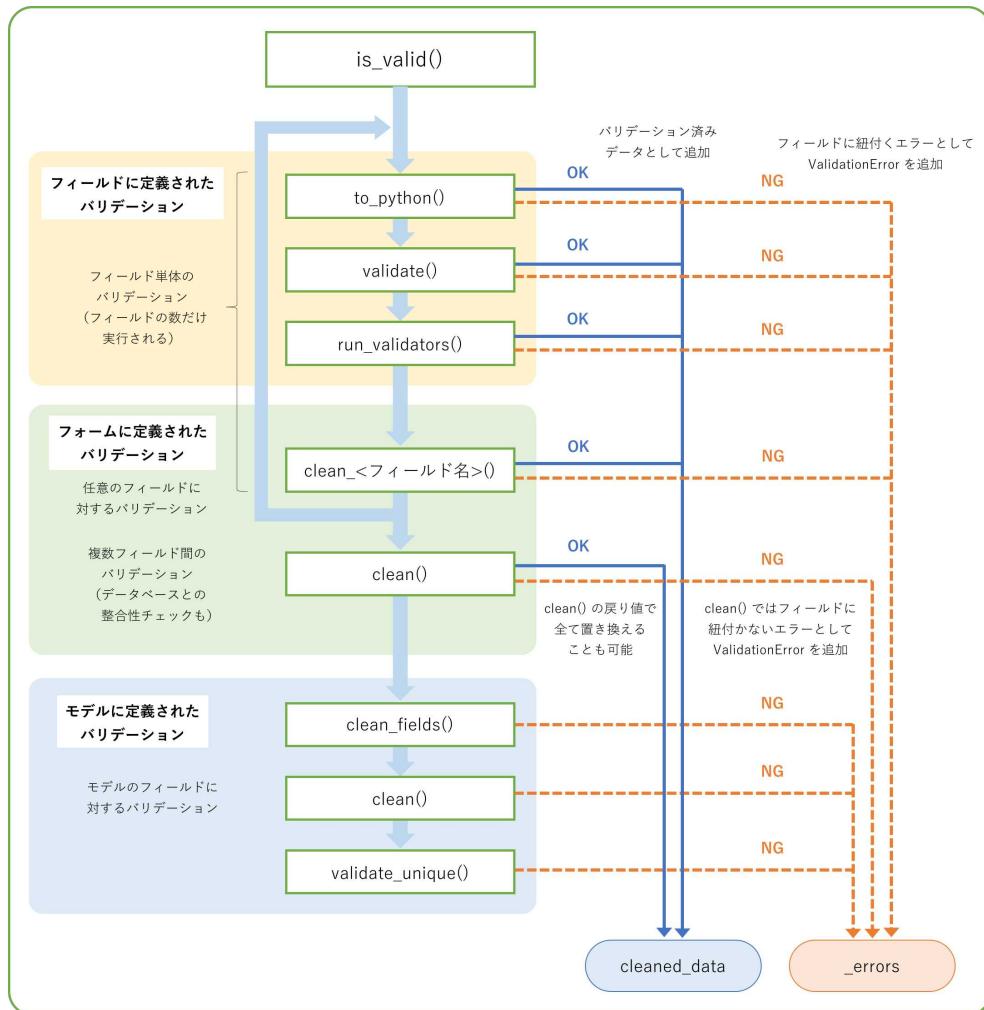
1. モデルに定義したフィールドのうちのいくつかが画面の入力フィールドと合致する
2. モデルの登録や変更を伴う

実際のところ、モデルの登録や変更を伴わない場合（検索フォームなど）でモデルのフィールドをいくつか利用したいというシーンはあまり無いように思います。もしあるとすればログイン画面でしょうか。そのような場合にも一応 ModelForm を使うことはできるのですが、ちょっとしたテクニックが必要になってきます（後述します）。

続いて、ModelForm のバリデーションの流れを追ってみましょう。ModelForm のバリデーションは図 8.6 のように、

1. フィールドのバリデーション（文字種などの形式チェック）
2. フォームのバリデーション（値の妥当性チェック）
3. モデルのバリデーション（ユニーク制約などのデータベースとの整合性チェック）

の順に実行されます。



▲図 8.6 ModelForm のバリデーションフロー

「フィールドのバリデーション」と「フォームのバリデーション」までは通常のフォームのバリデーションと同じです。ModelForm ではそれに加えて最後に、「モデルのバリデーション（モデルに定義されたバリデーション）」が実行されます。

特に最後の「`validate_unique()`」メソッドが重要で、これはモデルの各フィールドに定義された「`unique=True`」の制約にしたがってレコードがユニークになっているかどうかをチェックしてくれます。このユニーク制約チェックは、フォームの「`_validate_unique`」というメンバ変数が `True` になっていることが発動条件になっているのですが、ModelForm の親クラスである「`django.forms.models.ModelForm`」の `clean()` メソッドがこれを `True` に更新していることで最後に `validate_unique()` が実行される、という非常にややこしい仕組みになっています。

▼リスト 8.4 django.forms.models.ModelForm#clean

```
def clean(self):
    self._validate_unique = True
    return self.cleaned_data
```

それを踏まえると、モデルの登録や変更を伴うフォームでは、このユニーク制約チェックは非常に便利なので、次のように親クラスの `clean()` メソッドを明示的に呼び出した方がよいでしょう。

▼リスト 8.5 accounts.forms.RegisterForm#clean

```
def clean(self):
    # 明示的に親クラスの clean() を呼び出す
    super().clean()
    password = self.cleaned_data['password']
    password2 = self.cleaned_data['password2']
    if password != password2:
        raise forms.ValidationError("パスワードと確認用パスワードが合致しません")
```

しかしながらモデルの登録・変更を伴わないフォームでは、モデルのフィールドの定義次第ではユニーク制約に引っ掛かってしまうケースがあるため、そのような場合には親クラスの `clean()` を呼び出さないようにするなどのテクニックが必要になります。

ややこしい説明を長々と書きましたが、`ModelForm` にバリデーションメソッドを記述する際には通常のフォームと同様に、

- `clean_<フィールド名>` … フィールド単体のバリデーション
- `clean` … 複数フィールド間の相関チェックやデータベースとの整合性チェック

というメソッドを用意すればよいというのは変わりありません。 *3

最後に、ビューから `ModelForm` を使ったフォームを利用する方法について説明します。モデルを登録する場合と更新する場合で、`ModelForm` を継承したフォームをインスタンス化する方法が少し異なります。

登録の場合は、通常のフォームと同じくリクエストオブジェクトの属性の GET オブジェクトや POST オブジェクトをコンストラクタ引数に渡して初期化します。

```
form = RegisterForm(request.POST)
```

`ModelForm` には `save()` メソッドが用意されており、対象のモデルをデータベースに保存することができます。

*3 もし必要なら、複数フィールド間のユニーク制約 `validate_unique` のチェックも `clean()` に書けばよいでしょう。

```
user = form.save()
```

しかしこれではうまくいかない場合があります。モデルを登録する際、POST で送られてきたデータだけではモデルの登録に必須の情報が不足している場合があります。そのような場合でもエラーが出ないように、いったんモデルのオブジェクトをフォームから取り出して、データを補足してから保存してあげるというパターンがよく使われます。それが次の例に示した「`save(commit=False)`」です。この時点ではまだモデルのオブジェクトはデータベースに保存や更新はされていません。

```
user = form.save(commit=False)
user.set_password(form.cleaned_data['password'])
user.save()
```

1 行目で返された `user` オブジェクトに設定されたパスワードは平文（入力されたままの文字列）になっていて、そのまま保存してしまうと次回ログインするときにパスワードが合致しなくなってしまいます。そこで 2 行目の `set_password()` でパスワードをハッシュ化してセットしてあげる必要があるのです。そして 3 行目でようやくデータベースに保存、という流れになります。

一方、モデルを更新する場合のフォームのインスタンス化は次のようにおこないます。まず更新したいモデルのオブジェクトをデータベースから取得し、「`instance`」という名前のキーワード引数でフォームのコンストラクタに渡すことで、更新前のデータをベースにしてリクエストの入力データが上書きされるという仕組みになっています。次の例は、ユーザーの氏名やメールアドレスを変更するためのプロフィール変更画面のビューで利用する `ProfileForm` を想定したサンプル実装です。

```
user = User.objects.get(pk=request.user.id)
form = ProfileForm(request.POST, instance=user)
```

8.7 まとめ

- フォームは `forms.py` に書く
- フォームで入力データをクリーンにしよう
- フォームにはフィールド定義とバリデーションメソッドを記述する
 - フィールドには適切な `Field` クラスを使う
 - バリデーションには「`clean_<フィールド名>`」と「`clean`」という名前の 2 種類のメソッドを用意する
- `ModelForm` を使うことでモデルのフィールド定義を再利用することができる
- POST リクエストには CSRF 対策として「`{% csrf_token %}`」タグを利用する

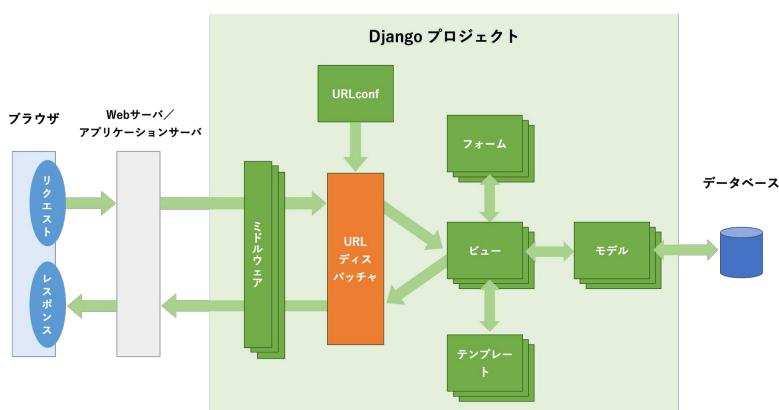
第9章

ミドルウェア (Middleware)

9.1 概要

ミドルウェアは、Django の主要な機能（モデル・テンプレート・ビュー）と Web サーバ／アプリケーションサーバの中間に位置し、ビューを出入りするすべてのリクエストとレスポンスをフックすることができます。さまざまなミドルウェアが組み込みで用意されており、リクエストやレスポンスにセキュリティに関する設定をしてくれるものもあれば、メッセージフレームワークという便利な機能を提供してくれるものもあります。そこから必要なものを設定ファイル（settings.py）に加えることで簡単に利用することができますし、自作して追加することも可能です。ミドルウェアの設定は必須ではありませんが、用途に合わせて積極的に使っていくことをお勧めします。

ここで再度、第2章「アーキテクチャ」に掲載した図を見てください。ビューに入る前に必ずすべてのミドルウェアが処理され、レスポンスをブラウザに返す前にも必ずすべてのミドルウェアが処理されます。なのでサイト全体に、リクエストやレスポンスを使用して一律に何らかの機能を加えたい場合などに利用されます。ちなみにリクエストやレスポンスに設定した属性は、セッションなどの仕組みを利用しない限り、次のリクエストには引き継がれないので注意が必要です。



▲図 9.1 ミドルウェアは Django プロジェクトの入口と出口に位置する

前述の通り、どのミドルウェアを有効にするかは設定ファイルの「MIDDLEWARE」にリスト形式で列举しますが、記載した順番通りにミドルウェアが実行されます。この順序は非常に重要で、記載する順番を間違えるとうまく動作しないこともありますので注意してください。たとえば、セッションを有効にしないと動作しない AuthenticationMiddleware は、セッションを有効化にするためのミドルウェアである SessionMiddleware よりも後に設置しなければいけない、といった具合です。

便利なミドルウェアですが、最後に大事な注意点がひとつあります。すべてのミドルウェアがリクエストの度に毎回実行されるので、あまり重い処理を書かないように気を付けてください。可能な限りデータベースへのアクセスはおこなわず、もし必要なら「`django.utils.functional.SimpleLazyObject`」を使った遅延評価やキャッシュを使うようにしましょう。

9.2 主なミドルウェアの役割

以降で、Django プロジェクトを作成したときにデフォルトで設定されるミドルウェアをひとつずつ紹介していきます。プロジェクト初期状態の設定ファイル (`settings.py`) では次の 7 つのミドルウェアが設定されています。

▼リスト 9.1 プロジェクト初期状態の `settings.py`

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',          # (1)
    'django.contrib.sessions.middleware.SessionMiddleware',   # (2)
    'django.middleware.common.CommonMiddleware',              # (3)
    'django.middleware.csrf.CsrfViewMiddleware',             # (4)
    'django.contrib.auth.middleware.AuthenticationMiddleware',# (5)
    'django.contrib.messages.middleware.MessageMiddleware',  # (6)
    'django.middleware.clickjacking.XFrameOptionsMiddleware',# (7)
]
```

9.2.1 SecurityMiddleware

リクエストおよびレスポンスへのセキュリティ強化をおこないます。その他、「HTTP」でアクセスした場合に「HTTPS」で始まる URL に自動的にリダイレクトするようにレスポンスを返すための機能もあり（デフォルトではオフ）、設定ファイルに次の設定をおこなうことで機能をオンにすることができます。

```
SECURE_SSL_REDIRECT = True
```

入れておいて損はないセキュリティ強化のためのミドルウェアなので、何も考えず設定

しておくのがよいかと思います。

9.2.2 SessionMiddleware

セッションを有効化するためのミドルウェアです。有効化する場合は設定ファイルの「INSTALLED_APPS」に「django.contrib.sessions」を入れておくこともお忘れなく。

デフォルトの設定ではセッションの格納先はデータベース（「django_session」テーブル）になっています。このミドルウェアが実行されると、Cookie から「sessionid」というキーで保管されたセッション ID を取り出し、データベースからセッション ID に紐付いたセッションデータを取得して、リクエストオブジェクトに「session」という属性でセッションオブジェクトをセットします。

セッションの使い方は、たとえば次のようになります。

```
language = request.session.get('_language')

if language is not None:
    request.session['_language'] = 'ja'
```

パフォーマンス向上のために、セッションの格納先をデータベースからキャッシュ（オンメモリ）に変更することも可能です。

```
SESSION_ENGINE = 'django.contrib.sessions.backends.cache'
```

しかしながら公式ドキュメント^{*1}に注意書きがされている通り、この設定はキャッシュバックエンドに「Memcached」^{*2}を利用している場合にのみするべきです。

セッションをまったく使わないという場合にはこのミドルウェアは利用しなくてよいですが、セッションが有効になっていないと利用できないミドルウェアもあるため、迷ったら有効化しておく、でよいでしょう。

^{*1} <https://docs.djangoproject.com/ja/2.2/topics/http/sessions/#using-cached-sessions>

^{*2} <https://docs.djangoproject.com/ja/2.2/topics/cache/#memcached>

9.2.3 CommonMiddleware

ユーザーエージェントによるアクセス制限や URL リライティングなど、いくつかの便利な機能を提供します。公式ドキュメント^{*3}にも「最低でも CommonMiddleware を使うことを強くお勧めします」と書かれているように、このミドルウェアの有効化は必須と考えてよいでしょう。

なおこのミドルウェアを導入すると、デフォルトで URL 末尾に「/」を付与する機能が有効化されます（「APPEND_SLASH」のデフォルト値が True のため）。これにより、リクエストされた URL がスラッシュで終わっておらず、URLconf にマッチするパターンが見つからない場合に、末尾に「/」を追加した URL に自動的にリダイレクトしてくれます。もし不要であれば OFF にしてしまいましょう。また、URL の先頭に「www.」がない場合に「www.」で始まる同じ URL にリダイレクトしてくれる機能はデフォルトでオフになっているので、必要に応じて「PREPEND_WWW」を True に設定するとよいでしょう。

9.2.4 CsrfViewMiddleware

第8章「フォーム (Form)」の「CSRF 対策」で説明した CSRF トークンを検証するために必要なミドルウェアです。

POST リクエスト^{*4}ごとに <input type="hidden" name="csrfmiddlewaretoken" value="...">> で送られてきた CSRF トークンの値を検証し、Cookie（「CSRF_USE_SESSIONS」の設定次第ではセッション）に格納していた値と異なっていた場合に 403 エラーを返してくれます。

9.2.5 AuthenticationMiddleware

リクエストのたびに、リクエストオブジェクトの「user」属性にユーザー情報をセットします。その際、ユーザーがログイン済みの場合にはセッションに保管されている User モデルのインスタンスを、未ログインの場合には AnonymousUser（匿名ユーザー）という特別なクラスのオブジェクトをセットします。

ここで、AuthenticationMiddleware の実装は次のようになっています。

^{*3} <https://docs.djangoproject.com/ja/2.2/topics/http/middleware/>

^{*4} 厳密には POST だけでなく PUT や DELETE リクエストについても検証がおこなわれます。

▼リスト 9.2 django.contrib.auth.middleware.AuthenticationMiddleware

```
class AuthenticationMiddleware(MiddlewareMixin):
    def process_request(self, request):
        ... (略) ...
        request.user = SimpleLazyObject(lambda: get_user(request))
```

この SimpleLazyObject の遅延評価により `get_user()` の処理は `request.user` にアクセスされて初めて動作するようになっているため、リクエストごとに重い処理が毎回実行されないような仕組みになっています。

なおこのミドルウェアはセッションを必要とするので、(2) の `SessionMiddleware` よりも後ろに設定する必要があります。

9.2.6 MessageMiddleware

画面に一度だけ出力する用途で使ういわゆる「フラッシュメッセージ」の機能を提供するためのミドルウェアです。この機構は「メッセージフレームワーク」と呼ばれます。よく使うので、章末の「9.4 ベストプラクティス 8：メッセージフレームワークを使う」で詳しく説明します。

9.2.7 XFrameOptionsMiddleware

クリックジャッキング対策のためのミドルウェアで、「X-Frame-Options」ヘッダをすべてのレスポンスに設定します。

9.3 ミドルウェアの書き方

Django ではバラエティ豊かなミドルウェアが数多く提供されていますが^{*5}、これらを拡張したり、新たに自作したりしたい場合があるかと思います。その場合は、リスト 9.3 のように決まった書き方をする `__init__()` と `__call__()` メソッドを持つ単純なクラスを実装することでミドルウェアを簡単に自作することができます。^{*6}

^{*5} <https://docs.djangoproject.com/ja/2.2/ref/middleware/>

^{*6} `django.utils.deprecation.MiddlewareMixin` を継承する書き方もありますが、非推奨にマークされているため今のうちに新しい書き方をしておくべきでしょう

▼リスト 9.3 シンプルなミドルウェアクラスの例

```
class SimpleMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        # リクエストへの前処理をここに記述

        response = self.get_response(request)

        # レスポンスへの後処理をここに記述

        return response
```

リスト 9.3 のコードをベースに、URL ディスパッチャの前段でリクエストに前処理を加えた場合には「# リクエストへの前処理をここに記述」の部分で引数の `request` に任意の処理を、ビュー呼び出し後にレスポンスに後処理をおこないたい場合には「# レスポンスへの後処理をここに記述」の部分でその直前に取得した `response` に任意の処理を加えるように実装します。そして、URL ディスパッチャのあと且つビューが呼び出される直前に処理をインサートしたい場合には「`process_view`」という名前の特別なメソッドを実装します。

次のコードはちょっとしたアクセス制限をするためのミドルウェアの実装例です。

▼リスト 9.4 accounts/middlewares.py

```
from django.core.exceptions import PermissionDenied
from django.urls import reverse

class SitePermissionMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        # リクエストへの前処理をここに記述

        response = self.get_response(request)

        # レスポンスへの後処理をここに記述

        return response

    def process_view(self, request, view_func, view_args, view_kwargs):
        """ビューを呼び出す直前に呼び出される処理"""
        has_site_permission = False
        if request.user.is_superuser or request.user.is_staff:
            has_site_permission = True

        admin_index = reverse('admin:index')
        # 権限を持っていないユーザーが「/admin/」配下にアクセスしたら 403 エラー
        if request.path.startswith(admin_index):
            if not has_site_permission:
                raise PermissionDenied

        request.user.has_site_permission = has_site_permission
```

`process_view()` では、後続の処理を続ける場合には戻り値を返さず（あるいは明示的に `None` を返す）、逆に処理を中断する場合には `django.http.response.HttpResponse` を返すようにします。`process_view()` から漏れた例外は URL ディスパッチャがエラーハンドリングをしてくれるので、リスト 9.4 のように `PermissionDenied` などの特殊な例外クラスを使うことも可能です。

作成したミドルウェアはもちろん、設定ファイルの「`MIDDLEWARE`」に追加しないと適用されないので注意しましょう。

このように、ミドルウェアを使うとサイト全体に渡るアクセス制限を簡単に適用することができます。

9.4 ベストプラクティス 8：メッセージフレームワークを使う

ユーザーが何らかのアクションを実行したあとに見せる画面に「ログインしました。」「商品を追加しました。」などといった通知メッセージを表示するケースはよくあるかと思います。そのような場合に簡単に使えて重宝するのが `MessageMiddleware` を使ったメッセージフレームワークです。この機能で利用できるメッセージは「フラッシュメッセージ」ともいわれ、一度表示させると保存領域から消去されてしまう一過性のメッセージです。逆にいえば表示されるまでは保存領域に持ち続けているため、テンプレート側にフラッシュメッセージを表示させる記述が抜けていると思わぬタイミングでメッセージが出ることがあります。そこで、`base.html` に記述してテンプレート全体に適用しておくか、`include` タグでメッセージを出す可能性があるテンプレートに適用しておくのがよいでしょう。

メッセージフレームワークを有効化する手順は次の通りなのですが、これらは Django プロジェクトを `startproject` で初期構築したときにデフォルトで設定済みなので、特に何も弄っていなければそのまま使い始めることができます。

- 設定ファイル (`settings.py`) の「`INSTALLED_APPS`」に「`django.contrib.messages`」を追加
- 同じく「`MIDDLEWARE`」に以下を追加
 - `django.contrib.sessions.middleware.SessionMiddleware`
 - `django.contrib.messages.middleware.MessageMiddleware` (SessionMiddleware より後に記載)
- 同じく「`TEMPLATES.OPTIONS.context_processors`」に「`django.contrib.messages.context_processors.messages`」を追加

フラッシュメッセージを保存する領域の初期設定は「`django.contrib.messages.storage.fallback.FallbackStorage`」になっていて、パフォーマンスのために基本的にCookieを使用し、Cookieで対応できないメッセージについてはセッションを使うという設定になっています。しかしながらこの設定では、一部の場合でリダイレクトしたときにフラッシュメッセージが表示されない場合があるため、セッションにのみ保存するように設定ファイルの「MESSAGE_STORAGE」の設定を次のように書き換えておくことを推奨します。

▼リスト 9.5 フラッシュメッセージの保存領域をセッションに変更 (config/settings.py)

```
MESSAGE_STORAGE = 'django.contrib.messages.storage.session.SessionStorage'
```

フラッシュメッセージを利用する場合は次のようにします。

▼リスト 9.6 ビュー側の実装例 (accounts/views.py)

```
from django.contrib import messages
from django.urls import reverse
from django.views import View

class LoginView(View):
    def post(self, request, *args, **kwargs):
        ... (略) ...
        # フラッシュメッセージを画面に表示
        messages.info(request, "ログインしました。")
        return redirect(reverse('shop:index'))
```

▼リスト 9.7 テンプレート側の実装例 (templates/_message.html)

```
{% if messages %}
<div class="ui relaxed divided list">
    {% for message in messages %}
    <div class="ui {% if message.tags %}{{ message.tags }}{% endif %} message">
        {{ message }}
    </div>
    {% endfor %}
</div>
{% endif %}
```



▲図 9.2 フラッシュメッセージの表示例

フラッシュメッセージではメッセージ文字列に加えて「タグ」を付けることができます。メッセージフレームワークのメッセージレベルには「error」「warning」「success」「info」などが用意されていますが、リスト 9.6 の `info()` メソッドでは（メソッド名と同じく）「info」というメッセージレベルがタグとして付けられたフラッシュメッセージが保存されます。メッセージを出力する際には リスト 9.7 に示したように各メッセージから「`{{ message.tags }}`」という形で参照することができます。

ちなみにちょっとした TIPS ですが、

```
messages.error(request, "想定外のエラーが発生しました。", extra_tags='danger')
```

と「`extra_tags`」に任意の文字列を渡すことで、「`{{ message.tags }}`」の出力結果に文字列を追加することができます（この場合は「`danger error`」が出力されます）。これをを利用して、アラートメッセージの CSS クラスとして「`alert-danger`」「`alert-warning`」「`alert-info`」「`alert-success`」が求められる Bootstrap4 *7 の仕様に合わせることもできます。

9.5 まとめ

- 見逃されがちなミドルウェアだけど、実は縁の下の力持ち
 - 普段何となく使っている機能が実はミドルウェアのお陰だったりする
- プロジェクト作成時に設定されるミドルウェアはとりあえずそのまま使うのが吉

*7 <https://getbootstrap.com/docs/4.3/components/alerts/>

第 10 章

設定オブジェクトと設定ファイル (`settings.py`)

10.1 概要

本章では、設定オブジェクト (`django.conf.settings`) と設定ファイル（プロジェクト作成時に生成される `settings.py`）について説明します。

まず、設定オブジェクトは Django の起動時に次のようにインスタンス化されるシングルトンなグローバルオブジェクトです。

▼リスト 10.1 `django/conf/__init__.py`

```
settings = LazySettings()
```

このオブジェクトは、Django のデフォルト設定ファイル (`django.conf.global_settings.py`) を読み込んだあと、プロジェクト作成時に生成される設定ファイル (`settings.py`) を読み込んで設定を上書きします。この設定ファイル (`settings.py`) には、Django のデフォルト設定とは異なるプロジェクト固有の設定値や自作のアプリケーションで利用する独自の変数を記述しておきます。

Django で利用できる設定変数とそのデフォルト値の一覧が公式ドキュメント ^{*1} に掲載されていますので、一読をお勧めします。

ビューやモデルから設定値を参照する際は、`django.conf.settings` はオブジェクトなので、

```
from django.conf.settings import LOGIN_URL
```

という形ではインポートすることができません。また、

^{*1} <https://docs.djangoproject.com/ja/2.2/ref/settings/>

```
from config import settings
print(settings.LOGIN_URL)
```

と設定ファイルにアクセスするのではなく、

```
from django.conf import settings
print(settings.LOGIN_URL)
```

のように設定オブジェクトにアクセスしないと Django のデフォルト設定が参照できないので注意が必要です（通常、設定ファイルに直接アクセスすることはありません）。

ところで、この `django.conf.settings` の「LazySettings」クラスは何が「Lazy」（怠け者）なのでしょうか。たとえば「誰かに結果を出せと言わされてからようやく仕事をやり始める人」をイメージしてもらえると分かりやすいかもしれません。LazySettings も同じように、その設定値にアクセスされてから慌てて設定ファイルを読み込み始めるというまさに怠け者なのです。裏を返せば、最初にその属性値にアクセスされるまでは設定をいくらでも追加・更新できるというカスタマイズしやすい特徴を持っているともいえるでしょう。

Django が起動されて設定ファイルの読み込みが終わったあとは（`settings` の属性に一度でもアクセスされてしまうと）、ビューなどで設定値を変更しようとしてもエラーが発生して変更できません。なので基本的には、設定ファイルの中で初期設定を済ませてしまします。

難しいことはさておき、設定ファイル（`settings.py`）にはプロジェクト固有の設定を記述し、ビューやモデルからは設定オブジェクト（`django.conf.settings`）を参照する、とだけ覚えておけばよいでしょう。以降で重要な設定を解説していきます。

10.2 インストールするアプリケーション一覧

「`INSTALLED_APPS`」に、インストールするアプリケーションをリスト形式で列挙します。自作のアプリケーションであれば

```
INSTALLED_APPS = [
    ... (略) ...
    'accounts.apps.AccountsConfig',
    'shop.apps.ShopConfig',
]
```

のように、各アプリケーションディレクトリ直下の `apps.py` に書かれた `AppConfig` クラスのサブクラスを追加します。 `AppConfig` のサブクラスは、アプリケーション作成時にリスト 10.2 のような最低限の定義が記述されているはずなのでそれを使います。

▼リスト 10.2 アプリケーション作成時の apps.py の例 (accounts/apps.py)

```
from django.apps import AppConfig

class AccountsConfig(AppConfig):
    name = 'accounts'
```

ところでよく、

```
INSTALLED_APPS = [
    ... (略) ...
    'accounts',
    'shop',
]
```

のようにパッケージ名だけを「INSTALLED_APPS」に追記する例を見かけることがあるかと思います。これは少し古い書き方で、後方互換のためにパッケージ名でも指定できるようになっているようです。パッケージ名でアプリケーションを登録すると、apps.py の AppConfig のサブクラスが自動的に認識されないなどという微妙な問題^{*2} もありますので、「INSTALLED_APPS」には apps.py の AppConfig のサブクラスを追加することを推奨します。

また、「INSTALLED_APPS」には優先順があり、上に書いた方が優先されます。プロジェクトの初期状態では次の 6 つのアプリケーションが設定済みですが、これらの下に追加していくべきよいでしょう。

1. django.contrib.admin (管理サイト)
2. django.contrib.auth (認証システム)
3. django.contrib.contenttypes (コンテンツタイプフレームワーク)
4. django.contrib.sessions (セッションフレームワーク)
5. django.contrib.messages (メッセージフレームワーク)
6. django.contrib.staticfiles (静的ファイル管理フレームワーク)

10.3 デバッグ設定

「DEBUG」は、開発モードと本番モードを切り替えられるようにする設定です。開発時は True にしておくことで、エラー発生時に画面にデバッグ情報が出力されるなど開発に便利な機能が提供されます。また、開発時に便利に使える「django-debug-toolbar」(第 14 章で解説) や SQL 文のロギング (第 10 章で解説) など、「DEBUG」が True になって

^{*2} アプリケーションディレクトリ直下の __init__.py に「default_app_config = 'accounts.apps.AccountsConfig'」などという記述をすれば解決できます。

いなければ利用できない機能もあります。

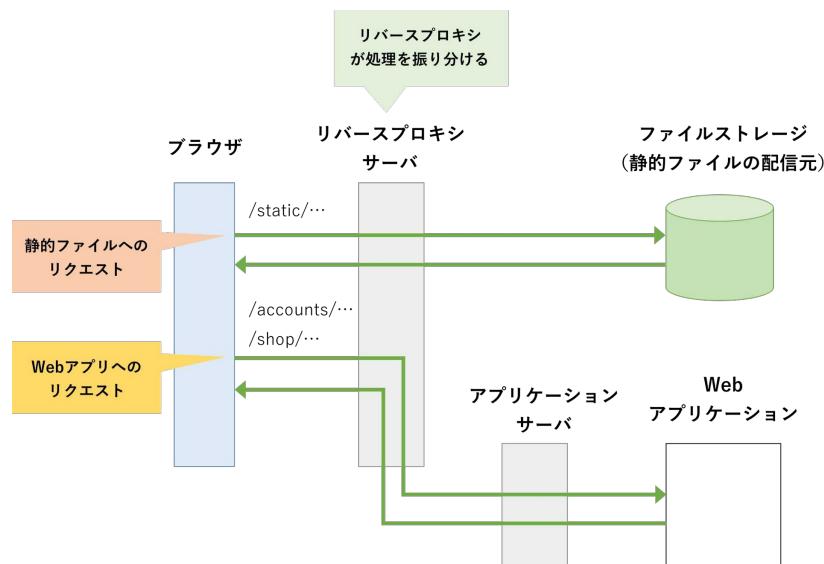
本番稼働時には、セキュリティ面を考慮して必ずこの「DEBUG」設定を False にしておくようにしましょう。

```
DEBUG = False
```

10.4 静的ファイル関連の設定

Web アプリケーションの世界では、CSS ファイルや JavaScript ファイル、画像ファイルなどのように、リクエストに応じて中身を変更せずにそのまま配信可能なファイルを「静的ファイル (static ファイル)」と呼びます。

単に静的ファイルをブラウザへ返すだけの処理をアプリケーションサーバで捌くと、無駄が多くなってしまいます。そこでアプリケーションサーバの前段に Nginx^{*3} に代表される「リバースプロキシ」と呼ばれるサーバを配置し、静的ファイルを返すだけの処理はリバースプロキシが担当し、Web アプリケーションの処理が必要なリクエストだけをアプリケーションサーバに振り分けることで効率よくリクエストを捌けるようにすることが多くなっています（下図 10.1 参照）。



▲図 10.1 リバースプロキシの仕組み

^{*3} <https://nginx.org/en/docs/>

またセキュリティなどの観点から、「静的ファイルの配信元」とプロジェクトで静的ファイルをバージョン管理する際の「プロジェクト内での置き場所」は別々にするケースが多いです。

以上を踏まえると、Django で静的ファイルを配信するためには、表 10.1 に示した 3 つの設定を最低限しておく必要があります。

▼表 10.1 静的ファイル配信に必要な設定

変数	説明
STATIC_URL	静的ファイル配信用のディレクトリで、URL の一部になる。 設定値はデフォルトの「/static/」のままでよい
STATICFILES_DIRS	アプリケーションに紐付かない静的ファイルの置き場所
STATIC_ROOT	静的ファイルの配信元。 <code>collectstatic</code> コマンドで静的ファイルを集約する際のコピー先でもある。 「STATICFILES_DIRS」とは別のディレクトリを指定する必要がある

上表 10.1 の設定例を次に示します。

▼リスト 10.3 静的ファイル関連の設定例 (config/settings.py)

```
STATIC_URL = '/static/'
STATICFILES_DIRS = [os.path.join(BASE_DIR, 'static')]
STATIC_ROOT = '/var/www/{}.static'.format(PROJECT_NAME)
```

ここでリスト 10.3 の設定をする前に、プロジェクトを配置したベースディレクトリ（たとえば「/home/ubuntu/mysite」や「/opt/webapps/mysite」など）を「BASE_DIR」、プロジェクト名（「mysite」など）を「PROJECT_NAME」などとして次のように定義しておくと何かと便利です。

```
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
PROJECT_NAME = os.path.basename(BASE_DIR)
```

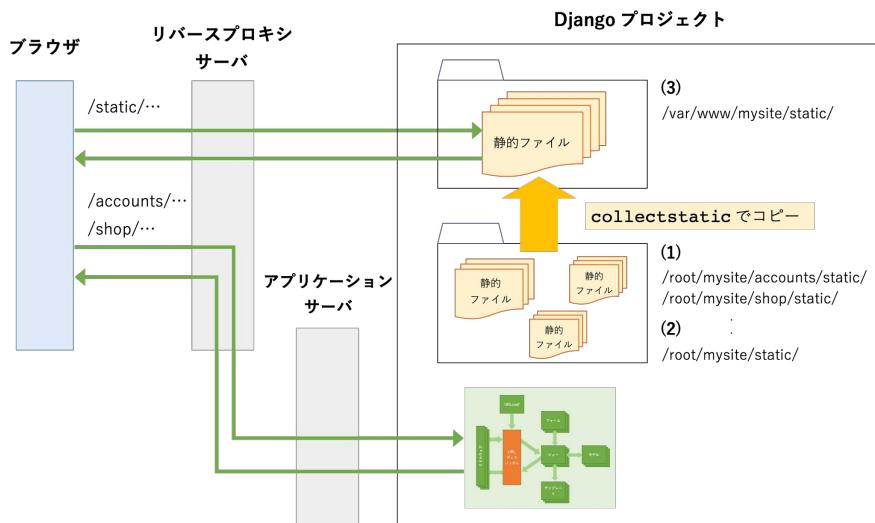
リスト 10.3 のように設定した場合、静的ファイルをプロジェクト内に保管しておくための置き場所は次の (1) や (2) になります（ここで (1) は Django が指定するデフォルトの場所で、(2) は「STATICFILES_DIRS」の場所です。ただし、第 3 章のベストプラクティスから (1) に静的ファイルを置くことは推奨しません）。

```
<ベースディレクトリ>
| -- <アプリケーションディレクトリ>
|   |
|   | -- static
|   |
|   |   | -- <アプリケーション名>
|   |   |   `-- (1)   例: images/no-image.png
|
| -- static
|   `-- (2)   例: images/logo.png
.
```

```
/var/www/<プロジェクト名>
|
`-- static
  `-- (3)
```

静的ファイル配信元である (3) の「STATIC_ROOT」は「DEBUG」が False の場合に必要になる設定で、すなわち本番環境で動作させることを想定した設定です（「DEBUG」が True の場合に設定していてもエラーになることはありません）。

runserver（開発用 Web サーバ：詳しくは 第 12 章 を参照）の静的ファイル自動配信機能は「DEBUG」が False の場合には動作しないため、runserver は (1) や (2) に置かれた静的ファイルを配信してくれません。その場合はリバースプロキシを使うなどして静的ファイルを自前で配信しなければならず、その配信元が (3) の「STATIC_ROOT」になると いうわけです（図 10.3 参照）。

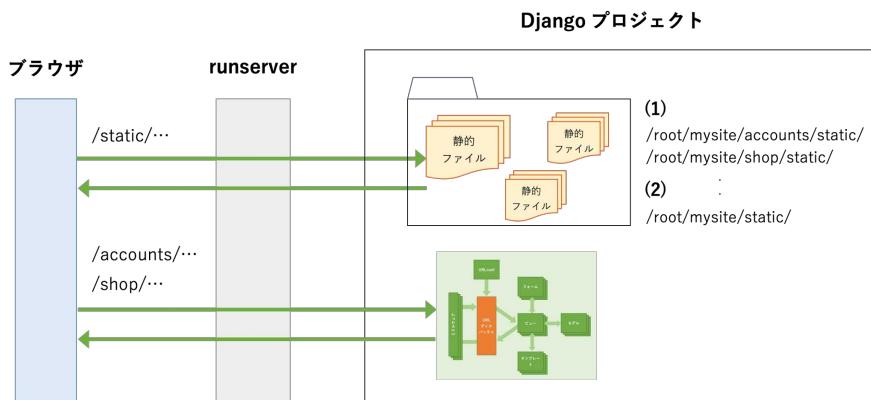


▲図 10.2 リバースプロキシによる静的ファイル配信例（DEBUG が False の場合）

したがって、「DEBUG」が False の場合には、リバースプロキシ側では(3)の配下の静的ファイルを「http(s)://<ホスト名>/<STATIC_URLの値>/...」との URL リクエストで参照できるように設定する必要があります、一方の Django プロジェクト側では(3)の配下に静的ファイルを集約する必要があります。そして静的ファイル集約のための管理コマンドが「collectstatic」です。次のコマンドを実行すると、(1) や (2) の静的ファイルが集約されて(3)にコピーされます。

```
$ python3 manage.py collectstatic
```

一方で「DEBUG」が True の場合には、runserver が、「http://localhost:8000/<STATIC_URLの値>/...」という URL へのアクセスに対して(1)や(2)に置かれた静的ファイルを配信してくれます(下図 10.3 参照)。この場合には「collectstatic」コマンドを事前に実行する必要はありません。



▲図 10.3 runserver による静的ファイル配信の仕組み (DEBUG が True の場合)

最後に、テンプレートで静的ファイルの配信用 URL を使って画像を表示する場合の実装例を挙げます。第 7 章で説明したように「static」タグを利用して静的ファイルの配信用 URL を取得しています。

```
{% load static %}

![Placeholder image for no image]({% static 'shop/images/no-image.png' %})
![Logo image]({% static 'images/logo.png' %})
```

10.5 メディアファイル関連の設定

Django では、静的ファイルのうち、（システム管理者を含めた）ユーザーがサイトを利用してアップロードする CSV や PDF、画像などのファイルを「メディアファイル」と呼んでいます。メディアファイルを配信する（ユーザーに見える状態にする）場合は、静的ファイルと同じようにファイルを配信する仕組みが必要になります。本番環境においては、静的ファイルと同様にメディアファイルについても、アプリケーションサーバで捌かずにリバースプロキシなどで捌くことで負荷を減らす方式が取られることが多いです。

本番環境用（「DEBUG」が False の場合）のメディアファイル関連の設定例を次に示します。

▼リスト 10.4 メディアファイル関連の設定例（config/settings.py）

```
MEDIA_URL = '/media/'
MEDIA_ROOT = '/var/www/{}/media'.format(PROJECT_NAME)
```

macOS や Windows 上でユニットテストやちょっとした検証をする場合には、次のように「MEDIA_ROOT」にプロジェクト内のディレクトリを指定すればよいでしょう。

```
MEDIA_ROOT = os.path.join(BASE_DIR, 'media_root')
```

メディアファイルのアップロードは、たとえば「`FileField`」（ファイルを扱うためのフィールド）や「`ImageField`」（画像を扱うためのフィールド）^{*4} をフィールドとして持つ、たとえば次のようなモデルを用意すれば、

▼リスト 10.5 ImageField を使ったモデルの例（shop/models.py）

```
class Book(models.Model):
    """本モデル"""

    class Meta(object):
        db_table = 'book'

    title = models.CharField(verbose_name='タイトル', max_length=255)
    image = models.ImageField(verbose_name='画像', null=True, blank=True)
    ... (略) ...
```

1. 管理サイト（第 13 章参照）のレコード編集画面
2. ModelForm を使った画面

^{*4} ImageField を利用するには Pillow ライブラリをインストールする必要があります。

などから簡単に利用することができます。1. で画面からアップロードしたり、2. でモデルを save() すると、「MEDIA_ROOT」で指定した配信先に直接ファイルが配置されます。ちなみにこの「MEDIA_ROOT」の設定を忘れる、(global_settings.py でのデフォルト値が””になっているため) ベースディレクトリの直下にメディアファイルがアップロードされてしましますので注意してください。

1. の場合は（モデルと admin.py を触るだけなので）特に問題はないと思いますが、2. の ModelForm を使った画面からのファイルアップロードを実装するにあたってはいくつか注意点があるので、次に挙げておきます。

1. <form> 要素に enctype="multipart/form-data" 属性を加える ^{*5}
2. ModelForm をインスタンス化する際の第二引数に request.FILES を指定する
3. FileField や ImageField の「upload_to」オプションを使うとアップロード先のディレクトリやファイル名を動的に変更できる ^{*6}
4. Django サイトを稼働させているサーバを冗長構成にしている場合は、S3 などのストレージサービスや NFS を利用する ^{*7}

モデルの FileField や ImageField のフィールドからは、アップロードしたメディアファイルの配信用 URL を取得することができます。ここで、テンプレートから配信用 URL を取得して画像を表示する場合の実装例をリスト 10.6 に挙げます。

▼リスト 10.6 templates/shop/book_list.html (抜粋)

```
{% load static %}

...
(略) ...

{% if book.image %}
    
{% else %}
    
{% endif %}
```

この例では、ファイルをアップロードしていない場合にエラーが出ないように「book.image」が存在する場合のみ配信用 URL を取得するようにし、未アップロードの場合は静的ファイルのデフォルト画像 (no-image.png) を表示するようにしています。

ところで開発者からすれば、開発時（「DEBUG」が True の場合）にわざわざメディアファイル配信の設定をするのは少し面倒です。そこで開発時にもメディアファイルの動作確認ができるように、runserver で静的ファイルと同様にメディアファイルの配信をして

^{*5} <https://docs.djangoproject.com/ja/2.2/topics/http/file-uploads/#basic-file-uploads>

^{*6} https://docs.djangoproject.com/ja/2.2/ref/models/fields/#django.db.models.FileField.upload_to

^{*7} 「django-storages」が非常に有用です。 <https://github.com/jschneier/django-storages>

くれる仕組みがあります。設定の手順としては、config/urls.py に次の末尾の URL 設定を加えるだけです。

▼リスト 10.7 config/urls.py

```
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = [
    ...
]
urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

次のようなサンプルをたまに見かけますが、static 関数の内部で「DEBUG」が True でないと動作しないようにチェックしているため、「if DEBUG:」の記述は不要です。

```
if DEBUG:
    urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

10.6 データベースの設定

「DATABASES」にはデータベースへのコネクションやその他オプションに関する設定を記述します。プロジェクト作成時にデフォルトで設定される SQLite のほか、MySQL や PostgreSQL などの主要なデータベースを利用することができる。組み込みでサポートしているデータベースエンジンは次の 4つです。

- `django.db.backends.postgresql` (PostgreSQL)
- `django.db.backends.mysql` (MySQL)
- `django.db.backends.sqlite3` (SQLite)
- `django.db.backends.oracle` (Oracle)

なお、組み込みでサポートされていない Microsoft SQL Server などのいくつかのデータベースについては、サードパーティ製のライブラリ (Django パッケージ) をインストールすることで対応が可能です。^{*8} しかしながら、ライブラリによっては新しいバージョンのデータベースに対応していないなどの問題があるケースもありますので、利用にあたっては注意が必要です。

^{*8} <https://docs.djangoproject.com/ja/2.2/ref/databases/#using-a-3rd-party-database-backend>

設定項目

データベースの設定には、次表のような項目を設定することができます。

▼表 10.2 データベース設定の項目（一部）

項目	説明	設定例
ENGINE	データベースエンジン	'django.db.backends.mysql'
NAME	データベース名	'mysite'
USER	接続するユーザー名	'mysiteuser'
PASSWORD	接続するユーザーのパスワード	'mysiteuserpass'
HOST	接続先ホスト。省略した場合は 'localhost'	'localhost'
PORT	接続先ポート。文字列か数値で指定する。 省略した場合はデフォルトのポート番号が使われる	'3306'
ATOMIC_REQUESTS	トランザクションの有効範囲をリクエストの 開始から終了までにするかどうかを指定	True (デフォルトは False)
OPTIONS	各種オプション	(後述)

4つのデータベースエンジンのうち、MySQL、PostgreSQL、SQLite の設定例を以降に示します。

MySQL の場合

▼リスト 10.8 MySQL の DATABASES 最小設定例 (config/settings.py)

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'mysite',
        'USER': 'mysiteuser',
        'PASSWORD': 'mysiteuserpass',
    }
}
```

▼リスト 10.9 MySQL の DATABASES 設定例 (config/settings.py)

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'mysite',
        'USER': 'mysiteuser',
        'PASSWORD': 'mysiteuserpass',
        'HOST': 'localhost',
        'PORT': '3306',
        'ATOMIC_REQUESTS': True,
        'OPTIONS': {
            'sql_mode': 'TRADITIONAL,NO_AUTO_VALUE_ON_ZERO',
        }
    }
}
```

```
    },
}
```

データベースがトランザクションをサポートしている場合は^{*9}、トランザクションの有効範囲についての設定を追加することができます。トランザクションの有効範囲をリクエストの開始から終了までにする設定については、第6章の「6.8 単体のオブジェクトを保存・更新・削除する」を合わせてご参照ください。

そのほか、「OPTIONS」にさまざまなオプション設定をすることができます。たとえば、トランザクションの分離レベル（「isolation_level」）や MySQL の場合の SQL モード（「sql_mode」）の設定が可能です。これらのオプションについては、デフォルトの設定値から変更したい場合に記述し、やむを得ない場合を除いて Django 側だけでなくデータベース側にも同じ設定をしておくのが望ましいでしょう。ちなみに MySQL では、SQL モードが「厳密モード」に設定されていないと、たとえば varchar 型のカラムの最大長を超える値で登録や更新をしようとした際にエラーが発生しないだけでなく、最大長以降の文字が勝手に切り捨てられて保存されてしまうといった微妙な振る舞いをするため、大変不便です。そのため MySQL では、SQL モードを「厳密モード」である「STRICT_TRANS_TABLES」や「STRICT_ALL_TABLES」、あるいはそれらを含む組み合わせモードの「TRADITIONAL」に設定し、データ更新時に桁数をオーバーする場合にエラーを出すようになります。^{*10} リスト 10.9 の「OPTIONS」の設定例は、SQL モードを最も厳しく設定したものになります。

なお、MySQL を利用する際はドライバをインストールする必要があります。Django 公式ドキュメント^{*11}では「mysqlclient」が推奨されていますので、pip でインストールしておきましょう。

```
$ pip3 install mysqlclient
```

PostgreSQL の場合

▼リスト 10.10 PostgreSQL の DATABASES 設定例 (config/settings.py)

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'mysite',
        'USER': 'mysiteuser',
```

^{*9} たとえば MySQL でストレージエンジンに「MyISAM」を利用している場合はトランザクションがサポートされません。

^{*10} <https://docs.djangoproject.com/ja/2.2/ref/databases/#setting-sql-mode>

^{*11} <https://docs.djangoproject.com/ja/2.2/ref/databases/#mysql-db-api-drivers>

```
'PASSWORD': 'mysiteuserpass',
'HOST': 'localhost',
'PORT': '5432',
}
}
```

「`django.db.backends.postgresql_psycopg2`」は Django 1.9 以降、「`django.db.backends.postgresql`」にリネームされました。

データベースの設定では通常、「HOST」を省略すると「localhost」として扱われますが、PostgreSQL では「HOST」を空文字で設定すると「`django.db.utils.OperationalError: FATAL: Peer authentication failed for user "mysiteuser"`」などといったエラーが出る場合があるので、設定値は省略せずに適切に設定する方がよいでしょう。

なお PostgreSQL を利用する際は、ドライバとして「`psycopg2-binary`」をインストールする必要があります。

```
$ pip3 install psycopg2-binary
```

SQLite の場合

▼リスト 10.11 SQLite の DATABASES 設定例 (config/settings.py)

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

SQLite の場合は「NAME」に使用するファイルの絶対パスを指定します。「USER」「PASSWORD」「HOST」「PORT」の項目は SQLite では使用しません。

ちなみに SQLite はセキュリティやスケールの問題があるため、特に大規模なシステムでの本番利用は推奨されません。

10.7 ロギングの設定

「LOGGING」はロギング（ログ出力）に関する設定です。プロジェクト作成時には設定ファイルにはロギングの設定は書かれておらず、Django 起動時に `django.utils.log.py` の「`DEFAULT_LOGGING`」の設定が読み込まれます。

本番稼働時はそのままでは使い物にならないので、リスト 10.12 のように書き換えてファイルにログを出力するような設定に変更します。

▼リスト 10.12 本番稼働時の LOGGING 設定例 (config/settings.py)

```
LOGGING = {
    # バージョンは「1」固定
    'version': 1,
    # 既存のログ設定を無効化しない
    'disable_existing_loggers': False,
    # ログフォーマット
    'formatters': {
        # 本番用
        'production': {
            'format': '%(asctime)s [%%(levelname)s] %(process)d %(thread)d '
                      '%(pathname)s:%(lineno)d %(message)s'
        },
    },
    # ハンドラー
    'handlers': {
        # ファイル出力用ハンドラー
        'file': {
            'level': 'INFO',
            'class': 'logging.FileHandler',
            'filename': '/var/log/{}app.log'.format(PROJECT_NAME),
            'formatter': 'production',
        },
    },
    # ロガー
    'loggers': {
        # 自作アプリケーション全般のログを拾うロガー
        '': {
            'handlers': ['file'],
            'level': 'INFO',
            'propagate': False,
        },
        # Django 本体が出すログ全般を拾うロガー
        'django': {
            'handlers': ['file'],
            'level': 'INFO',
            'propagate': False,
        },
    },
}
```

ファイル出力用ハンドラーとして「`logging.FileHandler`」の代わりに「`logging.handlers.RotatingFileHandler`」や「`logging.handlers.TimedRotatingFileHandler`」を使うと、ログファイルをローテーションすることができて非常に便利です。

開発時はログをコンソールに出力し、発行されるクエリをログ出力されるようにしてみます。リスト 10.13 は開発時のロギング設定の一例です。

▼リスト 10.13 開発時の LOGGING 設定例 (config/settings.py)

```
LOGGING = {
    # バージョンは「1」固定
    'version': 1,
    # 既存のログ設定を無効化しない
    'disable_existing_loggers': False,
```

```

# ログフォーマット
'formatters': {
    # 開発用
    'develop': {
        'format': '%(asctime)s [%(levelname)s] %(pathname)s:%(lineno)d '
                  '%(message)s'
    },
},
# ハンドラ
'handlers': {
    # コンソール出力用ハンドラ
    'console': {
        'level': 'DEBUG',
        'class': 'logging.StreamHandler',
        'formatter': 'develop',
    },
},
# ロガー
'loggers': {
    # 自作アプリケーション全般のログを拾うロガー
    '': {
        'handlers': ['console'],
        'level': 'DEBUG',
        'propagate': False,
    },
    # Django 本体が出すログ全般を拾うロガー
    'django': {
        'handlers': ['console'],
        'level': 'INFO',
        'propagate': False,
    },
    # 発行される SQL 文を出力するための設定
    'django.db.backends': {
        'handlers': ['console'],
        'level': 'DEBUG',
        'propagate': False,
    },
},
}

```

このように、本番時の設定と開発時の設定が異なることはよくあります。しかしながら、環境が切り替わる際にいちいち設定値を書き換えるのはあまりよろしくありません。

```
if DEBUG:
```

などとして「DEBUG」の値によって設定値を書き分けてもよいのですが、もっと便利なベストプラクティスがあります。章末の「10.9 ベストプラクティス 9：個人の開発環境の設定は local_settings.py に書く」を参照してください。

「LOGGING」の設定に話を戻します。ロガー (loggers) に「」という設定を書くと、独自に追加したアプリケーション全般のログを拾うことができます。一方、「django」ロガーは Django 本体が出すエラー全般を拾います。もし SQL 文を出力したくなれば「django.db.backends」ロガーの部分をコメントアウトすればよいでしょう。ちなみに SQL 文の出力はパフォーマンスの観点から、「DEBUG」が True でないと出力されないよ

うになっています。

その他の組み込みで提供されているロガーは公式ドキュメント^{*12}にまとめられています。ログフォーマットに利用できるプレースホルダの一覧も公式ドキュメント^{*13}にまとめられていますので、ぜひ参考にしてください。

最後に、ビューでログを出力するサンプルコードをリスト 10.14 に示します。

▼リスト 10.14 ビューでのログ出力例

```
import logging
import time
from django.views.generic import View

logger = logging.getLogger(__name__)

class SampleView(View):
    def post(self, request, *args, **kwargs):
        start_time = time.time()
        logger.info("User({}) posted.".format(request.user.id))

        ... (略) ...

    logger.debug("Finished in {:.2f} sec.".format(time.time() - start_time))
    return render(request, 'sample.html')
```

ここで、ロガーを取得するための

```
logger = logging.getLogger(__name__)
```

という一文は決まり文句ですので覚えててしまいましょう。

リスト 10.14 の出力結果は次のようになります。

```
2019-04-01 00:20:33,234 [INFO] /root/mysite/shop/views.py:76 User(1) posted.
2019-04-01 00:20:34,621 [DEBUG] /root/mysite/shop/views.py:98 Finished in 1.43 sec.
```

なおログレベルには強い順に「CRITICAL」「ERROR」「WARNING」「INFO」「DEBUG」が用意されていて、本番稼働時には「INFO」以上、開発時には「DEBUG」以上のログを拾うようにするなど「LOGGING」の設定を使い分けるようにした方がよいでしょう。

^{*12} <https://docs.djangoproject.com/ja/2.2/topics/logging/#id3>

^{*13} <https://docs.python.org/ja/3.7/library/logging.html#logrecord-attributes>

10.8 その他の重要な設定

数ある設定の中から、特に重要な設定を紹介します。

10.8.1 TEMPLATES (テンプレートに関する設定)

「TEMPLATES」にはテンプレートに関する設定を記述します。

まず「BACKEND」には、テンプレートエンジンを指定します。デフォルトの DTL (Django Template Language) であれば「django.template.backends.django.DjangoTemplates」が設定されています。

「DIRS」には、ビューから指定されるテンプレート名に対してどのディレクトリを優先してテンプレートを探しに行くかという順番をリスト形式で指定します。次の例では、ベースディレクトリ直下の「templates」を最初の検索候補として指定しています（第3章の「3.4 ベストプラクティス1：分かりやすいプロジェクト構成」に対応した設定です）。また「APP_DIRS」にはテンプレートを探す際に各アプリケーションディレクトリ直下の「templates」ディレクトリを優先するかどうかを指定しますが、この例では「APP_DIRS」が True になっているのでこちらが優先されます。^{*14}

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, 'templates')],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

「OPTIONS」には、テンプレートから参照できる変数を渡すための context_processors が列挙されています。

10.8.2 LANGUAGE_CODE (言語コード)

画面に表示されるメッセージなどを多言語化するときの言語コードを指定することができます。

^{*14} 「APP_DIRS」を False にすると、管理サイト（第13章参照）のテンプレートが見つからなくなるなどの影響が出るため、この設定は True のままでよいでしょう。

デフォルトでは 'en-us' となっているので、

```
LANGUAGE_CODE = 'ja'
```

に書き換えておきましょう。これを変更することで、第13章で説明する管理サイトの表示なども日本語化されて見やすくなります。

10.8.3 TIME_ZONE (タイムゾーン)

時刻を表示する際のタイムゾーンを指定することができます。

デフォルトでは 'UTC' となっているので、

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

に変更しておきましょう。なお、「USE_TZ」が True になっていないと「TIME_ZONE」の設定が反映されないので注意が必要です。

```
USE_TZ = True
```

10.8.4 MIDDLEWARE (ミドルウェアの設定)

第9章「ミドルウェア (Middleware)」で説明した通り、「MIDDLEWARE」には有効にしたいミドルウェアをリスト形式で列挙します。記載する順番を間違えるとうまく動作しないこともありますので注意してください。

10.8.5 ALLOWED_HOSTS (許可するホスト)

「ALLOWED_HOSTS」はセキュリティ対策のための設定で、Django サイトを配信するときのホストを指定します。「ALLOWED_HOSTS」には、

- FQDN (「www.example.com」など)
- サブドメインをワイルドカードにした形式 (「.example.com」など)
- IP アドレス
- 'localhost'
- '*' (すべてを許可)

などの文字列が利用できます。

「DEBUG」が True のときには特に意識する必要はありませんが、「DEBUG」が False のときにはこの設定が必須になります（何も設定されていなければエラーが出てしまいます）。開発時は次のように設定してしまっても構わないでしょう。

```
ALLOWED_HOSTS = ['*']
```

10.8.6 SECRET_KEY (シークレットキー)

「SECRET_KEY」は Django 内部で暗号署名やハッシュ生成時に利用されるシークレットな文字列で、悪意ある攻撃者に推測されないように十分に長くてランダムな（単純でない）文字列であることが求められます。

環境ごとに固有であることが望ましいため、テスト環境から本番環境に移行するときに新しいランダムな文字列に付け替えるのがよいでしょう。その際は、次のコマンドを実行することで「SECRET_KEY」用の新しい文字列を生成することができます（改行せずに実行してください）。なお可能な限り本番運用中に「SECRET_KEY」の値を書き換えないように注意してください。

```
$ python3 manage.py shell -c "from django.core.management import utils; print(utils.get_random_secret_key())"
```

10.8.7 SITE_ID (サイト ID)

「SITE_ID」は、動作させている Django サイトを識別するための ID で、「sites フレームワーク」(django.contrib.sites)^{*15} が作成する「django_site」テーブル内のレコードの主キー (id) の値を数値で指定します。この設定自体は特に重要というわけではありませんが、インストールした Django パッケージによってはこの設定が必須となる場合があるため、注意が必要です。

たとえば、第 14 章で紹介する「django-allauth」では sites フレームワークの利用が必須となっており^{*16}、加えて Django サイト起動時に設定ファイルに django_site テーブルのレコードに対応する「SITE_ID」が設定されていなければエラーを出す仕組みになっています。そのため、

^{*15} <https://docs.djangoproject.com/ja/2.2/ref/contrib/sites/>

^{*16} sites フレームワークは Django 1.6 以降、startproject 実行時に自動作成される settings.py の「INSTALLED_APPS」の初期設定に含まれなくなつたため、手動で追加する必要があります。

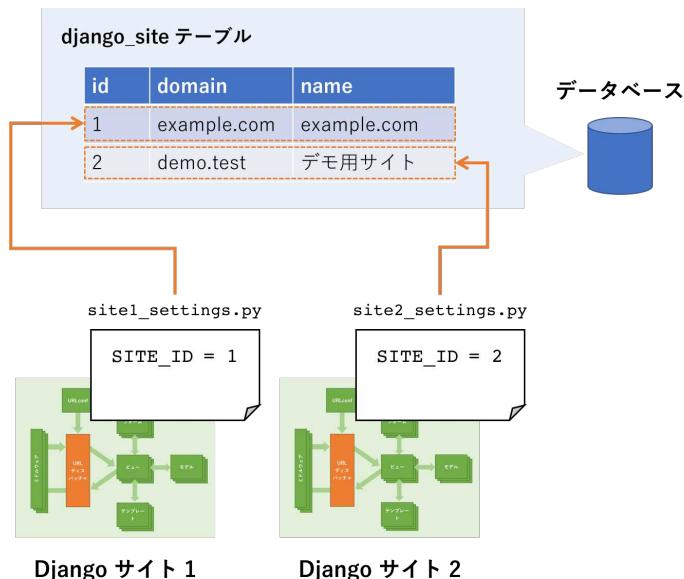
```
SITE_ID = 1
```

などという設定が必要になっているのです。

ところで、なぜ値が「1」なのでしょうか？

「INSTALLED_APPS」に「django.contrib.sites」を設定してマイグレーションを実行すると、自動的に「1 | example.com | example.com」というデフォルトのレコードが作成されます。つまり「1」という値自体には意味はなく、とりあえず存在するレコードのIDを設定しているだけなのです。なお、後述するような複数サイトでsiteフレームワークを利用するケースでは、適切なIDを設定する必要があります。

最後に、sitesフレームワークの利用イメージについて説明します。まず、複数のDjangoサイトがひとつのデータベースを共有しつつ稼働している状況があるとして、それぞれのサイトがsitesフレームワーク経由で自身のドメイン名を引き当てて画面や帳票に出力するといった使い方が考えられます（下図10.4参照）。また、このデータはサイト1のもの、このデータはサイト2のもの、といった具合にデータへのタグ付けの代わりに利用することもできます。なおこのsitesフレームワークは、サイトマップ機能を提供する組み込みの「サイトマップフレームワーク」（django.contrib.sitemaps）^{*17}でも利用されています。



▲図 10.4 sitesフレームワークの利用イメージ

^{*17} <https://docs.djangoproject.com/ja/2.2/ref/contrib/sitemaps/>

10.9 ベストプラクティス 9：個人の開発環境の設定は local_settings.py に書く

設定ファイル (settings.py) には本来、本番用の設定が書かれているべきであると筆者は考えます。片や開発時には、それぞれの開発者が自分の開発環境に合わせて「DEBUG」や「DATABASES」、「CACHES」、「ALLOWED_HOSTS」の設定値を書き換える必要があったり、あるいは検証のために「LOGGING」の設定を SQL 文が出力されるように変更したりするケースがあるかと思いますが、その際に settings.py をいちいち書き換えるのは面倒です。

そこでよく利用されるのが、settings.py の内容を読み込んだ個人用の設定ファイル「local_settings.py」を別に作り、そのファイルを使って runserver を起動するというものです。

次の例は local_settings.py のイメージサンプルです。

▼リスト 10.15 config/local_settings.py (例)

```
from .settings import *

DEBUG = True

ALLOWED_HOSTS = ['*']

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

このように書くことで、任意の設定値で settings.py の設定値を上書きすることができます。そして runserver 起動時に

```
$ python3 manage.py runserver 0.0.0.0:8000 --settings config.local_settings
```

と実行するか、もしくは環境変数「DJANGO_SETTINGS_MODULE」を使って、

```
$ export DJANGO_SETTINGS_MODULE=config.local_settings
$ python3 manage.py runserver 0.0.0.0:8000
```

と実行することで、local_settings.py を起動時の設定ファイルとして読みませることができます。^{*18}

なお、この local_settings.py は、チームメンバが共通で使うものではないためバージョン管理外とするのが通例です。

^{*18} Windows のコマンドプロンプトでは、export ではなく set コマンドを使って環境変数を設定します。

このベストプラクティスの発展型として、各環境（ローカル環境／テスト環境／本番環境）に合わせた設定ファイルを用意し、環境変数「DJANGO_SETTINGS_MODULE」によって Django プロジェクト起動時に読み込む設定ファイルを使い分けるというやり方がよく利用されます。具体的には、たとえば config/settings/ 以下を次のような構成にし、base.py には共通の設定を、それ以外からは「from .base import *」として差分となる設定だけを記述します。

```
config/settings
|-- __init__.py
|-- base.py          (共通の設定を記述)
|-- local.py         (ローカル環境用の設定ファイル)
|-- production.py   (本番環境用の設定ファイル)
`-- test.py          (テスト環境用の設定ファイル)
```

10.10 ベストプラクティス 10：シークレットな変数は .env ファイルに書く

前述の「SECRET_KEY」や AWS のシークレットアクセスキー、Stripe や PayPal などの外部サービスの API キー、パスフレーズなどは機密性が高いため、バージョン管理下に置かないように気を付けましょう。それらのシークレットな変数は設定ファイル (settings.py) には書かず、ファイル外から読み込む方法がよく利用されます。

シークレットな変数を設定ファイル外から読み込む方法として、環境変数を利用する方法やバージョン管理外のファイルを読み込む方式がよく用いられます。その際、「django-environ」^{*19} というパッケージを利用すると、環境変数を読み込むこともファイルに書かれた変数を読み込むこともできる、いわゆる 2-way な使い方ができます。

使い方は非常に簡単です。

まず、pip で django-environ をインストールします。

```
$ pip3 install django-environ
```

Django を起動する前にたとえば次のような環境変数を export しておき、

^{*19} <https://django-environ.readthedocs.io/>

```
$ export SECRET_KEY='^t74!xx0e!x*fq(bu3ypj)^%^al2n#+&vif1cqg_(me_fq(pb%'
```

設定ファイル側ではリスト 10.16 のように「`env = environ.Env()`」とするだけで、`export` した環境変数の値を読み込んでくれます。

▼リスト 10.16 settings.py (例)

```
import environ
env = environ.Env()

SECRET_KEY = env('SECRET_KEY')
```

環境変数を `export` したくない場合は、リスト 10.17 のような内容を記述した `.env` ファイル（ファイル名は任意）をベースディレクトリに用意しておき、

▼リスト 10.17 .env ファイル (例)

```
SECRET_KEY=^t74!xx0e!x*fq(bu3ypj)^%^al2n#+&vif1cqg_(me_fq(pb%
# MySQL
DATABASE_URL=mysql://mysiteuser:mysiteuserpass@localhost:3306/mysite
# SQLite
# DATABASE_URL=sqlite:///db.sqlite3
```

リスト 10.18 で示したように `read_env()` メソッドを使うことで `.env` ファイルを読み込むことが可能です。この書き方をすると、`.env` ファイルが存在しなくてもエラーは発生しません。そしてもちろん、`.env` ファイルはバージョン管理外にしておきます。

▼リスト 10.18 settings.py (例)

```
import environ
env = environ.Env()
# もし .env ファイルが存在したら設定を読み込む（ただし同じ変数の値は上書きされない）
env.read_env(os.path.join(BASE_DIR, '.env'))

SECRET_KEY = env('SECRET_KEY')

DATABASES = {
    'default': env.db()
}
```

`.env` ファイルにシークレットな変数を書いておく方式は使いやすいのですが、変数とその値を一行で書かないといけないので複雑な構造を書くのが難しいのが難点です。Env クラスには、決まった書き方をすればそれをペースして `list` や `dict` に展開してくれるメソッドがいくつか用意されています。リスト 10.18 の `env.db()` もそのひとつで、リスト 10.17 のように書いておけばデータベースの設定用オブジェクトに変換してくれるのです。

10.11 まとめ

- 設定ファイル（settings.py）にはプロジェクト固有の設定を記述する
 - local_settings.py には開発時の個人の設定を記述する
- 開発時は「DEBUG」は True、本番環境やステージング環境では False にする
- シークレットな変数はバージョン管理下のファイルには書かないようとする
- この章で紹介した重要な設定については最低限理解しておこう

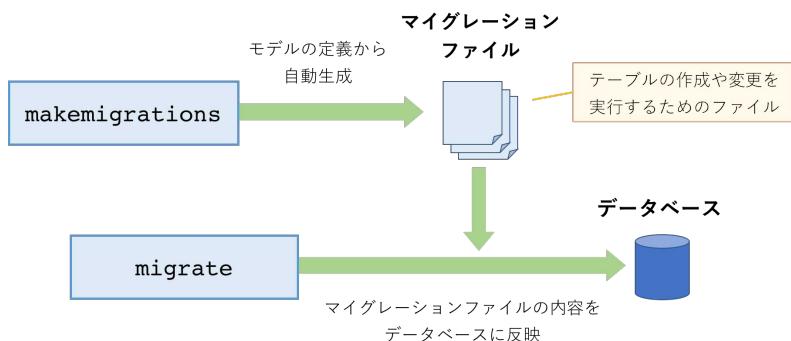
第 11 章

データベースのマイグレーション

11.1 概要

Web アプリケーションの世界では「マイグレーション」とはデータベースの移行作業を指すのが一般的ですが、Django で「マイグレーション」といえば、設定ファイルとモデルの定義からデータベースを作成したり定義を変更したりするためのコマンドを実行することを指します。Django プロジェクトでデータベースを利用するためには、Web サーバを起動する前にデータベースとテーブルの作成を済ませておく必要がありますが、そこで利用するのがマイグレーションなのです。

Django のマイグレーションコマンドは、二段構えになっています。ひとつ目がモデルの変更差分をチェックしてマイグレーションファイルを作成するための「makemigrations」、ふたつ目がマイグレーションファイルの内容をデータベースに反映してテーブルを作成・変更するための「migrate」です。マイグレーションの流れを図に示すと次の図 11.1 のようになります。



▲図 11.1 マイグレーションの流れ

コマンド	説明
<code>makemigrations</code>	モデルの変更差分をチェックしてマイグレーションファイルを作成
<code>migrate</code>	設定ファイルで定義したデータベースに、マイグレーションファイルの内容を反映してテーブルを作成・変更

11.2 makemigrations (マイグレーションファイルの作成)

自作したアプリケーションのモデルの変更差分からマイグレーションファイルを作成するには、makemigrations コマンドを実行します。引数なしでコマンドを実行すると、設定ファイルの「INSTALLED_APPS」に登録されたすべてのアプリケーションに対して処理がおこなわれます。

```
$ python3 manage.py makemigrations
```

個別のアプリケーションに限定するには、アプリケーション名を引数に指定します。複数のアプリケーション名をスペース区切りで列挙することも可能です。

```
$ python3 manage.py makemigrations <アプリケーション名>
$ python3 manage.py makemigrations <アプリケーション名 1> <アプリケーション名 2>
```

マイグレーションファイルは、各アプリケーションディレクトリの「`migrations`」パッケージ内に「<4桁の数字>_<自動生成されたマイグレーション名>.py」という名前で作成されます。4桁の数字は、migrations パッケージ配下のマイグレーションファイルの状況に応じて自動的にインクリメントされます。たとえば、shop アプリケーションに「0001_initial.py」および「0002_book_image.py」というマイグレーションファイルがすでに存在する状態で、Book モデルを若干修正して makemigrations を実行すると、次に示すような「0003_auto_20190401_0910.py」などというファイルが作成されます。

```
|-- shop
| .
| .
| |-- migrations
| | |-- 0001_initial.py
| | |-- 0002_book_image.py
| | |-- 0003_auto_20190401_0910.py
| | '-- __init__.py
```

<自動生成されたマイグレーション名> を任意の名前に変更したい場合は次のように「`--name`」オプションを利用します。

```
$ python3 manage.py makemigrations book --name add_fields_to_book
```

ちなみにモデルの作成や変更をしていない場合は、makemigrations コマンドは実行しなくとも構いません（実行してもマイグレーションファイルは作成されません）。

なお、第10章の「10.9 ベストプラクティス9:個人の開発環境の設定は local_settings.py に書く」で説明した個人開発環境用の「local_settings.py」を利用する場合は、コマンド実行時に「--settings」オプションで指定するか、もしくは環境変数「DJANGO_SETTINGS_MODULE」で指定します。

```
$ python3 manage.py makemigrations --settings config.local_settings
```

```
$ export DJANGO_SETTINGS_MODULE=config.local_settings  
$ python3 manage.py makemigrations
```

11.3 migrate（マイグレーションの実行）

作成したマイグレーションファイルの内容をデータベースに適用するには、migrate コマンドを実行します。migrate コマンドを実行すると、マイグレーションファイルの内容に応じた CREATE TABLE 文や ALTER TABLE 文などが発行されて、データベースに反映されます。

引数なしで実行すると、設定ファイルの「INSTALLED_APPS」に登録されたすべてのアプリケーションに対して処理がおこなわれます。

```
$ python3 manage.py migrate
```

個別のアプリケーションに限定するには makemigrations コマンドと同様、アプリケーション名を引数に指定します。複数のアプリケーション名をスペース区切りで列挙することも可能です。

```
$ python3 manage.py migrate <アプリケーション名>  
$ python3 manage.py migrate <アプリケーション名 1> <アプリケーション名 2>
```

なお、makemigrations コマンドと同様、「local_settings.py」を利用する場合は、コマンド実行時に「--settings」オプションで指定するか、もしくは環境変数「DJANGO_SETTINGS_MODULE」で指定します。

ここでもし、自作したアプリケーションにマイグレーションが適用されない場合は、

1. アプリケーションが設定ファイルの「INSTALLED_APPS」に登録されているか
2. アプリケーションディレクトリに「migrations」ディレクトリが存在するか

3. 「migrations」ディレクトリに「__init__.py」が存在するか（Python パッケージとして認識されているか）
4. データベースにマイグレーション履歴のレコード（次節を参照）が残ったままになっているいか

を確認してみてください。4. の「マイグレーション履歴のレコード」については次節で説明しますが、履歴レコードが不整合を起こして現在のマイグレーションファイルの番号よりも進んでしまっている場合はマイグレーションが実行されない場合があります。

11.4 マイグレーション履歴

Django はマイグレーションの履歴を「django_migrations」というテーブルで管理しています。「django_migrations」テーブルには、migrate コマンドで適用されたマイグレーションの履歴が、マイグレーションファイルごとに一行ずつのレコードとして保存されています（下表 11.2 参照）。

▼表 11.1 「django_migrations」テーブルの状況（一例）

id	app	name	applied
1	contenttypes	0001_initial	2019-04-01 00:00:00.000000
2	contenttypes	0002_remove_content_type_name	2019-04-01 00:00:00.000000
3	auth	0001_initial	2019-04-01 00:00:00.000000
...
19	shop	0001_initial	2019-04-01 00:00:00.000000
20	shop	0002_book_image	2019-04-02 06:11:00.000000

ここで、

```
$ python3 manage.py showmigrations
```

というコマンドを実行することで、makemigrations で作成されたマイグレーションファイルがどこまで migrate で適用されているかを確認することができます。コマンド実行結果は、

```
accounts
[X] 0001_initial
admin
[X] 0001_initial
[X] 0002_logentry_remove_auto_add
[X] 0003_logentry_add_action_flag_choices
auth
[X] 0001_initial
[X] 0002_alter_permission_name_max_length
```

```
[X] 0003_alter_user_email_max_length
[X] 0004_alter_user_username_opts
... (略) ...
contenttypes
[X] 0001_initial
[X] 0002_remove_content_type_name
sessions
[X] 0001_initial
shop
[X] 0001_initial
[X] 0002_book_image.py
[ ] 0003_auto_20190401_0910
```

のように表示されます。「X」が付いているものは migrate が適用されて履歴レコードとして保存されていること、「X」が付いていないものはマイグレーションファイルは存在するものの migrate がまだ実行されていないことを示しています。ここから、migrate を実行すると [] となっているマイグレーションファイルが適用されることが予想できます。

さらにありがたいことに、この履歴を使ってデータベースの状態を元に戻すことができます。たとえば、「0002_book_image」までが適用された shop アプリケーション関連のテーブルを、ひとつ前の「0001_initial」適用直後の状態に戻したいときは、次のようにアプリケーション名とマイグレーション名を指定して migrate コマンドを実行します。

```
$ python3 manage.py migrate shop 0001_initial
```

マイグレーション名はアプリケーション内で一意に特定できればよいので、上の例では「0001」のように省略することができます。

この機能は開発中にモデルの実装を試行錯誤しているフェーズで大変重宝しますが、マイグレーションファイルに記載していた関数を削除してしまった場合など、エラーが出て履歴を戻せなくなってしまうといったことが度々起ります。^{*1} どうしても元に戻せなくなった場合は、(レコードは無くなってしまいますが) いっそのことデータベースを最初から作り直してしまった方が早い場合があります。たとえば SQLite の場合であれば、データベースのファイルを削除するだけでリセットが可能です。

11.5 まとめ

- マイグレーションの目的は、データベースの準備
- マイグレーションコマンドは「makemigrations」と「migrate」の二段構え
 - makemigrations でマイグレーションファイルの作成
 - migrate でマイグレーションファイルをデータベースに反映

^{*1} 先にマイグレーションの適用を戻してからモデルを修正するように気を付ければ、多少軽減されます。

第 12 章

開発用 Web サーバ（runserver） を起動する

12.1 概要

Django には「runserver」と呼ばれる軽量の Web サーバが同梱されています。 runserver は開発時（基本は「DEBUG」が True）にのみで使われることを想定した Web サーバで、セキュリティの問題があったり本番環境で必要になりそうな機能が利用できなかったりするため、本番環境で利用するのは絶対にやめてください。本番環境では、 Gunicorn や Apache (mod_wsgi) などの WSGI アプリケーションに対応した Web サーバを利用するのが一般的です。

また runserver の大きな特徴として

- 自動リロード機能
- 静的ファイルの自動配信機能

があり、これらの機能はデフォルトでオンになっています。

自動リロードは、配下の Python コードの変更を検知して自動でサーバを再起動してくれる機能です。これにより、ビューやモデルのコード修正や設定ファイル (settings.py) の変更のたびにわざわざ手動で Web サーバを再起動する手間がなくなります。まさに開発に打ってつけの機能なのですが、敢えて自動リロード機能をオフにしたい場合は、「`--noreload`」オプションを付けて起動することで機能をオフにすることも可能です。

静的ファイルの自動配信は、runserver が設定ファイルから静的ファイルのパスを特定し、たとえば「`http://localhost:8000/static/...`」という URL で自動的に配信してくれる機能です（ただし、「DEBUG」が True の場合のみ）。これにより、静的ファイルを公開用ディレクトリに収集するための `collectstatic` コマンド（巻末の付録「E：覚えておきたい Django 管理コマンド 10 選」参照）を事前に実行する必要はなくなります。

このように、runserver は開発者に嬉しい機能がいくつも搭載された開発用の Web サー

バなのです。

12.2 runserver コマンドについて

runserver を実行するには次のコマンドを実行します。

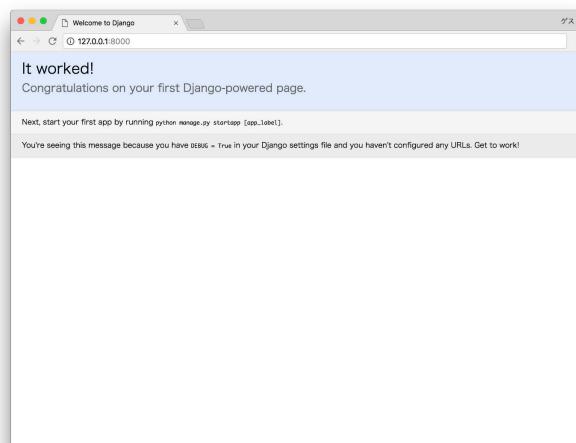
```
$ python3 manage.py runserver [<IP アドレス>:<ポート番号>]
```

IP アドレスとポート番号を省略した場合は、「127.0.0.1」というループバックアドレス（自分自身を示す特別なアドレス）とデフォルトポート番号の「8000」番で起動します。なお、IP アドレスを指定した場合はポート番号を省略することはできません。

実際にターミナルで実行してみましょう。

```
$ python3 manage.py runserver
```

実行後、ブラウザから「<http://127.0.0.1:8000>」あるいは「<http://localhost:8000>」にアクセスします。



▲図 12.1 ブラウザから <http://127.0.0.1:8000> にアクセス

プロジェクト作成直後の状態であればこのような画面が表示されれば OK です。

ここで、Docker 上で runserver を起動しようとした場合に、ホスト OS (Mac など) のブラウザから接続できない場合があります。「127.0.0.1」はゲスト OS の特別な IP アド

レスになっていてホスト OS と接続できない場合があります。その場合は、runserver の起動時に「0.0.0.0」という特別な IP アドレスで起動してみてください。

```
# python3 manage.py runserver 0.0.0.0:8000
```

12.3 まとめ

- runserver は開発用の軽量 Web サーバ。本番では使わない
 - 基本は「DEBUG」が True の場合に利用される
- runserver はコードの変更を検知して自動で再起動してくれる
- runserver は静的ファイルの自動配信してくれる
 - 事前に collectstatic コマンドを実行する必要なし

第 13 章

管理サイト（Django Admin）

13.1 概要

開発時にちょっとした検証をしたいときにテスト用のデータをデータベースに投入したり、テストや本番運用フェーズでデータの一部の値を変更したいケースがあるかと思います。そんなときに力を発揮するのが、管理サイト（Django Admin）です。Django はモデルに対応するデータベース内のレコードを簡単にメンテナンスできる専用の UI を提供してくれているのです。

しかも管理サイトの準備は簡単です。モデルクラスが準備できていれば、あとは「`django.contrib.admin.site`」という `AdminSite` クラスのグローバルオブジェクトにモデルを登録（register）するだけです。モデルごとに最短で 1 行加えるだけで登録ができます。なおこの登録処理は各アプリケーションの `admin.py` と呼ばれるモジュールに書くのが通例となっています。

この管理サイトは `superuser`（システム管理者）と特別な権限が与えられたユーザーのみが利用する想定になっており、専用のログイン画面が用意されています。管理サイトのログイン画面の URL は、プロジェクトの初回設定では「/admin/」になっています（リスト 13.1 を参照）、この URL を変更することも可能です。

▼リスト 13.1 プロジェクト初回作成時の `urls.py`

```
from django.urls import path
from django.contrib import admin

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

ここでシステム管理者とは、`is_superuser` 属性が `True` になっている `User` のことを指します。このユーザーは管理サイトで登録されたモデルのレコードを自由に追加・変更・削除することができてしまい、superuser のアカウントの取り扱いには厳重な注意が必要です。

13.2 モデルの登録方法

作成したモデルを管理サイトに登録するには、各アプリケーションディレクトリに配置された admin.py を編集します。たとえば、shop アプリケーションの「出版社 (Publisher)」「著者 (Author)」「本 (Book)」モデルを登録するには次のようなコードを記述します。

▼リスト 13.2 shop/admin.py

```
from django.contrib import admin
from .models import Publisher, Author, Book

admin.site.register(Publisher)
admin.site.register(Author)
admin.site.register(Book)
```

このようにそれぞれ 1 行ずつ書くだけで、モデルを管理サイトに登録することができます。なお、auth パッケージの Group モデルおよび User モデルは初めから管理サイトに登録されているので、パーミッショングループやユーザーの追加・変更・削除は自由におこなうことができます。

13.3 一部機能のカスタマイズ方法

「ModelAdmin」という管理サイトのオプションを管理しているクラスを使うことで、管理サイトの一部の機能をカスタマイズすることができます。ここでは、レコードの一覧画面や編集画面の表示内容を変更したり、レコードを保存する際のバリデーションを拡張したりする方法について説明します。

ここでいうカスタマイズは CSS のスタイルなどの見た目を変更するという意味ではありません。見た目のカスタマイズはやろうと思えばできなくもないですが、そもそもこの管理サイトはエンドユーザーに見せるためのものとしては作られていないため、あまりお勧めしません。

次の例では、リスト 13.2 のコードに以下の修正を加えています。

- 一覧画面で出力されるフィールドを「タイトル」「出版社」「価格」に変更
- 一覧画面でのソート順を「価格」の高い順に変更
- 編集画面に表示するフィールドを「タイトル」「出版社」「著者」「価格」のみに変更

▼リスト 13.3 ModelAdmin を使って管理サイトの表示フィールドを変更（shop/admin.py）

```
from django.contrib import admin
from django import forms

from .models import Author, Book, Publisher

class BookModelAdmin(admin.ModelAdmin):
    # 一覧表示のフィールドを変更
    list_display = ('title', 'publisher', 'price')
    # 一覧表示のソート順を変更
    ordering = ('-price',)
    # 編集画面のフィールドを変更
    fields = ('title', 'publisher', 'authors', 'price')

admin.site.register(Publisher)
admin.site.register(Author)
admin.site.register(Book, BookModelAdmin)
```

続いて、第8章「フォーム（Form）」で解説した ModelForm を使ったバリデーションを追加してみます。リスト 13.3 にさらに、レコードの保存時にタイトルに「Django」が含まれていなければエラーを表示するバリデーションを追加してみます。

▼リスト 13.4 管理サイトに ModelForm によるバリデーションを追加（shop/admin.py）

```
from django.contrib import admin
from django import forms

from .models import Author, Book, Publisher

class BookAdminForm(forms.ModelForm):
    def clean_title(self):
        value = self.cleaned_data['title']
        if 'Django' not in value:
            raise forms.ValidationError("タイトルには「Django」という文字を含めてください")
        return value

@admin.register(Book)
class BookModelAdmin(admin.ModelAdmin):
    list_display = ('title', 'publisher', 'price')
    ordering = ('-price',)
    fields = ('title', 'publisher', 'authors', 'price')
    form = BookAdminForm

admin.site.register(Publisher)
admin.site.register(Author)
```

保存時にバリデーションが NG になった場合は図 13.1 のようなエラーメッセージが表示されます。本番運用時にモデルの登録内容に何らかの制限を掛けたいときなどに便利に使えるでしょう。



▲図 13.1 バリデーション NG の際のエラー表示

ちなみに、リスト 13.4 の例では

```
admin.site.register(Book, BookModelAdmin)
```

と書く代わりに「`@admin.register`」デコレータを使って書いていますが、処理内容は同じです。

13.4 利用条件

管理サイトを有効化する条件は次の通りです。

- 設定ファイルの「`INSTALLED_APPS`」に「`django.contrib.admin`」を追加
- 同じく「`INSTALLED_APPS`」に以下を追加
 - `django.contrib.auth`
 - `django.contrib.contenttypes`
 - `django.contrib.messages`
 - `django.contrib.sessions`
- 同じく「`TEMPLATES.OPTIONS.context_processors`」に以下を追加
 - `django.contrib.auth.context_processors.auth`
 - `django.contrib.messages.context_processors.messages`
- 同じく「`MIDDLEWARE`」に以下を追加
 - `django.contrib.auth.middleware.AuthenticationMiddleware`
 - `django.contrib.messages.middleware.MessageMiddleware`
- URLconf に管理サイトの URL 設定を追加

やや複雑ですが、プロジェクト初回作成時のデフォルト設定のままですべての条件が満たされているので、あとは各アプリケーションディレクトリの `admin.py` にモデルの登録処理を書いていくだけよいはずです。

次に、管理サイトを利用するユーザーの権限について説明します。

まず管理サイトにログインできるユーザーは、`is_staff` と `is_active` 属性の値が `True` であることが条件になります。

サイト管理で各モデルの「追加」「変更」「削除」のアクションを実行するためには、モデルごと・アクションごとの「ユーザー権限」と呼ばれる権限が必要になります。たとえば、shop アプリケーションの Book モデルのレコードを追加する権限をユーザーに付与するには、「shop | book | Can add book」というユーザー権限を付与することになります。なお、`is_superuser` が `True` のユーザーは特別にすべてのユーザー権限が付与されていると見なされるため、superuser には追加で権限を付与する必要はありません。

13.5 使い方

管理サイトを使うためには、事前に

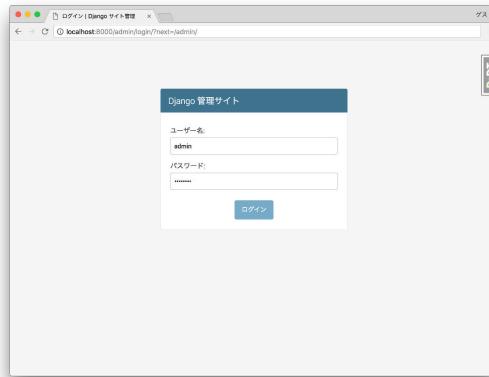
- 管理サイトの有効化（前節を参照）
- データベースのマイグレーション（第 11 章「データベースのマイグレーション」参照）
- superuser（システム管理者）の作成
- Web サーバの起動

が完了している必要があります。

管理サイトを利用するための superuser は事前に `createsuperuser` コマンドで作成しておいてください（巻末の付録「E：覚えておきたい Django 管理コマンド 10 選」を参照）。このコマンドで作成されたユーザーは `is_superuser`、`is_staff`、`is_active` がいずれも `True` となるので、管理サイトにログインすることができ、登録されたすべてのモデルの操作をおこなうことができます。

またここでは Web サーバとして `runserver` を利用します。まだ起動していない場合は、第 12 章「開発用 Web サーバ（`runserver`）を起動する」を参照して起動を済ませておいてください。

事前準備が整ったら、管理サイトのログイン画面（`/admin/`）にアクセスし、作成した superuser ユーザーでログインします。



▲図 13.2 管理サイトのログイン画面

ホーム画面にモデルの一覧が表示されます。ここにモデルが表示されていない場合は管理サイトへの登録がうまくできていない可能性がありますので、admin.py のコードを見直してください。



▲図 13.3 管理サイトのホーム画面

任意のモデル名をクリックすると、データベースに保存されたレコードの一覧が表示されます。新たにレコードを追加するためには、右上の「追加」ボタンをクリックします。



▲図 13.4 モデルのレコード一覧

任意の値を入力し、「保存してもうひとつ追加」「保存して編集を続ける」「保存」のいずれかのボタンをクリックして、レコードを保存します。



▲図 13.5 レコードの編集画面

追加が完了しました。同じようにして、画面を操作してレコードの変更や削除をおこなうことができます。

13.6 まとめ

- ・「管理サイト」と呼ばれる、開発者やシステム管理者向けのマスタメンテ機能が付属されている
- ・画面を使って、管理サイトに登録されたモデルのレコードを追加・変更・削除できる
- ・管理サイトを利用するには特別な権限が必要
- ・管理サイトに関するコードは admin.py に書く

第14章

便利な Django パッケージを使おう

14.1 概要

「Django パッケージ」は「Django Packages」^{*1} というサイトにまとめられた、Django の機能を拡張することに特化したパッケージ（プログラム集）です。pip でインストールして「INSTALLED_APPS」に追記して利用するタイプの Python パッケージ（サイトでは「App」と分類されています）がほとんどですが、その他にも「Mezzanine」^{*2} のように Django そのものを拡張した「Framework」や、補助的に使われる「Tool」なども含まれています。サイトには現在、3,900 を超える Django パッケージが掲載されています。

「Django Packages」にはありとあらゆるパッケージが掲載されていますが、中には長年放置されているものや、Python 3 や新しいバージョンの Django に対応していないものもあり、利用する際にはその選定がキモになります。選定する際は、上部に表示されているジャンルを選んでパッケージをグリッド表示するのがコツです。

▲図 14.1 Django パッケージをグリッドで比較

^{*1} <https://djangopackages.org/>

^{*2} <http://mezzanine.jupo.org/>

グリッド表示することで、各パッケージの特徴や開発状況、GitHub のスター数やフォーク数などを比較することができます。ある程度の選定基準にはなるものの、それでも選定は大変です。

そんなときには William S. Vincent^{*3} 氏が作成した「Awesome Django」^{*4} というサイトが有用です。このサイトには

- きちんとメンテナンスがされている
- 利用者が多い
- ドキュメントがしっかりしている
- ベストプラクティスに則っている

などの条件をクリアした、文字通り「Awesome」な選りすぐりの Django パッケージがリストアップされています。本章ではその中から筆者がお薦めするパッケージを少しだけ紹介します。

14.2 DRY 系パッケージ

共通化できるようなよくある機能は Django パッケージとしてすでに開発されているものが多いです。車輪の再発明をしないようにしましょう。

14.2.1 django-allauth（認証系機能の拡充）

「django-allauth」^{*5} では、ユーザー名ではなくメールアドレスをベースにしたログインや、パスワードを忘れた場合のフローなど、ログインまわりの細かな機能が多数提供されています。それだけでなく、Google や Twitter、GitHub などのソーシャルアカウントを利用したサードパーティ認証機能も含まれています。

14.2.2 django-tables2（テーブル表示とページネーション）

テーブル表示とページネーション（ページ移動の仕組み）は、一覧表示の際に常に常につきまとう厄介事です。汎用ビューの ListView を使うとビュー側である程度のサポートをしてくれますが、テンプレート側の実装は自力でやらなければいけません。そんな場合に役立つのが「django-tables2」^{*6} です。

^{*3} 『Django for Beginners』『REST APIs with Django』の著者としても有名。
ブログも非常に有用です。 <https://wsvincent.com/>

^{*4} <https://github.com/wsvincent/awesome-django>

以前は Roberto Rosario 氏が作成した同名のサイトがありましたが、現在は閉鎖されています。

^{*5} <https://django-allauth.readthedocs.io/>

^{*6} <https://django-tables2.readthedocs.io/>

pip でパッケージをインストールし、「INSTALLED_APPS」に追加した後に、ビューとテンプレートにそれぞれテーブル表示オブジェクトの作成と表示についてのコードを書くだけで、ページネーションを組み込んだテーブルを表示することができます。

▼リスト 14.1 ビュー側の実装例（shop/views.py）

```
import django_tables2 as tables

class BookTable(tables.Table):
    """本のテーブル表示クラス"""
    class Meta:
        model = Book

class BookListView(View):
    def get(self, request, *args, **kwargs):
        queryset = Book.objects.all()
        # テーブルオブジェクトを作成
        table = BookTable(queryset)
        table.paginate(page=request.GET.get('page', 1), per_page=10)
        context = {
            'table': table,
        }
        return render(request, 'shop/book_list.html', context)
```

▼リスト 14.2 テンプレート側の実装例（shop/book_list.html）

```
{% load django_tables2 %}
{% render_table table %}
```

また、設定ファイルの「DJANGO_TABLES2_TEMPLATE」変数を調整するだけで、Bootstrap 3 および 4、Semantic-UI 向けのスタイルが施されたテーブルを出力することもできます。

しかしながら、そもそもテーブル表示のページングをサーバ側で処理するのがベストなアプローチかといえば若干微妙です。その理由として、ページ移動のたびにデータベースへのアクセスが発生するために負荷が増えることやユーザーの UX が悪くなること、更新頻度が高いデータの場合にはページ移動時に不整合が発生する可能性があることなどが挙げられます。

その代替としてよく使われるのが、ビューからはデータの一覧を JSON 形式で一括で渡してしまい、クライアント側（テーブル表示を管理する JS）に検索やソート、ページングなどをすべて任せてしまうというアプローチです。この方が地雷を踏むことが少ないかもしれません。「Tabulator」⁷ は多機能でカスタマイズ性も高く、筆者お薦めの jQuery プラグインです。

⁷ <http://tabulator.info/>

14.3 開発補助系パッケージ

開発補助系のパッケージも多数あります。「10.10 ベストプラクティス 10：シークレットな変数は .env ファイルに書く」で紹介した「django-environ」もそのひとつです。

14.3.1 django-debug-toolbar (GUIによるデバッグ)

開発補助系パッケージで一番オススメなものがこれです。「SQL パネル」と呼ばれるデバッグ機能が強力で、runserver 起動中に画面を操作しながら実際に発行されたクエリを「SQL パネル」を開いて確認することができます。以降で、django-debug-toolbar を利用する手順について説明します。

まず、利用にあたっては以下の設定が必須です。

- 設定ファイル (settings.py) の「DEBUG」が True
- 同じく「INSTALLED_APPS」に「django.contrib.staticfiles」が設定済み

最初に、django-debug-toolbar をインストールします。

```
$ pip3 install django-debug-toolbar
```

次に、設定ファイルの「DEBUG」を True に変更し、次の設定を末尾に加えます。

▼リスト 14.3 config/settings.py (末尾に追加)

```
if DEBUG:  
    def show_toolbar(request):  
        return True  
  
    INSTALLED_APPS += (  
        'debug_toolbar',  
    )  
    MIDDLEWARE += (  
        'debug_toolbar.middleware.DebugToolbarMiddleware',  
    )  
    DEBUG_TOOLBAR_CONFIG = {  
        'SHOW_TOOLBAR_CALLBACK': show_toolbar,  
    }
```

「10.9 ベストプラクティス 9：個人の開発環境の設定は local_settings.py に書く」で説明したように、この設定は本来、「local_settings.py」に書くべきでしょう。

続いて、urls.py の末尾に次の設定を加えます。

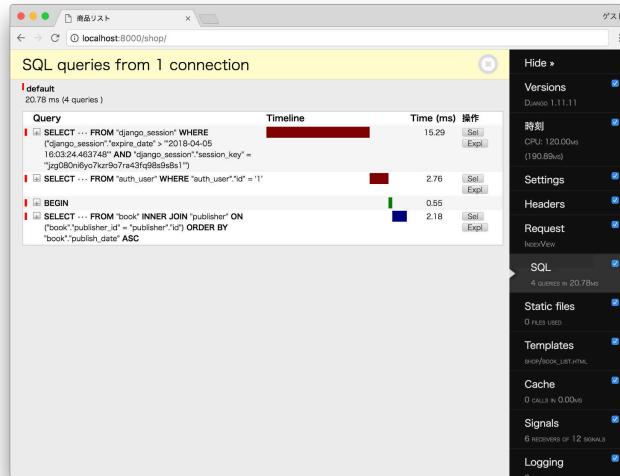
▼リスト 14.4 config/urls.py

```
if settings.DEBUG:
    import debug_toolbar

urlpatterns = [path('__debug__/', include(debug_toolbar.urls))] + urlpatterns
```

これで設定は完了です。

runserver を起動し、実際に画面を操作したあとに右側の「DJDT」と書かれたパネルを開くと、さまざまなデバッグ情報を確認することができます。デバッグが格段に捲るようになるため、初心者から中級者まで幅広い方にお薦めしたい Django パッケージです。



▲図 14.2 django-debug-toolbar の SQL パネル

14.4 まとめ

- 「Django Packages」は宝の山！（ただし玉石混淆なので注意）
- 「Awesome Django」にはその中から選りすぐりのパッケージが掲載されている
- お気に入りの Django パッケージ自分で見つけてみよう！

第 15 章

サンプルコード

15.1 概要

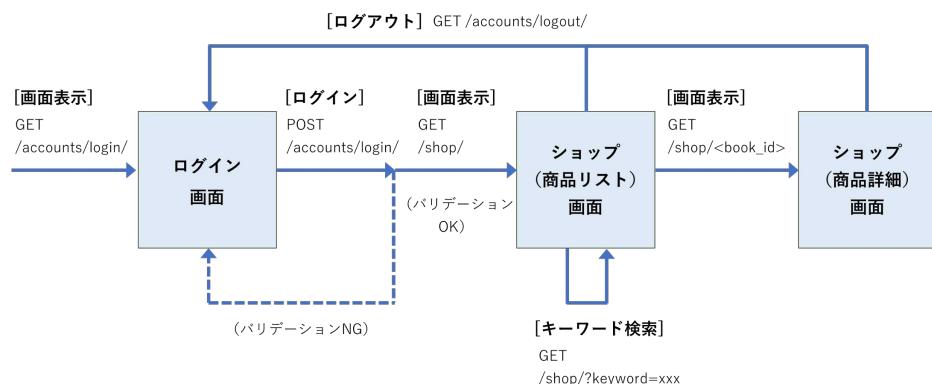
本書で説明した内容の総集編として、サンプルプロジェクトのコードを見ていきましょう。サンプルプロジェクトのテーマは「本のオンラインショップ」です。

サンプルコードは下記の GitHub リポジトリにアップしています。なおこのサンプルコードは予告なく変更されることがありますのでご了承ください。

サンプルコード

<https://github.com/akiyoko/django-book-mysite-sample>

サンプルコードでは主に、ログイン画面と商品リスト画面まわりの実装をしています。画面遷移は次の図 15.1 のようになっています。



▲図 15.1 サンプルプロジェクトの画面遷移

ログイン関連のアプリケーション名は「accounts」、ショップ関連は「shop」としています。また、第 6 章で掲示した 出版社 (Publisher) モデル、著者 (Author) モデル、本

(Book) モデルをほぼそのまま利用しています。

15.2 サンプルプロジェクトを動かすまでの最短ステップ

macOS でサンプルプロジェクトを動かすまでのステップについて説明します。Docker を利用しますので、巻末の付録「D:Docker でラクラク開発」を参考にして事前に「Docker Desktop for Mac」をインストールしておいてください。他の OS をお使いの方は適宜読み替えてください。

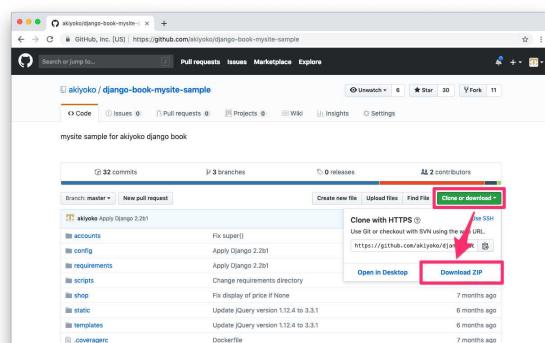
1. サンプルコードのダウンロード

まず、サンプルコードを GitHub からダウンロードします。作業場所はどこでも構いませんが、今回の例では PyCharm のデフォルトのワークスペース「~/PycharmProjects/」を作業場所にしています。

次の Git コマンドを実行して、master ブランチをダウンロードします。^{*1} ここではプロジェクトのベースディレクトリ名を「mysite-sample」としています。

```
$ cd ~/PycharmProjects/
$ git clone https://github.com/akiyoko/django-book-mysite-sample.git mysite-sample
```

Git が使えない場合は、GitHub の画面からサンプルコード一式を ZIP 形式でダウンロードすることも可能です。サンプルコードの URL（ブランチ名を確認）にアクセスし、「Clone or download」ボタンをクリックした後、「Download ZIP」をクリックします。



▲図 15.2 ZIP 形式でサンプルプロジェクトをダウンロード可能

^{*1} Windowsにおいて git clone をした際の改行コードを自動的に LF から CRLF にしてしまう設定になっていると、docker run 実行時にシェルの実行が失敗する場合があります。その場合は Git の設定を「git config --global core.autocrlf input」で変更してから clone するようにしてみてください。

2. サンプルプロジェクト用 Docker イメージの作成

clone したディレクトリに移動して、その直下にある Dockerfile から Docker イメージを作成します。Docker イメージは Ubuntu 18.04 LTS をベースに、python3、pip3 などの最低限必要となるソフトウェアと、サンプルプロジェクトの requirements.txt に記載された各種パッケージをインストールしています。

```
$ cd mysite-sample/  
$ docker build -t mysite-sample:1.0 .
```

3. Docker コンテナの実行

作成したイメージを使って Docker コンテナを実行し、bash を起動します。コンテナ実行時にデータベース（SQLite）の作成やマイグレーション、superuser の作成、および runserver の起動までをおこなっています。

```
$ docker run --rm -it -p 8000:8000 -v ~/PycharmProjects/mysite-sample/:/root/mysite  
--name mysite-sample mysite-sample:1.0 /bin/bash
```

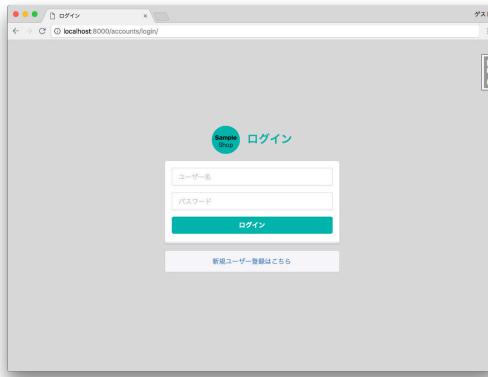
Docker コンテナを停止するときは、次のコマンドを実行してください。

```
$ docker stop mysite-sample
```

コンテナの実行を繰り返すと「django.db.utils.IntegrityError: UNIQUE constraint failed: custom_user.username」とのエラーが表示されますが、これはデータベースに同じユーザー名の superuser を登録しようとして出るエラーで、特に実害はないので無視して構いません。

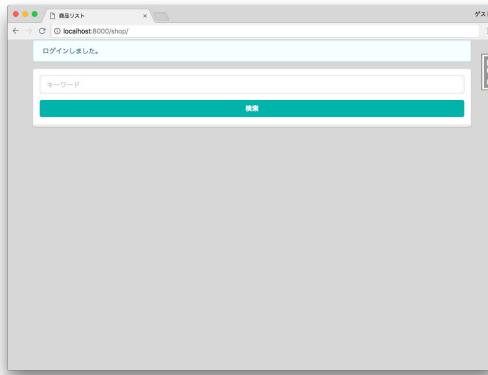
4. 動作確認

ブラウザから「<http://localhost:8000>」あるいは「<http://127.0.0.1:8000>」にアクセスします。次のようなログイン画面が表示されれば OK です。



▲図 15.3 ログイン画面

ユーザー名「admin」、パスワード「adminpass」の superuser が先の手順で作成されているはずなので、ログインをしてみましょう。



▲図 15.4 ショップ（商品リスト）画面

本の一覧が表示されるはずが、まだ本が登録されていないので何も表示されません。出版社、著者、本などのレコードは、管理サイト (<http://localhost:8000/admin/>) から登録することができます。

15.3 まとめ

- サンプルコードは本書の知識でほぼすべて理解できるはず
 - もし分からぬ部分があれば、それぞれの章に戻ってもう一度読み返そう

A : Python 3 のインストール手順

ローカル PC (macOS および Windows) に Python 3 をインストールする手順について説明します。前提とする OS バージョンはそれぞれ以下の通りとなります。バージョンが異なる場合はこの手順では上手くいかない可能性がありますのでご注意ください。

- macOS 10.15
- Windows 10 Home

すでにローカル PC に Python 3 がインストール済みかどうかについては、次のようにターミナル上で「python3」コマンドを実行することで確認できます（コマンドは「python」もしくは「python3.7」となっている場合があります）。

```
$ python3 --version  
Python 3.7.5
```

「Python 3.x.x」と出力されれば、Python 3 がインストール済みの状態です。デフォルトでは、macOS には Python 2.7 のみがインストールされており、Windows には Python 自体がインストールされていない状態になっています。以降で、macOS および Windows に Python 3.7 をインストールする手順を紹介します。 *2

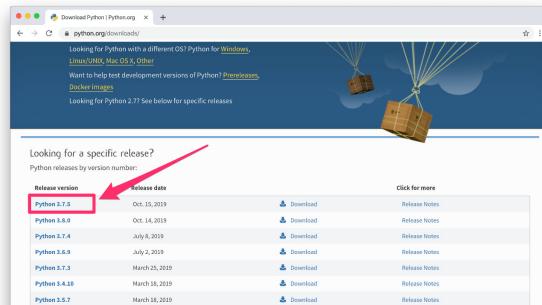
1. macOS 10.15 へのインストール手順

Python 公式サイトのインストーラを使う方法と、Python のバージョン管理ツール「Pyenv」を使う方法の 2 種類の手順について説明します。前者の方が簡単なので、慣れていない方には公式インストーラを利用する方法をお薦めします。Pyenv を利用する場合の手順は多少複雑になりますが、プロジェクトによって異なるバージョンの Python を使う必要がある場合にバージョンの切り替えがしやすくなるのがメリットになります。

*2 Django 2.2 がサポートしている Python 3.5 や 3.6 がすでにインストールされている場合は、無理にインストールし直す必要はありません。

1.1 公式インストーラを使う方法

Python 公式サイトのダウンロードページ^{*3}から Python 3.7 の最新バージョン（執筆時点では「3.7.5」）のリンクをクリックします。



「for macOS 10.9 and later」と書かれた「macOS 64-bit installer」のリンクからインストーラをダウンロードします。

The screenshot shows the Python 3.7.5 release page. A red arrow points to the 'macOS 64-bit installer' link in the 'Description' column of the file download table. The table includes columns for Version, Operating System, Description, MD5 Sum, File Size, and GPG.

Version	Operating System	Description	MD5 Sum	File Size	GPG
Gzipped source tarball	Source release		1cd071f70ff6d97524...aa3057aa	23126230	SIG
XZ compressed source tarball	Source release		0bedfb03b11...c45d2052e7a8f02	17236432	SIG
macOS 64-bit-32 bit installer	Mac OS X	(Deprecated) for Mac OS X 10.6 and later	c07...38c8e9648a591a92294464	35020778	SIG
macOS 64-bit installer	Mac OS X	for macOS 10.9 and later	20ef940498c5abab1d2bc1ad5069339	28196752	SIG
Windows help file	Windows		608caf6250f80aa11a690bfc13842c00	8141193	SIG
Windows x86-64 embeddable zip file	Windows	for AMD64/EM64T/x64	43b1bf0f02d2a03355003b1c595053	7550597	SIG
Windows x86-64 executable installer	Windows	for AMD64/EM64T/x64	6071ff88e0e0ccaa0ffdf3a77e979806	26777448	SIG
Windows x86-64 web-based installer	Windows	for AMD64/EM64T/x64	b95fe5e9e21209dd0f8136ed0ff8kf	1363032	SIG
Windows x86 embeddable zip file	Windows		7268f77d1af15a7dc58ff5a4ff48964cd1	6745126	SIG
Windows x86 executable installer	Windows		cfe9a282a611d5951b74093d70e0e89	25766192	SIG
Windows x86 web-based installer	Windows		e94494b76ce63d39664fed59e32c11370	1324872	SIG

最後に、インストーラをダブルクリックしてインストールを実行します。この手順でインストールした場合は、Python のインストールパスが .bash_profile の PATH に自動的に追加されます。

この手順では、インストールした Python 3 は「python3」というコマンドで実行できるようになります。インストールパスを確認するには「which python3」を実行します。

```
$ python3 --version
Python 3.7.5
$ which python3
/Library/Frameworks/Python.framework/Versions/3.7/bin/python3
```

^{*3} <https://www.python.org/downloads/>

1.2 Pyenv を使う方法

macOS では、Pyenv は Homebrew を使ってインストールするのが簡単です。

Homebrew がインストールされていない場合は、まずは Homebrew からインストールしていきます。ターミナルで次のコマンドを実行してください。念のため、実行するコマンドは公式サイト^{*4} からコピー&ペーストするのがよいでしょう。

```
$ /usr/bin/ruby -e "$(curl -fsSL \
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

次に、Homebrew で Pyenv をインストールします。インストール後に設定する環境変数の詳細については、公式の GitHub ページ^{*5} を参照してください。

```
$ brew install pyenv
$ echo 'export PYENV_ROOT=${HOME}/.pyenv' >> ~/.bash_profile
$ echo 'export PATH=$PYENV_ROOT/bin:$PATH' >> ~/.bash_profile
$ echo 'eval "$(pyenv init -)"' >> ~/.bash_profile
$ source ~/.bash_profile
```

最後に、Pyenv で Python 3.7.5 をインストールします。^{*6}

```
$ pyenv install 3.7.5
$ pyenv global 3.7.5
$ pyenv rehash
```

なお、pyenv install 実行時に「zipimport.ZipImportError: can't decompress data; zlib not available」とのエラーが出てビルドが失敗する場合は、

```
$ xcode-select --install
```

を実行してから再度 pyenv install を実行すると、エラーが解消する場合があります。

この手順では、「python3」のほか「python」や「python3.7」でも Python 3 が実行できるようになります。

^{*4} https://brew.sh/index_ja

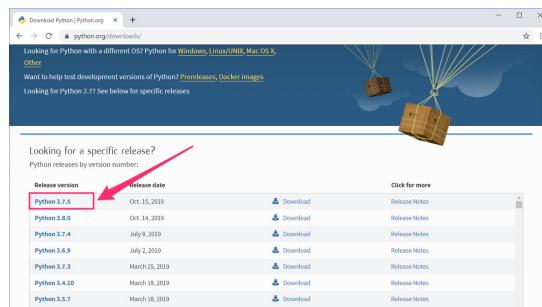
^{*5} <https://github.com/pyenv/pyenv>

^{*6} 「pyenv install --list」で最新バージョンがインストール候補に出てこない場合は、「brew upgrade pyenv」を実行して Pyenv をアップデートしてみてください。

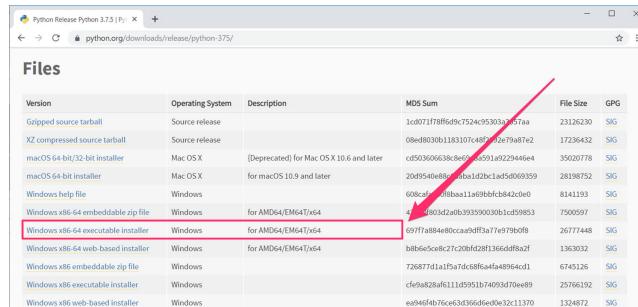
```
$ python3 --version
Python 3.7.5
$ which python3
/Users/akiyoko/.pyenv/shims/python3
$ pyenv which python3
/Users/akiyoko/.pyenv/versions/3.7.5/bin/python3
```

2. Windows 10へのインストール手順

Python 公式サイトのダウンロードページ^{*7}から Python 3.7 の最新バージョン（執筆時点では「3.7.5」）のリンクをクリックします。



OS が 64 bit 版の場合^{*8}は「Windows x86-64 executable installer」、32 bit 版の場合は「Windows x86 executable installer」と書かれたリンクからインストーラをダウンロードします。

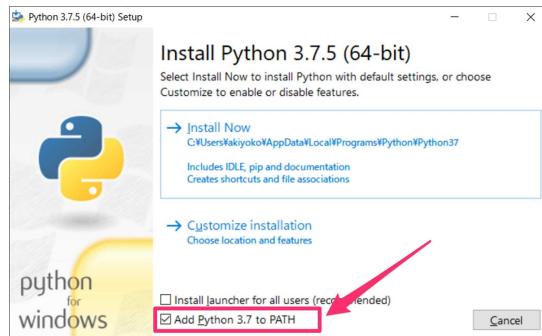


インストーラをダブルクリックしてインストールを実行します。この際、「Add Python 3.7 to PATH」にチェックを入れて、環境変数 PATH に Python のインストールパスを追加しておきます。

*7 <https://www.python.org/downloads/>

*8 [スタート] メニューのアプリ一覧から [Windows システムツール] > [コントロールパネル] をクリックし、[システムとセキュリティ] > [システム] を確認します。「システムの種類」に「64 ビット オペレーティングシステム、x64 ベース プロセッサ」と表示されていれば 64 bit 版です。

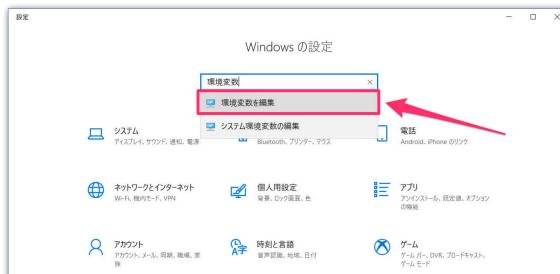
付録 A : Python 3 のインストール手順



これでインストールは完了です。

もし「Add Python 3.7 to PATH」にチェックを入れ忘れてインストールしてしまった場合は、以降の手順でインストールパスを PATH に追加することができます。

[スタート] メニューの [設定] から「環境変数を編集」を検索してクリックします。



ユーザー環境変数「Path」の「編集」をクリックして、環境変数の編集をおこないます。

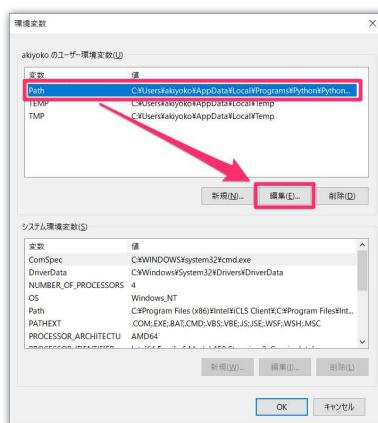


図: ユーザー環境変数の「Path」を「編集」

Python 3 のインストール先に合わせてたとえば次のようなパス 2 つを先頭に追加して

「OK」をクリックします。なお、「...\\Scripts\\」は pip などのツールやスクリプトを利用するためには必要なパスです。こちらも合わせて追加しておいてください。

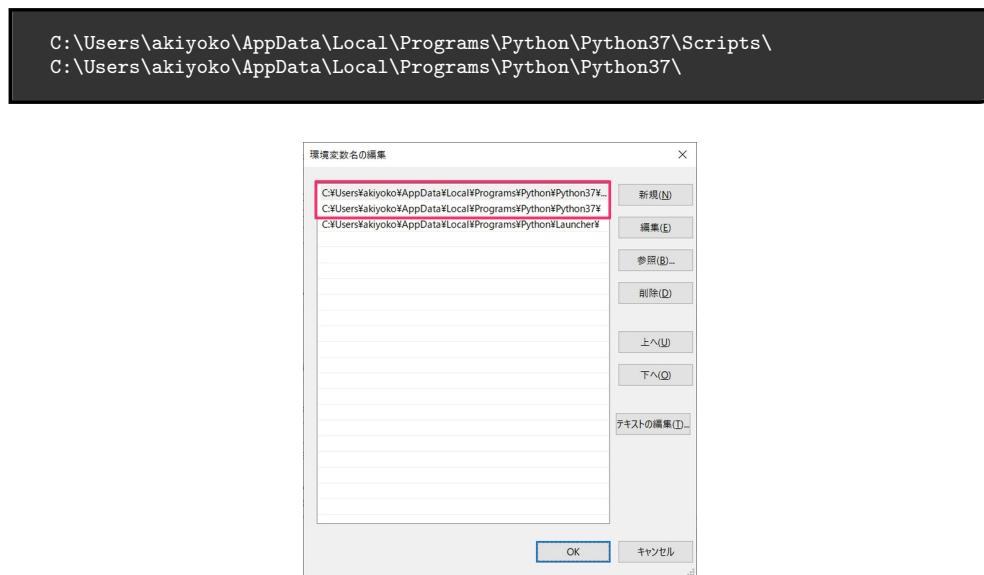


図: Python 3 のインストール先のパスを追加

この手順では、Python 3 は「python」として実行できるようになります。Python 3 のインストールパスを確認するには、コマンドプロンプト上で次のように「where python」を実行します。

```
> python --version
Python 3.7.5
> where python
C:\Users\akiyoko\AppData\Local\Programs\Python\Python37\python.exe
```

3. (参考) venv・pip・Pipenvについて

「venv」は、仮想環境を作成するためのモジュールです。仮想環境とは、プロジェクトごとに隔離された Python を実行するための仕組みで、任意に作成したディレクトリの下にベースとなる Python バイナリへのリンク、pip スクリプト、そしてその pip でインストールされたさまざまな Python パッケージが格納されます。これを使うことで、Python や Python パッケージのバージョンをプロジェクトごとに別々に管理することができます。通常、1 プロジェクトごとにそれぞれ 1 つの仮想環境を用意します。

Python 3.3 以降では「venv」モジュールが Python に標準で付属しており、Python 3.5 以降は仮想環境を作成するのに（「virtualenv」や「pyvenv」ではなく）「venv」が推

付録 A : Python 3 のインストール手順

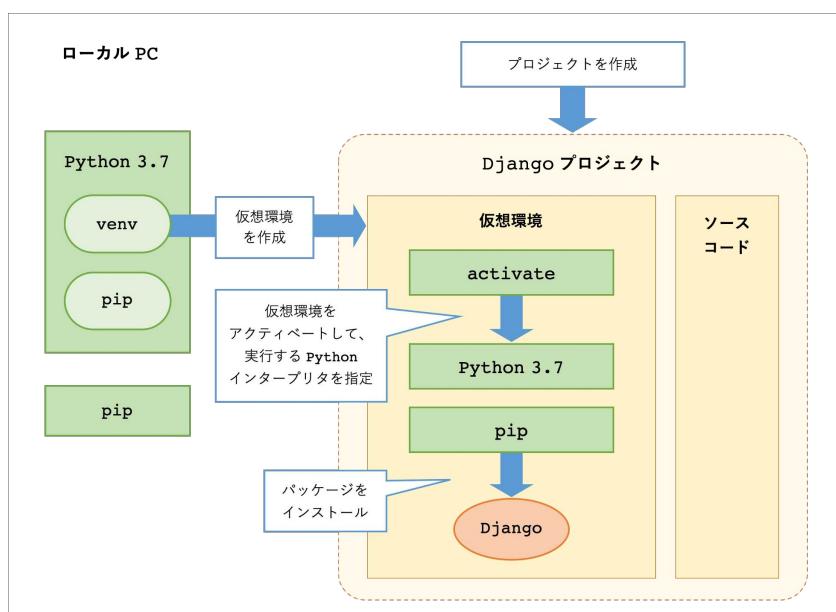
弊されています。^{*9} 今回の手順でインストールした Python 3.7 にも「venv」が同梱されているため、別途インストールする必要はありません。

「pip」は、PyPI ^{*10} などで提供されている Python パッケージを、バージョンを指定してインストールしたりアンインストールしたりするための便利ツールです。今回紹介した Python のインストール手順では、pip も合わせてインストールされます。

「venv」は仮想環境作成ツール、「pip」はパッケージ管理ツールですが、「Pipenv」はそれらを兼ね備えたようなツールです（内部では pip が使用されています）。後発のツールだけあってモダンで便利ですが、Python 自体に標準で付属されていないため、別途インストールする必要があります。

仮想環境の作成には venv か Pipenv のどちらかを使えばよいでしょう。本書では以降で venv を使った例を紹介します。

それでは実際に、次の手順にしたがって仮想環境を作成してみましょう。



macOS のターミナルや Windows のコマンドプロンプトを起動し、次のコマンドを実行してプロジェクトのベースとなるディレクトリを作成します。

```
$ mkdir mysite
```

^{*9} <https://docs.python.org/ja/3.7/library/venv.html>

^{*10} https://ja.wikipedia.org/wiki/Python_Package_Index

次のコマンドを実行して仮想環境を作成します。「-m venv」で「venv」モジュールを利用して仮想環境を作成せよ、という指定になります。右側の「venv」は仮想環境の情報を保管しておくためのディレクトリ名^{*11}で、絶対パスか相対パスのいずれかで指定します。この例では相対パスで指定しています。

```
$ cd mysite  
$ python3 -m venv venv
```

実行すると、仮想環境を格納しておくための「venv」ディレクトリが作成されます。

次に、仮想環境をアクティベート（有効化）します。

```
(macOS の場合)  
$ source venv/bin/activate  
(venv) $  
  
(Windows の場合)  
> venv\Scripts\activate  
(venv) >
```

アクティベート実行後、プロンプト表示部分の先頭に「(venv)」と仮想環境名が表示されますが、これが仮想環境がアクティベートされている合図です。仮想環境で利用するための特別な「python」や「pip」が用意されているため、アクティベート後は「python」および「pip」（それぞれ「3」や「3.7」を付けなくてもよい）で実行できるようになります。

最後に、pip を使って仮想環境に Django をインストールしてみます。「==」を付けることでインストールするバージョンを指定することができます。

```
(venv) $ pip install Django==2.2.6
```

なお、パッケージ名には大文字・小文字の区別は無いため、「Django」の代わりに「django」と指定しても OK です。

仮想環境から抜ける場合は、「deactivate」を実行します（macOS・Windows 共通）。

```
(venv) $ deactivate
```

^{*11} ディレクトリ名や配置場所は任意に指定できますが、慣例的に「venv」という名前のディレクトリをプロジェクトの直下に作成することが多いようです。

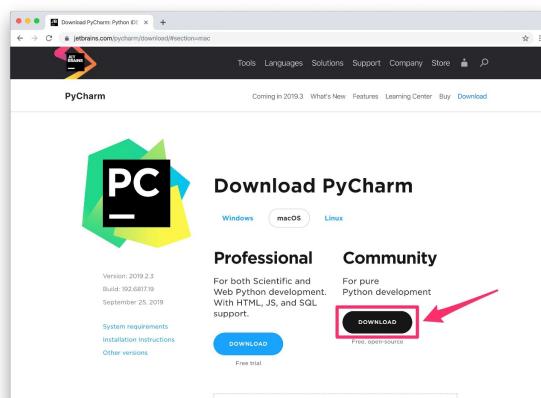
B : PyCharm のインストールと初期設定

PyCharm は JetBrains 社が開発・提供している IDE（統合開発環境）で、Python で開発をする上で便利な機能が多数取り揃えられています。「venv」や「pip」の使い方が分からなくても画面を操作するだけで簡単にパッケージ管理ができるため、Python に慣れていないエンジニアでも手軽に使い始めることができるのも大きな魅力です。デメリットとしては、一部の機能が有償版の「Professional Edition」でないと使えないことや、困ったときの日本語の情報が少ないといったことが挙げられますが、それでも PyCharm を使うメリットは非常に大きいと筆者は考えます。

なお PyCharm は、macOS、Windows および Linux で利用することができます。ここでは、macOS における「PyCharm CE」のインストールおよび初期設定の手順を紹介します。他の OS をお使いの方は適宜読み替えてください。

1. 無償版「PyCharm CE」のインストール

PyCharm の公式ダウンロードサイト^{*12} から、OS に合わせた「PyCharm CE」のインストーラをダウンロード後、インストーラをダブルクリックしてインストールを実行します。なお、2019 年 10 月時点の最新版は 2019.2 となっています。



^{*12} <https://www.jetbrains.com/pycharm/download/>

PyCharm CE を起動するといいくつか初回設定を選択する画面になりますが、「Skip Remaining and Set Defaults」をクリックしてデフォルトのスタイルを選択しておけばよいでしょう。

(参考) PyCharm の日本語化

「Pleiades 日本語化プラグイン」を利用すれば、PyCharm の UI を日本語化することも可能です。ただし JetBrains 公式のものではないため、動作は保証されていません。^{*13}

プラグインの導入手順は簡単です。まず、「Pleiades 日本語化プラグイン」のダウンロードサイト^{*14}から zip ファイルをダウンロードしてください。

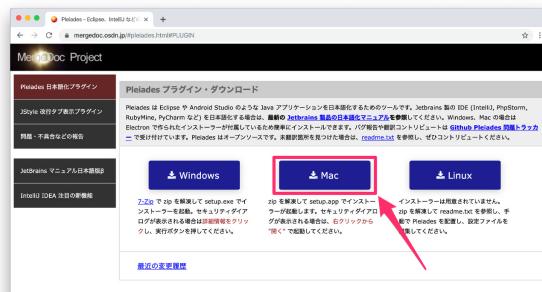


図: OS に合わせた「Pleiades 日本語化プラグイン」をダウンロード

あとは、解凍した setup.app をダブルクリックしてセットアップ画面を起動し、インストール先の「PyCharm CE」を選択して「日本語化する」をクリックするだけです。



図: 「PyCharm CE.app」を選択して「日本語化する」をクリック

^{*13} ちなみに筆者は日本語化はおこなっていません。本書も英語版のままでの説明をおこなっています。

^{*14} <https://mergedoc.osdn.jp/#pleiades.html#PLUGIN>

付録 B : PyCharm のインストールと初期設定

日本語化プラグインの適用を解除するには、上部メニューの [ヘルプ] > [カスタム VM オプションの編集...] から「pycharm.vmoptions」というファイルを開いて最後の行をコメントアウトした後、PyCharm を再起動してください。

```
#-javaagent...(略).../PyCharmCE2019.2/jp.sourceforge.mergedoc.pleiades/pleiades.jar
```

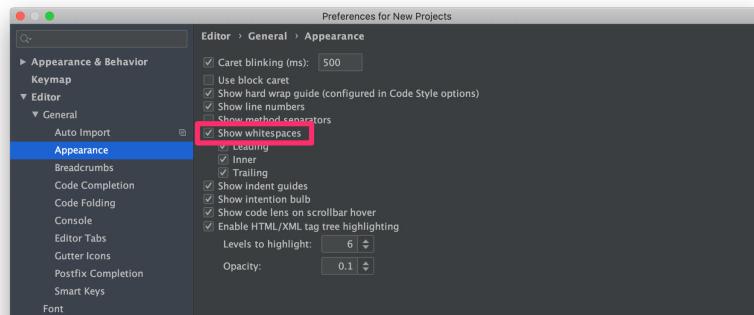
2. 初期設定

PyCharm は「out of the box（箱から取り出してすぐに使える）」というのが大きな魅力のひとつですが、多少の環境設定をすることでさらに便利に使えるようになります。環境設定画面は、macOS の場合は上部メニューの [PyCharm] > [Preferences]、Windows の場合は [File] > [Settings] から開くことができます。

ここでは最低限の初期設定のみをおこないます。 *15

コード表示の設定

ソースコード内の空白スペースを判別できるようにするために、環境設定画面の [Editor] > [General] > [Appearance] から、[Show whitespaces] にチェックを入れます。

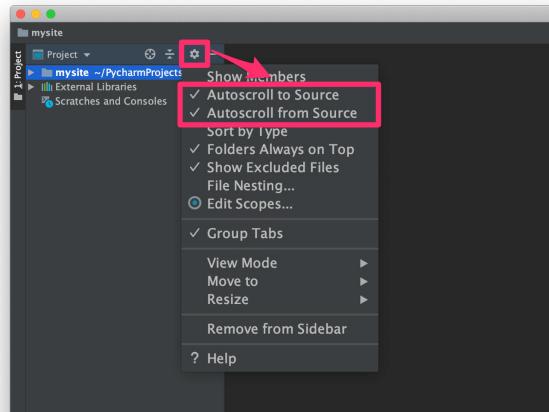


Project ビューの表示設定

Project ビューとソースビューを連動させるために、Project ビューの歯車アイコンをクリックして、[Autoscroll to Source] および [Autoscroll from Source] にチェックを入れます。

これにより、Project ビューのファイルをシングルクリックすることでソースビューが開き、ソースビューを開くことで Project ビューのファイルにフォーカスがあたるようになります。

*15 さらなる高度な環境設定については、拙ブログ記事 (<https://akiyoko.hatenablog.jp/entry/2017/03/10/082912>) を参考にしてください。



メモリ設定

パス内検索などの高負荷な処理をこなせるように、ヒープメモリの最大サイズを増やしておきます。

上部メニューの [Help] > [Edit Custom VM Options] を選択すると、次に示すような「pycharm.vmoptions」という名前のファイルが自動で作成されます。ここで、デフォルト値が「-Xmx750m」(750 MB) となっている場合は「-Xmx2048m」などと書き換えて、PyCharm を再起動します。ヒープメモリのサイズは環境に応じて適切な値に調整してください。

```
-Xms128m  
-Xmx2048m  
-XX:ReservedCodeCacheSize=240m  
... (略) ...
```

(参考) 便利なショートカット

よく使う便利なショートカットをいくつか紹介します。

なお、[Help] > [Keymap Reference] でキーマップ・リファレンス（チートシート）を見ることができます。PyCharm ヘルプの日本語版サイト^{*16}から参照できる Web 版チートシート（翻訳）^{*17}も非常に有用です。

^{*16} <https://pleiades.io/help/pycharm/>

^{*17} https://pleiades.io/sites/willbrains.jp/keymap/pdf/shortcut_pycharm_mac.pdf
https://pleiades.io/sites/willbrains.jp/keymap/pdf/shortcut_pycharm_windows.pdf

付録 B : PyCharm のインストールと初期設定

▼表 1 検索系のショートカット

macOS 版	Windows 版	説明
⌘ + F	Ctrl + F	ファイル内検索
⌘ + (shift +) G	(Shift +) F3	ファイル内検索結果を移動
⌘ + shift + F	Ctrl + Shift + F	パス内検索
⌘ + B	Ctrl + B	宣言箇所にジャンプ
option + F7	Alt + F7	使用箇所を検索して Find ウィンドウに表示
⌘ + option + ↓ (↑)	Ctrl + Alt + ↓ (↑)	Find ウィンドウの検索結果を移動
⌘ + option + ← (→)	Ctrl + Alt + ← (→)	履歴を移動
shift x 2 (素早く 2 回)	Shift x 2 (素早く 2 回)	クイック検索

▼表 2 その他のショートカット

macOS 版	Windows 版	説明
⌘ + ,	Ctrl + Alt + S	環境設定を開く
control + スペース	Ctrl + スペース	コード補完
⌘ + option + L	Ctrl + Alt + L	コードの自動整形
⌘ + control + O	Ctrl + Alt + O	import 文の最適化
⌘ + E	Ctrl + E	最近開いたファイルを開く
control + ← (→)	Alt + ← (→)	タブ移動

(参考) Professional Edition と Community Edition の違い

PyCharm には有償の「Professional Edition」と無償の「Community Edition」(CE) の 2 種類のエディションがあり、後者では一部機能が使えません（下図参照）。

Choose your edition	PyCharm Professional Edition	PyCharm Community Edition
Intelligent Python editor	✓	✓
Graphical debugger and test runner	✓	✓
Navigation and Refactorings	✓	✓
Code inspections	✓	✓
VCS support	✓	✓
Scientific tools	✓	
Web development	✓	
Python web frameworks	✓	
Python Profiler	✓	
Remote development capabilities	✓	
Database & SQL support	✓	

▲図 5 有償版と無償版の機能比較 (<https://www.jetbrains.com/pycharm/features/> より)

この中で著者が考える Professional Edition の優位点はズバリ、次の 4 点です。

1. ファイルの同期とリモートデバッグ (Remote development capabilities)
2. データベースのサポート (Database & SQL support)
3. Django などの各種フレームワークのサポート (Python web frameworks)
4. コードカバレッジのサポート

1. は、ローカル PC のファイル保存時に開発用サーバに自動的に同期してくれたり、開発サーバ上の Django のプロセスをブレークポイントで止めて PyCharm 上でステップ実行できるようにしてくれる便利機能です。ファイルの同期については「D : Docker でラクラク開発」を活用すれば、Community Edition でも同じようなことが実現できます。リモートデバッグについては、pdb を使ってコンソールでデバッグするのと比較して、見慣れた画面上でデバッグできるため（どこでエラーが発生したかがすぐに特定できるので Django に慣れないうちは特に）開発効率が劇的に上がります。

2. は、データベースの操作を PyCharm 上で統合的に利用できる機能です。SQLite については良さげなクライアントツールが他に無いので個人的にこの機能は重宝していますが、MySQL であれば「MySQL Workbench」や「Sequel Pro」^{*18}、PostgreSQL であれば「PSequel」^{*19} などのツールがあれば特に必須ではないと思います。ER 図の作成機能（下図参照）についてはプロジェクトによってはニーズがあるかもしれません。

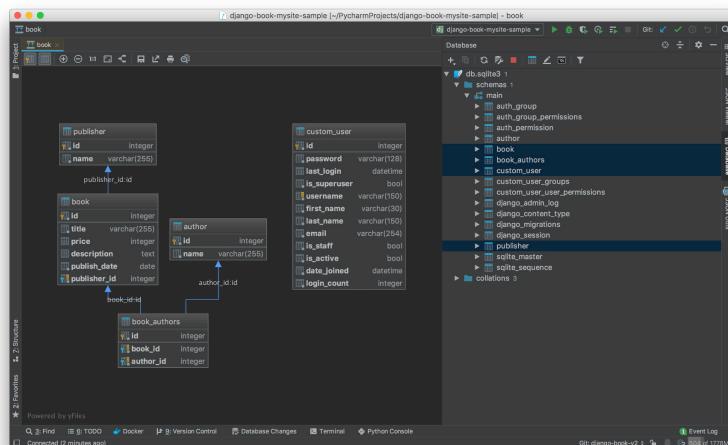


図: ER 図作成機能 (Professional Edition のみ)

3. は、Django のテンプレートエンジンに合わせたシンタックスハイライトやモジュールの種類に合わせたコード補完をサポートしてくれます。あればもちろん便利なのですが、必要不可欠とまではいえないよう思います。

*18 <https://www.sequelpro.com/>

*19 <http://www.psequel.com/>

4. については Coverage.py^{*20} の使い方が分かっていればこの機能が無くても問題ないでの、必須とまではいえないでしょう。

(参考) ライセンスと料金体系

Professional Edition のライセンスは 2017 年以降、「サブスクリプションライセンス」方式に統合されました。料金は、用途が「パーソナル」(個人向け) なのか「コマーシャル」(企業向け) なのに加え、継続年数によっても異なります(下表 3 参照)。この内容は 2019 年 10 月時点のものですので、正確な価格については公式サイト^{*21}をご確認ください。

▼表 3 PyCharm Professional Edition のサブスクリプション料金 (年額)

継続年数	パーソナルライセンス	コマーシャルライセンス
1 年目	10,300 円	22,900 円
2 年目	8,200 円	18,300 円
3 年目以降	6,200 円	13,700 円

筆者は Professional Edition を愛用していますが、まずは無償の Community Edition を使ってみてその便利さを体験してみることをお薦めします。

ちなみに、無料で使える IDE としては他に「Visual Studio Code」^{*22} が有名ですが、Python 開発を専用としたものではないため、Python 拡張機能のインストールなどといった手間が必要になります。

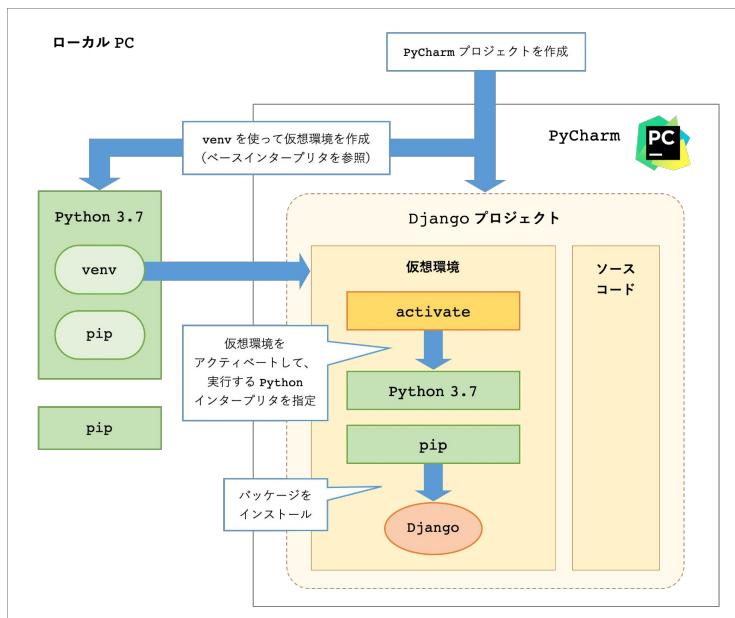
*20 <https://coverage.readthedocs.io/>

*21 <https://www.jetbrains.com/pycharm/buy/>

*22 <https://code.visualstudio.com/>

C : PyCharm による Django 開発 環境の構築手順

ローカル PC に無償版の PyCharm CE を使用して Django の開発環境を構築する手順について説明します。前提とする OS・バージョンは macOS 10.15 となります。Windows の場合は適宜読み替えてください。構築手順の全体像は次の通りです。 *23



1. PyCharm プロジェクトを作成

プロジェクト作成時に必要なものは、

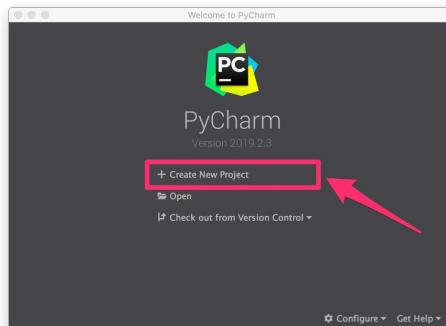
1. プロジェクトを格納するワークスペースのディレクトリのパス
2. 仮想環境のディレクトリのパス
3. 仮想環境にコピーする元となる Python インターパリタのパス

*23 PyCharm のバージョン 2018.2 以降で仮想環境作成時に「Pipenv」が選択できるようになりました。
しかしながら本書では Pipenv を使った手順については説明しません。

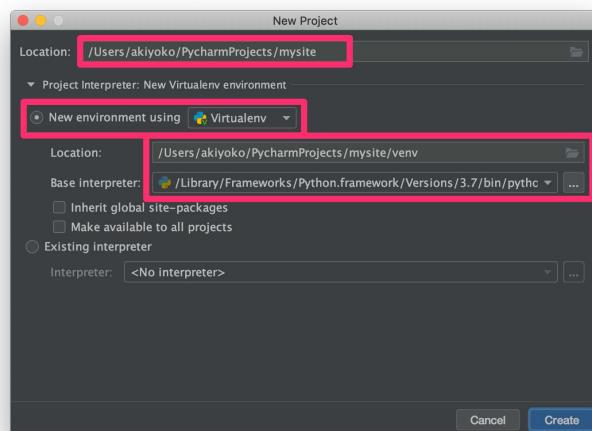
付録 C : PyCharm による Django 開発環境の構築手順

の 3 つです。2. や 3. はプロジェクト作成後でも指定できますが、プロジェクト作成時に指定しておくことをお薦めします。

PyCharm 起動後に「Create New Project」を選択し、プロジェクトを新規作成します。



新規作成画面で次の内容を入力します。



▼表 4 プロジェクト新規作成時の設定内容

設定項目	設定値
Location:	/Users/akiyoko/PycharmProjects/mysite
New environment using:	Virtualenv
Location:	/Users/akiyoko/PycharmProjects/mysite/venv
Base interpreter:	/Library/Frameworks/Python.framework/Versions/3.7/bin/python3 ('which python3' の実行結果をコピー&ペースト)

「~/PycharmProjects/」は、macOS 版 PyCharm のデフォルトのワークスペースです。新規作成時に「Virtualenv」を指定することで、(Python 3.3 以降の場合は)「venv」を利用してプロジェクトごとの仮想環境が作成されます。ここでは、仮想環境のディレクトリをプロジェクト直下に「venv」という名前で作成するようにしています。「Base interpreter」には仮想環境にコピーする元となる Python インタープリタのパスを指定します。具体的には、macOS の場合は「which python3」(Pyenv を利用している場合は「pyenv which python」)、Windows の場合は「where python」の実行結果で得られるパスを指定してください。

2. PyCharm で Django をインストール

PyCharm で Python パッケージをインストールするには、「Project Interpreter」の画面から操作をおこないます。環境設定画面を開き、下部の「+」ボタンをクリックして Django を仮想環境内にインストールします。

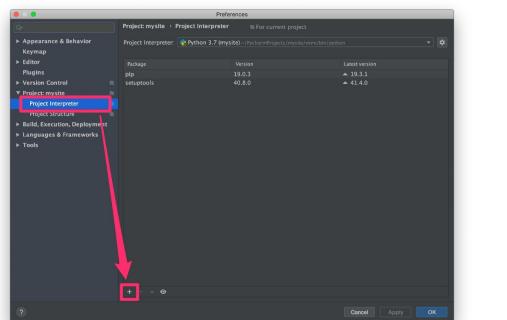


図: 環境設定画面の「Project Interpreter」から「+」ボタンをクリック

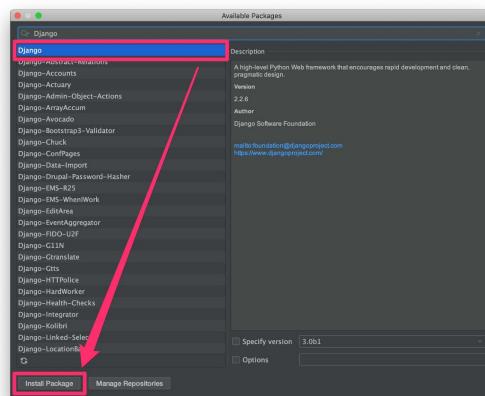
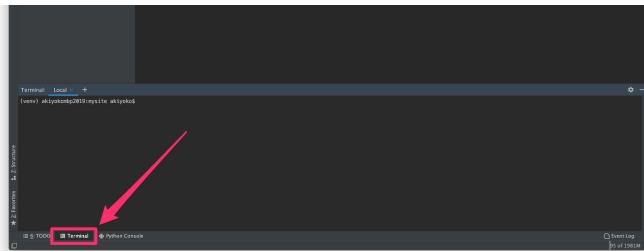


図: 「Django」を検索してインストール

このように PyCharm では、実際に venv や pip コマンドを利用しなくても、画面を操作して仮想環境の作成や必要なパッケージのインストールをすることができるのです。

3. Django プロジェクトのひな形を作成

左下の「Terminal」を選択すると、仮想環境がアクティベートされた状態のターミナルが起動します。 *²⁴



図：仮想環境がアクティベートされたターミナルを起動

ターミナル上で、次のコマンドを実行して Django プロジェクトのひな形を作成します。ちなみに最後の「.(ドット)」は、プロジェクトの箱となる「ベースディレクトリ」を作成しないようするための指定です。付け忘れないようしてください。

```
(venv) $ django-admin startproject config .
```

これで PyCharm で Django を開発する環境が整いました。

(参考) 既存のプロジェクトから新規作成する場合

Git で clone したプロジェクトなど、既存のプロジェクトから PyCharm プロジェクトを新規作成しようとするとき、プロジェクト作成時に仮想環境が設定できない場合があります。その場合はプロジェクト作成後に、環境設定画面の「Project Interpreter」から右上の歯車アイコンの「Add...」をクリックすると、仮想環境を設定する画面を開くことができます。

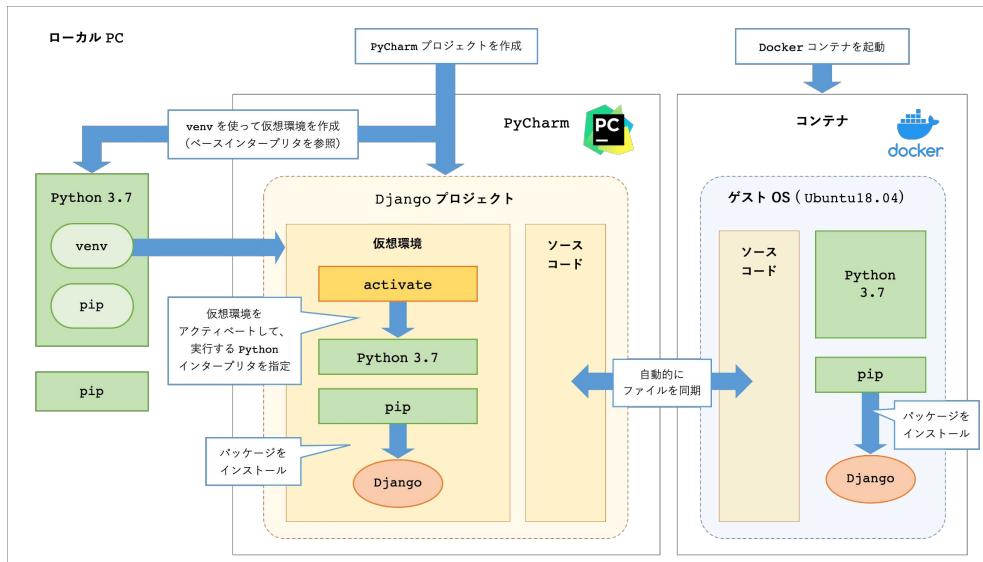
*²⁴ アクティベートされない場合は、環境設定画面から [Tool] > [Terminal] で [Activate virtualenv] にチェックが入っているかを確認してください。それでもアクティベートされない場合は、ターミナルで直接「source venv/bin/activate」を実行してください。

D : Docker でラクラク開発

「Docker Desktop for Mac」を利用して、ソースコードの開発を macOS 上の PyCharm で、Web アプリケーションの動作確認を Docker コンテナ（ゲスト OS は Ubuntu）上で実施できる環境を構築する手順について説明します。Docker を使えば macOS 側の環境を壊すことがないため、開発の効率が格段にアップします。

Windowsにおいても「Docker Desktop for Windows」がインストールできれば同様の開発スタイルが可能ですが、OS が 64 bit 版 Windows 10 Pro、Enterprise、Education のいずれかであること、Hyper-V が利用できることがインストールの最低要件になります。^{*25} ^{*26}

Docker を使った Django 実行環境の構築手順の全体像は次の通りです。



^{*25} <https://docs.docker.com/docker-for-windows/install/#what-to-know-before-you-install>

^{*26} 64 bit 版 Windows 10 Home の場合は、「Docker Toolbox」をインストールすることで Docker が使えるようになります。https://docs.docker.com/toolbox/toolbox_install_windows/

1. Docker Desktop for Mac をインストール

Docker Desktop for Mac の公式サイト ^{*27} から dmg ファイルをダウンロードしてインストールします。

2. Django プロジェクトを作成

「B : PyCharm のインストールと初期設定」に沿って「~/PycharmProjects/mysite」に Django プロジェクトが作成されているものとします。

3. Dockerfile から Docker イメージを作成

次に示す Dockerfile を Django プロジェクトの直下に作成します。ファイル名は「Dockerfile」（拡張子なし）してください。^{*28}

▼リスト 1 Dockerfile

```
FROM ubuntu:18.04

ENV PYTHONUNBUFFERED 1
ENV PYTHONIOENCODING utf-8

ENV HOME /root
ENV DEPLOY_DIR ${HOME}/mysite

RUN apt update

# Set locale
RUN apt install -y locales
RUN sed -i -e "s/# en_US.UTF-8 UTF-8/en_US.UTF-8 UTF-8/" /etc/locale.gen \
    && locale-gen
ENV LANG en_US.UTF-8
ENV LANGUAGE en_US:en
ENV LC_ALL en_US.UTF-8

# Install Python 3.7
RUN apt install -y wget \
    build-essential \
    zlib1g-dev \
    libssl-dev \
    libsqlite3-dev
WORKDIR ${HOME}
RUN wget https://www.python.org/ftp/python/3.7.5/Python-3.7.5.tgz \
    && tar zxf Python-3.7.5.tgz \
    && cd Python-3.7.5 \
    && ./configure --enable-optimizations \
    && make altinstall

# Set alias
RUN update-alternatives --install \
    /usr/local/bin/python3 python3 /usr/local/bin/python3.7 1
RUN update-alternatives --install /usr/local/bin/pip3 pip3 /usr/local/bin/pip3.7 1
```

^{*27} <https://www.docker.com/products/docker-desktop>

^{*28} <https://github.com/akiyoko/django-book-mysite-sample/blob/master/Dockerfile> を参考にしてください。途中まで同じです。

```
RUN pip3 install -U pip  
WORKDIR ${DEPLOY_DIR}  
CMD ["/bin/bash"]
```

次のコマンドを実行し、Dockerfile から Docker イメージを作成します。初回は少し時間が掛かります。

```
### 使い方  
$ docker build -t <イメージ名>:<タグ名> <Dockerfile を配置したディレクトリ>  
$ docker build -t mysite:1.0 ~/PycharmProjects/mysite/
```

この Dockerfile から、Python 3.7.5 をインストールした Ubuntu 18.04 ベースのイメージを作成することができます。Python 3 は「python3」、pip は「pip3」で実行可能です。

4. Docker コンテナを実行

macOS のターミナル上で次のコマンドを実行し、Docker コンテナを実行して bash で対話型シェルにログインします。その際、runserver のために 8000 番ポートを開けておきます。また、先に作成した Django プロジェクトのディレクトリを Docker 側の「/root/mysite」ディレクトリと同期させています。docker run 実行時の各種オプションについては、Docker ドキュメント日本語化プロジェクトのリファレンス ^{*29} を参照してください。

```
### 使い方  
$ docker run -it -p <ホスト OS 側のポート番号>:<コンテナ側のポート番号>  
      -v <ホスト OS 側のディレクトリ>:<マウント先のコンテナ側のディレクトリ>  
      --name <コンテナ名>  
      <イメージ名>:<タグ名>  
      <起動時のコマンド>  
  
$ docker run -it -p 8000:8000 -v ~/PycharmProjects/mysite:/root/mysite \  
      --name mysite mysite:1.0 /bin/bash
```

5. Docker 上で事前準備

Docker 上で次のコマンドを実行していきます(ユーザーは root のまま)。なお、Docker コンテナは使い捨てで利用するため、仮想環境は作成せず、Python パッケージはグローバルにインストールしてしまいます。

^{*29} <http://docs.docker.jp/engine/reference/run.html>

```
### 必要なものを適宜インストール  
# apt install -y sqlite3  
  
### requirements.txt でまとめてインストール  
# pip3 install -r requirements.txt  
  
### あるいは pip でひとつづつインストール  
# pip3 install "Django==2.2.6"
```

6. 動作確認

これで、PyCharm で追加・修正したソースコードを、Docker コンテナ上の Ubuntu で動作確認することができるようになりました。macOS 側の環境を壊すこと也没有。

最後に runserver を実行して、macOS 上のブラウザで動作確認をおこないます。

```
# python3 manage.py runserver 0.0.0.0:8000
```

E：覚えておきたい Django 管理コマンド 10 選

ぜひ覚えておきたい Django コマンドを 10 個紹介します。本書のカバーする範囲外のものもありますが、いずれ使うようになるかと思います。

他の管理コマンドの一覧については公式ドキュメント^{*30} を参照してください。

1. startproject (新しいプロジェクトのひな形を作成)

新しいプロジェクトのひな形を作成します。

```
$ django-admin startproject <プロジェクト名> [<ディレクトリ>]
```

第一引数のプロジェクト名と同名の「ベースディレクトリ」と「設定ファイルディレクトリ」(第 3 章を参照) が作成されます。第二引数にはベースディレクトリを作成する場所を指定します。もしこれを省略した場合は現在のディレクトリの直下にベースディレクトリが作成され、「. (ドット)」を指定した場合はベースディレクトリを作成せずに現在のディレクトリ直下にプロジェクト固有のモジュール群が生成されます。

2. startapp (プロジェクトにアプリケーションのひな形を追加)

プロジェクトにアプリケーションのひな形を追加します。このコマンドを実行しても、設定ファイルの「INSTALLED_APPS」にはアプリケーションは追加されないため、手動で追加する必要がありますのでご注意ください。

```
$ python3 manage.py startapp <アプリケーション名>
```

^{*30} <https://docs.djangoproject.com/ja/2.2/ref/django-admin/#available-commands>

3. makemigrations (マイグレーションファイルを生成)

モデルの変更差分から、マイグレーションファイルを自動生成します。

```
$ python3 manage.py makemigrations [<アプリケーション名>]
```

4. migrate (マイグレーションファイルからテーブルを作成・変更)

マイグレーションファイルの内容に基づいて、データベースのテーブルを作成・変更します。

```
$ python3 manage.py migrate [<アプリケーション名>]
```

5. createsuperuser (システム管理者を作成)

対話式で superuser (システム管理者) のレコードを作成することができます。

```
$ python3 manage.py createsuperuser
```

```
Username (leave blank to use 'root'): admin
Email address: admin@example.com
Password: admin12345
Password (again): admin12345
Superuser created successfully.
```

このコマンドで作成されたユーザーは `is_superuser`、`is_staff`、`is_active` がいずれも `True` となるため、第 13 章で解説する「管理サイト」にログインすることができ、登録されたすべてのモデルの操作をおこなうことができます。

6. loaddata (ファイルに書かれたレコードをデータベースにインポート)

ファイルに書かれたデータレコードをデータベースにインポートすることができます。テスト実施前にまとめたデータを準備する必要があるときなどで重宝します。

```
$ python3 manage.py loaddata <ファイルパス>
```

なお、データファイルは `dumpdata` コマンドで出力することが可能です。デフォルトの出力フォーマットは JSON 形式なので、その際のファイルの拡張子は「.json」としておきます。

7. runserver（開発用の Web サーバを起動）

開発用の Web サーバを起動します。詳細については 第 12 章 で解説をしています。

```
$ python3 manage.py runserver [<IP アドレス>:<ポート番号>]
```

8. shell（Django のプロジェクト設定を読み込んだ REPL を起動）

Django のプロジェクト設定を読み込んだ REPL（対話型評価環境）を起動します。見た目は通常の Python の REPL と同じですが、Django の設定ファイルが読み込まれてセットアップが完了した状態になっているので、モデルクラスを使って設定ファイルに定義したデータベースにクエリを発行するなどの検証をおこなうことができます。

```
$ python3 manage.py shell
```

9. collectstatic（静的ファイルを公開用ディレクトリに収集）

規定の場所に配置した静的ファイルを公開用ディレクトリに収集します。`runserver` には静的ファイルの自動配置機能が備わっているので、`runserver` で動作確認をおこなう場合には事前に実行しておく必要はないでしょう。静的ファイルのパスについては、第 10 章の「10.4 静的ファイル関連の設定」で詳しく説明しています。

```
$ python3 manage.py collectstatic
```

10. test（ユニットテストを実行）

本書では説明していませんが、ユニットテストを実行する際に使用するコマンドです。Django 標準のテストランナーは、名前が「test」で始まるモジュール内の、`TestCase` クラスを継承したクラスの、名前が「test」で始まるメソッドをテスト実行対象として自動で収集して実行してくれます。

```
$ python3 manage.py test
```

現場で使える Django の教科書 《基礎編》

2018 年 8 月 21 日 v1.0.0

2019 年 10 月 26 日 v1.2.0 (第 5 版)

著 者 横瀬 明仁 (akiyoko)

デザイン 橋本 OSO 鉄也

(C) 2019 akiyoko blog