Глава 1: Начальная настройка

В данной главе рассматривается, как правильно настроить ваш компьютер под управлением Windows или macOS для работы с проектами Django. Мы начнем с обзора командной строки, мощного текстового интерфейса, который разработчики широко используют для установки и настройки проектов Django. Затем мы установим последнюю версию Python, научимся создавать отдельные виртуальные окружения и установим Django. В заключительном этапе мы рассмотрим использование Git для контроля версий и работу с текстовым редактором. К концу этой главы вы создадите свой первый проект Django с нуля и, что более важно, сможете создавать и модифицировать новые проекты Django буквально в несколько нажатий клавиш.

Командная строка

Если вы когда-либо видели в телешоу или фильме сцену, где хакер с жаром набирает текст в черном окне, то это и есть командная строка. Это альтернатива мыши или интерфейсу, основанному на нажатии пальцем, знакомому большинству пользователей компьютера. Обычному пользователю компьютера никогда не потребуется использовать командную строку, но разработчики программного обеспечения делают это, так как некоторые задачи можно выполнить только с ее помощью. К ним относятся запуск программ, установка программного обеспечения, использование Git для контроля версий или подключение к серверам в облаке. С небольшой практикой большинство разработчиков обнаруживают, что командная строка на самом деле быстрый и более мощный способ навигации и управления компьютером.

Из-за минимального пользовательского интерфейса, состоящего всего из пустого экрана и мигающего курсора, командная строка может пугать новичков. Часто после выполнения команды нет обратной связи, и при неверном использовании команды можно случайно удалить содержимое всего компьютера: предупреждающее окно не появится! Поэтому командной строкой следует использовать осторожно. Убедитесь, что вы не слепо копируете и вставляете команды, которые находите в Интернете; полагайтесь только на надежные ресурсы для выполнения команд, которые вы полностью понимаете.

++++++++++++++++++++++

На практике существует несколько терминов, используемых для обозначения командной строки: интерфейс командной строки (Command Line Interface, CLI), консоль (console), терминал (terminal), оболочка (shell) или приглашение (prompt). Технически говоря, терминал - это программа, которая открывает новое окно для доступа к командной строке, консоль - это текстовое приложение, оболочка - это программа, которая выполняет команды в базовой операционной системе, и приглашение - это место, где вводятся и выполняются команды. Изначально можно запутаться в этих терминах, но в конечном итоге они в основном означают одно и то же: командная строка - это место, где мы запускаем и выполняем текстовые команды на нашем компьютере.

На операционной системе Windows встроенный терминал и оболочка оба называются PowerShell. Для доступа к нему найдите панель задач внизу экрана рядом с кнопкой Windows и введите "powershell", чтобы запустить приложение. Оно откроет новое окно с темно-синим фоном и мигающим курсором после приглашения ">". Вот как это выглядит на моем компьютере.

++++++++++

Перед приглашением ">" находится PS, что означает PowerShell, затем идет начальный каталог C операционной системы Windows, за которым следует каталог Users и имя текущего пользователя, которое на моих личных компьютерах - wsv. Ваше имя пользователя, очевидно, будет отличаться. На данном этапе не беспокойтесь о том, что находится слева от приглашения ">": это зависит от каждого компьютера и может быть настроено позже. В дальнейшем мы будем использовать более короткое приглашение ">" для Windows.

+++++++++

Существует множество доступных команд оболочки, но большинство разработчиков постоянно используют одни и те же несколько команд и ищут более сложные, когда это необходимо.

В большинстве случаев команды для Windows (PowerShell) и macOS схожи. Например, команда whoami возвращает имя компьютера/имя пользователя на Windows и только имя пользователя на macOS. Как и в случае с любыми командами оболочки, введите саму команду, а затем нажмите клавишу "Ввод". Обратите внимание, что символ "#" обозначает комментарий и не будет выполнен на командной строке.

++++++++++++++

Иногда, однако, команды оболочки на Windows и macOS полностью разные. Хорошим примером является команда для вывода простого сообщения "Hello, World!" в консоль. На Windows используется команда Write-Host, в то время как на macOS используется команда echo.

Частой задачей на командной строке является навигация в файловой системе компьютера. На Windows и macOS команда **pwd** (print working directory) выводит текущее расположение в файловой системе.

Вы можете сохранить свой код Django в любом месте на вашем компьютере, но для удобства мы разместим наш код в каталоге рабочего стола (desktop directory). Команда **cd** (change directory), за которой следует желаемое местоположение, работает на обеих системах.

Совет: клавиша Tab автоматически дополняет команду. Например, если вы введете **cd d** и затем нажмете клавишу Tab, она автоматически заполнит остальную часть имени. Если существует более двух каталогов, начинающихся с "d", нажмите клавишу Tab еще раз, чтобы перебрать их.

Чтобы создать новый каталог, используйте команду **mkdir**, за которой следует имя. Мы создадим каталог с именем "code" на Рабочем столе, а затем внутри него новый каталог с именем "ch1-setup".

Вы можете проверить, что каталог был создан, посмотрев на свой Рабочий стол или выполнить команду **ls**. Весь вывод для Windows немного длиннее, но здесь он сокращен для краткости.

Чтобы выйти из командной строки, вы можете закрыть терминал с помощью мыши, но "хакерский" способ - использовать команду оболочки **exit**. Это работает по умолчанию на Windows, но на macOS требуется изменить настройки Терминала. В верхней части экрана выберите "Terminal," затем "Preferences" в выпадающем меню. Выберите "Profiles" в верхнем меню и затем "Shell" из списка ниже. Там есть радиокнопка для "When the shell exits:". Выберите "Close the window" ("Закрыть окно").

Виртуальные окружения

Установка последних версий Python и Django - правильный подход для любого нового проекта. Однако в реальном мире часто бывает так, что существующие проекты зависят от старых версий каждого из них. Рассмотрим следующую ситуацию: проект A использует Django 2.2, но проект B использует Django 4.0. По умолчанию Python и Django устанавливаются глобально на компьютере, что означает, что достаточно сложно устанавливать и переустанавливать разные версии каждый раз, когда вы переключаетесь между проектами.

К счастью, существует простое решение. Виртуальные окружения позволяют вам создавать и управлять отдельными окружениями для каждого проекта Python на одном и том же компьютере. Существует много способов создания виртуальных окружений, но самый простой - это использовать модуль **venv**, который уже установлен в составе стандартной библиотеки Python 3. Для попробовать это, перейдите в уже существующий каталог "ch1-setup" на вашем Рабочем столе.

Разработчику предоставляется выбор подходящего имени для виртуального окружения, но обычно используется общепринятое имя **.venv**. Например, чтобы создать виртуальное окружение с именем **.venv**, выполните следующие команды:

На Windows: **python -m venv .venv** На macOS: **python3 -m venv .venv**

Теперь у вас будет создано виртуальное окружение внутри каталога "ch1-setup".

Если вы используете команду **ls** для просмотра текущего каталога, он может показаться пустым. Однако каталог **.venv** присутствует - просто он "скрыт" из-за точки **.** в начале имени. Скрытые файлы и каталоги - это способ, которым разработчики указывают, что содержимое важно и должно рассматриваться иначе, чем обычные файлы. Для просмотра скрытых файлов и каталогов используйте команду **ls -la**, которая покажет все каталоги и файлы, включая скрытые.

После создания виртуального окружения его необходимо активировать.

На Windows, для выполнения сценариев, необходимо настроить политику выполнения. Это мера безопасности. Документация Python рекомендует разрешить выполнение сценариев только для текущего пользователя (CurrentUser), именно это мы и сделаем. На macOS нет подобных ограничений для выполнения сценариев, поэтому можно просто выполнить команду **source .venv/bin/activate** для активации виртуального окружения. Вот как выглядают полные команды для создания и активации нового виртуального окружения с именем **.venv**:

На Windows:

1. **python -m venv .venv** (Создание виртуального окружения)
2. **Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy RemoteSigned** (Настройка политики выполнения)
3. **.venv\Scripts\Activate.ps1** (Активация виртуального окружения)

На macOS:

1. **python3 -m venv .venv** (Создание виртуального окружения)
2. **source .venv/bin/activate** (Активация виртуального окружения)

После активации виртуального окружения вы будете работать в изолированной среде с отдельными версиями Python и библиотеками, что позволит избежать конфликтов между проектами.

Теперь, когда Python установлен и мы знаем, как использовать виртуальные окружения, пришло время установить Django впервые.

Django размещен в Python Package Index (PyPI), центральном репозитории для большинства пакетов Python. Для установки последней версии Django используйте pip, самый популярный установщик пакетов, который поставляется вместе с Python 3. Для установки Django версии 4.0.0 с последующими обновлениями безопасности используйте команду:

python -m pip install django~=4.0.0

Оператор сравнения **~=** гарантирует автоматическую установку последующих обновлений безопасности для Django, таких как 4.0.1, 4.0.2 и так далее. Обратите внимание, что хотя возможно использовать более короткую версию **pip install <пакет>**, практикой считается использовать более длинную, но более явную форму **python -m pip install <пакет>**, чтобы обеспечить правильную версию Python. Это может быть проблемой, если на вашем компьютере установлено несколько версий Python.

После выполнения этих команд вы можете увидеть предупреждение (WARNING) о необходимости обновления pip. Всегда хорошо находиться на последней версии программного обеспечения и избавиться от раздражающего предупреждения при каждом использовании pip. Вы можете либо скопировать и вставить рекомендуемую команду, либо выполнить **python -m pip install --upgrade pip**, чтобы обновить pip до последней версии.

Проект Django может иметь практически любое имя, но в этой книге мы будем использовать имя **django\_project**. Для создания нового проекта Django используйте команду:

django-admin startproject django\_project .

то стоит пояснить, почему в конце предыдущей команды следует добавить точку (.) в конце. Если вы просто выполните команду **django-admin startproject django\_project**, то по умолчанию Django создаст следующую структуру каталогов:

Вы видите несколько каталогов с именем **django\_project**? Сначала создается каталог верхнего уровня с именем **django\_project**, а затем внутри него создается еще один каталог с тем же именем, который содержит файлы, необходимые для вашего проекта Django. Это кажется избыточным, поэтому предпочтительнее добавить точку в конце команды, чтобы установить Django в текущем каталоге. Это создаст структуру каталогов без лишнего уровня вложенности.

Командная строка - это место, где мы выполняем команды для наших программ, но текстовый редактор - это место, где фактически пишется код. Компьютеру не важно, какой текстовый редактор вы используете - конечный результат всегда является только кодом. Однако хороший текстовый редактор может предоставлять полезные подсказки и помогать обнаруживать опечатки.

Существует множество современных текстовых редакторов, и одним из очень популярных является Visual Studio Code, который бесплатен, легко устанавливается и пользуется широкой популярностью. Если вы еще не используете текстовый редактор, скачайте и установите VSCode с официального веб-сайта.

Дополнительным, но настоятельно рекомендуемым шагом, является использование большой экосистемы расширений, доступных в VSCode. На Windows перейдите в "File" -> "Preferences" -> "Extensions", а на macOS в "Code" -> "Preferences" -> "Extensions". Это откроет строку поиска в магазине расширений. Введите "python", что приведет к появлению расширения Microsoft Python как первого результата. Установите его.

Второе расширение, которое стоит установить, - это Black, который является форматировщиком кода Python и быстро стал стандартом в сообществе Python. Чтобы установить Black, откройте окно терминала в VSCode, перейдя в "Terminal" -> "New Terminal" в верхней части страницы. В новом окне терминала, открывшемся в нижней части страницы, введите команду:

python -m pip install black

Затем откройте настройки VSCode, перейдя в "File" -> "Preferences" -> "Settings" на Windows или "Code" -> "Preferences" -> "Settings" на macOS. В поиске введите "python formatting provider" и выберите "black" из выпадающих опций. Затем найдите "format on save" и включите "Editor: Format on Save". Теперь Black будет автоматически форматировать ваш код при сохранении файла с расширением \*.py.

Для проверки, что это работает, используйте свой текстовый редактор, чтобы создать новый файл с именем **hello.py** в каталоге "ch1-setup" на вашем Рабочем столе и введите следующий код, используя одинарные кавычки. Сохраните файл, и Black автоматически отформатирует его код.

Последний шаг - установить Git, систему контроля версий, которая является неотъемлемой частью современной разработки программного обеспечения. С помощью Git вы можете сотрудничать с другими разработчиками, отслеживать все свои изменения через коммиты и восстанавливать предыдущие версии вашего кода, даже если вы случайно удалили что-то важное!

На Windows перейдите на официальный веб-сайт по адресу <https://git-scm.com/> и нажмите на ссылку "Download", которая должна установить подходящую версию для вашего компьютера. Сохраните файл, а затем откройте папку "Downloads" и дважды щелкните по файлу. Это запустит установщик Git for Windows.

Пройдите почти все ранние значения по умолчанию, нажимая кнопку "Next", так как они подходят, и их всегда можно будет обновить позже, если потребуется. Есть два исключения: в разделе "Choosing the default editor used by Git" выберите VS Code, а не Vim. И в разделе "Adjusting the name of the initial branch in new repositories" выберите опцию "main" вместо "master" в качестве имени ветки по умолчанию. В остальном рекомендуется оставить значения по умолчанию, и их всегда можно будет настроить позже, если потребуется.

Для подтверждения установки Git на Windows закройте все текущие окна командной строки и откройте новое, что позволит загрузить изменения в переменной PATH. Введите команду **git --version**, которая должна показать, что Git установлен.

После установки Git на вашем компьютере нам нужно выполнить одноразовую системную настройку, указав имя и адрес электронной почты, связанные с вашими коммитами Git, а также установив имя ветки по умолчанию "main". В командной строке введите следующие две строки, убедившись, что вы обновили их с вашим именем и адресом электронной почты.

В этой главе мы создадим проект Django, который просто выводит "Hello, World" на главной странице. Это традиционный способ начать работу с новым языком программирования или фреймворком. Мы также впервые поработаем с Git и разместим наш код на GitHub.

Если вы застрянете на каком-либо этапе, полный исходный код для этой и всех будущих глав доступен онлайн на официальном репозитории GitHub для книги.

Для начала откройте новое окно командной строки или используйте встроенный терминал в VS Code. Если вы используете VS Code, нажмите "Terminal" в верхней части окна и затем "New Terminal", чтобы открыть его внизу экрана.

Убедитесь, что вы не находитесь в существующем виртуальном окружении, проверив, что перед вашим приглашением в командной строке нет никаких скобок. Вы даже можете ввести **deactivate**, чтобы быть абсолютно уверенными. Затем перейдите в каталог "code" на вашем Рабочем столе и создайте каталог с именем "helloworld" с помощью следующих команд:

Создайте новое виртуальное окружение с именем **.venv**, активируйте его и установите Django с помощью Pip, как мы делали в предыдущей главе.

Теперь мы будем использовать команду **startproject** Django для создания нового проекта с именем **django\_project**. Не забудьте добавить точку (.) в конце команды, чтобы проект был установлен в текущем каталоге.

django-admin startproject django\_project .

Давайте на мгновение остановимся и рассмотрим структуру проекта по умолчанию, которую Django предоставил нам. Вы можете визуально рассмотреть эту структуру, открыв новый каталог с помощью мыши на Рабочем столе. Возможно, каталог **.venv** изначально не виден, потому что он является "скрытым", но тем не менее он существует.

• Файл **\_\_init\_\_.py** указывает, что файлы в каталоге являются частью Python-пакета. Без этого файла мы не сможем импортировать файлы из другого каталога, что мы будем делать много раз в Django.

• **asgi.py** предоставляет возможность использования опционального интерфейса Asynchronous Server Gateway Interface (ASGI).

• **settings.py** управляет общими настройками нашего проекта Django.

• **urls.py** сообщает Django, какие страницы создавать в ответ на запросы браузера или URL.

• **wsgi.py** означает интерфейс веб-сервера (Web Server Gateway Interface), который помогает Django обслуживать наши будущие веб-страницы.

Файл **manage.py** не является частью проекта **django\_project**, но используется для выполнения различных команд Django, таких как запуск локального веб-сервера или создание нового приложения.

Давайте попробуем наш новый проект, используя встроенный легкий веб-сервер Django для локальной разработки. Команда, которую мы будем использовать, называется **runserver** и находится в файле **manage.py**.

Обратите внимание, что полный вывод командной строки будет содержать дополнительную информацию, включая предупреждение о 18 непримененных миграциях. Технически это предупреждение пока не имеет значения. Django жалуется на то, что мы еще не "мигрировали" нашу начальную базу данных. Поскольку в этой главе мы фактически не будем использовать базу данных, это предупреждение не повлияет на конечный результат.

Однако, поскольку предупреждения все равно немного раздражают, мы можем их убрать, сначала остановив локальный сервер с помощью команды **Control+c**, а затем выполнив **python manage.py migrate**.

Цикл HTTP-запроса/ответа HTTP (Протокол передачи гипертекста) был создан Тимом Бернерс-Ли в 1989 году и является основой Всемирной паутины. Протокол передачи данных - это набор правил для форматирования и обработки данных. Это похоже на общий язык для компьютеров, который позволяет им общаться друг с другом, даже если они находятся на разных концах земли и имеют сильно разное аппаратное и программное обеспечение.

HTTP является протоколом запроса-ответа, который работает в модели вычислительного клиента-сервера. Каждый раз, когда вы посещаете веб-страницу, начальный "запрос" отправляется "клиентом" (то есть вашим компьютером), и "ответ" отправляется "сервером". Однако клиент не обязательно должен быть компьютером, это также может быть мобильный телефон или любое устройство, подключенное к Интернету. Но процесс одинаков: клиент отправляет HTTP-запрос на URL, и сервер отправляет HTTP-ответ.

В конечном итоге, весь веб-фреймворк, такой как Django, просто принимает HTTP-запросы по указанному URL и возвращает HTTP-ответ, содержащий информацию, необходимую для отображения веб-страницы. Вот и всё. Обычно этот процесс включает в себя идентификацию правильного URL, подключение к базе данных, добавление логики, применение стилей с помощью HTML/CSS/JavaScript/статических ресурсов, а затем возвращение HTTP-ответа. Это и всё.

С течением времени шаблон Model-View-Controller (MVC) стал популярным способом внутреннего разделения данных, логики и отображения приложения на отдельные компоненты. Это упрощает задачу разработчика в понимании кода. Шаблон MVC широко используется во многих веб-фреймворках, включая Ruby on Rails, Spring (Java), Laravel (PHP), ASP.NET (C#) и многие другие.

В традиционной модели MVC существуют три основных компонента:

* Модель (Model): Управляет данными и основной бизнес-логикой.
* Вид (View): Отображает данные из модели в определенном формате.
* Контроллер (Controller): Принимает ввод пользователя и выполняет логику, специфичную для приложения.

Django следует традиционному подходу MVC нестрого и предлагает собственную версию, часто называемую Model-View-Template (MVT). Это может изначально сбивать с толку разработчиков с опытом использования других веб-фреймворков. На самом деле, подход Django включает в себя 4 компонента, который также включает в себя конфигурацию URL, поэтому более точным описанием было бы что-то вроде MVTU (Model-View-Template-URL).

Подход MVT в Django выглядит следующим образом:

* Модель (Model): Управляет данными и основной бизнес-логикой.
* Вид (View): Описывает, какие данные отправляются пользователю, но не их представление.
* Шаблон (Template): Представляет данные в виде HTML с возможными CSS, JavaScript и статическими ресурсами.
* Конфигурация URL: Компоненты с регулярными выражениями, настроенные на Вид (View).

Это взаимодействие фундаментально для Django, но может быть очень запутанным для новичков, поэтому давайте определим порядок цикла HTTP-запроса/ответа. Когда вы вводите URL, такой как [https://djangoproject.com](https://djangoproject.com/), первое, что происходит внутри нашего проекта Django, - это находится URL-шаблон (содержащийся в urls.py), который соответствует ему. URL-шаблон связан с одним видом (содержащимся в views.py), который объединяет данные из модели (хранящейся в models.py) и стили из шаблона (любой файл с расширением .html). Затем вид возвращает HTTP-ответ пользователю.

Полный процесс в Django выглядит следующим образом:

Если вы совершенно новичок в веб-разработке, то различие между MVC и MVT не будет иметь большого значения. В этой книге демонстрируется способ работы Django, поэтому не должно возникать путаницы.

Однако, если вы веб-разработчик с опытом работы с MVC, может потребоваться немного времени, чтобы перестроить свое мышление в соответствии с "джанговским способом" вещей, который более свободно связан и позволяет более легкие модификации.

Django использует концепцию проектов и приложений, чтобы сохранить код чистым и читаемым. Один верхнеуровневый проект Django может содержать несколько приложений. Каждое приложение управляет изолированным функционалом. Например, сайт электронной коммерции может иметь одно приложение для аутентификации пользователей, другое приложение для платежей и третье приложение для отображения деталей о товарах. Это три отдельных приложения, которые все находятся внутри одного верхнеуровневого проекта. Как и когда разделять функциональность на приложения, в некоторой степени субъективно, но, в целом, каждое приложение должно иметь четкую функцию.

Чтобы добавить новое приложение, перейдите к командной строке и остановите работающий сервер с помощью **Control+c**. Затем используйте команду **startapp**, за которой следует имя нашего приложения, которое будет **pages**.

Давайте рассмотрим, что делает каждый новый файл приложения **pages**:

* **admin.py** - это конфигурационный файл для встроенного приложения Django Admin.
* **apps.py** - это конфигурационный файл для самого приложения.
* **migrations/** - отслеживает любые изменения в файле **models.py**, чтобы он оставался синхронизированным с нашей базой данных.
* **models.py** - здесь мы определяем модели базы данных, которые Django автоматически преобразует в таблицы базы данных.
* **tests.py** - предназначен для тестирования, специфичного для приложения.
* **views.py** - здесь мы обрабатываем логику запроса/ответа для нашего веб-приложения.

Заметьте, что модель, вид (view) и URL из шаблона MVT уже присутствуют с самого начала. Единственное, чего у нас пока нет, - это шаблон, который мы добавим в ближайшее время.

Несмотря на то, что наше новое приложение существует внутри проекта Django, Django не "знает" о нем, пока мы явно не добавим его в файл **django\_project/settings.py**. Откройте этот файл в вашем текстовом редакторе и прокрутитесь вниз к разделу **INSTALLED\_APPS**, где вы увидите уже шесть встроенных приложений Django. Добавьте **pages.apps.PagesConfig** внизу списка.

Если у вас установлен Black в вашем текстовом редакторе и вы сохраняете файл, все одинарные кавычки '' здесь будут заменены двойными кавычками "". Это нормально. Как уже отмечалось ранее, в будущем Django планирует полностью внедрить форматирование Python с использованием Black.

Что такое **PagesConfig**, вы можете спросить? Это имя единственной функции в файле **pages/apps.py** на данном этапе.

Не беспокойтесь, если вы сейчас чувствуете себя запутанными: для усвоения структуры проектов и приложений Django требуется практика. На протяжении этой книги мы будем создавать множество проектов и приложений, и со временем эти шаблоны станут вам знакомыми.

В Django для создания одной динамической (связанной с базой данных) веб-страницы требуется четыре отдельных файла, соответствующих этому шаблону MVT:

* **models.py**
* **views.py**
* **template.html** (подходит любой файл HTML)
* **urls.py**

Однако, чтобы создать статическую веб-страницу (не связанную с базой данных), мы можем жестко закодировать данные в виде, поэтому модель не требуется. Именно это мы и сделаем здесь, чтобы упростить задачу. Начиная с главы 4, мы будем использовать модель во всех наших проектах.

Следующим шагом будет создание нашего первого вида (view). Начнем с обновления файла **views.py** в нашем приложении **pages** следующим образом:

По сути, мы говорим, что когда вызывается функция представления **homePageView**, нужно вернуть текст "Hello, World!" Более точно, мы импортировали встроенный метод **HttpResponse**, чтобы вернуть объект ответа пользователю. Мы создали функцию с именем **homePageView**, которая принимает объект запроса и возвращает ответ с строкой "Hello, World!".

В Django существует два типа представлений: представления на основе функций (FBV) и представления на основе классов (CBV). Наш код в этом примере является представлением на основе функций: он относительно прост в реализации и явно выражен. Django изначально начал с FBV, но с течением времени добавил CBV, что позволяет достичь гораздо большей переиспользуемости кода, соблюдать принцип DRY (Don't Repeat Yourself) и расширять его с помощью mixin. Дополнительная абстракция в CBV делает их мощными и лаконичными, но также делает их менее читабельными для новичков в Django.

Поскольку разработка веб-приложений быстро становится повторяющейся, Django также предоставляет ряд встроенных обобщенных классовых представлений (GCBV) для обработки общих случаев, таких как создание новых форм, представления списков, постраничный вывод и так далее. В последующих главах этой книги мы будем активно использовать GCBV.

Таким образом, технически в Django существует три способа написания представлений: представления на основе функций (FBV), представления на основе классов (CBV) и обобщенные классовые представления (GCBV). Эта настраиваемость полезна для опытных разработчиков, но может быть запутывающей для новичков. Многие разработчики Django, включая вашего автора, предпочитают использовать GCBV, когда это возможно, и возвращаться к CBV или FBV при необходимости. К концу этой книги вы попробуете все три способа и сможете решить, какой подход вам более нравится.

Продолжая, нам нужно настроить наши URL-адреса. В вашем текстовом редакторе создайте новый файл с именем **urls.py** в приложении **pages** и затем обновите его с помощью следующего кода:

На первой строке мы импортируем **path** из Django для настройки нашего URL-шаблона, а на следующей строке мы импортируем наши представления. Обратившись к файлу **views.py** как **.views**, мы говорим Django искать файл **views.py** в текущем каталоге и импортировать представление **homePageView** оттуда.

Наш URL-шаблон состоит из трех частей:

* регулярное выражение Python для пустой строки ""
* ссылка на представление с именем **homePageView**
* необязательный именованный URL-шаблон "home"

Другими словами, если пользователь запрашивает домашнюю страницу, представленную пустой строкой "", Django должен использовать представление с именем **homePageView**.

Мы почти завершили этот этап. Последний шаг - обновить файл **django\_project/urls.py**. Обычно в одном проекте Django могут быть несколько приложений, таких как **pages** в данном случае, и каждое из них нуждается в собственном пути URL.

Мы импортировали **include** на второй строке рядом с **path**, а затем создали новый URL-шаблон для нашего приложения **pages**. Теперь, когда пользователь посещает домашнюю страницу, он сначала направляется в приложение **pages**, а затем к представлению **homePageView**, установленному в файле **pages/urls.py**.

Эта необходимость в двух отдельных файлах **urls.py** часто смущает начинающих. Представьте верхнеуровневый файл **django\_project/urls.py** как ворота к различным URL-шаблонам, присущим каждому приложению.

У нас есть весь необходимый код. Для подтверждения того, что все работает ожидаемым образом, перезапустите наш сервер Django.

В предыдущей главе мы установили систему контроля версий Git. Давайте использовать её здесь. Первый шаг - инициализировать (или добавить) Git в наш репозиторий. Убедитесь, что вы остановили локальный сервер с помощью **Control+c**, затем выполните команду **git init**.

Если вы затем введете **git status**, вы увидите список изменений с момента последнего коммита Git. Поскольку это наш первый коммит, этот список содержит все наши изменения до сих пор.

Обратите внимание, что наш виртуальный окружение .venv включено в список изменений, что не является хорошей практикой. Его следует исключить из контроля версий Git, так как в нем часто содержатся секретные данные, такие как ключи API и подобное. Решение - создать новый файл с именем **.gitignore**, который сообщает Git, что игнорировать. Точка в начале файла указывает, что это "скрытый" файл. Файл все равно существует, но это способ сообщить разработчикам, что его содержимое, вероятно, предназначено для настройки, а не для контроля версий. В этом новом файле добавьте одну строку для **.venv**.

Если вы снова выполните команду **git status**, вы увидите, что **.venv** больше не отображается. Git перестал отслеживать этот файл.

В то же время, мы хотим иметь запись о установленных пакетах в нашем виртуальном окружении. Наилучшей практикой в настоящее время является создание файла **requirements.txt** с этой информацией. Команда **pip freeze** выведет содержимое вашего текущего виртуального окружения, и с помощью оператора **>** можно выполнить все это в один шаг: вывести содержимое в новый файл с именем **requirements.txt**. Если ваш сервер все еще работает, введите **Ctrl+c** и **Enter**, чтобы выйти, прежде чем вводить эту команду.

Новый файл **requirements.txt** будет содержать все наши установленные пакеты и их зависимости. Если вы посмотрите внутрь этого файла, вы увидите, что там фактически четыре пакета, хотя мы установили только один.

Это потому, что Django зависит от других пакетов для поддержки. Часто бывает так, что, устанавливая один пакет Python, вы также устанавливаете несколько других пакетов, от которых он зависит. Поскольку сложно отслеживать все устанавливаемые пакеты, файл **requirements.txt** является очень важным.

Затем мы хотим добавить все недавние изменения с помощью команды **add -A**, а затем зафиксировать изменения вместе с сообщением (-m), описывающим, что изменилось.

В профессиональных проектах файл **.gitignore** обычно довольно обширен. По эффективности и безопасности часто требуется исключить из системы управления исходным кодом множество каталогов и файлов. Мы рассмотрим это более подробно в главе 16, когда будем проводить развертывание профессионального уровня.

Хорошей практикой является создание удаленного репозитория для каждого проекта. Таким образом, у вас есть резервная копия в случае каких-либо проблем с вашим компьютером, и, что более важно, это позволяет сотрудничать с другими разработчиками. Популярными вариантами для хранения кода являются GitHub, Bitbucket и GitLab. При изучении веб-разработки лучше придерживаться закрытых репозиториев, чтобы случайно не размещать критическую информацию, такую как пароли, в открытом доступе в Интернете.

В этой книге мы будем использовать GitHub, но все три сервиса предоставляют схожую функциональность для новичков. Зарегистрируйтесь на бесплатной учетной записи на домашней странице GitHub и подтвердите свой адрес электронной почты. Затем перейдите на страницу "Создать новый репозиторий", которая находится по адресу <https://github.com/new>.

Введите имя репозитория "hello-world" и установите переключатель рядом с "Частный" вместо "Общедоступный". Затем нажмите на кнопку внизу "Создать репозиторий".

Ваш первый репозиторий теперь создан! Однако в нем еще нет кода. Прокрутите страницу вниз к разделу "...или отправьте существующий репозиторий из командной строки." Именно это нам и нужно. Скопируйте текст, находящийся непосредственно под этим заголовком, и вставьте его в командную строку. Вот, как это выглядит для меня с моим именем пользователя GitHub - wsvincent. Ваше имя пользователя будет другим.

К сожалению, существует хорошая вероятность того, что последняя команда вызвала ошибку, если вы новый разработчик и у вас еще не настроены SSH-ключи.

Это загадочное сообщение означает, что нам нужно настроить SSH-ключи. Это действие выполняется один раз, но, честно говоря, оно может быть немного неудобным.

SSH - это протокол, используемый для обеспечения частных соединений с удаленным сервером. Представьте его как дополнительный уровень конфиденциальности поверх имени пользователя и пароля. Процесс включает в себя создание уникальных SSH-ключей и их сохранение на вашем компьютере, чтобы только GitHub имел доступ к ним.

Сначала проверьте, есть ли у вас уже существующие SSH-ключи. GitHub предоставляет руководство по этому25, которое подходит для Mac, Windows и Linux. Если у вас нет существующих открытых и закрытых ключей, вам потребуется их создать. GitHub также предоставляет руководство по этому.

После завершения настроек вы должны сможете успешно выполнить команду **git push -u origin main**!

Если вы чувствуете себя перегруженными и разочарованными, если у вас возникли проблемы с SSH-ключами, не волнуйтесь. У GitHub есть множество ресурсов, которые помогут вам разобраться, но, честно говоря, это может показаться интимным в первый раз. Если вы по-настоящему застряли, продолжайте работу с книгой и вернитесь к разделам о SSH-ключах и GitHub, отдохнув полной ночью. Не могу пересчитать, сколько раз ясный разум помог мне разрешить трудную проблему программирования.

Предполагая успех с GitHub, выходите из виртуальной среды с помощью команды **deactivate**.

Теперь на вашей командной строке больше не должно быть скобок, что указывает на то, что виртуальная среда больше не активна.

В этой главе мы создадим, протестируем и развернем приложение Pages, содержащее домашнюю страницу и страницу "О нас". Мы все еще не будем касаться базы данных, это произойдет в следующей главе, но мы узнаем о представлениях на основе классов и шаблонах, которые являются строительными блоками для более сложных веб-приложений, создаваемых позже в книге.

Как и во второй главе: Привет, мир!, наше начальное настроение включает следующие шаги: • создайте новый каталог для нашего кода с именем "pages" и перейдите в него. • создайте новое виртуальное окружение с именем ".venv" и активируйте его. • установите Django. • создайте новый проект Django с именем "django\_project". • создайте новое приложение с именем "pages". На командной строке убедитесь, что вы не работаете в существующем виртуальном окружении. Если перед приглашением командной строки есть текст - ">" в Windows или "%" на macOS, значит, вы находитесь в виртуальном окружении. Убедитесь, что вы введите "деактивировать", чтобы выйти из него. В новой оболочке командной строки начните с ввода следующего:

Помните, что даже если мы добавили новое приложение, Django не будет его распознавать, пока оно не будет добавлено в параметр INSTALLED\_APPS в django\_project/settings.py. Откройте ваш текстовый редактор и добавьте его внизу файла.

Каждый веб-фреймворк нуждается в удобном способе создания HTML-файлов, и в случае Django используются шаблоны: отдельные HTML-файлы, которые можно связывать между собой и включать в них базовую логику.

Напомним, что в предыдущей главе наш сайт "Hello, World" имел фразу, закодированную в файле views.py. С технической точки зрения, это работает, но это не масштабируется хорошо! Более правильный способ - связать представление с шаблоном, разделяя информацию, содержащуюся в каждом из них.

В этой главе мы узнаем, как использовать шаблоны для более легкого создания желаемой домашней страницы и страницы "О нас". А в будущих главах использование шаблонов будет поддерживать создание веб-сайтов, которые могут обрабатывать сотни, тысячи и даже миллионы веб-страниц с минимальным количеством кода.

Первым вопросом является то, где разместить шаблоны в структуре проекта Django. Есть два варианта. По умолчанию загрузчик шаблонов Django будет искать соответствующие шаблоны в каждом приложении. Однако структура этих шаблонов внутри приложения может быть verwegunsu.

По умолчанию загрузчик шаблонов Django будет искать соответствующие шаблоны внутри каждого приложения. Однако структура этих шаблонов внутри приложения может быть довольно запутанной: каждое приложение требует новый каталог "templates", еще один каталог с тем же именем, что и у приложения, и затем файл шаблона.

Следовательно, в нашем приложении pages, Django ожидает следующую структуру:

Это означает, что нам нужно создать новый каталог "templates", новый каталог с именем нашего приложения, "pages", и, наконец, сам шаблон "home.html". Почему такой повторяющийся подход? Краткий ответ заключается в том, что загрузчик шаблонов Django хочет быть уверенным, что он нашел правильный шаблон! Что произойдет, если файлы "home.html" находятся в двух разных приложениях? Эта структура гарантирует, что таких конфликтов не будет.

Тем не менее, есть еще один подход: создать единый каталог "templates" на уровне проекта и поместить все шаблоны в него. Сделав небольшую настройку в файле django\_project/settings.py, мы можем указать Django искать шаблоны также в этом каталоге. Именно этот подход мы и используем.

Для начала завершите работу сервера с помощью команды Control+c. Затем создайте каталог с именем "templates".

Затем нам нужно обновить файл **django\_project/settings.py**, чтобы сообщить Django о местоположении нашего нового каталога с шаблонами. Это однострочное изменение для параметра "DIRS" внутри "TEMPLATES".

Внутри каталога templates создайте новый файл с именем home.html. Вы можете сделать это с помощью вашего текстового редактора: в Visual Studio Code перейдите в верхний левый угол вашего экрана, нажмите "Файл", затем "Новый файл". Убедитесь, что вы даете файлу правильное имя и сохраняете его в правильном месте.

Файл home.html будет содержать простой заголовок на данный момент.

Хорошо, наш шаблон готов! Следующим шагом будет настройка файлов URL и view.

Ранние версии Django поставлялись только с функциональными представлениями, но разработчики вскоре обнаружили, что они снова и снова повторяют те же шаблоны. Создать представление, которое перечисляет все объекты в модели. Создать представление, которое отображает только один детализированный элемент из модели. И так далее.

Обобщенные функциональные представления на основе функций были введены, чтобы абстрагировать эти шаблоны и упростить разработку общих случаев. Однако не было простого способа расширить или настроить эти представления. В результате Django ввел классовые обобщенные представления, которые упрощают использование и расширение представлений, охватывающих общие случаи.

Классы являются фундаментальной частью Python, но подробное обсуждение их находится вне рамок этой книги. Если вам нужно введение или обновление, я предлагаю ознакомиться с официальной документацией Python, где есть отличный учебник о классах и их использовании.

В нашем представлении мы будем использовать встроенный **TemplateView** для отображения нашего шаблона. Обновите файл **pages/views.py**.

Обратите внимание, что мы написали имя нашего представления, **HomePageView**, с заглавной буквы, так как это теперь класс Python. В отличие от функций, классы всегда должны начинаться с заглавной буквы. **TemplateView** уже содержит всю логику, необходимую для отображения нашего шаблона, нам просто нужно указать имя шаблона.

Последний шаг - обновить наши URL. Помните из главы 2, что нам нужно внести изменения в двух местах. Сначала мы обновляем файл django\_project/urls.py, чтобы указать наше приложение pages, а затем внутри pages мы сопоставляем представления с маршрутами URL.

Начнем с файла django\_project/urls.py.

Здесь код должен вам уже быть знаком. На второй строке мы добавляем **include**, чтобы указать существующий URL для приложения pages. Затем создайте файл **pages/urls.py** и добавьте следующий код. Этот шаблон почти идентичен тому, что мы делали в главе 2, с одним основным отличием: при использовании Class-Based Views вы всегда добавляете **as\_view()** в конце имени представления.

Процесс добавления страницы "О нас" очень похож на то, что мы только что сделали. Мы создадим новый файл шаблона, новое представление и новый маршрут URL. Начнем с создания нового файла шаблона с именем **about.html** в директории **templates** и добавим в него короткий HTML-заголовок.

И, наконец, импортируйте имя представления и свяжите его с URL по пути **about/**.

Мощь шаблонов заключается в их способности быть расширяемыми. Если подумать о большинстве веб-сайтов, то есть контент, который повторяется на каждой странице (заголовок, нижний колонтитул и так далее). Не было бы здорово, если бы у нас, как у разработчиков, было одно каноническое место для нашего заголовка, которое наследовалось бы всеми другими шаблонами?

Ну, у нас есть такая возможность! Давайте создадим файл base.html, содержащий заголовок с ссылками на наши две страницы. Мы могли бы назвать этот файл как угодно, но использование base.html - это общепринятая конвенция. В вашем текстовом редакторе создайте этот новый файл с именем templates/base.html.

У Django есть мини-язык шаблонов для добавления ссылок и базовой логики в наши шаблоны. Вы можете увидеть полный список встроенных тегов шаблона здесь в официальной документации. Теги шаблона имеют форму {% что-то %}, где "что-то" - это сам тег шаблона. Вы даже можете создавать свои собственные теги шаблона, хотя мы не будем этого делать в этой книге.

Чтобы добавить ссылки на URL в нашем проекте, мы можем использовать встроенный тег шаблона url, который принимает имя URL-шаблона в качестве аргумента. Помните, как мы добавили имена URL в наши две маршруты в pages/urls.py? Вот для чего это было. Тег url использует эти имена, чтобы автоматически создавать ссылки для нас. URL-маршрут для нашей домашней страницы называется home. Чтобы настроить ссылку на нее, мы используем следующий синтаксис: {% url 'home' %}.

Внизу мы добавили блочный тег с именем content. Блоки могут быть перезаписаны дочерними шаблонами через наследование. Хотя закрывающий тег endblock можно назвать по желанию - вы можете просто написать {% endblock %}, если предпочитаете - это помогает сделать код более читаемым, особенно в больших шаблонах. Давайте обновим наши файлы home.html и about.html, чтобы расширить шаблон base.html. Это означает, что мы можем использовать тот же код из одного шаблона в другом шаблоне. В языке шаблонов Django есть метод extends, который мы можем использовать для этого.

Теперь, если вы запустите сервер с помощью **python manage.py runserver** и откроете наши веб-страницы снова по адресу <http://127.0.0.1:8000/>, заголовок будет включен.