ROZDZIAŁ 4

DJANGO

4 Django

4.1 Podstawowe pojęcia w tworzeniu stron internetowych

Tworzenie aplikacji internetowych wymaga zrozumienia podstawowych pojęć, które kształtują sposób działania nowoczesnych stron i systemów webowych. Załóżmy, że zamierzamy zbudować sklep internetowy i opublikować go pod adresem *ShopifyX.com*. Taka witryna składa się z dwóch głównych części: front-endu, który odpowiada za interfejs użytkownika i jest wyświetlany w przeglądarce, oraz back-endu, który działa na serwerze i odpowiada za przetwarzanie danych, zarządzanie użytkownikami, obsługę zamówień i integrację z bazą danych. W naszym przypadku do budowy back-endu wykorzystamy Django – popularny framework webowy w Pythonie.

Wyobraźmy sobie, że użytkownik chce odwiedzić sklep internetowy. Wpisuje w przeglądarce adres i naciska Enter. W tym momencie jego przeglądarka wysyła żądanie HTTP do serwera hostującego stronę. Serwer odbiera żądanie, przetwarza je i zwraca odpowiedź – na przykład stronę HTML z listą produktów. Przeglądarka klienta wyświetla stronę na podstawie otrzymanych danych. Do wymiany danych między klientem a serwerem wykorzystywany jest protokół HTTP, który określa zasady komunikacji w Internecie.

Tradycyjnie serwer zwracał całe strony HTML, ale obecnie coraz częściej stosuje się podejście API-first, gdzie back-end odpowiada jedynie za przetwarzanie i udostępnianie danych w formacie JSON, a front-end zajmuje się ich wyświetlaniem. Takie rozwiązanie pozwala na większą elastyczność, umożliwiając jednoczesne tworzenie różnych wersji aplikacji, na przykład strony internetowej i aplikacji mobilnej, które korzystają z tego samego API. Współczesne technologie takie jak React, Angular i Vue umożliwiają dynamiczne generowanie stron po stronie klienta, co poprawia wydajność i wygodę użytkowania.

Django, w przeciwieństwie do tych narzędzi, jest frameworkiem back-endowym i służy do budowy warstwy serwerowej aplikacji. Jego głównym zadaniem jest obsługa żądań od klientów, zarządzanie bazą danych i dostarczanie odpowiedzi w formie dynamicznych stron lub API. W porównaniu do innych frameworków serwerowych, takich jak ASP.NET Core dla C#, czy Express dla JavaScriptu, Django wyróżnia się wysoką produktywnością i wbudowanymi mechanizmami zabezpieczeń.

W nowoczesnym podejściu do aplikacji webowych serwer staje się bramą do danych, dostarczając punkty końcowe (endpointy), z którymi klient może się komunikować w celu pobierania lub zapisywania informacji. Przykładowo, jeden endpoint może zwracać listę produktów, a inny przyjmować zamówienia. Zbiór tych punktów tworzy spójny interfejs komunikacyjny — API (Application

Programming Interface) — stanowiący standardowy sposób interakcji między systemami.

W naszym projekcie skupimy się na używaniu Django do budowy API dla sklepu internetowego. Jeśli znasz podstawy front-endu, możesz później stworzyć aplikację kliencką, która będzie komunikować się z naszym API.

Teraz, gdy mamy ogólny obraz działania aplikacji webowej, jesteśmy gotowi do stworzenia pierwszego projektu w Django.

4.2 Instalacja i konfiguracja Django

Aby rozpocząć pracę z Django, musimy zainstalować tę bibliotekę w środowisku Pythona. Na potrzeby instalacji przyjmijmy, że mamy zainstalowanego Pythona w wersji 3.12, a sama instalacja będzie instalacją globalną. Proces instalacji Django rozpoczynamy od otwarcia terminala systemu operacyjnego i wpisaniu jednego z poleceń:

```
/usr/local/bin/python3.12 -m pip install django
```

Listing 4.1 Instalacja Django dla systemu macOS

```
py -3.12 -m pip install django
```

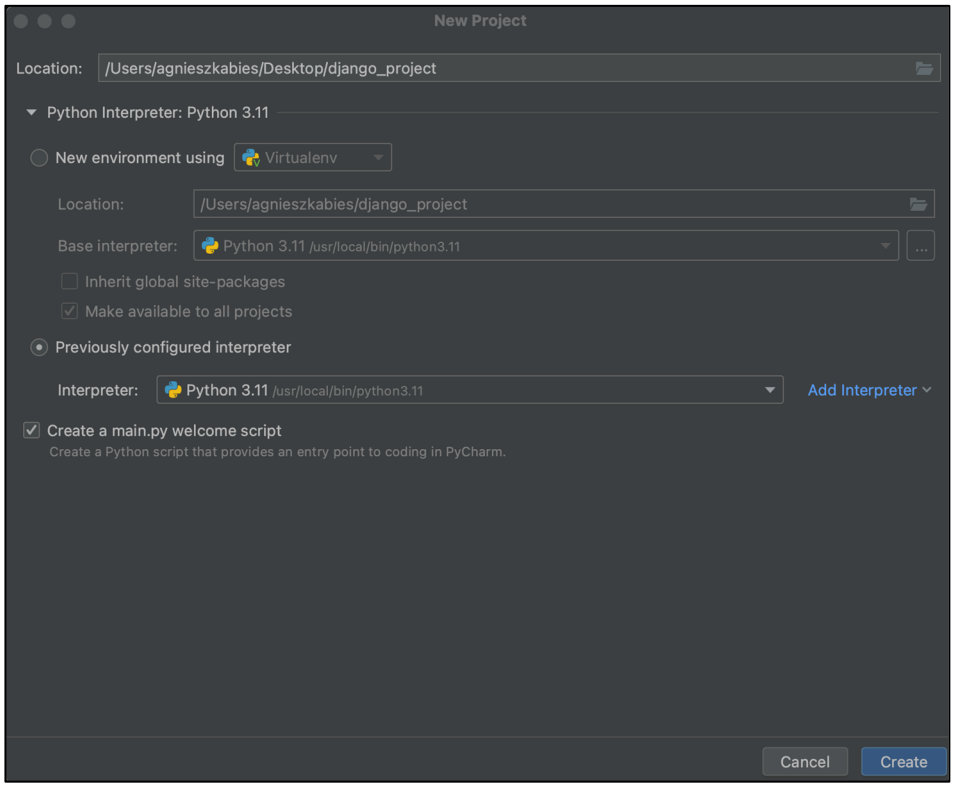
Listing 4.2 Instalacja Django dla systemu Windows

```
/usr/bin/python3.12 -m pip install django
```

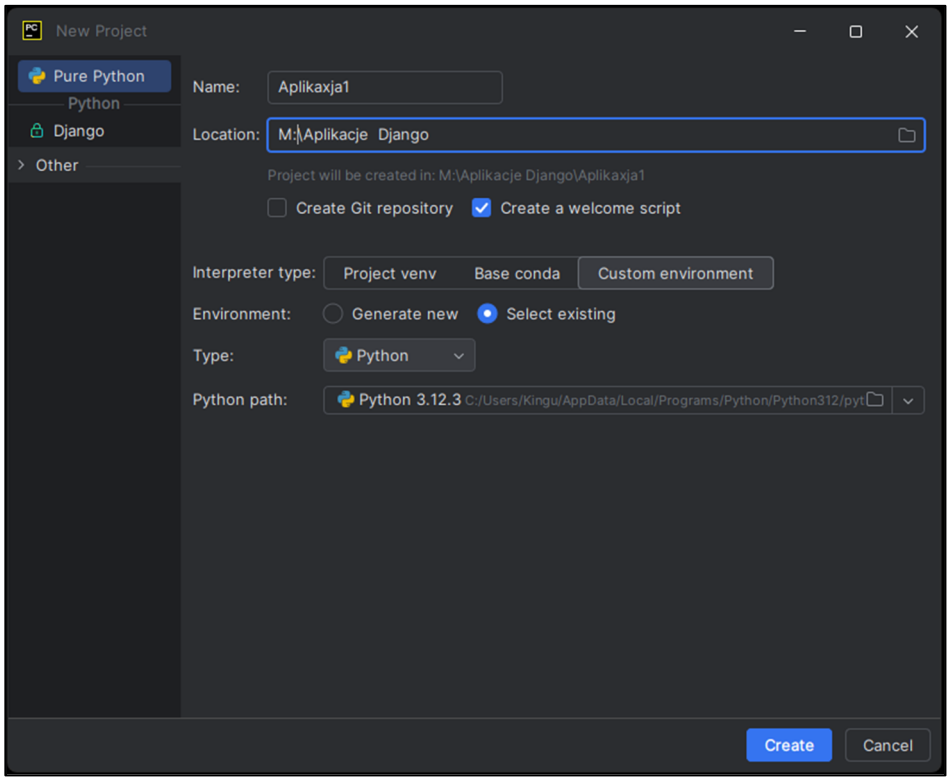
Listing 4.3 Instalacja Django dla systemu Linux

Kolejnym krokiem jest utworzenie nowego projektu w środowisku programistycznym PyCharm. Uruchamiamy program i wybieramy opcję „Create New Project" w oknie startowym lub przez menu „File" w górnej części ekranu. Podczas tworzenia nowego projektu w PyCharm, zostaniemy poproszeni o podanie kilku kluczowych informacji:

1. Nazwa i lokalizacja projektu: Wybieramy folder, w którym chcemy przechowywać swój projekt. Możemy użyć domyślnej ścieżki sugerowanej przez PyCharm lub wybrać własną lokalizację, klikając przycisk „Browse".
2. Wybór środowiska projektu (environment): W oknie tworzenia nowego projektu wybieramy istniejące środowisko Pythona wraz z interpreterem, który był użyty do instalacji Django.



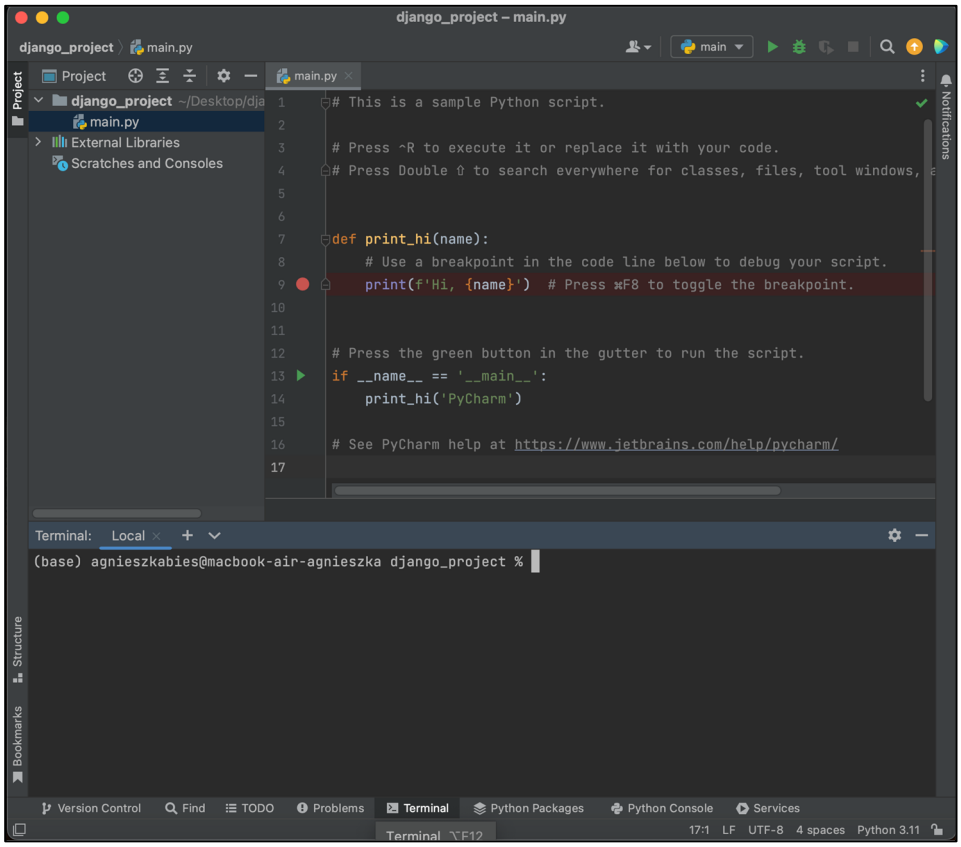
Rysunek 4.1 Ustawienia nowego projektu w PyCharm dla systemu MacOS



Rysunek 4.2 Ustawienia nowego projektu w PyCharm dla systemu Windows

Wybór opcji potwierdzamy przyciskiem „Create” na dole okna dialogowego. PyCharm utworzy nowy projekt z wybraną konfiguracją. Następnie przechodzimy do katalogu naszego projektu i usuwamy plik *main.py*.

W kolejnym etapie przechodzimy do terminala w PyCharm.



Rysunek 4.3 Wyświetlenie Terminala w PyCharm

Następnie wpisujemy komendę:

```
django-admin startproject nazwa_projektu
```

Listing 4.4 Tworzenie nowego projektu Django

Posłuży ona do utworzenia nowego projektu Django. Aby utworzyć nową aplikację w projekcie Django, należy przejść w terminalu do folderu projektu (`cd nazwa_projektu`) i skorzystać z komendy:

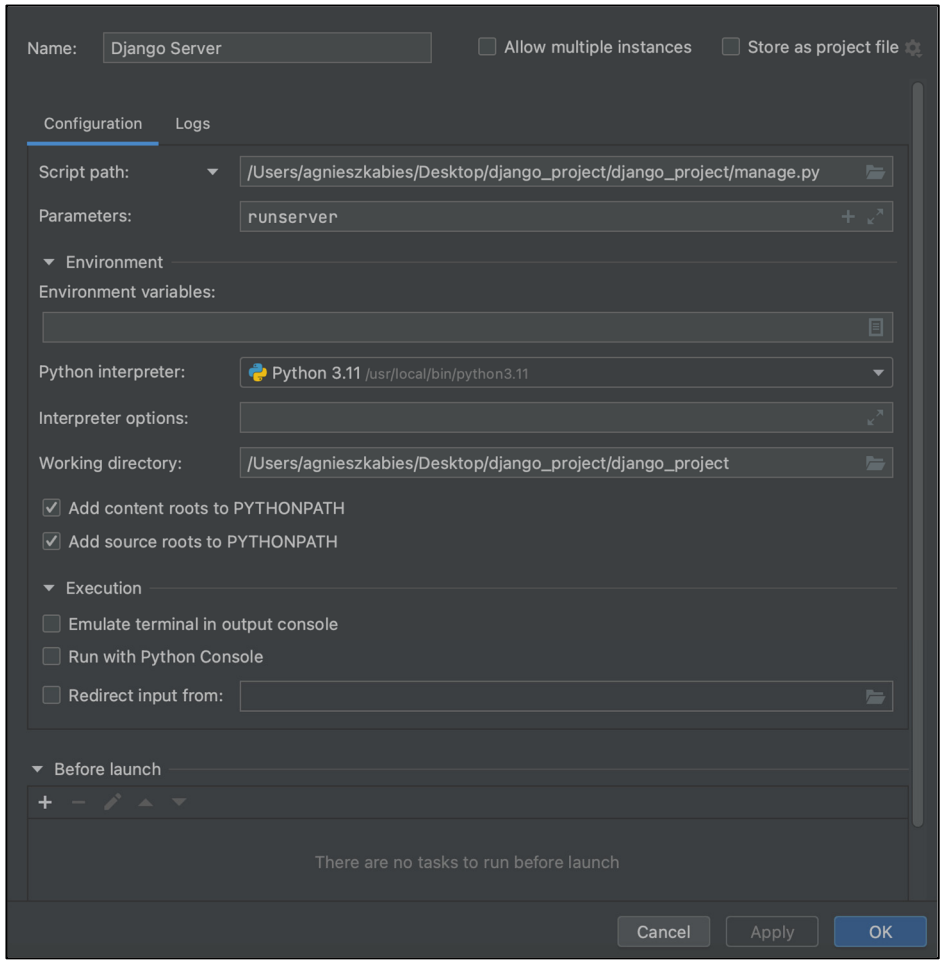
```
django-admin startapp nazwa_aplikacji
```

Listing 4.5 Tworzenie nowej aplikacji w projekcie Django

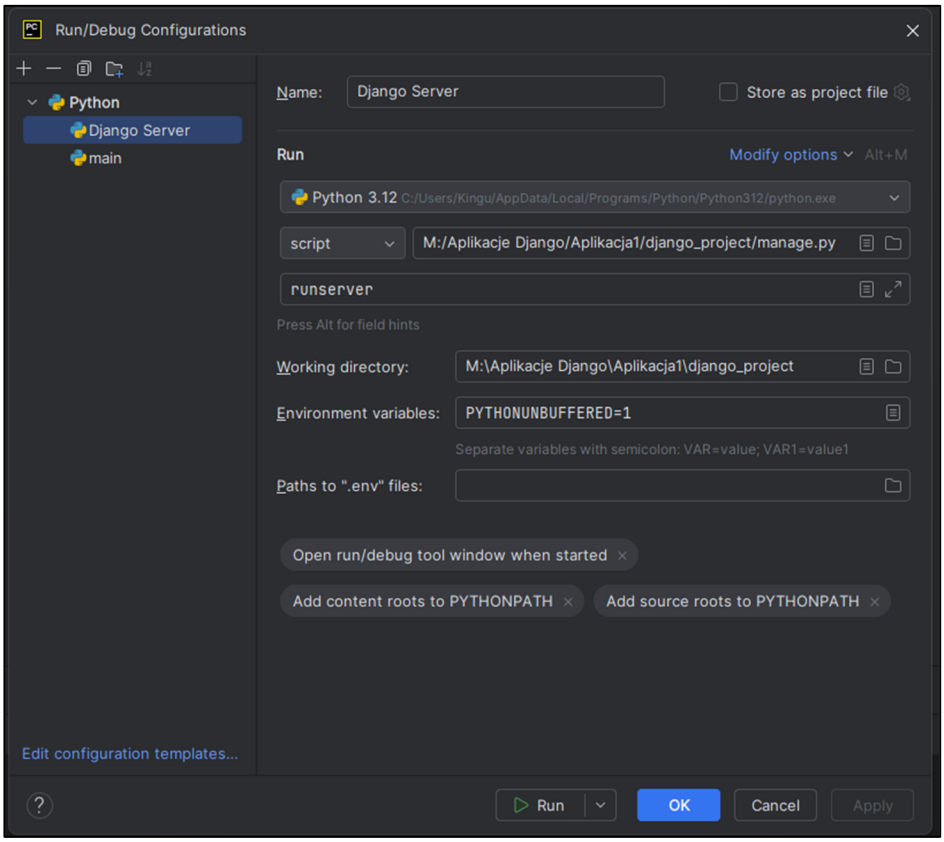
Aby uruchomić serwer Django bezpośrednio z PyCharm, należy skonfigurować środowisko uruchamiania. W górnym menu wybieramy *Run* → *Edit Configurations*, co otworzy okno konfiguracji. Klikamy "+" i wybieramy nową konfigurację Python, a w niej:

- nadajemy nazwę, np. "Django Server";
- wskazujemy lokalizację zawierający plik *manage.py* projektu;
- w polu *Parameters* wpisujemy *runserver*.
- wskazujemy lokalizację projektu Django.

Na koniec potwierdzamy konfigurację. Od tej chwili serwer Django można uruchamiać bezpośrednio z PyCharm (zielony przycisk w kształcie strzałki).



Rysunek 4.4 Ustawienia konfiguracji środowiska uruchomieniowego Django w PyCharm w systemie MacOS



Rysunek 4.5 Ustawienia konfiguracji środowiska uruchomieniowego Django w PyCharm w systemie Windows

Serwer Django możemy również uruchomić poprzez Terminal PyCharm. Wystarczy przejść do lokalizacji, w której znajduje się plik *manage.py* naszego projektu za pomocą poniższego polecenia.

```
cd nazwa_projektu_django
```

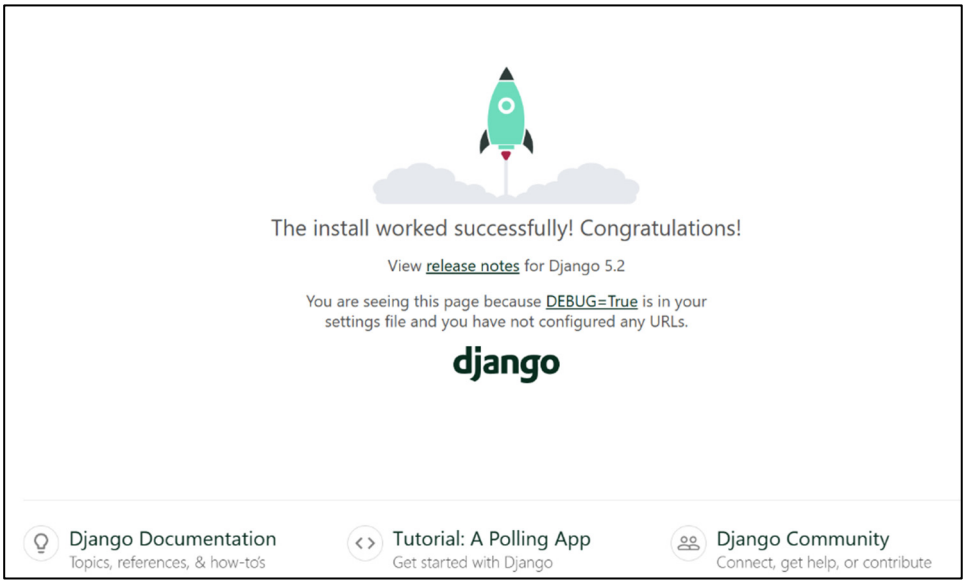
Listing 4.6 Przejście w terminalu do lokalizacji projektu Django

Następnie uruchamiamy serwer poleceniem:

```
python manage.py runserver
```

Listing 4.7 Uruchomienie serwera Django w terminalu

Aby uzyskać dostęp do naszego serwera Django, otwieramy przeglądarkę internetową i wpisujemy lokalny adres IP wraz z portem: *127.0.0.1:8000* lub *localhost:8000*.



Rysunek 4.6 Strona powitalna Django w przeglądarce.

Zobaczymy domyślną stronę powitalną Django, która potwierdza, że serwer działa poprawnie. Strona ta zwykle wyświetla komunikat „The install worked successfully! Congratulations!" wraz z wersją Django i krótkim przewodnikiem co robić dalej.

4.2.1 Sprawdź się!

1. Czym jest front-end w aplikacji internetowej?
   a) Część aplikacji działająca na serwerze, odpowiedzialna za przetwarzanie danych
   b) Część aplikacji, która odpowiada za logikę biznesową
   c) Część aplikacji ładowana w przeglądarce użytkownika, z którą użytkownik wchodzi w interakcję
   d) System do zarządzania bazą danych

2. Jaki protokół definiuje sposób wymiany danych między klientem a serwerem?
   a) FTP
   b) SMTP

c) HTTP
d) SSH

3. Jakie narzędzia są używane do generowania stron internetowych po stronie klienta?
   a) Django, Flask, ASP.NET
   b) MySQL, MongoDB, PostgreSQL
   c) React, Angular, Vue
   d) Apache, Nginx, IIS

4. Co to jest API w kontekście aplikacji internetowych?
   a) System zarządzania bazą danych
   b) Interfejs do komunikacji między różnymi aplikacjami
   c) Narzędzie do tworzenia interfejsów graficznych
   d) Rodzaj bazy danych

5. Co zobaczymy po uruchomieniu serwera Django i otwarciu przeglądarki pod adresem 127.0.0.1:8000?
   a) Stronę błędu
   b) Domyślną stronę powitalną Django
   c) Stronę logowania do Django
   d) Listę plików projektu

4.3 Struktura projektu Django

Podczas tworzenia projektu Django za pomocą polecenia `django-admin startproject`, w katalogu projektu zostaną utworzone następujące pliki i foldery:

- *__init__.py* – plik ten informuje Pythona, że dany katalog jest modułem.
- *settings.py* – zawiera konfigurację projektu Django, m.in. ustawienia bazy danych, debugowania, zainstalowanych aplikacji itp.
- *urls.py* – definiuje główne mapowanie adresów URL. Na jego podstawie Django kieruje żądania HTTP do odpowiednich widoków lub dalszych mapowań w aplikacjach.
- *asgi.py* i *wsgi.py* – pliki służące jako punkty wejścia dla serwerów ASGI i WSGI, umożliwiające komunikację między serwerem, a aplikacją Django podczas wdrożenia.

- *manage.py* – skrypt służący do zarządzania projektem Django. Umożliwia uruchamianie poleceń administracyjnych, takich jak migracje, uruchamianie serwera deweloperskiego czy tworzenie aplikacji.

Polecenie `startproject` tworzy też podkatalog o tej samej nazwie co projekt, który zawiera właśnie pliki: *__init__.py*, *settings.py*, *urls.py*, *asgi.py* i *wsgi.py*. W katalogu nadrzędnym znajduje się natomiast tylko plik *manage.py*.

Plik *manage.py* służy do zarządzania projektem Django. Oto niektóre z najważniejszych poleceń:

- `runserver` – uruchamia wbudowany serwer deweloperski HTTP Django, który pozwala testować aplikację lokalnie.
- `startapp` – tworzy nową aplikację Django w ramach projektu.
- `shell` – uruchamia interaktywny interpreter Pythona z załadowanymi ustawieniami projektu Django.
- `dbshell` – otwiera interaktywną powłokę połączoną z bazą danych skonfigurowaną w ustawieniach projektu.
- `makemigrations` – generuje pliki migracji na podstawie zmian w modelach danych.
- `migrate` – stosuje migracje utworzone poleceniem `makemigrations`, aktualizując strukturę bazy danych.
- `test` – uruchamia testy automatyczne zdefiniowane w projekcie.

W pliku *settings.py* znajdujemy różnorodne konfiguracje, ale skoncentrujemy się na sekcji `INSTALLED_APPS`, która wygląda następująco:

```
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
]
```

Listing 4.8 Sekcja INSTALLED_APPS w pliku setting.py

Domyślnie każdy projekt Django zawiera kilka podstawowych aplikacji:

- `Admin` – dostarcza interfejs administracyjny do zarządzania danymi w projekcie.
- `Auth` – obsługuje uwierzytelnianie, autoryzację i zarządzanie użytkownikami.
- `ContentTypes` – umożliwia śledzenie modeli zarejestrowanych w projekcie i zarządzanie różnymi typami treści.
- `Sessions` – służy do przechowywania danych sesyjnych użytkowników po stronie serwera.
- `Messages` – umożliwia wyświetlanie jednorazowych komunikatów (powiadomień) użytkownikom.
- `StaticFiles` – odpowiada za obsługę plików statycznych, takich jak obrazy, arkusze CSS czy skrypty JavaScript.

Po utworzeniu nowej aplikacji w naszym projekcie Django (np. first_app), pojawia się folder z charakterystyczną dla aplikacji strukturą.

W folderze znajdują się m.in.:

- *migrations/* – służy do przechowywania migracji bazy danych,
- *admin.py* – miejsce, w którym definiuje się interfejs administracyjny,
- *apps.py* – zawiera konfigurację aplikacji (AppConfig),
- *models.py* – do definiowania klas modeli odpowiadających tabelom w bazie danych,
- *tests.py* – do pisania testów jednostkowych,
- *views.py* – zawiera widoki, czyli funkcje lub klasy obsługujące żądania HTTP.

Taka struktura tworzy spójny „szkielet" aplikacji, który ułatwia organizację kodu, testowanie i późniejsze rozbudowywanie funkcjonalności projektu.

Po stworzeniu nowej aplikacji, konieczne jest jej zarejestrowanie w module ustawień. Dodajemy jej nazwę do listy `INSTALLED_APPS` w pliku *setting.py*. Dzięki temu Django będzie wiedziało, że ma uwzględnić nową aplikację w procesie działania projektu.

```
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "first_app",
]
```

Listing 4.9 Sekcja INSTALLED_APPS w pliku setting.py

wraz z dodaną aplikacją

4.4 Widoki i szablony

W aplikacjach Django większość logiki przetwarzania żądań znajduje się w widokach. Kiedy użytkownik odwiedza stronę, jego przeglądarka wysyła żądanie HTTP do serwera w celu pobrania danych (szczegóły dotyczące żądań HTTP omówimy w kolejnym punkcie).

Widok to funkcja (lub klasa), która odbiera to żądanie jako obiekt Pythona - konkretnie `HttpRequest` platformy Django. Widok decyduje, jak odpowiedzieć na żądanie i jakie dane przesłać użytkownikowi. Wynikiem działania widoku musi być obiekt `HttpResponse`, który zawiera treść odpowiedzi, status HTTP oraz ewentualne nagłówki.

Widok może również pobierać dane przekazane w adresie URL, np. identyfikator konkretnego zasobu. Zgodnie z typowym wzorcem projektowym widok wykonuje zapytanie do bazy danych przy użyciu ORM Django, korzystając z otrzymanego identyfikatora. Następnie może wyrenderować szablon HTML, wypełniając go danymi pobranymi z bazy. Wyrenderowany szablon jest osadzany jako treść obiektu `HttpResponse` i zwracany do przeglądarki użytkownika. Django automatycznie przesyła te dane z powrotem do klienta.

4.4.1 Tworzenie pierwszego widoku

Aby utworzyć naszą pierwszą funkcję widoku (nazwiemy ją `greeting()`) przechodzimy do pliku *views.py* naszej aplikacji. Funkcja powinna przyjąć obiekt żądania i zwrócić odpowiedź. Na wstępie z pakietu `django.http`

zaimportujemy klasę `HttpResponse`. W funkcji możemy realizować różne działania, takie jak pobieranie danych z bazy danych, ich przekształcanie, wysyłanie e-maili, itp. Na początek jednak zwrócimy prostą odpowiedź za pomocą instancji klasy `HttpResponse`, zawierającą tekst "Hello User!".

```
from django.shortcuts import render

from django.http import HttpResponse


def greeting(request):

    return HttpResponse("Hello User!")
```

Listing 4.10 Funkcja widoku zwracająca napis „Hello User”

4.4.2 Konfiguracja URL dla widoku

Załóżmy teraz, że za każdym razem, gdy wysyłamy żądanie do adresu *first_app/greeting*, nasza funkcja widoku powinna być wywoływana i zwracać "Hello User!". W folderze *first_app* dodajmy nowy plik o nazwie *urls.py*. Moglibyśmy nazwać go inaczej, ale standardowo nazywamy go właśnie *urls*.

Zamierzamy zmapować nasze URL-e do funkcji widoków. Na początku importujemy funkcję `path` z `django.urls`. Musimy również zaimportować moduł `views` z bieżącego folderu, aby móc odwołać się do naszej funkcji widoku. Następnie definiujemy specjalną zmienną `urlpatterns`, którą ustawiamy jako tablicę obiektów wzorców URL. Używamy funkcji `path()`, aby stworzyć obiekt wzorca URL. Funkcja ta ma kilka parametrów. Pierwszy to `route`, który jest ciągiem znaków określającym ścieżkę URL. Drugi parametr to `view`, który jest funkcją zwracającą obiekt odpowiedzi HTTP.

Oto jak wygląda kod w pliku *urls.py* w folderze *first_app*.

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

urlpatterns = [
    path('greeting/', views.greeting)

]
```

Listing 4.11 Deklaracja URL w stworzonym pliku urls.py

4.4.3 Rejestracja URL-i aplikacji w głównym projekcie

Każda aplikacja może mieć własną konfigurację URL. Teraz musimy zaimportować tę konfigurację URL do głównej konfiguracji URL projektu (plik *urls.py* w folderze projektu). Należy również zaimportować `include` z `django.urls`.

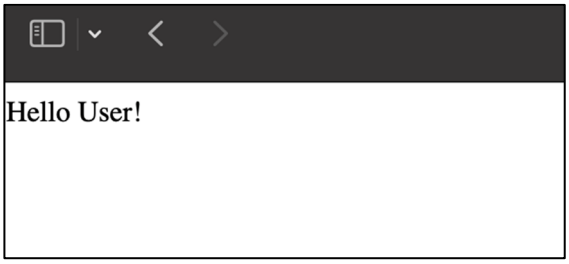
```
from django.contrib import admin
from django.urls import path, include

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

Listing 4.12 Deklaracja URL-i w głównym pliku urls.py

W ten sposób deklarujemy, że wszystkie URL-e, które zaczynają się od *first_app/*, powinny być przekierowywane do naszej aplikacji `first_app`. Funkcja `include` wymaga parametru, który jest ciągiem znaków; tutaj odwołujemy się do `first_app.urls`. Dzięki temu, jeśli zostanie wysłane żądanie do *first_app/greeting*, Django wie, że wszystkie żądania zaczynające się od *first_app* powinny być obsługiwane przez tę aplikację. Django odcina pierwszą część URL-a i przekazuje resztę do modułu konfiguracji URL w aplikacji `first_app` (plik *urls.py* w folderze *first_app*).

Upewnijmy się, że wszystko jest poprawnie skonfigurowane. Otwórzmy okno terminala (należy uruchomić serwer) i sprawdźmy, czy nie ma błędów. Po zweryfikowaniu wracamy do przeglądarki i wysyłamy żądanie do utworzonego punktu końcowego (*http://127.0.0.1:8000/first_app/greeting/*). Powinniśmy zobaczyć na ekranie "Hello User!".



Rysunek 4.7 Wynik żądania końcowego http://127.0.0.1:8000/first_app/greeting/

4.4.4 Przekazywanie danych w żądaniach HTTP

W ramach żądania HTTP dane mogą być przekazywane jako parametry w adresie URL (metoda GET) lub w ciele żądania (metoda POST). Metoda GET umieszcza parametry bezpośrednio w adresie URL, co sprawia, że są one widoczne dla użytkownika w pasku adresu przeglądarki (parametry znajdują się za znakiem zapytania). Przykładowo, w adresie:

https://www.example.com/search?query=Django&category=tutorial&page=1,

parametry to "query", "category" i "page".

Metoda POST z kolei przekazuje parametry w ciele żądania, co oznacza, że są one niewidoczne dla użytkownika. Metoda POST jest często wykorzystywana do przekazywania większych ilości danych lub poufnych informacji, ponieważ zapewnia większą prywatność i bezpieczeństwo. Django automatycznie przekształca te parametry w obiekty `QueryDict`, które są dostępne w obiekcie `HttpRequest` przekazywanym do widoku. Możemy uzyskać do nich dostęp poprzez atrybuty `HttpRequest.GET` i `HttpRequest.POST`, odpowiednio dla parametrów z adresu URL i ciała żądania.

Oto kolejny krok w ulepszeniu naszego widoku. Naszym celem jest teraz stworzenie spersonalizowanego powitania dla konkretnego użytkownika. W pliku widoku naszej aplikacji w funkcji `greeting()` dodajmy zmienną o nazwie `name`, która będzie przechowywała imię użytkownika pobrane z parametrów GET. Funkcja będzie zwracała tekst powitalny składający się z "Hello" i imienia przekazanego poprzez parametr GET (metoda `format()` wstawia wartość zmiennej `name` w miejsce `{}`).

```
from django.shortcuts import render
from django.http import HttpResponse


def greeting(request):
    name = request.GET.get("name") or "User"
    return HttpResponse("Hello {}!".format(name))
```

Listing 4.13 Plik widoku views.py z obsługą parametru GET

Aby przetestować działanie funkcji powitalnej w przeglądarce, wystarczy do adresu *http://127.0.0.1:8000/first_app/greeting/* dopisać parametr *name* z dowolnym imieniem, np. *http://127.0.0.1:8000/first_app/greeting/?name=Jan*.

4.4.5 Zastosowanie szablonów w Django

To, co często nazywamy widokiem w innych frameworkach, w Django nazywane jest szablonem. Szablony to pliki HTML (lub inne pliki tekstowe), które zawierają specjalne znaczniki umożliwiające zastępowanie ich wartościami zmiennych z aplikacji. Na przykład, aplikacja może generować listę produktów w postaci tabeli lub galerii. Szablony mogą również importować inne pliki HTML, aby w różny sposób prezentować dane.

Django dba o bezpieczeństwo, automatycznie kodując znaki specjalne w zmiennych, co zapobiega atakom typu XSS (Cross-Site Scripting). Szablony w Django mogą wykorzystywać zmienne, pętle i instrukcje warunkowe, co pozwala na dynamiczne generowanie zawartości stron. Dodatkowo można stosować filtry i tagi szablonowe, które ułatwiają formatowanie danych i wykonywanie bardziej złożonych operacji. Dzięki temu szablony są elastycznym narzędziem do tworzenia responsywnych i interaktywnych stron internetowych.

Zatem jak możemy użyć szablonu, aby zwrócić treść w HTML klientowi? W aplikacji `first_app` dodajmy nowy folder o nazwie *templates*, a w tym folderze dodajmy nowy plik *page.html* (oczywiście możemy go nazwać jakkolwiek). Tutaj możemy urozmaicić nasz tekst „Hello User!”.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    <h1>Hello User!</h1>
</body>
</html>
```

Listing 4.14 Przykładowy szablon HTML - plik page.html

Teraz wróćmy do naszej funkcji widoku. Zamiast zwracać `HttpResponse`, użyjemy funkcji `render()`, przekazując dla niej instancję `request` i nazwę naszego szablonu.

```
from django.shortcuts import render
from django.http import HttpResponse


def greeting(request):
    return render(request, "page.html")
```

Listing 4.15 Przykład prostego widoku w Django, który renderuje szablon HTML 'page.html' w odpowiedzi na zapytanie HTTP - plik views.py

Upewnijmy się, że nasz projekt działa. Sprawdźmy to w przeglądarce pod adresem *http://127.0.0.1:8000/first_app/greeting/.*

Rysunek 4.8 Wynik działania widoku w Django, który wyświetla stronę HTML z komunikatem 'Hello User!'

Oczywiście, szablony w Django to nie tylko statyczne pliki. Mogą zawierać zmienne, które są interpretowane podczas renderowania. Większość tych zmiennych jest przekazywana do szablonu przez widok za pomocą słownika. Aby urozmaicić dotychczasowy kod, możemy zamiast statycznego "User", dynamicznie renderować jakąś wartość. Aby wyrenderować zmienną w szablonie, umieszczamy ją w dwóch nawiasach klamrowych {{ }}.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    <h1>Hello {{ name }}!</h1>
</body>
</html>
```

Listing 4.16 Przykładowy szablon HTML ze zmienną name

Teraz, po odświeżeniu przeglądarki, zobaczymy tylko "Hello!". Stało się tak, ponieważ szablon nie wyrenderował żadnej zawartości. Jest to wynik braku konfiguracji w naszym widoku. Otwórzmy więc plik *views.py* i dodajmy zmienną, `name` z wartością "User" i przypiszmy ją do klucza `name` słownika przekazanego do funkcji `render()`. Wartości przypisane do kluczy słownika zostaną wyrenderowane w szablonie.

```
from django.shortcuts import render
from django.http import HttpResponse


def greeting(request):
    name = "User"
    return render(request,"page.html", {"name": name})
```

Listing 4.17 Widok funkcji greeting() renderującej szablon HTML z dynamiczną zmienną

Po wprowadzonych zmianach i ponownym odświeżeniu strony, powinniśmy zobaczyć napis "Hello User". Manipulując zmienną `name`, możemy dostosować powitanie dla każdego użytkownika, który odwiedza naszą stronę. Na przykład, możemy dodać obsługę parametrów GET, aby dynamicznie dostosować powitanie na podstawie danych przekazanych w URL. Zmodyfikujmy ponownie plik *views.py*.

```
from django.shortcuts import render
from django.http import HttpResponse


def greeting(request):
    name = request.GET.get("name") or "User"
    return render(request, "page.html", {"name": name})
```

Listing 4.18 Widok funkcji greeting() renderującej szablon HTML z dynamiczną zmienną name pobraną z parametrów GET

Możemy teraz przetestować dynamiczne powitanie, odwiedzając np. adres *http://127.0.0.1:8000/first_app/greeting/?name=Wojtek.* Powinniśmy zobaczyć "Hello Wojtek!". Jeśli parametr "name" nie zostanie przekazany, zobaczymy domyślną wartość "Hello User!".

4.4.6 Sprawdź się!

1. Co zawiera plik *settings.py* w projekcie Django?

a) Kod źródłowy aplikacji
b) Konfiguracje projektu, takie jak baza danych i ustawienia debugowania
c) Mapowanie adresów URL
d) Plik szablonów HTML

2. Które z poniższych poleceń Django służy do uruchamiania serwera roboczego HTTP?

a) runserver
b) startapp
c) migrate
d) shell

3. Jaki jest cel pliku *manage.py* w projekcie Django?

a) Przechowywanie konfiguracji serwera
b) Zarządzanie migracjami bazy danych
c) Zarządzanie projektem Django, w tym uruchamianie serwera i tworzenie aplikacji
d) Przechowywanie szablonów HTML

4. Czym jest widok w Django?

a) Szablon HTML używany do generowania stron
b) Narzędzie do zarządzania bazą danych
c) Funkcja obsługująca żądania HTTP i zwracająca odpowiedź HTTP
d) Moduł do obsługi plików statycznych

5. Jaka jest rola pliku *urls.py* w aplikacji Django?

a) Definiuje mapowanie adresów URL do widoków
b) Przechowuje widoki aplikacji
c) Przechowuje konfiguracje bazy danych
d) Przechowuje szablony HTML

6. Jakie parametry przekazuje się do funkcji path() w pliku *urls.py* aplikacji?

a) Nazwa szablonu i dane do renderowania
b) Ścieżka URL i widok, który ma zostać wywołany
c) Typ metody HTTP i nagłówki odpowiedzi
d) Nazwa aplikacji i konfiguracja bazy danych

7. Co się dzieje, gdy w przeglądarce odwiedzasz adres URL zawierający parametry GET?

a) Parametry są przekazywane jako nagłówki żądania
b) Parametry są widoczne w adresie URL i dostępne w widoku poprzez HttpRequest.GET
c) Parametry są ukryte w ciele żądania i dostępne w widoku poprzez HttpRequest.POST
d) Parametry są ignorowane przez serwer

8. Która funkcja Django jest używana do renderowania szablonów HTML?

a) HttpResponse
b) include
c) path
d) render

9. Jakie dane można przekazywać do szablonów w Django?

a) Dowolne zmienne w postaci słownika
b) Tylko statyczne teksty
c) Tylko adresy URL
d) Tylko obiekty baz danych