

Geekbrains

Backend - разработка web - приложения на языке программирования Python с использованием фреймворка Django

Программа: «Python - разработчик»

Специализация: «Backend - разработчик»

Касимов И. Т.

Магнитогорск

2024

Дипломный проект «Backend - разработка web - приложения на языке программирования Python с использованием фреймворка Django».

Содержание

Введение	5
Глава 1. Программирование и его направления.....	8
1.1 Что такое программирование	8
1.2 Направление языков программирования	10
Глава 2. Виды разработки в программировании.....	11
2.1 Основные виды.	11
2.2 Backend – разработка.....	14
2.1.1 Что такое backend – разработка?	14
2.1.2 Backend - разработка на языке программирования Python.....	15
Глава 3. Основные понятия web – технологий.....	16
3.1 Основы HTML.	16
3.1.1 Что такое HTML?	16
3.1.2 Структура элементов разметки HTML	17
3.2 Основы CSS.....	19
3.2.1 Что такое CSS?	19
3.2.2 Синтаксис CSS	20
3.3 Основы JavaScript	21
Глава 4. Git. Знакомство с контролем версий	22
4.1 Основы систем контроля версий.....	22
4.2 Преимущества систем контроля версий.....	23
Глава 5. Основы языка Python	25
5.1 Знакомство с языком программирования Python	25
5.2 Базовые понятие ООП Python	27
Глава 6. Фреймворк Django.....	30
6.1 Что такое Django?	30
6.2 Установка и настройка Django	32
6.2.1 Создание проекта и запуск проекта	33
6.2.2 MVT концепция. Создание первого приложения.....	35
6.2.3 Маршрутизация и функции представления	38
6.2.4 Динамические URL	40

6.2.5 Обработчики исключений при запросе	41
6.2.6 Перенаправления redirect. Функция reverse	42
6.3 Шаблоны в Django	43
6.3.1 Введение в шаблоны	43
6.3.2 Передача данных (контекста) в шаблон	45
6.3.3 Теги шаблонов. Теги if, for и url.....	46
6.3.4 Наследование шаблонов	47
6.3.5 Подключение статистических файлов.....	49
6.3.6 Пользовательские теги шаблонов	50
6.4 Введение в ORM и модели	52
6.4.1 Что такое базы данных, SQL и ORM.....	52
6.4.2 Понятие CRUD.....	55
6.4.3 Слагги (slug) в URL-адресах. Метод get_absolute_url().....	57
6.5 Связи между таблицами.....	59
6.5.1 Типы связей между моделями.....	59
6.5.1.1 ForeignKey – один ко многим	60
6.5.1.2 ManyToMany - многие ко многим.....	62
6.5.1.3 OneToOne – один к одному	64
6.5.2 Создание связи many-to-one	66
6.5.3 Создаем связь many-to-many	68
6.5.4 Агрегирование и группировка.....	70
6.6 Оптимизация с Django Debug Toolbar	71
6.7 Работа с админ-панелью	74
6.7.1 Создание суперпользователя. Регистрация моделей.	75
6.7.2 Настройка отображения списка из БД в админ-панели. Атрибуты.....	77
6.7.3 Поиск и фильтрация объектов.....	78
6.7.4 Настройка формы редактирования записей.....	79
6.8 Работа с формами	81
6.8.1 Что такое HTML-формы. Отправка данных через GET и POST	81
6.8.2 Класс Form. Атрибуты, классы полей формы.	85
6.8.3 Сохранение переданных данных в БД.....	87
6.8.4 Загрузка файлов с использованием классов моделей	88
6.8.5 Отображение изображений.....	89
6.9 Классы представлений	91
6.9.1 Класс ListView	92

6.9.2 Класс FormView	94
6.9.3 Класс CreateView	95
6.9.4 Собственный класс DataMixin.....	96
6.10 Авторизация и регистрация.....	98
6.10.1 Авторизация пользователей	100
6.10.2 Классы LoginView и AuthenticationForm.....	102
6.10.3 Декоратор login_required() и класс LoginRequiredMixin.....	104
6.10.4 Функционал реализации отзывов	105
6.10.5 Регистрация пользователей с классом UserCreationForm.....	107
Заключение	109
Список используемой литературы.....	110

Введение

Актуальность.

В современном мире веб-разработка играет ключевую роль в создании эффективных и функциональных онлайн-решений для широкого круга пользователей. Для достижения этой цели разработчики часто обращаются к использованию различных фреймворков, среди которых популярностью особенно выделяется Django.

Данное учебно-исследовательское задание нацелено на создание веб-приложения с использованием фреймворка Django – мощного инструмента для быстрой и эффективной разработки веб-приложений на языке программирования Python. Django обеспечивает удобную инфраструктуру для работы с базами данных, обработки запросов и создания динамического контента.

Изучение объектно-ориентированного программирования и его применение в проекте на Django позволит глубже понять основные принципы разработки программного обеспечения, а также на практике оценить эффективность использования фреймворка Django для создания веб-приложений.

В дальнейшем работе будут рассмотрены основные шаги по созданию веб-приложения, описаны ключевые компоненты проекта, а также представлены результаты исследований, сделанных в ходе выполнения данного дипломного проекта.

Цель проекта.

Целью данного дипломного проекта является исследование базовых принципов объектно-ориентированного программирования, а также их практическое применение при создании веб-приложения с использованием фреймворка Django. Проект будет включать в себя реализацию базовых функциональностей, таких как аутентификация пользователей, управление контентом и взаимодействие с базой данных.

Задачами дипломной работы являются:

1. Изучение ООП и его применение в Python:
 - Изучение основных принципов объектно-ориентированного программирования.
 - Применение принципов ООП при разработке веб-приложения на Django.
2. Исследование фреймворка Django:
 - Изучение основных возможностей и особенностей фреймворка Django.
 - Ознакомление с инструментами Django для работы с базами данных, маршрутизацией запросов и шаблонами.

3. Разработка архитектуры веб-приложения:

- Определение структуры проекта, включая модели данных, представления (views) и шаблоны (templates).
- Разработка схемы базы данных, определение связей между моделями.

4. Реализация основных функциональностей:

- Аутентификация и авторизация пользователей.
- Создание CRUD (Create, Read, Update, Delete) операций для управления контентом.
- Работа с формами и обработка пользовательских данных.

5. Оптимизация производительности и безопасности:

- Внедрение механизмов кэширования для оптимизации работы приложения.
- Обеспечение безопасности приложения, включая обработку пользовательского ввода и защиту от уязвимостей.

Каждый из этих пунктов важен для успешной реализации дипломного проекта по созданию сайта на Django и способствует получению глубоких знаний и навыков в веб-разработке на основе данного фреймворка.

Научная и практическая значимость.

Научная значимость:

1. Исследовательский аспект: В процессе разработки дипломного проекта и изучения фреймворка Django студент получает возможность глубоко погрузиться в тему веб-разработки, применить знания о объектно-ориентированном программировании и изучить современные подходы к созданию веб-приложений.
2. Прикладной аспект: Создание функционального веб-приложения позволяет применить теоретические знания на практике, разработать полноценный проект с учетом требований пользователя и возможностей фреймворка.

Практическая значимость:

1. Получение опыта: Разработка дипломного проекта на Django позволяет студенту приобрести практический опыт работы с веб-технологиями, базами данных, архитектурой приложения и другими аспектами разработки.
2. Повышение конкурентоспособности: Освоение фреймворка Django и разработка качественного веб-приложения способствуют расширению круга профессиональных навыков студента, что делает его более конкурентоспособным на рынке труда.
3. Применение результатов: Законченный дипломный проект может быть использован в будущем для демонстрации навыков и компетенций при поиске работы или участии в профессиональных проектах.

Инструменты и технологии

В дипломном проекте будут использованы следующие инструменты и технологии: Python, Django, HTML, CSS, Git, PyCharm, SQLite, SQLiteStudio.

Рекомендации по изучению

Дипломный проект представляет из себя курс по освоению (или расширению уже имеющихся) знаний в области web – разработке с помощью фреймворка Django на языке программирования Python.

Глава 1. Программирование и его направления

1.1 Что такое программирование

Программирование - процесс создания и модификации компьютерных программ. Мы пользуемся запрограммированной техникой каждый день. Обычно программирование выглядит как написание программного кода на каком-нибудь языке программирования. С его помощью разработчики создают сайты, приложения, умные устройства, роботов и разнообразные цифровые сервисы.

В более широком смысле программирование - любое создание инструкций для исполнителя. Инструкции пишут в виде кода на различных языках для исполнителя. Исполнитель - устройства выполняющие эти инструкции. Когда человек программирует, он, по сути, переводит задачи для машины на язык, который ей понятен. Инструкции, написанные на этом языке, могут запускаться и выполняться - компьютер им следует. Так получаются программы. Люди общаются друг с другом на естественных языках, разработчики же с компьютерами взаимодействуют на языках программирования. Написанный программистом код — это алгоритм действий, который должно выполнить устройство. Таким образом, язык программирования помогает человеку записать понятную компьютеру последовательность операций.

Все виды языков программирования, суть которых сводится к преобразованию понятных человеку команд в машинный код, обладают строгим синтаксисом. К примеру, конец строки должен заканчиваться точкой с запятой. Такое правило позволяет компьютеру отделять команды друг от друга.

В современном мире программирование стало неотъемлемой частью IT - индустрии, и его роль постоянно растет. Основной целью программирования является написание программ, которые выполняют определенные задачи или решают определенные проблемы. Программы могут быть разработаны для различных платформ и устройств, начиная от компьютеров и мобильных устройств, и заканчивая встроенными системами и сетевыми устройствами.

Программирование включает в себя не только написание кода, но и планирование, анализ, тестирование и отладку программного обеспечения. В процессе программирования программисты используют различные инструменты и технологии для создания эффективных и надежных программ.

В современном мире программирование играет ключевую роль в различных отраслях, таких как разработка программного обеспечения, веб-разработка, машинное обучение,

искусственный интеллект, кибербезопасность и другие. Понимание основ программирования является важным навыком для специалистов IT - индустрии и позволяет создавать инновационные решения и продукты.

Таким образом, программирование представляет собой творческий процесс, направленный на создание программного обеспечения для удовлетворения потребностей пользователей и решения различных задач. В рамках дипломной работы по программированию можно изучить различные методы разработки программ, алгоритмы, структуры данных, технологии программирования и другие аспекты, связанные с созданием качественного программного обеспечения.

1.2 Направление языков программирования

Направления языков программирования охватывают широкий спектр технологий и методов, используемых для разработки программного обеспечения. Различные языки программирования имеют свои уникальные особенности, преимущества и области применения, что делает их подходящими для различных задач и проектов. В рамках данного исследования рассмотрим основные направления языков программирования и их роль в современной IT - индустрии.

Одним из основных направлений языков программирования являются процедурные языки. Они используются для описания последовательности действий, выполняемых компьютером для достижения определенной цели. Примерами процедурных языков являются C, Pascal, и BASIC. Эти языки часто используются для написания системного программного обеспечения, операционных систем, и встроенных приложений.

Другим важным направлением языков программирования являются объектно-ориентированные языки. В данном подходе программа строится из объектов, которые содержат данные и методы для их обработки. Языки, такие как Java, C++, и Python, широко используются для разработки сложных приложений, веб-сервисов, и игр.

Функциональное программирование также представляет важное направление языков программирования. Основным принципом этого подхода является работа с функциями как основными строительными блоками программы. Языки, такие как Haskell, Lisp, и Scala, позволяют писать более выразительный и функциональный код, что способствует улучшению производительности и читаемости программ.

Еще одним важным направлением языков программирования являются скриптовые языки. Они обеспечивают быструю разработку и исполнение программы без предварительной компиляции. Примерами скриптовых языков являются JavaScript, PHP, и Ruby. Эти языки широко используются для создания веб - страниц, обработки данных, и автоматизации задач.

Таким образом, направления языков программирования представляют собой разнообразные подходы к разработке программного обеспечения, каждое из которых имеет свои особенности и преимущества. Изучение и понимание различных языков программирования позволяет разработчикам выбирать наиболее подходящий язык для конкретного проекта и достигать оптимальных результатов. В дальнейшем исследовании будут рассмотрены более детально особенности каждого из направлений языков программирования и их влияние на развитие современной IT - индустрии.

Глава 2. Виды разработки в программировании.

2.1 Основные виды.

1. Backend - разработчик. Backend Developer - наиболее популярный тип разработчиков. Они создают серверно - административную часть продукта, то есть пишут код для внутренней стороны сайта, которую не видит пользователь. Такой программист разрабатывает фундамент проекта и настраивает его работу.

За что отвечает: база данных, архитектура, логика продукта, системы обработки и хранения данных.

Языки и технологии: Python, C++, Java, PHP, Ruby, Go. Хороший бэкенд-разработчик также должен разбираться в базе данных MySQL, PostgreSQL или NoSQL.

2. Frontend - разработчик. Входят в тройку самых востребованных программистов. Он отвечает за пользовательскую сторону приложения и работают на стороне клиента. Цель frontend - разработчика - сделать удобный и интуитивно понятный интерфейс для пользователя. Например, интерактивные переходы страниц, всплывающее меню, меняющие цвет кнопки — всем этим занимается frontend - developer. Также в его задачи входит правильное отображение сайта или приложения на компьютере и на разных электронных гаджетах. Ну а если в проекте отсутствует верстальщик, берет его задачи на себя - работает с HTML-кодом.

За что отвечает: пользовательский интерфейс, кросс-браузерные ошибки, верстка шаблона сайта, адаптивная и мобильная верстка.

Языки и технологии: JavaScript, CSS, интерфейсные среды (React, jQuery или Angular), HTML, препроцессоры SASS/LESS. Дополнительные знания: серверные технологии, основы SEO-оптимизации, веб-шрифты, графические редакторы.

3. Fullstack - разработчик. Это универсальный программист полного цикла, который сочетает в себе знания frontend - и backend -разработчика. Fullstack - разработчик полностью разрабатывает web - проект или сайт: от программно-административной части до клиентского интерфейса. Такие разработчики нужны для создания полного проекта и востребованы на рынке.

За что отвечает: пользовательская и серверная часть сайта, кроссплатформенные приложения, интеграция сервисов на frontend - и backend.

Языки и технологии: JavaScript для браузерной части, PHP, Java или Python для серверной части, HTML, CSS, базы данных, фреймворки.

4. Мобильный разработчик. К ним относятся программисты, которые делают приложения для мобильных устройств ОС Android и iOS. Помимо телефонов, к таким устройствам относятся: GPS - навигаторы, умные часы, электронные книги.

Как правило, Android - программисты разбираются в обеих частях приложения, отвечая за полный цикл разработки, поэтому строгого деления на frontend - и backend у них нет.

За что отвечает: полный цикл разработки приложений для мобильных устройств.

Языки для Android - разработчика: Java, Kotlin, OpenGL, Android SDK.

Языки для iOS-разработчика: Objective-C, Swift, CoreData, CoreGraphics.

5. Разработчик видеоигр. Программисты, которые разрабатывают видеоигры, а именно:

- desktop - приложения (программы, которые работают под управлением ОС и не зависят от других приложений);
- мобильные игры;
- ролевые онлайн-игры MMORG.

Они работают над ПО, которое предназначено для ОС Windows, Apple OSX или Linux. Некоторые Game-разработчики создают бизнес-приложения для конкретной компании.

За что отвечает: игровая разработка, веб-дизайн, обновления.

Языки и технологии: C/C++, C#, Java, Open GL/DirectX, игровые движки (Unity, Unreal Engine, Torque), графические библиотеки.

6. Инженер - программист. Разрабатывает программы для разных устройств на предприятиях и заводах: станки ЧПУ, конвейеры, хлебопечки и другие. Основное отличие от остальных разработчиков - техническая экспертиза с опытом более 10 лет. Инженер-программист обладает глубокими знаниями дискретной математики и физики, понимает промышленные и технологические процессы.

За что отвечает: автоматизация производства и программирование внутренних устройств.

Языки и технологии: C, C++, C#, Delphi, Assembler, технические знания.

7. DevOps - инженер. Одна из самых сложных позиций в ИТ-рекрутменте, потому что не всегда понятно, чем занимается такой специалист. DevOps — это методика повышения качества программного обеспечения, а DevOps - инженер использует ее для синхронизации всех этапов создания ИТ-продукта и совмещает в себе разработчика, тестировщика, менеджера и сисадмина. Он контролирует и автоматизирует работу разработчиков и других ИТ - специалистов, которые связаны с продуктом.

За что отвечает: кодирование, тестирование, запуск приложения, автоматизация, внедрение программных инструментов.

Языки и технологии: PHP, Perl, Ruby, Python, C++, Cloud Automation (Azure, GCP, Alibaba), Jira, системы мониторинга сетевых устройств, ОС Windows/Linux, ПО для автоматизации (Docker, Jenkins, Puppet).

8. Embedded - разработчик. Относительно новая и узкая специализация разработчиков, которые работают со встроенными устройствами, то есть с девайсами и гаджетами, в которых есть аппаратная платформа. Например, микроконтроллеры, встроенные программы и устройства. Embedded - developer умеет создавать ПО, а также должен разбираться в физических процессах электрических компонентов.

За что отвечает: разработка ПО, тестирование и отладка, оптимизация оборудования, разработка решений для диагностики сбоев устройств.

Языки и технологии: C/C ++, Assembler, компьютерные алгоритмы, инженерные и математические знания.

Мы рассказали об основных видах разработчиков, которые встречаются в ИТ - вакансиях. Наши рекрутеры знают, как подбирать таких разработчиков и помогут найти классного специалиста. Оставляйте заявку на нашем сайте.

2.2 Backend – разработка.

2.1.1 Что такое backend – разработка?

Backend - разработка, или разработка серверной части программного обеспечения, играет ключевую роль в создании современных веб-приложений. Backend отвечает за обработку запросов от клиентской стороны, взаимодействие с базами данных, бизнес-логику приложения и многие другие аспекты работы системы. Основная задача backend - разработчиков - создать структуру, которая обеспечивает надежную и эффективную работу приложения.

Backend - разработка основывается на использовании различных технологий и языков программирования. Часто для создания серверных приложений используются языки программирования, такие как Java, Python, PHP, Ruby, и другие. Эти языки позволяют разработчикам реализовать функционал серверной части приложения, обеспечивают возможность взаимодействия с базами данных, обработку запросов от клиентов, а также обеспечивают безопасность и масштабируемость системы.

Важным аспектом backend - разработки является обеспечение безопасности приложения. Backend - разработчики должны уделять особое внимание защите данных пользователей, обработке запросов безопасным образом, предотвращению уязвимостей, и многим другим аспектам, связанным с кибербезопасностью.

Еще одной важной составляющей backend - разработки является оптимизация производительности. Разработчики должны создавать эффективные и масштабируемые решения, которые обеспечивают быструю обработку запросов и отзывчивость системы даже при больших нагрузках.

2.1.2 Backend - разработка на языке программирования Python.

Backend - разработка на Python представляет собой процесс создания серверной части веб-приложений с использованием языка программирования Python. Python — это мощный и гибкий язык программирования, который широко применяется в веб-разработке благодаря своей простоте и выразительности.

Backend - разработчики, работающие с Python, отвечают за реализацию функционала серверной части приложения. Их задачи включают в себя разработку API, взаимодействие с базами данных, обработку запросов от клиентов, реализацию бизнес - логики приложения и обеспечение безопасности данных.

Python имеет богатое экосистему инструментов и фреймворков, которые облегчают разработку серверной части приложений. Некоторые из наиболее популярных фреймворков для Backend - разработки на Python включают Django, Flask, FastAPI и другие. Эти фреймворки предоставляют разработчикам готовые инструменты для работы с HTTP - запросами, маршрутизацией, авторизацией и многими другими задачами.

Python отличается высокой читаемостью кода, простотой синтаксиса и богатой стандартной библиотекой, что делает его привлекательным языком для создания серверных приложений. Кроме того, Python поддерживает асинхронное программирование, что позволяет создавать высокопроизводительные и отзывчивые веб-приложения.

Backend - разработка на Python представляет собой увлекательный и перспективный направление в сфере веб - разработки. Использование Python в качестве основного инструмента для создания серверных приложений позволяет разработчикам эффективно реализовывать функционал приложения, обеспечивая высокую производительность и безопасность.

Глава 3. Основные понятия web – технологий.

3.1 Основы HTML.

3.1.1 Что такое HTML?

HTML (Hyper Text Markup Language) — стандартизированный язык гипертекстовой разметки документов для просмотра веб-страниц в браузере. Веб-браузеры получают HTML документ от сервера по протоколам HTTP/HTTPS или открывают с локального диска, далее интерпретируют код в интерфейс, который будет отображаться на экране монитора. Понимание основ HTML и его особенностей имеет важное значение для любого web – разработчика.

HTML использует теги, которые представляют собой метки, используемые для разметки и классификации информации, что упрощает для браузеров интерпретацию и форматирование контента на веб-сайте. Проще говоря, теги служат ключевыми словами, которые указывают браузеру, как обрабатывать и отображать содержимое сайта.

3.1.2 Структура элементов разметки HTML

Ниже представлена структура HTML элемента:

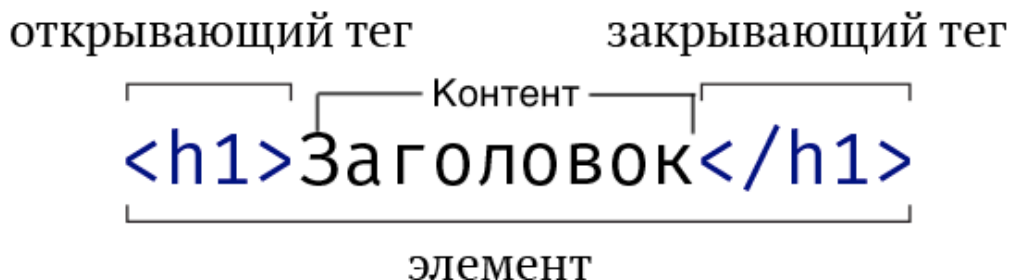


Рисунок 1. Структура элементов разметки HTML.

Главными частями нашего элемента являются:

1. Открывающий тег. Состоит из имени элемента (в данном случае, "h1"), заключённого в открывающие и закрывающие угловые скобки. Открывающий тег указывает, где элемент начинается или начинает действовать, в данном случае — где начинается заголовок первого уровня.
2. Закрывающий тег — это то же самое, что и открывающий тег, за исключением того, что он включает в себя косую черту перед именем элемента (в данном случае, "/h1"). Закрывающий элемент указывает, где элемент заканчивается, в данном случае — где заканчивается заголовок первого уровня. Отсутствие закрывающего тега является одной из наиболее распространённых ошибок начинающих.
3. Контент - то контент элемента, который в данном случае является просто текстом.
4. Элемент - Открывающий тег, закрывающий тег и контент вместе составляют элемент.

HTML элементы могут иметь атрибуты. Атрибуты предназначены для добавления дополнительной информации об элементе и/или для изменения его стандартного поведения. Атрибуты всегда указываются внутри открывающего тега. В большинстве случаев атрибуты являются необязательными и указываются только при необходимости. Один элемент никогда не должен содержать в себе два и более атрибутов, имеющих одинаковое имя. Синтаксис элемента с атрибутом:

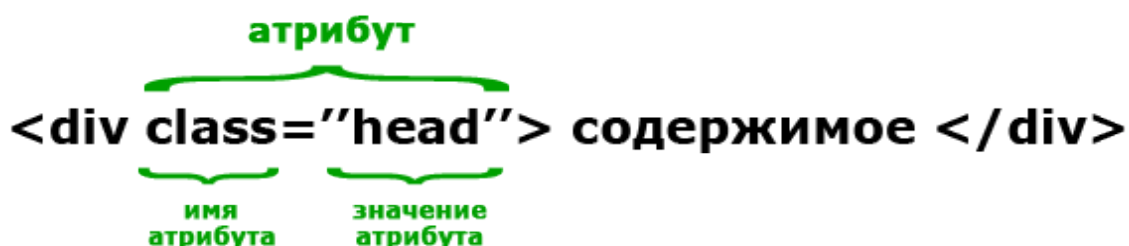


Рисунок 2. Атрибут элемента.

Атрибуты состоят из пары: имя_атрибута="значение". Имена атрибутов не чувствительны к регистру символов и могут быть записаны с любым сочетанием строчных и заглавных букв. Значение атрибута может состоять из текста, цифр и других символов (единственным исключением является символ амперсанда (&), его использование запрещено). Дополнительные ограничения на использование различных символов в значении атрибута зависят от способа записи значения. При указании нескольких атрибутов в одном элементе, они должны быть отделены друг от друга символом пробела. Порядок следования атрибутов не имеет значения.

Атрибуты часто используются для поиска элементов на странице, поэтому необходимо обратить на них внимание при изучении HTML.

3.2 Основы CSS

3.2.1 Что такое CSS?

CSS (Cascading Style Sheets) — это код, используемый для настройки внешнего вида вашей веб-страницы. Каскадные таблицы стилей предоставляют дополнительные возможности для разметки html-документа и свободу по созданию уникального дизайна для веб-страниц.

Когда браузер обрабатывает HTML-код, он использует встроенный по умолчанию стиль представления HTML-элементов на веб-странице. Чтобы понять, что такое "стиль по умолчанию" рассмотрим в качестве примера элементы `<h1>` - `<h6>`: заголовки являются блочными элементами, занимают всю доступную ширину в родительском элементе, имеют разрыв строки до и после элемента, текст заголовка отображается жирным начертанием и имеет определённый размер, в зависимости от уровня заголовка, всё это вместе является встроенным стилем для заголовков.

С помощью CSS можно переопределить установленный для элементов стиль по умолчанию на свой собственный, создав тем самым уникальный стиль оформления для элементов веб-страницы, например изменить цвет текста заголовка и размер шрифта, выделить изображение красной рамкой и т.д.

Чем полезны таблицы стилей, помещенные в отдельный документ? Ответ очень прост, можно собрать все стили, которые используются на сайте, в один внешний файл с расширением `.css` и связать его со всеми страницами сайта. После этого, когда вы будете редактировать стиль, изменения моментально затрагивают все элементы на страницах сайта, где есть ссылка на данный внешний файл со стилями. Таким образом, вы можете полностью изменить внешний вид путем редактирования единственного файла таблицы стилей, что существенно упрощает работу, нежели редактировать стили на каждой странице в отдельности.

3.2.2 Синтаксис CSS

CSS представляет из себя правило или набор правил, описывающих форматирование (изменение внешнего вида) отдельных элементов на веб-странице. Правило, состоит из двух частей: селектора и следующим за ним блоком объявлений. На изображении ниже показана структура (синтаксис) правила:



Рисунок 3. Синтаксис CSS.

- Первым всегда указывается селектор, он сообщает браузеру, к какому элементу или элементам веб-страницы будет применен стиль.
- Далее следует блок объявлений, который начинается с открывающей фигурной скобки {и заканчивается закрывающей}, между фигурными скобками указываются форматирующие команды (объявления), которые используются браузером для стилизации выбранного селектором элемента. В блоке может быть указано сколько угодно объявлений. Сам блок объявлений так же называют просто стилем.
- Каждое объявление состоит из двух частей: свойства и его значения. Объявление всегда должно заканчиваться точкой с запятой (;). Опустить указание ";" можно только в конце последнего объявления перед закрывающей фигурной скобкой.
- Свойство — это команда форматирования, определяющая конкретный стилевой эффект для элемента. Само свойство представляет из себя слово или несколько написанных через дефис слов. Каждое свойство имеет свой предопределенный набор значений. После имени свойства указывается двоеточие, которое отделяет название свойства от допустимого значения.

3.3 Основы JavaScript

JavaScript – это интерпретируемый язык программирования, разработанный для взаимодействия с веб-страницами. JavaScript представляет собой реализацию ECMAScript. ECMAScript – это просто описание языка, который определен в стандарте ECMA-262.

Возможности:

Итак, небольшой список того, что позволяет JavaScript:

- Добавлять различные эффекты анимации
- Реагировать на события - обрабатывать перемещения указателя мыши, нажатие клавиш с клавиатуры
- Осуществлять проверку ввода данных в поля формы до отправки на сервер, что в свою очередь снимает дополнительную нагрузку с сервера
- Создавать и считывать cookie, извлекать данные о компьютере посетителя
- Определять браузер
- Изменять содержимое HTML-элементов, добавлять новые теги, изменять стили
- Этим конечно же список не ограничивается, так как помимо перечисленного JavaScript позволяет делать и многое другое.
-

Ограничения:

Существуют так же и некоторые ограничения, распространяемые на данный язык:

- JavaScript не может закрывать окна и вкладки, которые не были открыты с его помощью. Не может защитить исходный код страницы и запретить копирование текста или изображений со страницы
- Не может осуществлять кросс доменные запросы, получать доступ к веб-страницам, расположенным на другом домене. Даже когда страницы из разных доменов отображаются в одно и тоже время в разных вкладках браузера, то код JavaScript принадлежащий одному домену не будет иметь доступа к информации о веб-странице из другого домена. Это гарантирует безопасность частной информации, которая может быть известна владельцу домена, страница которого открыта в соседней вкладке
- Не имеет доступа к файлам, расположенным на компьютере пользователя, и доступа за пределы самой веб-страницы, единственным исключением являются файлы cookie, это небольшие текстовые файлы, которые JavaScript может записывать и считывать
- В целом, можно сказать, что он разработан таким образом, чтобы затруднить выполнение вредоносного кода.

Глава 4. Git. Знакомство с контролем версий

4.1 Основы систем контроля версий

Система контроля версий — это система, записывающая изменения в файл или набор файлов в течение времени и позволяющая вернуться позже к определённой версии. Для контроля версий файлов в этой книге в качестве примера будет использоваться исходный код программного обеспечения, хотя на самом деле вы можете использовать контроль версий практически для любых типов файлов.

Система контроля версий Git является распределенной, она используется для отслеживания изменений в файлах и директориях проекта. Позволяет разработчикам контролировать и отслеживать изменения в проекте.

Программное обеспечение контроля версий отслеживает все вносимые в код изменения в специальной базе данных. При обнаружении ошибки разработчики могут вернуться назад и выполнить сравнение с более ранними версиями кода для исправления ошибок, сводя к минимуму проблемы для всех участников команды.

Практически во всех программных проектах исходный код является сокровищем: это ценный ресурс, который необходимо беречь. Для большинства команд разработчиков программного обеспечения исходный код — это репозиторий бесценных знаний и понимания проблемной области, которые они скрупулезно собирали и совершенствовали. Контроль версий защищает исходный код от катастрофических сбоев, от случайных ухудшений, вызванных человеческим фактором, а также от непредвиденных последствий.

Группы разработчиков программного обеспечения, не использующие какую-либо форму управления версиями, часто сталкиваются с такими проблемами, как незнание об изменениях, выполненных для пользователей, или создание в двух несвязанных частях работы изменений, которые оказываются несовместимыми и которые затем приходится скрупулезно распутывать и перерабатывать. Если вы как разработчик ранее никогда не применяли управление версиями, возможно, вы указывали версии своих файлов, добавляя суффиксы типа «финальный» или «последний», а позже появлялась новая финальная версия. Возможно, вы использовали комментирование блоков кода, когда хотели отключить определенные возможности, не удаляя их, так как опасались, что этот код может понадобиться позже. Решением всех подобных проблем является управление версиями.

4.2 Преимущества систем контроля версий

Программное обеспечение контроля версий рекомендуется для продуктивных команд разработчиков и команд DevOps. Управление версиями помогает отдельным разработчикам работать быстрее, а командам по разработке ПО — сохранять эффективность и гибкость по мере увеличения числа разработчиков.

За последние несколько десятилетий системы контроля версий (Version Control Systems, VCS) стали гораздо более совершенными, причем некоторым это удалось лучше других. Системы VCS иногда называют инструментами SCM (управления исходным кодом) или RCS (системой управления редакциями). Один из наиболее популярных на сегодняшний день инструментов VCS называется Git. Git относится к категории распределенных систем контроля версий, известных как DVCS (эта тема будет рассмотрена подробнее чуть позже). Git, как и многие другие популярные и доступные на сегодняшний день системы VCS, распространяется бесплатно и имеет открытый исходный код. Независимо от того, какую систему контроля версий вы используете и как она называется, основные ее преимущества заключаются в следующем.

1. Полная история изменений каждого файла за длительный период. Это касается всех изменений, внесенных огромным количеством людей за долгие годы. Изменением считается создание и удаление файлов, а также редактирование их содержимого. Различные инструменты VCS отличаются тем, насколько хорошо они обрабатывают операции переименования и перемещения файлов. В историю также должны входить сведения об авторе, дата и комментарий с описанием цели каждого изменения. Наличие полной истории позволяет возвращаться к предыдущим версиям, чтобы проводить анализ основных причин возникновения ошибок и устранять проблемы в старых версиях программного обеспечения. Если над программным обеспечением ведется активная работа, то «старой версией» можно считать почти весь код этого ПО.

2. Ветвление и слияние. Эти возможности полезны не только при одновременной работе участников команды: отдельные сотрудники также могут пользоваться ими, занимаясь несколькими независимыми направлениями. Создание «веток» в инструментах VCS позволяет иметь несколько независимых друг от друга направлений разработки, а также выполнять их слияние, чтобы инженеры могли проверить, что изменения, внесенные в каждую из веток, не конфликтуют друг с другом. Многие команды разработчиков ПО создают отдельные ветки для каждой функциональной возможности, для каждого релиза либо и для того, и для другого. Имея множество различных рабочих процессов, команды могут выбирать подходящий для них способ ветвления и слияния в VCS.

3. Отслеживаемость. Возможность отслеживать каждое изменение, внесенное в программное обеспечение, и связывать его с ПО для управления проектами и отслеживания багов, например Jira, а также оставлять к каждому изменению комментарий с описанием цели и назначения изменения может помочь не только при анализе основных причин возникновения ошибок, но и при других операциях по исследованию. История с комментариями во время чтения кода помогает понять, для чего этот код нужен и почему он структурирован именно так. Благодаря этому разработчики могут вносить корректные и совместимые изменения в соответствии с долгосрочным планом разработки системы. Это особенно важно для эффективной работы с унаследованным кодом, поскольку дает специалистам возможность точнее оценить объем дальнейших задач.

Глава 5. Основы языка Python

5.1 Знакомство с языком программирования Python

Python – это интерпретируемый, высокоуровневый язык программирования, который известен своей простотой и эффективностью. В данной работе мы рассмотрим основы Python, включая его синтаксис, типы данных, структуры данных и другие важные аспекты.

Синтаксис Python.

Одной из ключевых особенностей Python является его удобный и понятный синтаксис. Python использует отступы для обозначения блоков кода, что делает программы на этом языке более читаемыми. Например, циклы и условные операторы в Python записываются так:

```
1. for i in range(5):
2.     if i % 2 == 0:
3.         print(i)
```

Типы данных в Python.

Python поддерживает различные встроенные типы данных, такие как целые числа (int), числа с плавающей запятой (float), строки (str), списки (list), кортежи (tuple), множества (set) и словари (dict). Каждый тип данных имеет свои особенности и методы работы, что делает Python мощным инструментом для решения различных задач.

Структуры данных в Python

В Python существует множество встроенных структур данных, которые облегчают работу с информацией. Например, списки позволяют хранить упорядоченные коллекции элементов, а словари позволяют ассоциировать ключи и значения.

Python также известен своей обширной стандартной библиотекой, которая включает в себя множество полезных модулей и функций для работы с различными аспектами программирования, такими как работа с файлами, сетевое программирование, веб-разработка и многое другое. Это делает Python универсальным инструментом для создания разнообразных приложений.

Другим важным аспектом Python является его поддержка объектно-ориентированного программирования (ООП). В Python можно создавать классы и объекты, что позволяет организовывать код в логические блоки и повторно использовать его. Наследование,

инкапсуляция и полиморфизм – основные принципы ООП, которые легко реализуются в Python.

Кроме того, Python активно используется в различных областях, таких как наука о данных, машинное обучение, искусственный интеллект, веб-разработка и другие. Благодаря своей простоте, гибкости и эффективности Python стал одним из самых популярных языков программирования в мире.

Итак, изучение основ Python является отличным стартом для того, чтобы погрузиться в мир программирования и начать создавать свои проекты. Благодаря обширным возможностям и простоте использования Python отлично подходит как для начинающих программистов, так и для опытных специалистов.

Заключение. Python - мощный и гибкий язык программирования, который позволяет разработчикам создавать разнообразные приложения и решать сложные задачи. Изучив его основы, включая синтаксис, типы и структуры данных, можно стать более продуктивным и эффективным программистом. В данной работе мы кратко рассмотрели основы Python, однако изучение этого языка не ограничивается этими темами – Python поистине бесконечен в своих возможностях.

5.2 Базовые понятие ООП Python

Объектно-ориентированное программирование (ООП) — это парадигма программирования, основанная на концепции объектов, которые могут содержать данные (атрибуты) и функции (методы), работающие с этими данными. ООП — это мощная парадигма программирования, которую можно использовать для создания более модульного, повторно используемого и поддерживаемого кода.

Однако количество кода растёт по экспоненте с количеством классов, которые вы внедряете по мере разработки.

Существует четыре основных принципа ООП:

1. **Инкапсуляция.** Относится к практике сокрытия внутренних деталей объекта от внешнего мира и раскрытия только общедоступного интерфейса, который можно использовать для взаимодействия с объектом. Это помогает защитить целостность объекта и упрощает поддержку и изменение кода с течением времени.
2. **Абстракция.** Относится к практике представления сложных систем или понятий с использованием более простых и общих понятий. Абстракция позволяет нам сосредоточиться на основных характеристиках объекта и игнорировать детали, которые не имеют отношения к нашей текущей задаче.
3. **Наследование.** Относится к практике создания новых классов, которые наследуют атрибуты и методы существующих классов. Наследование позволяет нам повторно использовать код и создавать более специализированные классы, основанные на функциональности более общих классов.
4. **Полиморфизм.** Относится к практике использования одного интерфейса для представления нескольких типов объектов. Полиморфизм позволяет нам писать более общий код, который может работать с широким спектром объектов, без необходимости знать конкретные детали каждого объекта.

Инкапсуляция:

Инкапсуляция является одним из ключевых принципов ООП, который позволяет объединять данные и методы работы с ними в единый объект, скрывая детали реализации от внешнего мира. В Python инкапсуляция достигается с помощью использования классов и их методов. Пример:

```
1. class Person:
2.     def __init__(self, name, age):
3.         self._name = name
4.         self._age = age
```

```

5.
6.         def get_name(self):
7.             return self._name
8.
9.         def get_age(self):
10.            return self._age
11.
12.         person1 = Person("Alice", 30)
13.         print(person1.get_name()) # Выведет: Alice

```

Наследование:

Наследование позволяет создать новый класс на основе существующего, заимствуя у него свойства и методы. В Python класс наследуется с помощью указания родительского класса в скобках при создании нового класса. Пример:

```

1.     class Student(Person):
2.         def __init__(self, name, age, student_id):
3.             super().__init__(name, age)
4.             self._student_id = student_id
5.
6.         def get_student_id(self):
7.             return self._student_id
8.
9.     student1 = Student("Bob", 25, 12345)
10.    print(student1.get_name()) # Выведет: Bob
11.    print(student1.get_student_id()) # Выведет: 12345

```

Полиморфизм:

Полиморфизм позволяет использовать один и тот же метод для разных типов данных. В Python полиморфизм проявляется в том, что одинаковые методы могут работать с различными объектами. Пример:

```

1. class Shape:
2.     def area(self):
3.         pass
4.
5. class Circle(Shape):
6.     def __init__(self, radius):
7.         self._radius = radius
8.
9.     def area(self):

```

```
10.         return 3.14 * self._radius**2
11.
12. class Square(Shape):
13.     def __init__(self, side):
14.         self._side = side
15.
16.     def area(self):
17.         return self._side**2
18.
19. circle = Circle(5)
20. square = Square(4)
21. print(circle.area()) # Выведет: 78.5
22. print(square.area()) # Выведет: 16
```

Изучение этих принципов ООП в Python поможет вам создавать более структурированный и гибкий код, упростить его поддержку и расширение. В дипломном проекте можно привести больше примеров, подробно описать применение каждого принципа и продемонстрировать их в действии в различных сценариях.

Глава 6. Фреймворк Django

6.1 Что такое Django?

Django — это высокоуровневый фреймворк для веб-приложений на языке Python. Он был создан в 2005 году и с тех пор активно развивается и обновляется сообществом разработчиков по всему миру.

Но первый вопрос, почему именно Django? Наверное, этих фреймворков очень много и это действительно так. Однако именно Django завоевал огромную популярность благодаря простоте разработки даже очень сложных сайтов, например, таких как youtube, google search, dropbox и instagram. Все они сделаны с использованием фреймворка Django. И давайте теперь разберемся, для чего он вообще нужен. Если очень кратко, то для обеспечения работы сайтов на стороне сервера. То есть, это инструмент для бэкенд разработчиков. А если немного подробнее, то процесс взаимодействия пользователя с сайтом можно представить следующим образом:

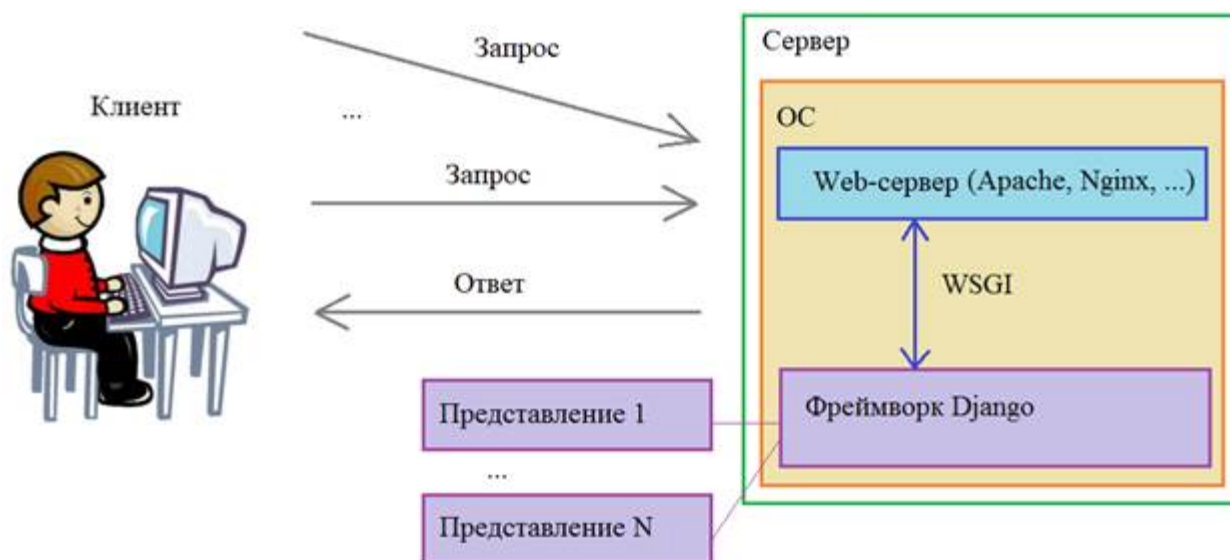


Рисунок 4. Взаимодействие пользователя с сайтом.

Сначала от клиента поступает запрос, например, он вводит в браузере адрес <https://youtube.com>. Информационные пакеты начинают двигаться в сторону сервера с сайтом Youtube. Вопрос о том, как они это делают, мы оставим в стороне, это называется маршрутизацией в сети Интернет. Главное, что они доходят до сервера (то есть, компьютера), где расположен Youtube. На этом компьютере установлено специальное программное обеспечение – Web-сервер. Очень часто используют уже готовые разработки с открытым кодом: Apache и Nginx. Реже какие-либо свои программные продукты. Так вот, этот Web-

сервер постоянно «слушает» каналы связи и в момент поступления запроса от пользователя должен перенаправить его на обработку соответствующему сайту (так как на одном сервере может находиться множество сайтов). Сайты способны по-разному обрабатывать входящие запросы, например, используя PHP и CGI скрипты, или же могут использоваться фреймворки, значительно упрощающие обработку запросов от клиентов сайта. В частности, Django – это и есть такой фреймворк, значительно упрощающий написание скриптов на языке Python. А взаимодействие с сервером происходит по интерфейсу WSGI, который передает обработку запроса в Django. При этом отрабатывает определенное представление, отвечающее за текущий запрос, и результатом обработки, как правило, является HTML-страница, которая сначала передается Web-серверу также по WSGI-интерфейсу, а он уже передает страницу конечному пользователю. Страница отображается в браузере клиента, и он видит заветный сайт Youtube. Вот так в очень упрощенном виде происходит взаимодействие между пользователем, сервером и фреймворком. WSGI (Web Server Gateway Interface) — стандарт взаимодействия между Python-программой, выполняющейся на стороне сервера, и самим веб-сервером.

Далее, мы установим интегрированную среду программирования PyCharm, разработанную специально для написания программ на Python. Она довольно удобна для разработки программ с использованием Django, поэтому все дальнейшие действия мы будем производить в PyCharm.

6.2 Установка и настройка Django

Так как разработка велась на операционной системе MacOS, все команды, используемые в терминале, будут записанные в этом проекте для данное ОС.

Python помогают решить проблему управления зависимостями проектов. Если у вас есть несколько проектов на Python, то вероятность того, что вам придется работать с разными версиями библиотек или самого Python, очень высока. Но использование виртуальных сред позволяет создавать независимые группы библиотек для каждого проекта, что предотвращает конфликты между версиями и не дает одному проекту повлиять на другой. В Python уже есть модуль `venv` для создания виртуальных сред, который можно использовать как в разработке, так и в производстве.

Создаём виртуальное окружение в Linux или MacOS:

```
mkdir project
```

```
cd project
```

```
python3 -m venv venv
```

Далее активируем виртуальное окружение, чтобы все дальнейшие действия выполнялись внутри него: `venv/bin/activate`

Для установки Django рекомендуется использовать менеджер пакетов `pip`. Для этого необходимо открыть терминал и выполнить команду: `pip3 install Django`.

Помимо самого фреймворка будет установлено несколько обязательных зависимостей. В прочем сам Django содержит в себе около 15 пакетов, позволяющих решать большинство задач разработки без установки дополнительных пакетов и модулей.

6.2.1 Создание проекта и запуск проекта

Первым шагом создания проекта мы определим папку, в которой будем вести разработку данного проекта. После выполним все действия, связанные с установкой и активацией виртуального окружения и установкой самого Django.

Используя ядро Django, мы можем создавать множество разных сайтов под управлением Django в рамках текущего виртуального окружения. Чтобы посмотреть список команд ядра, достаточно в терминале записать: `django-admin`.

Для создания нового проекта в Django необходимо выполнить команду `django-admin startproject <project_name>`. Здесь «`project_name`», обычно, является доменным именем. Например, если мы собираемся располагать сайт на домене `cinemasite`, то пропишем: `django-admin startproject cinemasite`. Эта команда создаст структуру проекта, которая будет содержать все необходимые файлы и папки для работы с фреймворком.

Структура проекта Django имеет следующий вид:

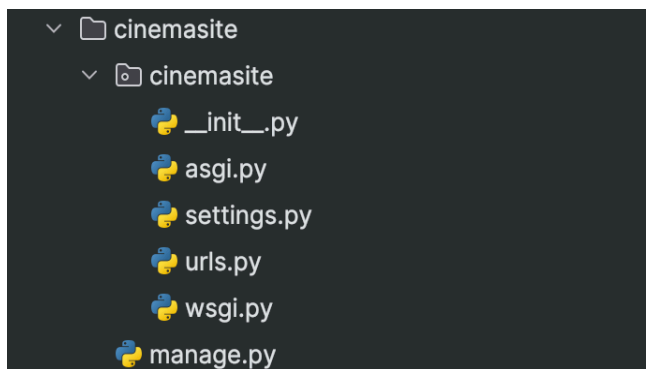


Рисунок 5. Структура проекта Django.

Рассмотрим каждый из созданных файлов.

- `manage.py` - файл, который используется для управления проектом. С его помощью можно запустить сервер, создать миграции, создать суперпользователя и т.д.
- `cinemasite/` - директория, которая содержит основные файлы проекта.
- `__init__.py` - файл, который сообщает Python, что директория `myproject` является пакетом.
- `settings.py` - файл, который содержит настройки проекта, такие как базы данных, шаблоны, статические файлы и т.д.
- `urls.py` - файл, который содержит маршруты приложения.
- `asgi.py` - файл, который используется для запуска проекта в ASGI-совместимых серверах.
- `wsgi.py` - файл, который используется для запуска проекта в WSGI-совместимых серверах.

Далее мы можем запустить наш сервер на нашем компьютере используя команду: `python manage.py runserver`. После запуска сервера можно открыть браузер и перейти по адресу <http://127.0.0.1:8000/>. Если все настроено правильно, то вы увидите страницу "The install worked successfully! Congratulations!" с ракетой.

Сервер Django используется лишь при разработке и тестировании проекта. Он способен раздавать статику в виде изображений, CSS файлов, JavaScript кода и т.п. Так же сервер отслеживает изменения, которые вы вносите в файлы проекта. После каждого такого изменения сервер автоматически перезагружается. Встроенный сервер нельзя использовать в продакшене. Он подходит только для разработки проекта.

При первом запуске сервера автоматически создается файл `db.sqlite3` - файл БД SQLite3. Дело в том, что по умолчанию Django использует именно такую СУБД. В дальнейшем, мы можем это изменить и указать любую другую СУБД, которую поддерживает данный фреймворк. Это может быть: PostgreSQL, MariaDB, MySQL, Oracle и SQLite.

Но вернемся к запуску сервера. Его можно запускать также и со следующими параметрами: `python manage.py runserver 192.168.1.1:4000`. Это значит, что теперь мы можем перейти по ссылке <http://192.168.1.1:4000/> и увидеть наш проект.

6.2.2 MVT концепция. Создание первого приложения

Концепция Модель-Представление-Шаблон (MVT) является основой фреймворка Django. Она разделяет приложение на три основных компонента: модель (Model), представление (View) и шаблон (Template).

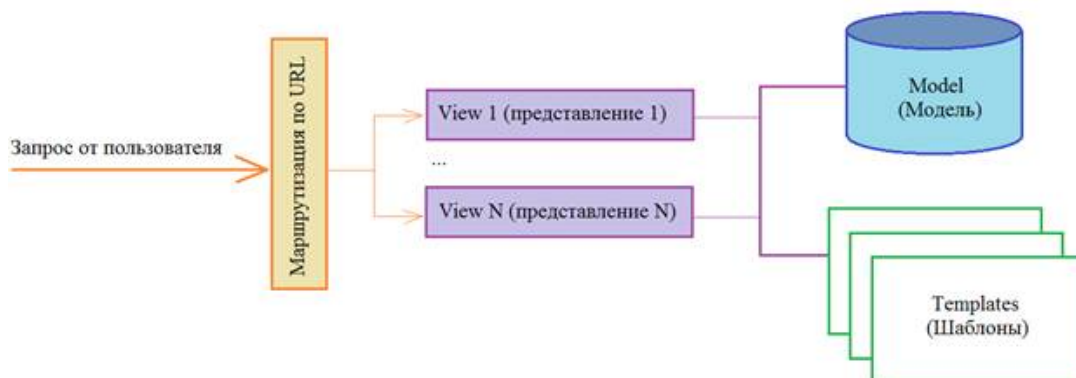


Рисунок 6. MVT модель.

Модель — это объект, который представляет данные в приложении и отвечает за их хранение и манипуляцию. Модели Django обычно соответствуют таблицам в базе данных. Представление — это компонент, который отвечает за обработку запросов и формирование ответов на основе данных из моделей. В Django представления обычно реализуются в виде функций или классов.

Шаблон — это компонент, который отвечает за отображение данных на странице. Шаблоны Django используют язык шаблонов, который позволяет вставлять данные из моделей и представлений в HTML-код.

Отдельно стоит упомянуть маршруты. Маршруты в Django играют роль посредника между пользователем и представлением. Они определяют, какой URL-адрес должен быть связан с каким представлением. Когда пользователь запрашивает URL-адрес, Django использует маршруты для определения соответствующего представления и передачи запроса ему. Представление обрабатывает запрос и формирует ответ, который затем возвращается пользователю.

Добавление первого приложения.

Согласно философии Django мы должны создать новое приложение в рамках нашего сайта. Что это за приложение и зачем оно вообще нужно? Разработчики фреймворка решили, что каждая самостоятельная часть сайта должна представляться в виде своего отдельного приложения. Например, создавая информационный сайт, мы должны будем определить приложение для отображения страниц этого сайта по определенным запросам. Далее, к нам приходит руководитель проекта и сообщает, что еще нужно реализовать форум на сайте. И, так как это функционально независимая часть сайта, то мы создаем еще одно приложение для

форума. Затем, руководитель почесал свою репу и вспомнил, что еще нужно сделать раздел с опросом пользователей по разным тематикам. И на сайте появляется еще одно приложение – для опроса. И так далее. Каждая логически и функционально независимая часть сайта предполагает его реализацию в виде отдельного приложения.

Приложения в Django следует реализовывать максимально независимыми, в идеале – полностью независимыми, чтобы в дальнейшем мы могли их просто скопировать в другой сайт и там оно сразу же начинало бы работать. Это не всегда удастся, но к этому нужно стремиться.

Итак, давайте создадим в нашем сайте первое приложение, которое возьмет на себя базовый функционал, то есть, оно и будет являться ядром нашего сайта. Для этого мы откроем терминал и, находясь в каталоге `cinemasite/`, выполню команду: `python manage.py startapp cinema_app`. Окончание `app` не является обязательным. Приложение может иметь любое имя, которое отражает его назначение. Но наличия суффикса `app` позволяет быстрее отличить пакет приложения от другого каталога внутри проекта.

Структура проекта после создания приложения имеет следующий вид:

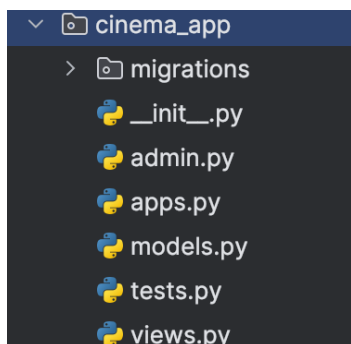


Рисунок 7. Структура приложения в Django

После выполнения команды у нас в проекте появилась еще одна папка – `cinema_app`, которая уже содержит несколько файлов, в том числе, и файл `__init__.py`. Следовательно, приложение в Django реализуется как пакет языка Python. Также здесь присутствует одна вложенная папка `migrations` для хранения миграций БД нашего приложения. Подробнее о ней мы поговорим позже. Остальные файлы имеют следующее назначение:

- `admin.py` – для настройки админ-панели сайта (админ-панель поставляется совместно с Django и каждый сайт может сразу ее использовать);
- `apps.py` – для настройки (конфигурирования) текущего приложения;
- `models.py` – для хранения ORM-моделей для представления данных из базы данных;
- `tests.py` – модуль с тестирующими процедурами;
- `views.py` – для хранения представлений (контроллеров) текущего приложения.

После создания приложения его необходимо зарегистрировать в проекте нашего сайта, чтобы фреймворк Django «знал» о его существовании и корректно с ним работал. Для этого

нужно перейти в пакет конфигурации сайта (cinemasite), открыть файл settings.py и в списке INSTALLED_APPS прописать новое приложение. В нем уже прописаны несколько стандартных приложений самого фреймворка и к ним мы просто добавим свое:

```
1. INSTALLED_APPS = [  
2. ...  
3.     cinema_app,  
4. ]
```

Этого вполне достаточно и все будет работать, но в действительности Django обращаясь к этому пакету находит файл apps.py, откуда и берет настройки приложения из класса CinemaAppConfig. Чтобы в дальнейшем каждый раз не конкретизировать этот путь, я пропишу его сразу в списке приложений:

```
1. INSTALLED_APPS = [  
2. ...  
3.     cinema_app.apps.CinemaAppConfig,  
4. ]
```

6.2.3 Маршрутизация и функции представления

Следующий пункт работы - создать представление. Так же их называют view или просто “вьюшками”.

Давайте для начала создадим обработчик главной страницы приложения `cinema_app` с помощью функции представления. Как я уже отмечал, представления в Django можно реализовывать или в виде функций, или в виде классов. Но для лучшего понимания механики обработки запросов мы воспользуемся именно функцией. Эта функция будет срабатывать при обращении к главной странице приложения `cinema_app`, называться `index` и формировать ответ в виде простой строки:

```
1. def index(request):  
2.     return HttpResponse("Страница приложения cinema_app.")
```

Здесь указывается первый обязательный параметр `request` — это ссылка на экземпляр класса `HttpRequest`, который содержит информацию о запросе, о сессии, куках и так далее. То есть, через переменную `request` нам доступна вся возможная информация в рамках текущего запроса. На выходе эта функция возвращает экземпляр объекта `HttpResponse`, который будет автоматически формировать нужный заголовок ответа, а содержимое будет представлено простой строкой.

Теперь нам нужно связать эту функцию представления с соответствующим URL-адресом. Для настройки URL в Django необходимо определить маршруты (routes), которые будут связывать определенные URL с соответствующими представлениями (views) в приложении. Маршруты определяются в файле `urls.py`, который находится в корневой директории проекта и в директории приложения.

Открываем файл `urls.py` в корневой директории проекта. Файл уже содержит комментарий с описанием трёх способов добавления маршрутов:

- функции представления
- классы представления
- подключение других файлов с настройками URL

Пропишем добавление файла конфигурации маршрутов из приложения `cinema_app` в проект:

```
1. from django.contrib import admin  
2. from django.urls import path, include  
3. urlpatterns = [
```

```
4.     path('admin/', admin.site.urls), path('', include('cinema_app.urls')),
5. ]
```

Создаем файл `urls.py` в директории приложения.

Убедитесь, что вы создаёте файл внутри каталога `cinema_app`. Когда в вашем проекте будет несколько приложений, каждое будет иметь собственный `urls.py` с локальными маршрутами. А `urls.py` проекта будет включать (`include()`) их в глобальный список адресов. Внутри `cinema_app/urls.py` пропишем следующий код:

```
1. from django.urls import path
2. from . import views
3. urlpatterns = [
4.     path('', views.index, name='index'),
5. ]
```

Импортируем функцию `path` и файл с представлениями из текущего каталога (импорт через точку).

Функция `path()` принимает два аргумента: первый аргумент - это URL-адрес, а второй - это представление, которое будет обрабатывать запрос на этот URL. Также можно задать имя маршрута с помощью параметра `name`. Таким образом, настройка URL в Django позволяет определить маршруты для обработки запросов на определенные URL и связать их с соответствующими представлениями.

6.2.4 Динамические URL

В Django можно описывать динамические URL следующим образом:

```
1. urlpatterns = [  
2.     path('cats/<int:cat_id>/', views.categories, name='cat'),  
3.     path('cats/<slug:cat_slug>/', views.categories_by_slug, name='cat_slug'),  
4. ]
```

Здесь в угловых скобках записан параметр `cat_id`, который имеет тип `int` – целочисленный и `slug` – слаг. Этот тип в маршрутах называется конвертером. И указанный путь будет соответствовать любым комбинациям URL с фрагментом `'cats/число/'` и `'cats/slug/'`. Далее, в функции представления `categories` мы уже можем использовать этот параметр:

```
1. def categories(request, cat_id):  
2.     return HttpResponse(f"<h1>Фильмы по категориям</h1><p>id:{cat_id}</p>")
```

Или через `slug`:

```
1. def categories(request, cat_slug):  
2.     return HttpResponse(f"<h1>Фильмы по категориям</h1><p>id:{cat_slug}</p>")
```

Помимо конвертера `int` в Django можно использовать и другие:

- `str` – любая не пустая строка, исключая символ `'/'`;
- `int` – любое положительное целое число, включая 0;
- `slug` – слаг, то есть, латиница ASCII таблицы, цифры, символы дефиса и подчеркивания;
- `uuid` – цифры, малые латинские символы ASCII, дефис;
- `path` – любая не пустая строка, включая символ `'/'`.

Слаги понятнее для пользователя и поисковых систем. Сайты с такими URL, в среднем, лучше индексируются и занимают более высокие позиции в поисковой выдаче. Поэтому принято прописывать конвекторы включающее в себя слаг.

6.2.5 Обработчики исключений при запросе

Следующий важный аспект – это обработка исключений при запросах к серверу. Самым распространенным из них является обращение к несуществующей странице, когда сервер возвращает страницу с кодом 404.

Мы можем создать вывод информации об не существующей страницы в любом формате, в котором хотим ее видеть. Для этого пропишем в файле `views.py` нашего приложения код функцию, которая возвращает информацию, которую мы хотим видеть:

```
1. def page_not_found(request, exception):  
2.     return HttpResponseNotFound("<h1>Страница не найдена")
```

А в файле `urls.py` нашего проекта импортируем эту функцию и пропишем код:

```
1. from cinema_app.views import page_not_found  
2. handler404 = page_not_found
```

Существуют следующие коды состояния ответа HTTP:

1. Информационные ответы (100 – 199)
2. Успешные ответы (200 – 299)
3. Сообщения о перенаправлении (300 – 399)
4. Ошибки клиента (400 – 499)
5. Ошибки сервера (500 – 599)

6.2.6 Перенаправления redirect. Функция reverse

Очень часто при развитии сайта некоторые его страницы переносятся на другой URL-адрес. И чтобы не потерять позиции этих страниц в поисковой выдаче, поисковым системам нужно явно указать, что страница перемещена либо временно, либо постоянно на новый URL. Это делается с помощью перенаправлений с кодами:

- 301 – страница перемещена на другой постоянный URL-адрес;
- 302 – страница перемещена временно на другой URL-адрес.

В Django данные перенаправления (функция `redirect`) выполняются с помощью струнёной функции `django.shortcuts.redirect`:

```
1. from django.shortcuts import redirect
2. def categories(request, cat_id):
3.     return redirect ("/")
```

Тем самым мы перенаправим себя на главную страницу нашего сайта с кодом ответа HTTP 302. Если же мы хотим сделать перенаправление с кодом 301 - страница перемещена на другой постоянный URL-адрес, то нужно прописать `return redirect ("/", permanent=True)`.

В первый параметр `redirect` мы можем указать имя, данное нашей функции например `'cats'`.

Если же маршрут помимо имени содержит еще параметры, как например, маршрут `'cats'` с параметром `slug`, то для корректного перенаправления необходимо в функции `redirect()` вторым и последующими аргументами передать требуемые параметры. В нашем случае это можно сделать так: `return redirect('cats', 'comedy')`. В результате, функция `redirect()` вычислит следующий URL: <http://127.0.0.1:8000/cats/comedy/> и сделает на него перенаправление.

Но мы можем разделить операции вычисления URL и непосредственно перенаправление. Для этого в Django имеется функция: `django.urls.reverse()` которая возвращает строку URL-адреса, вычисленный на основе переданного имени и набора аргументов. Например, для вычисления адреса маршрута `cats` с параметром `'comedy'` функцию `reverse()` можно вызвать следующим образом:

```
1. from django.urls import reverse
2. def re(request):
3.     url_redirect = reverse('cats', args=('comedy',))
4.     return redirect(url_redirect)
```

Результатом перенаправления будет тот же URL: <http://127.0.0.1:8000/cats/comedy/>

6.3 Шаблоны в Django

6.3.1 Введение в шаблоны

Познакомимся со второй компонентой паттерна проектирования MTV – шаблонами (templates). Что это такое? Вот смотрите, если мы откроем наш проект и запустим тестовый веб-сервер, то на главной странице увидим отображение одной короткой строчки. Как вы понимаете, полноценная HTML-страница содержит гораздо больше информации, в том числе, заголовок и подключаемые статические файлы. Конечно, если решать эту задачу простым способом, то можно было бы написать в функции представления что-то вроде:

```
1. def index(request):
2.     return HttpResponse(''<!DOCTYPE html>
3. <html>
4.     <head>
5.         <title></title>
6.     </head>
7.     <body>
8.     </body>
9. </html>'')
```

Но если данным образом прописывать весь HTML код в представления, то его будет сложно читать, исправлять и, кроме того, изменение HTML-страницы повлечет изменение и самого приложения. Это не правильный подход к исполнению. Поэтому все это выносится за пределы приложения и организуется в виде шаблонов HTML-страниц. Работа с ними очень похожа на работу шаблонизатора Jinja.

Шаблоны для приложения `cinema_app` должны располагаться в подкаталоге `templates`, для этого создаем его. По умолчанию, Django ищет шаблоны в подкаталоге `templates` нашего приложения. Мы можем располагать здесь наши файлы шаблонов и все должно работать. Но есть один важный нюанс. В каком-либо другом приложении также может оказаться файл с именем `index.html`. Тогда фреймворк Django будет использовать тот, что встретится первым. Чтобы этого не происходило, в `templates` приложения принято создавать еще один подкаталог с именем приложения. В нашем случае – `cinema_app`. И уже в него помещать файлы шаблонов. Тогда все наши файлы шаблонов будут отделяться от других дополнительным подкаталогом и это позволит избежать коллизий имен файлов.

Обратите внимание, для корректного отображения кириллицы все шаблоны рекомендуется сохранять в кодировке utf-8. Тем более что сам Python, начиная с версии 3, по умолчанию использует юникод.

Далее нужно воспользоваться функцией `render()` и передать в нее сначала наш параметр `request`, а потом путь до шаблона относительно нашего приложения:

```
1. from django.shortcuts import render
2. def                                     index(request):
    return render(request, 'cinema_app/index.html')
```

При работе функций и `render()` фреймворк Django использует шаблонизатор, указанный в параметре `TEMPLATES` файла `settings.py` пакета конфигурации. В частности мы там видим строку: `'BACKEND': 'django.template.backends.django.DjangoTemplates'`, означающую, что используется встроенный шаблонизатор Django. Кроме того, здесь есть параметры `DIRS` и `APP_DIRS`. Параметр `DIRS` позволяет прописывать нестандартные пути к файлам шаблонов, а `APP_DIRS` со значением `True` указывает шаблонизатору Django искать шаблоны также и внутри приложений. Причем, шаблоны сначала ищутся по коллекции `DIRS`, а затем уже в приложениях. В частности, благодаря этому параметру успешно обнаруживаются наши шаблоны `index.html` и `about.html` внутри приложения `cinema_app`. Если установить параметр `APP_DIRS` в значение `False`, то при запуске проекта и перехода на главную страницу: `http://127.0.0.1:8000` появится ошибка: `TemplateDoesNotExist at /` говорящая, что шаблон не был найден.

6.3.2 Передача данных (контекста) в шаблон

Для передачи данных в шаблон, в функцию `render` нужно передать третий параметр (`context`), который в свою очередь принимает словарь с любыми коллекциям данных.

```
1. def index(request):
2.     context = {
3.         'title': 'Главная страница',
4.         'menu': menu,
5.         'float': 28.56,
6.         'lst': [1, 2, 'eee'],
7.         'set': {1, 2, 3, 2, 5},
8.         'dict': {'key_1': 'value_1', 'key_2': 'value_2'},
9.     }
10.    return render(request, 'cinema_app/index.html', context=context)
```

Для отображения на странице данных переданные в шаблон в файле HTML нужно прописать эти переменные в фигурных скобках:

```
1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4.     <meta charset="UTF-8">
5.     <title>{{ title }}</title>
6. </head><body>
7. <p>{{ menu }}</p>
8. <p>{{ float }}</p>
9. <p>{{ dict.key_1 }}</p>
10. <h1>{{ title }}</h1>
11. </body>
12. </html>
```

Обращается к ключам словаря нужно через точку: `{{ dict.key_1 }}`, Прописывать квадратные скобки с указанием ключа недопустимо: `{{dict['key_1']}}` – получим ошибку. Также следует иметь в виду, что каждый отображаемый параметр должен быть записан в отдельных двойных фигурных скобках. Прописывать несколько параметров нельзя.

6.3.3 Теги шаблонов. Теги if, for и url

В шаблонах фреймворка Django можно прописывать специальные теги. Стандарт записи тегов выглядит следующим образом: `{% <название тега> [параметры] %}`. Любой шаблонный тег нужно обязательно закрывать: `{% end<название тега> %}`.

```
1. {% for f in films %}
2.     <li>
3.         <h2>{{ f.title }}</h2>
4.         <p>{{ f.content }}</p>
5.         {% if f!= films|last %}
6.             <hr>
7.         {% endif %}
8.     </li>
9. {% endfor %}
```

Но не нужно путать теги: есть теги разметки HTML-документа, а есть теги шаблонизатора. Теги шаблонизатора обрабатываются на стороне сервера и служат для формирования общего вида HTML-документа, который, затем, возвращается пользователю. А в браузере клиента обрабатываются уже HTML-теги, в частности, для форматирования выводимой информации на экране устройства. Так вот, тег `for` – это тег шаблонизатора Django, который работает по аналогии с оператором цикла `for` языка Python и в нашем примере перебирает переданную коллекцию `cinema`. Конец тега-цикла `for` обязательно должно быть отмечено тегом `endfor`. Все, что попадает между этими тегами, образует тело цикла и повторяется на каждой итерации. То есть, на каждой итерации в HTML-документ будет добавляться тег `li` с соответствующим содержимым: заголовком `h2`, абзацем `p` с текстом статьи и разделительной линией (тег `hr`). Поэтому backend – разработчик должен знать основные правила HTML – разметки.

Тег `url` формируется обычно внутри HTML – тега `<a>` : `Название ссылки`. Определим маршрут представления: `path('film/<int:film_id>/', views.show_film, name='film')`, URL-адрес страницы передаем с помощью именованной функции представления и далее можем передать параметр. Данный код будет выглядеть так: `Читать описание`.

6.3.4 Наследование шаблонов

В Django есть возможность наследовать шаблоны, что позволяет создавать базовый шаблон и на его основе создавать другие, которые будут иметь общий вид и функциональность. Наследование шаблонов позволяет избежать дублирования кода и упростить процесс разработки. Мы перестаем нарушать принцип DRY.

В начале мы с вами определим базовый шаблон. Django позволяет создать базовый шаблон на уровне проекта. Поэтому создадим каталог `templates` в папке `cinemasite`, а в нем файл с именем `base.html`. Однако если мы сейчас попробуем обратиться к этому шаблону, о увидим ошибку, что шаблон не был найден. Это связано с тем, что маршрут `templates /base.html` не стандартный и его нужно явно прописать для шаблонизатора. Для этого нужно перейти в файл `settings.py` пакета конфигурации, найти параметр `TEMPLATES` и в коллекции `DIRS` прописать путь к каталогу `templates` `BASE_DIR / 'templates'`:

```
1. TEMPLATES = [  
2.     {  
3.         'BACKEND': 'django.template.backends.django.DjangoTemplates',  
4.         'DIRS': [  
5.             BASE_DIR / 'templates',  
6.         ],  
7.         'APP_DIRS': True,  
8.         ...  
9.     },  
10. ]
```

Теперь в базовый шаблон – `base.html` нужно разместить весь дублирующийся код, например главное меню нашего сайта. В блок `<body>` разместим этот код:

```
1. <body>  
2. <ul>  
3.     <li><a href="{% url 'home' %}">Главная страница</a></li>  
4.     {% for m in menu %}  
5.         {% if not forloop.last %}<li>{% else %}<li class="last">{% endif %}  
6.             <a href="{% url m.url_name %}">{{ m.title }}</a>  
7.         </li>  
8.     {% endfor %}  
9. </ul>  
10. {% block content %} {% endblock %}  
11. </body>
```

Как мы видим у нас появился тег `{% block content %} {% endblock %}`, он нужен для вставки контента из наследуемого шаблона. То есть, в него попадет весь код который мы укажем в данном блоке в наследуемом шаблоне в данном случае переменная `title`:

```
1. {% extends 'base.html' %}
2.
3. {% block content %}
4. <h1>{{ title }}</h1>
5. {% endblock %}
```

Теперь мы имеем новый тег: `{% extends 'base.html' %}`, он служит для того, чтобы дочерний шаблон понимал откуда мы наследуем базовый шаблон, в данном случае из файла `base.html`. Название блоков должны совпадать, если мы хотим, чтобы наш код в базовый шаблон попал именно в `block content`, то мы в дочернем шаблоне должны прописать то же `block content` и в нем писать наше содержимое сайта.

Из всего этого следует, что теперь вся общая информация, которая будет дублироваться на разных страницах нашего сайта, будет помещена в базовый шаблон `base.html`.

6.3.5 Подключение статистических файлов

Следующий важный шаг – научиться подключать к шаблонам статические файлы, например, оформление CSS, JavaScript, изображения и так далее. Здесь есть свои тонкости. Как мы помним, наш проект может работать в двух режимах: отладки – на тестовом веб-сервере; эксплуатации – на реальном веб-сервере. Так вот, в режиме отладки статические файлы Django ищет во всех подкаталогах static приложений, которые прописаны в коллекции `INSTALLED_APPS`. То есть, статические файлы при отладке совершенно спокойно можно размещать в подкаталоге static нашего приложения. Но, в режиме эксплуатации реальный веб-сервер будет брать все статические файлы из папки static, расположенной, как правило, в каталоге всего проекта. Для этого, при подготовке сайта к эксплуатации, выполняется специальная команда фреймворка Django: `python manage.py collectstatic`, которая собирает все статические файлы из разных приложений и модулей и помещает в одну общую папку проекта.

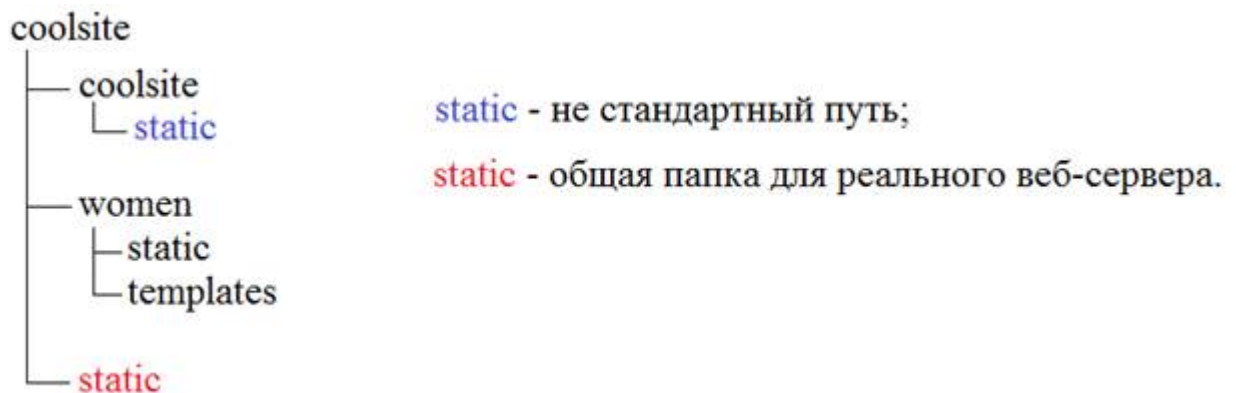


Рисунок 8. Подключение статистических файлов

Для того что бы корректно собиралась вся статика при выполнении команды `python manage.py collectstatic`, нужно в файле `settings.py` нашего проекта приписать константы:

- `STATIC_URL` – префикс URL-адреса для статических файлов;
- `STATIC_ROOT` – путь к общей статической папке, формируемой при запуске команды `collectstatic` (для сбора всей статика в единый каталог при размещении сайта на реальном веб-сервере);
- `STATICFILES_DIRS` – список дополнительных (нестандартных) путей к статическим файлам, используемых для сбора и для режима отладки.

6.3.6 Пользовательские теги шаблонов

В Django можно создавать свои собственные шаблонные теги и использовать их при формировании HTML-страниц. Для этого Django позволяет использовать два вида пользовательских тегов:

- simple tags – простые теги;
- inclusion tags – включающие теги.

Для начала создадим в нашем приложении еще один подкаталог с названием `templatetags`. Этот каталог должен быть пакетом, значит в нем создаем файл `__init__.py` и сам файл `cinema_tags`, в котором будем создавать наши теги. Импортируем в него модуль `template` для работы с шаблонами и модуль `views`:

```
1. from django import template
2. import cinema_app.views as views
```

Следующим шагом нам нужно создать экземпляр класса `Library`, через который происходит регистрация собственных шаблонных тегов: `register = template.Library()`

Simple Tags выглядит так:

```
1. @register.simple_tag()
2. def get_categories():
3.     return views.cats_db
```

Он возвращает любые данные, которые мы импортировали из вью представления. Далее нужно передать это тег в базовый шаблон что бы с ним можно было работать. Нужно прописать в нашем шаблоне `base.html` - `{% load cinema_tags %}`. После этого в шаблоне (и во всех его дочерних шаблонах) доступен тег по имени `get_categories`. Если прописать сам тег в шаблоне так `{% get_categories %}`, то перебрать его объекты нельзя. Для этого в Django в тегах можно использовать специальное ключевое слово `as`, которое сформирует ссылку на данные тега. В нашем случае это можно записать так: `{% get_categories as categories %}`. Сформируется переменная `categories`, которую уже можно использовать в теге цикла `for`.

Второй тип пользовательских тегов – включающий тег (Inclusion Tags), он позволяет дополнительно формировать свой собственный шаблон на основе некоторых данных и возвращать фрагмент HTML-страницы.

В нашем проекте он выглядит следующим образом:

```
1. @register.inclusion_tag('cinema_app/list_categories.html',
    takes_context=True)
2. def show_categories(context):
3.     context = {
4.         'cats': views.cats_db,
5.         'cat_selected': context['cat_selected'],
6.     }
7.     return context
```

Внутри тега мы передаем путь к шаблону и параметр `takes_context=True` для того, чтобы все данные, которые передаем с переменного `context` в нашем представлении, мы могли использовать в шаблоне `list_categories.html`. В нашем случае мы используем переменную `cat_selected`, для подсветки выбранной нами категории, через класс `selected` тега ``:

```
1. {% for cat in cats%}
2. {% if cat.id == cat_selected %}
3.     <li class="selected">{{ cat.name }}</li>
4. {% else %}
5.     <li><a href="{% url 'category' cat.id %}">{{ cat.name }}</a></li>
6. {% endif %}
7. {% endfor %}
```

В итоге пользовательские теги позволяют отдельно создавать шаблон для использования конкретного блока задачи. И данный шаблон мы можем вставить в базовый шаблон для его использования с помощью заданного нам тега.

6.4 Введение в ORM и модели

6.4.1 Что такое базы данных, SQL и ORM

База данных — это организованная коллекция данных, которая обычно хранится в электронном виде на компьютере или сервере. Она предназначена для эффективного хранения, управления и доступа к информации. БД имеют структуру в виде таблиц.

Для создания таблиц, описания их структуры и наполнения данными используется специальный язык, который сокращенно называется SQL. Это аббревиатура от фразы: SQL (Structured Query Language) которая переводится как язык структурированных запросов. Именно на нем пишутся запросы к БД. На самом деле SQL используется всегда и, мало того, для каждого типа СУБД имеет свои характерные особенности. Но, чтобы программист мог создавать универсальный программный код, не привязанный к конкретному типу БД, в Django встроен механизм взаимодействия с таблицами через объекты классов языка Python посредством технологии ORM (Object-Relational Mapping). Причем этот интерфейс универсален и на уровне WSGI-приложения не привязан к конкретной СУБД.

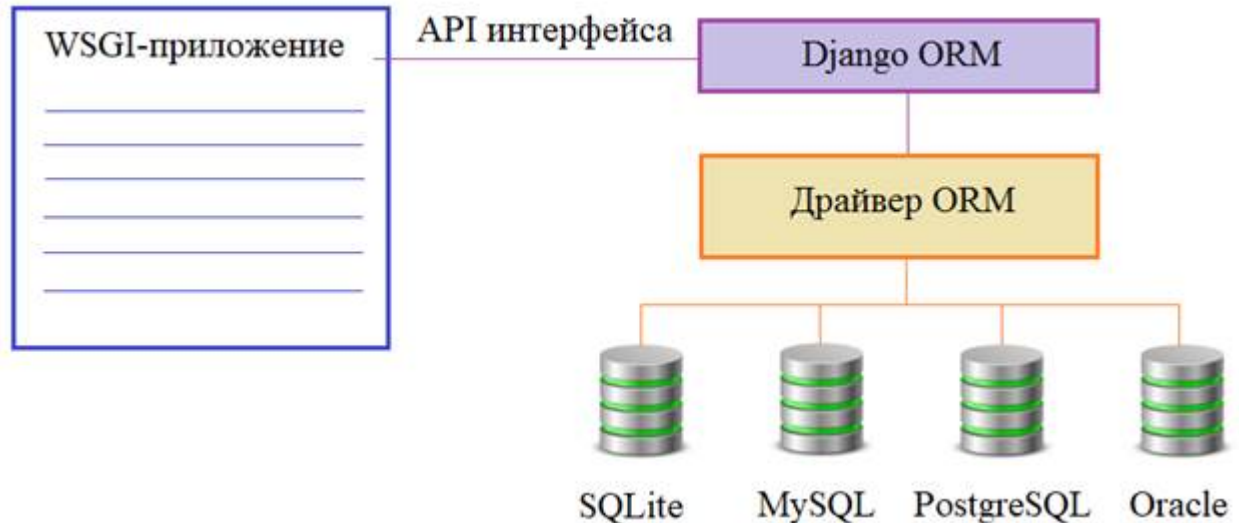


Рисунок 9. Объектная модель ORM в Django

Основные концепции ORM:

1. Отображение объектов: ORM позволяет создавать классы в объектно-ориентированном коде, которые отображают таблицы и их отношения в базе данных. Каждый объект класса соответствует записи в таблице.

2. Автоматическое создание запросов: ORM автоматически генерирует SQL-запросы для выполнения операций, таких как выборка, вставка, обновление и удаление данных из базы данных.
3. Отношения между объектами: ORM позволяет определять и управлять отношениями между объектами, отражая связи между таблицами в базе данных.
4. Объектно-ориентированный код: Программисты работают с объектами и классами, а не с языком SQL и таблицами баз данных, что делает код более понятным и поддерживаемым.

При работе с Django разработчику не нужно беспокоиться о подключении к БД и ее закрытию, когда пользователь покидает сайт. Фреймворк такие действия берет на себя и делает это весьма эффективно. Все что нам остается, это через модель взаимодействия выполнять команды API интерфейса, записывать, считывать и обновлять данные. По умолчанию Django сконфигурирован для работы с БД SQLite.

Модели в Django — это классы Python, которые определяют структуру таблиц базы данных. Каждый атрибут класса соответствует полю таблицы, а экземпляры класса представляют записи в таблице. Django использует ORM (Object-Relational Mapping), чтобы автоматически создавать SQL-запросы для работы с базой данных. Модель отвечает за хранение и оперирование данными сайта. Django поддерживает следующие стандартные СУБД: PostgreSQL, MariaDB, MySQL, Oracle и SQLite. Все эти БД относятся к реляционным, то есть, позволяют хранить данные в виде связанных таблиц. Причем, для каждой таблицы мы сами определяем число и тип столбцов. Эти столбцы называются полями таблицы и определяют ее структуру. А набор конкретных данных полей — записями таблицы.

Рассмотрим следующий пример создания таблицы БД:

```
1. class Cinema(models.Model):
2.     title = models.CharField(max_length=255)
3.     descriptions = models.TextField(blank=True)
4.     time_create = models.DateTimeField(auto_now_add=True)
5.     time_update = models.DateTimeField(auto_now=True)
6.     is_active = models.BooleanField(default=True)
```

Класс Film наследуются от базового класса `models.Model`, именно это наследование превращает наш класс в класс модели. Первое поле `id` не прописывается, Django создаст его автоматически. Далее мы прописываем поля (столбцы), которые будем использовать в таблице, а типы полей — это параметры (характеристики) для записей в данные столбцы.

По умолчанию имя таблицы будет совпадать с именем класса, а ее структура определяться атрибутами, которые мы здесь определили. Далее, атрибут `title` будет определять одноименное поле как текстовую строку с максимальным числом символов 255. Более подробно можно ознакомиться по ссылке: <https://docs.djangoproject.com/en/4.2/ref/models/fields/> - официальная документация Django.

Для того что бы наша таблица создавалась в соответствии с классом `Film` в Django существует механизм, известный как создание и выполнение миграций. Фактически, каждая миграция представляет собой отдельный файл (модуль) с текстом программы на языке Python, где описаны наборы команд на уровне ORM интерфейса, для создания таблиц определенных структур. При выполнении файла миграции в БД автоматически создаются новые или изменяются прежние таблицы, а также связи между ними. Каждый новый файл миграции помещается в папку `migrations` текущего приложения и описывает лишь изменения, которые произошли в структурах таблиц с прошлого раза. Их можно воспринимать как контролеры версий: всегда можно откатиться к предыдущей структуре и продолжить работу с прежней версией структур и связей между таблицами. Выполним следующую команду: `python3 manage.py makemigrations`. Команду `makemigrations` следует выполнять каждый раз, когда у нас меняется хотя бы одна модель, иначе изменения не отразятся в файлах миграций и, как следствие, в самой БД. Создается файл миграции, но он сам по себе не создает таблицу, для этого нужно его применить SQL запрос, который в Django выполняется командой: `python3 manage.py migrate`.

Для работы с БД мы будем использовать программу SQLiteStudio. Откроем ее и увидим, что в нашей БД создались таблицы по мимо нашей `Cinema`. Django создает вспомогательные таблицы.

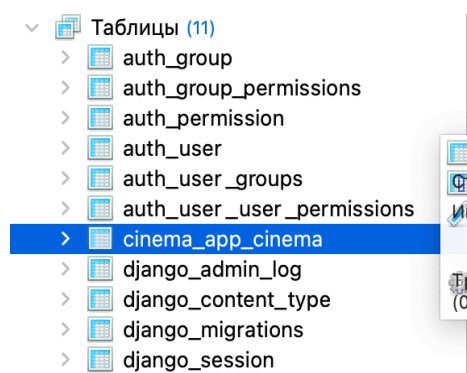


Рисунок 10. Таблицы в БД.

Таблица, которую мы создали называется `cinema_app_cinema` и в ней имеются поля, описанные в нашем классе `Cinema`:

id	title	descriptic	time_crea	time_upd	is_active
----	-------	------------	-----------	----------	-----------

Рисунок 11. Поля в таблицы БД.

6.4.2 Понятие CRUD

В Django работа с данными в моделях осуществляется через объекты моделей, которые представляют отдельные записи в базе данных.

CRUD - акроним, обозначающий четыре базовые функции, используемые при работе с базами данных:

- Create – создание;
- Read – чтение;
- Update – изменение;
- Delete – удаление.

Почти все проекты, построенные на Django, используют его встроенную ORM, не переходя на уровень SQL-запросов. В этом часто просто нет необходимости, так как ORM предоставляет богатейшие возможности по работе с БД. Кроме того, это обеспечивает независимость программного кода от конкретной используемой СУБД и если в будущем потребуется изменить тип БД, то сделать это будет предельно просто. Наконец, ORM в Django хорошо оптимизирует запросы по скорости выполнения и частоте обращения к таблицам БД, а также обеспечивает защиту от SQL-инъекций. Благодаря этому, даже начинающий веб-программист сможет создавать грамотный код по работе с БД.

Create – используется для добавления в таблицу данных. Существуют разные способы добавления данных:

1. Создание и сохранение экземпляров модели:

```
1. from myapp.models import MyModel
2. obj = Cinema(title='Фильм 1', descriptions='Описание фильма 1')
3. obj.save()
```

2. Массовое добавление данных с использованием метода bulk_create():

```
1. Cinema.objects.bulk_create([
2.     Cinema(title='Фильм 1', 'Описание фильма 1'),
3.     Cinema(title='Фильм 2', 'Описание фильма 2'),
4.     ...
5. ])
```

3. Использование метода create() менеджера объектов:

Метод create() позволяет создать и сохранить экземпляр модели в одном вызове.

```
1. Cinema.objects.create(title = 'Фильм 1', 'Описание фильма 1')
```

Read - менеджер объектов предоставляет следующие методы для доступа к объектам в базе данных:

1. `all()`:

Метод `all()` возвращает все объекты модели в виде `QuerySet` (коллекция объектов из базы данных): `all_objects = Cinema.objects.all()`.

2. `filter()`:

Метод `filter()` возвращает набор объектов, отфильтрованных по указанным условиям: `filtered_objects = Cinema.objects.filter(field1=value1, field2=value2)`.

3. `exclude()`:

Метод `exclude()` возвращает набор объектов, исключая те, которые соответствуют указанным условиям: `excluded_objects = Cinema.objects.exclude(field1=value1)`.

4. `get()`:

Метод `get()` возвращает один объект, соответствующий указанным условиям. Если найдено более одного объекта или ни одного объекта, сработает исключение `MultipleObjectsReturned` или `DoesNotExist` соответственно: `my_object = Cinema.objects.get(pk=1)`.

5. `order_by()`:

Используется для сортировки результатов запроса к базе данных по заданному или нескольким полям: `MyModel.objects.order_by('title')`.

Update – изменение. Метод `update()` позволяет обновить значения полей для всех объектов, удовлетворяющих определенным условиям, без необходимости извлекать их из базы данных: `Cinema.objects.filter(condition).update(field1=new_value1, field2=new_value2)`

Delete - в Django можно удалить отдельные объекты модели, вызвав метод `delete()` на объекте модели: `my_object = Cinema.objects.get(pk=1); my_object.delete()`

6.4.3 Слагги (slug) в URL-адресах. Метод `get_absolute_url()`

Слагги (slug) — это короткие текстовые метки, обычно используемые в веб-разработке для создания читабельных и оптимизированных для поисковых систем URL-адресов. Они обычно формируются путем преобразования заголовков или других текстовых данных в URL-дружественный формат.

Преимущества использования слаггов:

1. Улучшение SEO: Слагги позволяют включать ключевые слова в URL, что может помочь улучшить SEO и увеличить видимость контента в поисковых системах.
2. Читаемость URL: Слагги делают URL-адреса более читаемыми для людей, поскольку они обычно содержат понятные слова или фразы, отражающие содержание страницы.
3. Уникальность и однозначность: Слагги обычно создаются на основе заголовков или других уникальных идентификаторов, что гарантирует их уникальность и однозначность.
4. Удобство использования в ссылках и социальных сетях**: Слагги удобны для передачи и обмена ссылками в социальных сетях и других интернет-ресурсах.
5. Безопасность и защита от атак: Использование слаггов помогает защитить сайт от некоторых видов атак, таких как атаки инъекцией или кросс-сайтовый скриптинг, так как слагги могут быть легко проверены и очищены от потенциально вредоносного кода.

Для того что бы мы переходили по ссылке по слаггу, мы должны каждую запись связать с полем slug, то есть создадим еще одно поле в модели Cinema и назовем его slug. Пропишем метод `save()` для автоматического формирования записи в это поле при создании новой записи:

```
1. class Cinema(models.Model):
2.     title = models.CharField(max_length=255)
3.     slug = models.SlugField(max_length=255, unique=True, db_index=True)
4.     ...
5.
6. def save(self, *args, **kwargs):
7.     if not self.slug:
8.         self.slug = slugify(unidecode(str(self.title)))
9.     super().save(*args, **kwargs)
```

Теперь мы можем прописать в файле `urls.py` формирование пути следующим образом: `path('film/<slug:film_slug>/', views.show_film, name='film')`. Далее для правильных формирований ссылок нужно определить метод `get_absolute_url()` в классе Cinema:

```
1. def get_absolute_url(self):  
2.     return reverse('film', kwargs={'film_slug': self.slug})
```

Теперь в шаблоне можно прописать так: `<p>Читать описание</p>`, и он будет формировать правильный путь по слаг. Этот пример показывает, как в Django легко и просто можно менять URL-адреса и вместо id использовать другие поля, в частности, слаг. При этом в шаблоне мы использовали метод `get_absolute_url()` модели `Cinema` для формирования корректного URL-адреса. Кроме того, Django автоматически защищает такие адреса от SQL-инъекций, когда злоумышленник пытается выполнить SQL-запрос, прописывая его в адресной строке браузера.

6.5 Связи между таблицами

6.5.1 Типы связей между моделями

Разберемся с разными типами связей, которые можно устанавливать между таблицами БД. Очевидно, что главное преимущество реляционных баз данных заключается в способности задавать отношения между таблицами. Этот процесс получил название нормализации данных и составляет отдельный раздел знаний по базам данных. Django предоставляет возможности для определения трех наиболее распространенных типов отношений:

1. `ForeignKey` – один ко многим
2. `ManyToManyField` -многие ко многим
3. `OneToOneField` – один к одному

6.5.1.1 ForeignKey – один ко многим

Один ко многим – ForeignKey. Тип связи "Один ко многим" (или ForeignKey в Django) представляет отношение, при котором каждый объект одной модели связан с одним объектом другой модели, но объекты второй модели могут быть связаны с несколькими объектами первой модели.

Пример с двумя моделями: Customers и Orders.

У каждого пользователя может быть несколько заказов, но каждый заказ принадлежит только одному пользователю.

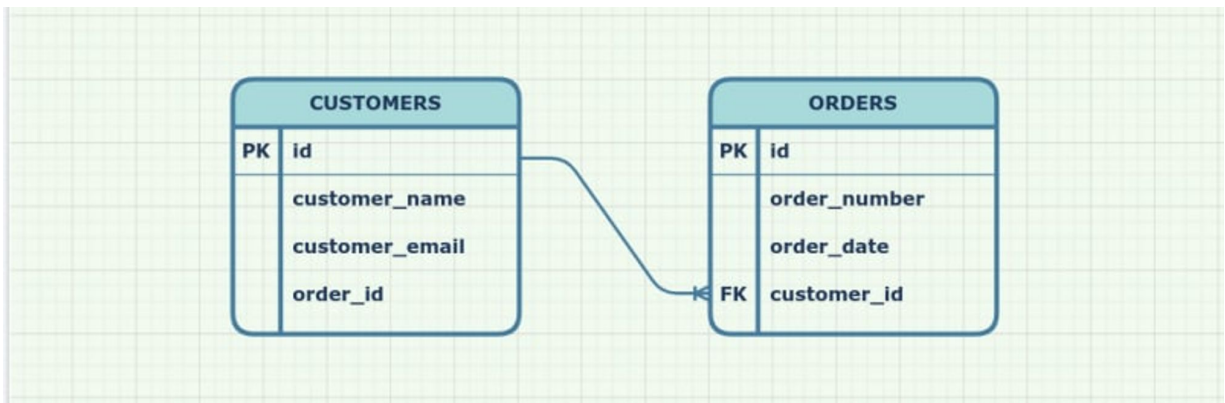


Рисунок 12. Один ко многим – ForeignKey

Реализация:

```
1. from django.db import models
2.
3. class Customers(models.Model):
4.     name = models.CharField(max_length=100)
5.
6. class Orders(models.Model):
7.     customer = models.ForeignKey(User, on_delete=models.CASCADE)
```

- В модели Orders определено поле ForeignKey, которое связывает каждый заказ с объектом пользователя. Это поле указывает на модель Customers и определяет тип связи.
- Параметр `on_delete=models.CASCADE` указывает, что при удалении пользователя все связанные с ним заказы также должны быть удалены.
- За кулисами Django добавляет "_id" к имени поля, чтобы создать имя столбца базы данных. В данном примере поле ForeignKey создает в базе данных столбец с именем `customer_id`, который содержит идентификатор пользователя, связанного с каждым заказом.

Преимущества:

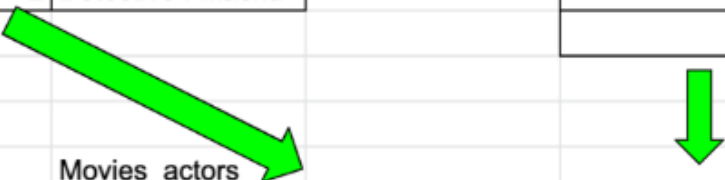
- Позволяет устанавливать однозначное отношение между объектами разных моделей.
- Обеспечивает простоту и эффективность доступа к связанным объектам.
- Упрощает структурирование данных и выполнение запросов к базе данных.

6.5.1.2 ManyToMany - многие ко многим

Тип связи "Многие ко многим" (или `ManyToManyField` в Django) представляет отношение, при котором каждый объект одной модели может быть связан с несколькими объектами другой модели, и наоборот.

Пример с двумя моделями: `Movie` и `Actor`:

У каждого фильма может быть несколько актеров, и каждый актер может участвовать в нескольких фильмах.



Movies Table		Actors Table	
ID	Movie Title	ID	Name
1	Deadpool	1	Ryan Reynolds
2	Detective Pikachu	2	Karan Soni
		3	Morena Baccarin

Movies_actors		
ID	movies_id	actors_id
1	1	1
2	2	1
3	1	3
4	1	2
5	2	2

Рисунок 13. Многие ко многим - `ManyToManyField`

Реализация:

```
1. from django.db import models
2.
3. class Movie(models.Model):
4.     movie_title = models.CharField(max_length=100)
5.     actors = models.ManyToManyField('Actor')
6.
7. class Actor(models.Model):
8.     name = models.CharField(max_length=100)
```

- В модели `Movie` определено поле `ManyToManyField`, которое связывает каждый фильм с несколькими актерами (ссылается на модель 'Actor')
- Взаимосвязь между моделями `Movie` и `Actor` обрабатывается в Django автоматически с использованием промежуточной таблицы в базе данных, которая отслеживает связи между объектами двух моделей.

Преимущества:

- Позволяет описывать сложные отношения между объектами различных моделей.
- Обеспечивает гибкость при работе с данными, так как позволяет добавлять и удалять связанные объекты без изменения схемы базы данных.
- Упрощает выполнение запросов, связанных с данными, которые имеют многие ко многим отношения.

6.5.1.3 OneToOne – один к одному

Тип связи "Один к одному" (или `OneToOneField` в Django) представляет отношение, при котором каждый объект одной модели связан только с одним объектом другой модели, и наоборот.

Пример с двумя моделями: `UserProfile` и `User`:

Каждый пользователь может иметь только один профиль, и каждый профиль связан только с одним пользователем.

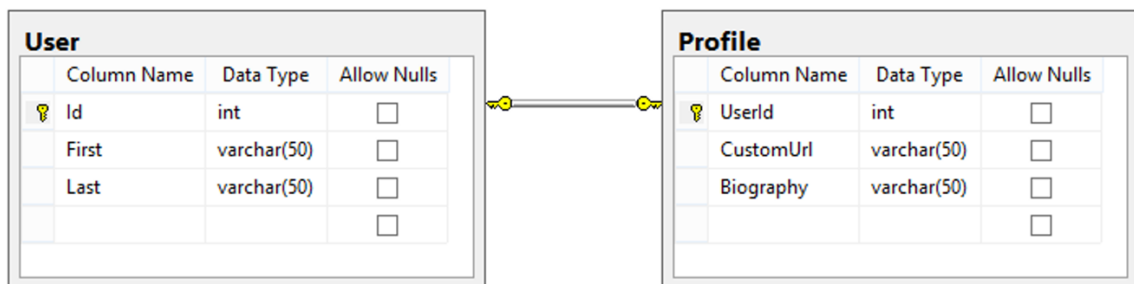


Рисунок 14. Один к одному – `OneToOneField`

Реализация:

```
1. from django.db import models
2. from django.contrib.auth.models import User
3.
4. class UserProfile(models.Model):
5.     user = models.OneToOneField(User, on_delete=models.CASCADE)
6.     custom_url = models.CharField(max_length=50, blank=True)
7.     biography = models.CharField(max_length=50, blank=True)
```

- В модели `UserProfile` определено поле `OneToOneField`, которое связывает каждый профиль с одним пользователем из модели `User`.
- `on_delete=models.CASCADE` указывает, что при удалении пользователя также будет удален его профиль. Это может быть изменено в зависимости от требований приложения.
- Связь "Один к одному" обычно используется для создания дополнительных профилей или расширенной информации о существующих пользователях.

Преимущества:

- Гарантирует наличие только одного связанного объекта для каждого объекта модели.
- Упрощает доступ к дополнительным данным о сущностях, таких как профили пользователей или дополнительные сведения о продуктах.
- Обеспечивает эффективное использование базы данных, поскольку связь описывается через уникальные ключи и индексы.

6.5.2 Создание связи many-to-one

Тип связи "Один ко многим" (или `ForeignKey` в Django) представляет отношение, при котором каждый объект одной модели связан с одним объектом другой модели, но объекты второй модели могут быть связаны с несколькими объектами первой модели.

При определении связи `ForeignKey` Django добавляет внутренний ключ (foreign key) к родительской модели в дочернюю модель. Затем Django автоматически создает поле в дочерней таблице базы данных, добавляя к имени поля `"_id"`. Это поле будет содержать идентификатор родительской записи. А также формирует связи между таблицами на уровне базы данных, обеспечивая согласованность данных в соответствии с определением моделей.

Класс `ForeignKey` принимает два обязательных аргумента:

- `to` – ссылка или строка класса первичной модели, с которой происходит связывание (в нашем случае это класс `Category` – модели для категорий);
- `on_delete` – определяет тип ограничения при удалении внешней записи (в нашем примере – это удаление из первичной таблицы `Category`).

В свою очередь, опция `on_delete` может принимать следующие значения:

- `models.CASCADE` – удаление всех записей из вторичной модели (например, `Cinema`), связанных с удаляемой категорией;
- `models.PROTECT` – запрещает удаление записи из первичной модели, если она используется во вторичной (выдает исключение);
- `models.SET_NULL` – при удалении записи первичной модели (`Category`) устанавливает значение foreign key в `NULL` у соответствующих записей вторичной модели (`Cinema`);
- `models.SET_DEFAULT` – то же самое, что и `SET_NULL`, только вместо значения `NULL` устанавливает значение по умолчанию;
- `models.SET()` – то же самое, только устанавливает пользовательское значение;
- `models.DO_NOTHING` – удаление записи в первичной модели (`Category`) не вызывает никаких действий у вторичных моделей.

В нашем случае свяжем таблицу с фильмами и категориями. Для этого создадим таблицу с категориями, код будет выглядеть так:

```
1. class Cinema(models.Model):
2.     ...
3.     cat = models.ForeignKey('Category', on_delete=models.PROTECT,
                             related_name='cinema')
```

```

4. class Category(models.Model):
5.     name = models.CharField(max_length=150, db_index=True)
6.     slug = models.SlugField(max_length=250, db_index=True, unique=True)

```

Для обратной связи моделей Category и Cinema, можно использовать конкретное название, которое мы сами можем задать в параметре `related_name`, его обычно называют – менеджер записей для обратного связывания. В данном случае `'cinema'`.

После создания таблицы с категориями можно привязать категории к фильмам и выводить фильмы по категориям (сделаем своеобразный фильтр). Для этого опишем нашу функцию представления `show_category()` следующим образом:

```

1. def show_category(request, cat_slug):
2.     cat = get_object_or_404(Category, slug=cat_slug)
3.     films = Cinema.active.filter(cat_id=cat.pk)
4.     context = {
5.         'title': f'Категория - {cat.name}',
6.         'menu': menu,
7.         'films': films,
8.         'cat_selected': cat.pk,
9.     }
10.    return render(request, 'cinema_app/index.html', context=context)

```

Переходя по ссылке `{{ cat.name }}`, которая формируется с помощью знакомого метода `get_absolute_url()` мы тем самым передаем наш слаг в url маршрут: `path('category/<slug:cat_slug>', views.show_category, name='category')`. Тот в свою очередь обрабатывает функцию представления `show_category()`, в которую передает слаг, и исходя из этого слага, мы получаем категорию с помощью метода `get_object_or_404()`, который в случае если категория не будет найдена выдаст ошибку 404. Далее фильтруем наши фильмы по присвоенным категориям и передаем их в наше представление `index.html`. Тем самым мы выберем конкретные фильмы по категории.

6.5.3 Создаем связь many-to-many

Тип связи "Многие ко многим" (или `ManyToManyField` в Django) представляет отношение, при котором каждый объект одной модели может быть связан с несколькими объектами другой модели, и наоборот.

Синтаксис: `ManyToManyField(to, **options)`. `ManyToManyField` требует один обязательный аргумент: класс, с которым связана модель - `to`. Неважно, в какой модели находится поле `ManyToManyField`, но вы должны поместить его только в одну из моделей - не в обе. Обычно ее помещают в ту модель, которая в дальнейшем будет редактироваться в форме.

Для создания связи опишем новый класс `GenreFilm` (жанры фильмов) и добавим поле для связи в класс `Cinema`:

```
1. class Cinema(models.Model):
2.     ...
3.     genres = models.ManyToManyField('GenreFilm', related_name='cinema_genres')
4.
5. class GenreFilm(models.Model):
6.     genre = models.CharField(max_length=150, db_index=True)
7.     slug = models.SlugField(max_length=250, db_index=True, unique=True)
```

После применения миграции создается таблица `cinema_app_genrefilm` и промежуточная таблица `cinema_app_cinema_genre`, через которую реализуется связь Many-To-Many. Если мы хотим задать имя промежуточной таблицы вручную, мы можем использовать параметр `db_table`. Это позволит контролировать имя таблицы и может быть полезным, если мы хотим сделать его более описательным или если у нас есть какие-то особые требования к именам таблиц в вашей базе данных.

Следует заметить, что в таблицах не создается дополнительное поле, поля для связи будут реализованы в промежуточной таблице `cinema_app_cinema_genre`:

id	cinema_id	genrefilm_id
1	1	8
2	1	2
3	1	3
4	1	4
5	2	1
6	2	5
7	2	6
8	2	7
9	3	6
10	3	9
11	3	10
12	3	11

Рисунок 15. Таблица many-to-many.

В поле `cinema_id` создаются номера, связанные с таблицей `cinema_app_cinema`, в поле `genrefilm_id` создаются связи (номера) с таблицей `cinema_app_gengrefilm`, а вот поля `cinema_id` и `genrefilm_id` имеют связь между собой, что характерно для `ManyToManyField`.

Таким образом мы имеем фильм, связанный с разными жанрами и жанры, связанные с разными фильмами.

Заполним таблицу с жанрами и добавим в наше представление функцию `show_genre()` для отображения всех жанров на странице:

```
1. def show_genre(request, genre_slug):
2.     genre = get_object_or_404(GenreFilm, slug=genre_slug)
3.     films = genre.cinema_genres.filter(is_active=Cinema.Status.ACTIVE)
4.     context = {
5.         'title': f'Категория - {genre.genre}',
6.         'menu': menu,
7.         'films': films,
8.         'cat_selected': None,
9.     }
10.    return render(request, 'cinema_app/index.html', context=context)
```

6.5.4 Агрегирование и группировка

Для агрегирования данных из БД можно использовать метод `annotate()`. При использовании `annotate()` с агрегирующей функцией, например, `count()`, Django автоматически генерирует SQL запрос с оператором `GROUP BY`, чтобы сгруппировать данные по полям. Данную конструкцию можно использовать в нашем проекте, например для вычисления количества привязанных категорий к нашим фильмам и выводить категорию если она привязана хоть к одному фильму и также выводить жанр если он имеет связь с фильмом. Это можно сделать следующим образом – в файле `cinema_tags.py` изменим запрос к БД: `cats = Category.objects.annotate(total=Count('cinema')).filter(total__gt=0)` – меняем запрос на вывод категорий, `genres = GenreFilm.objects.annotate(total=Count('cinema_genres')).filter(total__gt=0)` – меняем запрос на вывод жанров.

Можно использовать не только функцию `count()`, но в нашем случае она очень подходит для проекта. Существует еще агрегирующие функции такие как `SUM`, `AVG`, `MIN`, `MAX` и другие.

6.6 Оптимизация с Django Debug Toolbar

Django Debug Toolbar (DjDT) - это инструмент, который предназначен для упрощения процесса отладки при разработке веб-приложений на основе фреймворка Django. Этот инструмент добавляет визуальную панель отладки, которая позволяет разработчикам получать полезную информацию о запросах, базе данных, производительности и других аспектах приложения прямо в браузере.

Некоторые ключевые возможности:

- Информация о запросах: DDT показывает список всех SQL-запросов, выполняемых во время запроса страницы, а также время, затраченное на каждый запрос.
- Профилирование производительности: позволяет анализировать производительность каждого запроса, включая время выполнения представления и время, затраченное на выполнение различных частей запроса.
- Информация о шаблонах: для каждой загружаемой страницы показывается список всех шаблонов Django, которые были использованы для её формирования, и время, затраченное на обработку каждого шаблона.
- Панель отладки запросов: отображает информацию о текущем запросе, параметры запроса, заголовки и другие детали.
- Информация о сессиях и пользователя: позволяет просматривать текущие сессии и информацию о пользователях, включая их атрибуты и данные.
- Отладка кэша: показывает информацию о кэшировании, включая количество кэш-промахов и время, затраченное на кэширование.
- Пользовательские панели: разработчики могут создавать собственные панели для отображения дополнительной отладочной информации.

Для установки выполним команду `pip3 install django-debug-toolbar`.

Проверяем наличие необходимых настроек в нашем `settings.py`:

```
1. INSTALLED_APPS = [  
2.     ...  
3.     "django.contrib.staticfiles",  
4.     ...  
5. ]  
6.  
7. STATIC_URL = "static/"  
8.
```

```

9. TEMPLATES = [
10.     {
11.         "BACKEND": "django.template.backends.django.DjangoTemplates",
12.         "APP_DIRS": True,
13.         ...
14.     }
15. ]

```

В том же файле, добавляем необходимые конфигурации для toolbar-a:

```

1. INSTALLED_APPS = [
2.     ...
3.     'debug_toolbar',
4.     ...
5. ]
6.
7. MIDDLEWARE = [
8.     ...
9.     'debug_toolbar.middleware.DebugToolbarMiddleware',
10.    ...
11. ]
12.
13. INTERNAL_IPS = ['127.0.0.1',]

```

Добавляем маршрут до toolbar в маршруты проекта:

```

1. urlpatterns = [ ... path('__debug__/', include('debug_toolbar.urls')), ]

```

При запуске сервера появиться иконка DjDT, нажав на нее мы можем увидеть список различных разделов. В нашем проекте нас интересует оптимизация SQL запросов. Выбрав это раздел мы видим, что у нас есть два повторяющихся запроса в шаблоне выводющий список категорий, а именно название категорий.

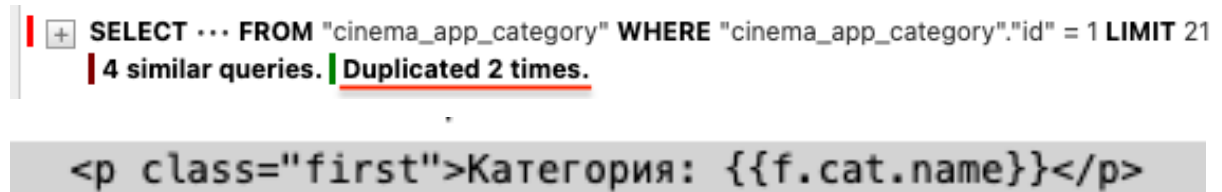


Рисунок 16. Оптимизация SQL запроса.

Дублирующие запросы к БД приводят к низкой скорости работы нашего сайта, поэтому нужно от них избавиться путем жадной загрузки данных из БД.

Для этого в Django имеются два полезных метода:

- `select_related(key)` – «жадная» загрузка связанных данных по внешнему ключу `key`, который имеет тип `ForeignKey`;
- `prefetch_related(key)` – «жадная» загрузка связанных данных по внешнему ключу `key`, который имеет тип `ManyToManyField`.

Так как мы используем `ForeignKey` для связи таблицы `Cinema` с таблицей `Category`, то нам нужен метод `select_related(key)`. Пропишем код в файле `views.py` нашего приложения `cinema_app` в функции представления `index()`: `films = Cinema.active.all().select_related('cat')`. Тем самым мы сразу загружаем все связанные данные из таблицы `Category` в переменную `films` и используем эти данные в шаблоне исключая дублирующий запрос к БД.

6.7 Работа с админ-панелью

Admin-панель в Django представляет собой встроенный интерфейс, который обеспечивает административный доступ к данным веб-приложения. Этот инструмент позволяет администраторам управлять содержимым сайта без необходимости написания дополнительного кода для создания пользовательского интерфейса. Она предоставляет возможность создавать, редактировать и удалять записи в базе данных, а также осуществлять множество других действий, связанных с управлением приложением.

Зачем нужна административная панель? Во-первых, она значительно упрощает работу разработчиков, позволяя им быстро и удобно управлять данными. Во-вторых, административная панель может быть использована в качестве удобного инструмента для администрирования приложения, что позволяет упростить жизнь не только разработчикам, но и другим пользователям приложения. В этой лекции мы рассмотрим основы работы с административной панелью в Django, настроим ее и создадим пользовательские модели. Также мы рассмотрим примеры использования административной панели и обсудим основные моменты ее работы.

6.7.1 Создание суперпользователя. Регистрация моделей.

Для входа нам нужно ввести имя пользователя и пароль. Но у нас пока их нет и вообще нет ни одного зарегистрированного пользователя в админ-панели. Поэтому, сначала необходимо создать суперпользователя – администратора сайта. Для этого я перейду в терминал и в новой вкладке консоли выполню команду: `python3 manage.py createsuperuser`.

Далее регистрируем нашего суперпользователя: вводим имя, почту и пароль.

В админ-панели можно отображать наши приложения с прописанными в них классами (таблицами), для этого нужно добавить наши приложения. Для этого в приложении в каждом приложении, в данном случае `cinema_app`, в файле `admin.py` нужно импортировать класс и зарегистрировать его:

```
1. from django.contrib import admin
2. from .models import Cinema
3.
4. admin.site.register(Cinema)
```

Мы можем поменять отображение имени в админ-панели для класса `Cinema`, для этого в данном классе мы уже обозначили класс `Meta` и именно в него внесем изменения:



Рисунок 17. Отображение класса

```
1. class Meta:
2.     verbose_name = 'ФИЛЬМЫ'
3.     verbose_name_plural = 'ФИЛЬМЫ'
```



Рисунок 18. Отображение класса

Так же мы можем поменять отображения нашего имени приложения.



Для этого нужно в файле `apps.py` прописать так:

Рисунок 19. Отображение имени приложения

```
1. class CinemaAppConfig(AppConfig):  
2.     verbose_name = 'Приложение с фильмами'
```



Рисунок 20. Отображение имени приложения

6.7.2 Настройка отображения списка из БД в админ-панели. Атрибуты

Для более подробного отображения списка мы можем использовать Администратор модели.

Администратор модели (ModelAdmin) — это класс, который определяет, как модель будет отображаться и какие действия можно выполнять с ней в административной панели Django. Он позволяет настраивать внешний вид и поведение модели в административной панели.

В предыдущем пункте мы уже описывали один способ создания администратора модели вместе с моделью через метод `register()`. Это также можно сделать через декоратор и добавить нужные выводимые нам столбцы из модели Cinema, и столбец по которому можно получать более подробную информацию о записи:

```
1. @admin.register(Cinema)
2. class CinemaAdmin(admin.ModelAdmin):
3.     list_display = ['id', 'title', 'time_create', 'is_active']
4.     list_display_links = ['title']
```

Атрибуты, которые мы можем настроить внутри класса администратора:

1. `list_display` - список полей, которые отображаются на странице списка объектов модели в административной панели. `list_display = ('field1', 'field2', 'field3')`
2. `list_display_links` - определяет, какое поле или поля будут использоваться в качестве ссылок для редактирования объектов на странице списка. Если вы установите `list_display_links`, поля, указанные в этом параметре, будут представлять собой активные ссылки на страницу редактирования объекта. `list_display_links = ('field1',)`
3. `ordering` - порядок сортировки объектов на странице списка. `ordering = ('-field1', 'field2')`
4. `list_editable` - список полей, которые можно редактировать прямо на странице списка. `list_editable = ('field1', 'field2')`
5. `list_per_page` - количество объектов, отображаемых на одной странице списка. Этот параметр позволяет настроить количество объектов, которые будут отображаться на странице списка, что может быть полезно при работе с большими объемами данных. `list_per_page = 50`
6. `list_filter` - список полей, по которым можно фильтровать объекты на странице списка. `list_filter = ('field1', 'field2')`.

Это только основные атрибуты. Полный список можно посмотреть в документации.

6.7.3 Поиск и фильтрация объектов

Django Admin по умолчанию предоставляет панель поиска - `search_fields`, которая позволяет искать по всем полям модели.

Django разбивает поисковый запрос на слова и возвращает все объекты, содержащие каждое из слов, без учета регистра - `icontains`, где каждое слово должно быть хотя бы в одном из полей `search_fields`.

Если мы не хотим использовать `icontains` в качестве настройки по умолчанию, мы можем использовать любой `lookup`, добавив его к полю. Например, можно использовать `exact`, установив `search_fields` в `['first_name__exact']`.

Поиск по связанным моделям. Мы также можем искать по связанным полям, используя `__`: `search_fields = ["cat__name"]`.

В нашем случае дополнительный поиск можно описать так:

```
1. @admin.register(Cinema)
2. class CinemaAdmin(admin.ModelAdmin):
3.     ...
4.     search_fields = ['title', 'cat__name']
5.     ...
```

Для добавления фильтрации мы можем использовать атрибут `list_filter`. Этот атрибут принимает коллекцию имен полей, по которым будет доступна фильтрация. Сделаем фильтрацию по полю категории и активным фильмам: `list_filter = ['cat__name', 'is_active']`. В админ-панели появится список, по которому мы можем делать отбор:

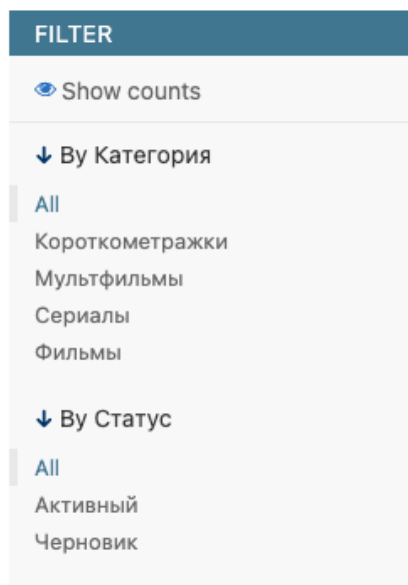


Рисунок 21. Настраиваемый фильтр в админ-панели

6.7.4 Настройка формы редактирования записей

Если перейти на редактирование какой-либо записи, то фреймворк Django автоматически сформирует страницу с набором всех редактируемых полей, которые мы с вами определяли для модели Cinema. Мы здесь не видим поля `time_create` и `time_update`, потому что значения их формируются автоматически.

Набор и свойства отображаемых полей можно настраивать. Например, в классе `CinemaAdmin` можно определить атрибут `fields` со списком отображаемых на форме полей:

```
1. @admin.register(Cinema)
2. class CinemaAdmin(admin.ModelAdmin):
3.     ...
4.     fields = ['title', 'descriptions', 'year_production', 'cat', 'slug']
5.     ...
```

Поля будут отображаться в том порядке, в котором они прописаны в атрибуте `fields`. В данном атрибуте необходимо указать все поля, которые имеют обязательное заполнение, так как при редактировании или добавлении новой записи Django выдаст ошибку.

Чтобы отобразить несколько полей в одной строке, объедините их в кортеж.

```
1. @admin.register(Cinema)
2. class CinemaAdmin(admin.ModelAdmin):
3.     ...
4.     fields = (('title', 'descriptions'), 'year_production', 'cat', 'slug')
5.     ...
```

Атрибут `exclude` позволяет указать, какие поля модели не должны быть включены в формы добавления ("add") и редактирования ("change") в админ-панели, другими словами, исключает указанные поля из этих форм. При перечислении одних и тех же полей в `fields` и `exclude` - будет вызвана ошибка.

Далее, мы можем сделать поля только для чтения. Для этого используется атрибут `readonly_fields`. Допустим укажем что поле слаг не редактируемое:

```
1. @admin.register(Cinema)
2. class CinemaAdmin(admin.ModelAdmin):
3.     ...
4.     readonly_fields = ['slug']
```

Если у нас есть поле в классе которое имеет связь ForeignKey, то мы можем вывести его и применить к нему настройку вида выборки параметров с помощью атрибута `filter_horizontal` или `filter_vertical`.

```
1. @admin.register(Cinema)
2. class CinemaAdmin(admin.ModelAdmin):
3.     ...
4.     filter_horizontal = ['genres']
5.     ...
```


6.8 Работа с формами

6.8.1 Что такое HTML-формы. Отправка данных через GET и POST

HTML-форма – это раздел веб-страницы, предназначенный для ввода и отправки данных пользователем. Она представляет собой контейнер, в котором размещаются различные элементы управления, такие как:

- Поля ввода текста: для ввода имени, адреса электронной почты, пароля и т.д.
- Текстовые области: для ввода более длинных текстов, например, комментариев или вопросов.
- Флажки: для выбора одного или нескольких вариантов из списка.
- Радиокнопки: для выбора одного варианта из группы.
- Выпадающие списки: для выбора одного варианта из списка.
- Кнопки: для отправки формы, сброса значений или выполнения других действий.

Как работает HTML-форма

- Пользователь заполняет поля формы, вводя необходимые данные.
- При нажатии кнопки "Отправить" данные формы отправляются на сервер.
- Сервер обрабатывает данные и выполняет необходимые действия, например, сохраняет информацию в базе данных, отправляет письмо или перенаправляет пользователя на другую страницу.

Теги HTML-форм:

- `<form>` Самый главный тег. Используется для создания HTML-формы, в которую пользователь вводит данные.
- `<input>` Многоцелевой тег, наиболее часто используемый для создания полей ввода.

Может иметь различные типы для задания видов вводимой информации:

- `type="text"`: простое текстовое поле
- `type="password"`: поле ввода пароля (с маскировкой)
- `type="email"`: поле для электронного адреса
- `type="checkbox"`: флажок (опция, которую можно включить или отключить)
- `type="radio"`: одна из группы радиокнопок (где возможен выбор только одного варианта)

- ...и многие другие
- `<label>` Связывает текстовую метку с определенным элементом формы (полем ввода, кнопкой и т.д.). Улучшает доступность формы.
- `<button>` Создает кликабельную кнопку в форме. Чаще всего используется для отправки формы (`type="submit"`) или для других действий.

Пример тегов в HTML-файле:

```
1. <form action="" method="post">
2.   <h2>Форма обратной связи</h2>
3.   <p>
4.     <label for="name">Ваше имя:</label>
5.     <input type="text" id="name" name="name" required>
6.   </p>
7.   <p>
8.     <label for="email">Ваш email:</label>
9.     <input type="email" id="email" name="email" required>
10.  </p>
11.  <p>
12.    <label for="comment">Ваш комментарий:</label>
13.    <textarea id="comment" name="comment" rows="5" required></textarea>
14.  </p>
15.  <p>
16.    <button type="submit">Отправить</button>
17.  </p>
18. </form>
```

Форма в браузере будет выглядеть так:

Рисунок 22. Вид формы в браузере

GET и POST запросы. HTTP-методы GET и POST являются основными для работы с формами.

Форма входа в Django возвращается с использованием метода 'POST'. В этом случае браузер упаковывает данные формы, кодирует их для передачи, отправляет на сервер и получает ответ.

Метод 'GET', напротив, включает отправленные данные в строку и использует ее для формирования URL-адреса. Таким образом, URL содержит адрес, куда должны быть отправлены данные, а также ключи и значения данных. Это можно увидеть, например, при поиске в документации Django, когда URL будет выглядеть как: <https://docs.djangoproject.com/search/?q=forms&release=1>.

Методы GET и POST обычно используются для разных целей:

- Любые запросы, которые могут изменить состояние системы (например, изменения в базе данных), должны использовать 'POST'. 'GET' следует использовать только для запросов, которые не влияют на состояние системы.
- 'GET' не подходит для формы пароля, потому что пароль будет отображаться в URL, а значит, и в истории браузера, и в логах сервера в открытом виде. Он также не подходит для больших объемов данных или бинарных данных, таких как изображения.
- Использование 'GET' для форм администратора является риском безопасности, так как злоумышленник может легко имитировать запрос формы, чтобы получить доступ к чувствительным частям системы. 'POST' в сочетании с другими защитами, такими как защита CSRF в Django, предоставляет больший контроль над доступом.
- С другой стороны, 'GET' подходит для таких вещей, как веб-форма поиска, потому что URL-адреса, представляющие запрос 'GET', можно легко добавлять в закладки, делиться ими или повторно отправлять.

Хотелось бы рассказать немного об CSRF токене. CSRF (Cross-Site Request Forgery) — это тип атаки на веб-приложения, при которой злоумышленник вынуждает пользователя выполнить действие на веб-сайте, на котором этот пользователь авторизован.

Процесс атаки обычно выглядит следующим образом:

1. Злоумышленник создает поддельный сайт или внедряет зловерный код на легитимный сайт.
2. Пользователь, уже авторизованный на целевом веб-сайте, посещает этот поддельный или компрометированный сайт.
3. На фоне пользователь может не заметить, что в его браузере отправляются запросы к целевому веб-сайту (через скрытые формы, JavaScript или другие методы), которые

выполняют нежелательные операции, такие как изменение пароля, отправка сообщения, выполнение финансовых операций и т. д.

Django предоставляет встроенную защиту от CSRF с помощью middleware и шаблонных тегов.

1. Генерируется случайное секретное значение CSRF-cookie, которое отправляется в ответе сервера.
2. Во все исходящие POST-формы добавляется скрытое поле "csrfmiddlewaretoken" со специальным значением.
3. При всех POST, PUT, DELETE запросах проверяется наличие и правильность CSRF-токена.
4. Также проверяется заголовок Origin или Referer для защиты от межсубдоменных атак.

Для того, чтобы использовать эту функциональность “из коробки”, добавляем тег `{% csrf_token %}` в наш html-шаблон, например:

```
1. <form action="/your-url/" method="post">
2.     {% csrf_token %}
3.     ...
4.     <input type="submit" value="Отправить">
5. </form>
```

Этот код создает форму с методом POST и добавляет скрытое поле с CSRF токеном. При отправке формы на сервер, CSRF токен будет автоматически включен в запрос, и Django сравнит его с CSRF токеном, хранящимся в куки. Если они совпадают, запрос будет считаться доверенным, и Django примет его.

6.8.2 Класс Form. Атрибуты, классы полей формы.

В Django, класс Form представляет собой специальный тип класса, который используется для создания и обработки HTML форм.

Классы Form позволяют определить поля и их характеристики, такие как тип данных, метки, виджеты (отображение на странице), а также правила валидации. После определения класса Form можно использовать его в представлении для отображения формы на веб-странице и обработки данных, отправленных пользователем.

Для отображения формы нужно создать файл forms.py в каталоге нашего приложения cinema_app. Импортируем в него класс для работы с формами и классы моделей. Далее нужно создать класс, который назовем AddFilmsForm и укажем наследования от forms.ModelForm. Этот класс будет создавать форму на основе нашего класса из файла models.py. Для этого в классе AddFilmsForm напишем еще один класс Metta, который в свою очередь имеет атрибут model для связи класса моделей и атрибут field, в котором мы и укажем выводимые поля формы в соответствии с классом модели. Но поля, связанные с другими таблицами, вынесем отдельными атрибутами.

```
1. from django import forms
2. from .models import Cinema
3.
4.
5. class AddFilmsForm(forms.ModelForm):
6.
7.     class Metta:
8.         model = Cinema
9.         fields = ['title', 'descriptions', 'year_production',
                  'is_active', 'cat', 'genres']
```

После этого данную форму мы можем передать в HTML-шаблон для отображения вводимых полей.

```
1. def views_form(request):
2.     form = AddFilmsForm()
3.     context = {
4.         'menu': menu,
5.         'cat_selected': 0,
6.         'form': form,
7.     }
```

```
8.         return render(request, 'cinema_app/ views_form.html',
                           context=context)
```

В шаблоне для перебора полей необходимо прописать переменную, которую мы передаем в контексте по одноименному названию, обычно ее так и называют form. Для того, чтобы все поля выводились с новой строки напишем переменную так `{{ form.as_p }}`.

```
1. {% extends 'base.html' %}
2.
3. {% block content %}
4. {{ form.as_p }}
5. {% endblock %}
```

В итоге браузер нам выведет форму для заполнения данных которую после мы можем преобразовать в POST запрос.

Заголовок:

Описание фильма:

Год производства:

Статус:

Категория:

Жанры:

Рисунок 23. Вывод формы

6.8.3 Сохранение переданных данных в БД

Для сохранения данных в БД нужно в шаблоне указать тег `<form>` с методом `post`: `<form action="" method="post"></form>`. Указать `{% csrf_token %}` и прописать кнопку отправить: `<button type="submit">Отправить</button>`. Далее в представление нужно прописать такой код:

```
1. def views_form(request):
2.     if request.method == 'POST':
3.         form = AddFilmsForm(request.POST)
4.         if form.is_valid():
5.             form.save()
6.             return redirect('home')
7.     else:
8.         form = AddFilmsForm()
9.     context = {
10.         'menu': menu,
11.         'cat_selected': 0,
12.         'form': form,
13.     }
14.     return render(request, 'cinema_app/ views_form.html',
        context=context)
```

В коде выше мы проверяем запрос наш имеет ли метод `POST`. Если да, то мы проверяем на валидность наш приходящий запрос и сохраняем введенные нами данные в БД и срабатывает перенаправление на главную страницу. Если был `GET` запрос, то мы выводим нашу форму для заполнения.

6.8.4 Загрузка файлов с использованием классов моделей

Также мы можем загружать файлы в наш проект из форм в нашу БД. Для начала нужно прописать в терминале команду: `python -m pip install Pillow`. После этого мы укажем путь куда будет происходить загрузка файлов из форм. Для этого нужно в файле проекта `settings.py` написать следующую константу: `MEDIA_ROOT = BASE_DIR / 'media'`. Далее в классе `Cinema` нашего приложения добавим поле для сохранения файла к фильму:

```
1. class Cinema(models.Model):
2.     ...
3.     photo = models.ImageField(upload_to='photos_films/', default=None,
4.                               blank=True, null=True, verbose_name='Фото')
```

Применим миграции для добавления этого поля в таблицу с БД. После нужно добавить поле в форму, а именно в класс `AddFilmsForm`, для вывода его в браузере. Но для корректной отправки файла нужно провисать в шаблоне представления параметр для тега формы: `enctype="multipart/form-data"`.

После нужно этот файл извлечь из отправленных нами данных и сохранить в БД следующим образом. Нам нужно дописать `request.FILES`:

```
1. def views_form(request):
2.     if request.method == 'POST':
3.         form = AddFilmsForm(request.POST, request.FILES)
4.         ...
```

Данные файлы теперь буду загружаться и размещаться в папке `media/photos_films`, а в БД в таблице с фильмами в поле `photo` будет ссылка на этот файл, так как хранить файлы в ней не целесообразно.

6.8.5 Отображение изображений

В нашем шаблоне представления `film.html` уже прописан атрибут для вывода изображений на экран:

```
1. {% if film.photo %}
2. <p></p>
3. {% endif %}
```

Но в данный момент мы их не увидим так как путь к ним сформирован не верно. Для верного формирования пути нужно прописать `MEDIA_URL = '/media/'` в файле `settings.py` нашего проекта. `MEDIA_URL` — это переменная, которая определяет публичный URL-префикс, используемый для доступа к загруженным медиафайлам. Она используется вместе с `MEDIA_ROOT` для построения правильных URL-адресов для доступа к файлам.

Основные задачи `MEDIA_URL`:

- Построение URL-адресов: Django использует `'MEDIA_URL'` для генерации корректных URL-адресов для доступа к загруженным файлам. Например, при использовании `{{instance.image.url}}` в шаблоне, будет сгенерирован URL, основанный на `'MEDIA_URL'`.
- Настройка веб-сервера: `'MEDIA_URL'` помогает правильно настроить веб-сервер (например, Nginx) для обработки запросов к медиафайлам. Сервер должен быть настроен на обработку запросов к `'MEDIA_URL'`.
- Безопасность: Разделение статических файлов приложения и пользовательских медиафайлов через `'MEDIA_URL'` помогает обеспечить безопасность, ограничивая доступ к критически важным файлам приложения.

Далее выполним связывание маршрутов в файле `urls.py` нашего проекта:

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

Это связывание необходимо для того, чтобы Django мог правильно обрабатывать запросы к медиафайлам во время разработки (в режиме `'DEBUG=True'`).

Когда `'DEBUG=True'`, Django предоставляет встроенный механизм для обработки статических файлов (включая медиафайлы), используя `'django.contrib.staticfiles.views.serve()'`.

Этот механизм позволяет Django обрабатывать запросы к медиафайлам, не полагаясь на внешний веб-сервер.

Однако, в production-окружении (когда `DEBUG=False`) это связывание не нужно, так как обычно медиафайлы обрабатываются внешним веб-сервером (например, Nginx), который должен быть настроен на обработку запросов к `MEDIA_URL`.

Для отображения фото в админ-панели нужно сформировать новое вычисляемое поле для и дадим ему название «изображение» с помощью `display_image.short_description`:

```
1. @admin.display()
2. def display_image(self, obj):
3.     if obj.photo:
4.         return mark_safe(f'<img src={obj.photo.url} width=40>')
5.     return 'Фото отсутствует'
6. display_image.short_description = 'Изображение'
```

6.9 Классы представлений

Функции-представления являются наиболее простым и базовым способом определения логики обработки запросов в Django. Они представляют собой обычные Python-функции, которые принимают запрос в качестве первого аргумента и возвращают HTTP-ответ. Этот подход хорошо подходит для простых сценариев, где требуется минимальная логика обработки запросов. Функции-представления просты в написании и понимании, что делает их привлекательным вариантом, особенно для начинающих разработчиков.

С другой стороны, классы-представления предоставляют более структурированный и расширяемый подход. Они основаны на ООП-принципах и наследуют функциональность от базовых классов, таких как `'View'` или его производные. Это позволяет легко повторно использовать код, переопределять и расширять поведение представлений. Классы-представления также предоставляют специальные методы, такие как `'get()'`, `'post()'`, `'put()'` и т.д., которые делают код более читаемым и понятным. Этот подход особенно полезен для более сложных приложений, где требуется гибкость и возможность расширения.

Выбор между функциями-представлениями и классами-представлениями зависит от конкретных требований вашего проекта. Для простых сценариев функции-представления могут быть более подходящими, поскольку они проще в реализации. Однако для более сложных приложений, где важны повторное использование кода, расширяемость и структурированность, классы-представления, как правило, являются предпочтительным выбором. Многие опытные разработчики Django предпочитают использовать классы-представления, поскольку они предоставляют более масштабируемый и поддерживаемый подход.

6.9.1 Класс ListView

ListView — это один из самых мощных и гибких классов в фреймворке Django, который позволяет легко создавать представления, отображающие список объектов из базы данных. Этот класс является частью пакета `django.views.generic`, который предоставляет набор готовых генерических представлений, упрощающих разработку веб-приложений.

При отображении главной страницы мы использовали в качестве представления функцию. Теперь же реализуем наше представление с помощью класса ListView следующим образом:

```
1. class CinemaHome(ListView):
2.     # model = Cinema
3.     template_name = 'cinema_app/index.html'
4.     context_object_name = 'films'
5.     extra_context = {
6.         'title': 'Главная страница',
7.         'menu': menu,
8.         'cat_selected': 0,
9.     }
10.
11.     def get_queryset(self):
12.         return Cinema.active.all().select_related('cat')
```

На первый взгляд может показаться, что мы особо не сократили программный код. Это действительно так. Объясняется очень просто. У нас с вами была достаточно простая реализация в функции представления `index()` и, как следствие, не имеем ничего нового в классе представления `CinemaHome`. Но дальше мы увидим, как, например, пагинация (разбивка на страницы) может быть легко реализована в классе буквально несколькими дополнительными атрибутами. Вообще, преимущества классов представлений перед функциями все больше и больше проявляются с ростом сложности проекта. Это происходит за счет большей модульности программного кода (благодаря классам и механизму наследования) и вынесения типовых решений в базовые классы.

Реализуем поиск на странице с помощью класса ListView:

```
1. class Search(ListView):
2.     template_name = 'cinema_app/index.html'
3.     context_object_name = 'films'
4.     extra_context = {
5.         'title': 'Результат поиска',
```

```
6.         'menu': menu,
7.         'cat_selected': None,
8.     }
9.
10.    def get_queryset(self):
11.        return
12.        Cinema.active.filter(title__icontains=self.request.GET.get('res'))
13.
14.    def get_context_data(self, **kwargs):
15.        context = super().get_context_data(**kwargs)
16.        context['res'] = self.request.GET.get('res')
17.        return context
```

6.9.2 Класс FormView

FormView является мощным инструментом для разработчиков Django, помогающий значительно упростить и ускорить процесс создания веб-форм. Его использование позволяет сосредоточиться на реализации бизнес-логики, а не на рутинной работе с обработкой HTTP-запросов и валидацией данных.

Основные преимущества использования FormView:

1. Разделение логики. FormView четко разделяет логику обработки формы, что делает код более модульным, читаемым и легко тестируемым.
2. Автоматическая обработка. Класс автоматически обрабатывает GET-запросы для отображения формы и POST-запросы для валидации и сохранения данных.
3. Встроенная валидация. FormView использует механизм форм Django для валидации пользовательских данных, что существенно упрощает эту задачу.
4. Гибкость. Благодаря наследованию от базового класса 'View', FormView можно легко расширять и переопределять необходимые методы для реализации специфичной логики.

Используем данный класс в нашем проекте. Очевидно, применим его для представления заполнения формы, а именно в добавлении фильма на сайт:

```
1. class AddFilm(FormView):
2.     form_class = AddFilmsForm
3.     template_name = 'cinema_app/ add_films.html'
4.     success_url = reverse_lazy('home')
5.     extra_context = {
6.         'title': 'Добавление фильма',
7.         'menu': menu,
8.         'cat_selected': None,
9.     }
10.
11.     def form_valid(self, form):
12.         form.save()
13.         return super().form_valid(form)
```

В данном классе мы использовали следующие методы:

- `form_class` - класс формы, который будет использоваться в представлении.
- `success_url` - URL-адрес, на который будет перенаправлен пользователь после успешной обработки формы.

6.9.3 Класс CreateView

CreateView - это класс-представление в Django, который отображает форму для создания объекта, повторно отображает форму с ошибками валидации (если таковые имеются) и сохраняет объект. Он наследуется от нескольких миксинов и базовых классов, таких как SingleObjectTemplateResponseMixin, BaseCreateView, ModelFormMixin и других.

Преимущества использования CreateView:

- Уменьшение количества кода. CreateView берет на себя большую часть работы по созданию и обработке форм, что позволяет писать меньше кода.
- Встроенная проверка данных. CreateView автоматически проверяет данные формы перед сохранением объекта в базе данных.
- Легкая настройка. Вы можете легко настроить поведение CreateView, переопределив его методы и атрибуты.

Следовательно, в нашем проекте можно заменить класс FormView, для добавления фильма на сайт на класс CreateView:

```
1. class AddFilm(CreateView):
2.     form_class = AddFilmsForm
3.     template_name = 'cinema_app/add_films.html'
4.     success_url = reverse_lazy('home')
5.     extra_context = {
6.         'title': 'Добавление фильма',
7.         'menu': menu,
8.         'cat_selected': None,
9.     }
```

Класс CreateView не нуждается в методе проверки формы на валидность, поэтому метод `def form_valid(self, form)` теперь не нужен.

6.9.4 Собственный класс DataMixin

В разных языках программирования миксины реализуются по-разному. В частности, в Python, благодаря наличию механизма множественного наследования, примеси можно добавлять в виде отдельного базового класса:

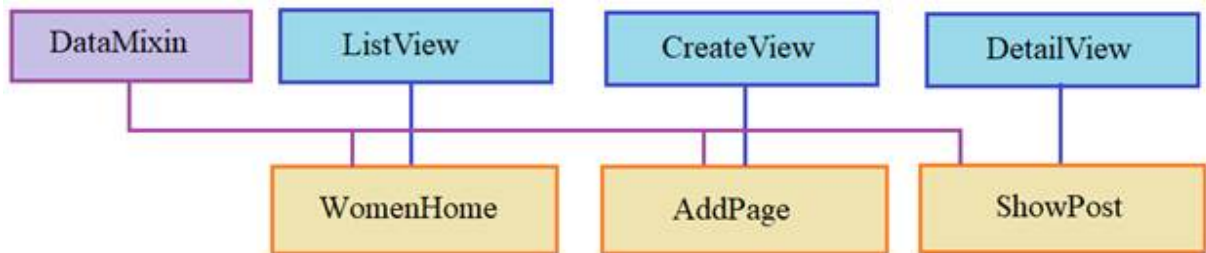


Рисунок 24. Класс DataMixin

Например, наши классы представлений можно дополнительно унаследовать от класса DataMixin, который будет отвечать за наполнение шаблонов стандартной информацией. Благодаря этому мы сможем сократить дублирование кода в тексте программы. Классы миксинов нужно записывать первыми в списке наследования. Это объясняется тем, что в Python, класс, записанный первым, первым и обрабатывается. Поэтому данные базового класса DataMixin не будут переопределять какие-либо атрибуты следующего класса ListView. Чтобы объявить класс DataMixin обычно создают файл в приложении под названием utils.py:

```
1. class DataMixin:
2.     title_page = None
3.     cat_selected = None
4.     extra_context = {}
5.
6.     def __init__(self):
7.         if self.title_page:
8.             self.extra_context['title'] = self.title_page
9.         if 'menu' not in self.extra_context:
10.            self.extra_context['menu'] = menu
11.
12.        if self.cat_selected is not None:
13.            self.extra_context['cat_selected'] = self.cat_selected
14.
15.    def get_mixin_context(self, context, **kwargs):
16.        context['menu'] = menu
17.        context['cat_selected'] = None
18.        context.update(kwargs)
19.        return context
```


Теперь наш класс `DataMixin` мы можем наследовать в наших классах представления. Давайте в классе для добавления фильмов унаследуем данный класс:

```
1. class AddFilm(DataMixin, CreateView):  
2.     form_class = AddFilmsForm  
3.     template_name = 'cinema_app/add_films.html'  
4.     success_url = reverse_lazy('home')  
5.     title_page = 'Добавление фильма'
```

Мы видим, как миксины в Django позволяют устранять дублирование кода и добавлять новый функционал в классы представления.

6.10 Авторизация и регистрация

Авторизация пользователей является важной составляющей всех сайтов сети интернет. Все мы постоянно с этим сталкиваемся, когда нас просят ввести логин и пароль для входа в личный кабинет и доступа к закрытой части сайта. Существует два вида авторизации которые нельзя путать:

Аутентификация (Authentication) — это процесс проверки личности пользователя. Она отвечает на вопрос: "Кто вы?". Другими словами, аутентификация подтверждает, что пользователь является тем, за кого себя выдает. Обычно это происходит через ввод логина и пароля, но также могут использоваться и другие методы, такие как биометрические данные или токены.

Авторизация (Authorization) — это процесс определения того, какие действия и ресурсы доступны аутентифицированному пользователю. Она отвечает на вопрос: "Что вам разрешено делать?". После успешной аутентификации, система авторизации проверяет права и разрешения пользователя и решает, может ли он выполнять определенные действия или получать доступ к определенным ресурсам.

Механизм авторизации:

1. Запрос от пользователя: Пользователь отправляет запрос на защищенную страницу или ресурс на сервере.
2. Проверка сессии. Django проверяет, есть ли у пользователя активная сессия. Если сессия существует и содержит информацию об аутентифицированном пользователе, Django извлекает объект пользователя (`request.user``) из сессии.
3. Проверка авторизации. Django проверяет, имеет ли пользователь необходимые права доступа для запрошенной страницы или ресурса. Это может быть реализовано с помощью декораторов (`'login_required'`, `'permission_required'` и др.), примесей (mixins) или путем явной проверки прав доступа в самом представлении (view).
4. Разрешение доступа. Если пользователь аутентифицирован и имеет необходимые права доступа, Django разрешает доступ к запрошенной странице или ресурсу и возвращает соответствующий ответ.
5. Перенаправление на страницу входа. Если пользователь не аутентифицирован (нет активной сессии), Django перенаправляет его на страницу входа. Обычно URL страницы входа указывается в настройках `'settings.py'` в параметре `'LOGIN_URL'`.
6. Аутентификация пользователя. На странице входа пользователь вводит свои учетные данные (логин и пароль) в форму входа и отправляет их на сервер. Django получает эти

данные и передает их функции `authenticate()` из модуля ``django.contrib.auth``, которая проверяет их на соответствие зарегистрированным пользователям в базе данных.

7. Создание сессии. Если учетные данные верны, Django создает новую сессию для пользователя и сохраняет информацию о нем в сессии с помощью функции `login()`. Обычно информация о пользователе хранится в виде ключа ``_auth_user_id`` в сессии.
8. Перенаправление на изначально запрошенную страницу. После успешной аутентификации Django перенаправляет пользователя на изначально запрошенную страницу или ресурс. Теперь пользователь аутентифицирован и имеет доступ к защищенным частям сайта.
9. Последующие запросы. При последующих запросах от того же пользователя Django будет извлекать информацию о нем из сессии и использовать ее для авторизации доступа к защищенным страницам и ресурс

Этот процесс повторяется при каждом запросе пользователя к серверу, пока пользователь не выйдет из системы (`logout`) или его сессия не истечет.

Django сразу формирует набор таблиц для хранения информации по пользователям при создании БД в файле `db.sqlite`. Если открыть эту БД, то мы увидим таблицы с префиксом `auth` и среди них таблицу `auth_user`. В них хранятся записи по всем пользователям нашего сайта. Помимо таблицы `auth_user` есть и другие вспомогательные таблицы, в частности `auth_permission` для разрешений зарегистрированных пользователей; или таблица `auth_group` для хранения информации о группах пользователей.

6.10.1 Авторизация пользователей

Для работы с авторизацией пользователей создадим новое приложение с именем `users_app` в нашем проекте командой `python3 manage.py startapp users_app`. Зарегистрируем его в установленных приложениях. Подключим его в пакете конфигурации в проекте: `path('users/', include('users_app.urls', namespace='users'))`. Параметр `namespace` в функции `include()` для создания пространства имен маршрутов приложения `"users_app"`. Однако если мы его используем нужно задать переменную `app_name` в нашем приложении в файле `urls.py` и присвоить ей как раз, то значение что и в `namespace`.

Теперь начнем с создания формы отображения ввода логина и пароля. Для этого создадим файл `forms.py` в приложении `users` и объявим в нем класс `LoginUserForm` следующим образом:

```
1. class LoginUserForm(forms.Form):
2.     username = forms.CharField(label='Логин',
    widget=forms.TextInput(attrs={'class': 'form-input'}))
3.     password = forms.CharField(label='Пароль',
    widget=forms.PasswordInput(attrs={'class': 'form-input'}))
```

Далее сформируем шаблон для отображения этой формы. Так же, как и в приложении `cinema_app`, создадим подкаталог `templates`, а в нем подкаталог `users_app`. И уже здесь разместим файл `login.html` со следующим содержимым:

```
1. {% extends 'base.html' %}
2.
3. {% block content %}
4. <h1>Авторизация</h1>
5.
6. <form method="post">
7.     {% csrf_token %}
8.     {{ form.as_p }}
9.     <p ><button type="submit">Войти</button></p>
10.</form>
11.
12.{% endblock %}
```

Теперь реализуем функцию представления. Далее мы ее преобразуем в класс, как и принято делать в Django:

```

1. def login_user(request):
2.     if request.method == 'POST':
3.         form = LoginUserForm(request.POST)
4.         if form.is_valid():
5.             cl_dt = form.cleaned_data
6.             user = authenticate(request, username=cl_dt['username'],
password=cl_dt['password'])
7.             if user and user.is_active:
8.                 login(request, user)
9.                 return HttpResponseRedirect(reverse('home'))
10.        else:
11.            form = LoginUserForm()
12.        return render(request, 'users_app/login.html', {'form': form})

```

Мы в данном случае создаем форму `LoginUserForm` с набором принятых данных `request.POST`. Затем, проверяем форму на корректность (валидность) и если проверка проходит, то создаем временную переменную `cl_dt`, которая ссылается на очищенные принятые данные формы. На основе полей `username` и `password` мы пытаемся с помощью функции `authenticate()` аутентифицировать пользователя по таблице `user` БД. Если пользователь с указанной парой логин/пароль находится в БД и является активным (не забанен, например), то вызывается ключевая функция `login()`, которая создает запись в сессии, авторизуя текущего пользователя на сайте. После этого делается перенаправление на главную страницу. В случае каких-либо ошибок снова форма отображается в браузере пользователя, предлагая ему еще раз попробовать ввести логин и пароль.

6.10.2 Классы LoginView и AuthenticationForm

LoginView — это класс представления (view) в Django, который значительно упрощает создание страницы аутентификации пользователей на сайте. Он инкапсулирует в себе всю необходимую логику для отображения формы входа и обработки данных при её отправке.

Основные преимущества использования LoginView:

1. Готовая функциональность "из коробки". LoginView избавляет нас от необходимости писать код для отображения формы логина и обработки ее отправки. Всё, что нужно — это указать путь к нему в файле `urls.py` и создать шаблон страницы входа. Об остальном позаботится Django.
2. Гибкая конфигурация через атрибуты класса. У LoginView есть набор атрибутов, которые позволяют легко кастомизировать его поведение. Например, с помощью `template_name` можно указать свой шаблон, через `authentication_form` - задать форму авторизации, а `extra_context` позволяет передать в шаблон дополнительные переменные.
3. Встроенная обработка ошибок и edge-cases. LoginView сам умеет обрабатывать различные ситуации. Если пользователь ввел неверные данные - он покажет ошибку. Если пользователь уже авторизован, а вы включили `redirect_authenticated_user` - он будет перенаправлен на другую страницу. Если не указан `next` в URL - используется `LOGIN_REDIRECT_URL` из настроек проекта. И так далее.
4. Расширяемость через наследование. Если вам нужна не просто страница логина, а что-то более специфичное - не проблема! Создайте свой класс, унаследуйте его от LoginView и переопределите необходимые методы и атрибуты. Вы получите всю мощь LoginView и свою кастомную логику сверху.

Класс AuthenticationForm в Django играет ключевую роль в процессе аутентификации пользователей. Он представляет собой форму, которая принимает имя пользователя и пароль, и проверяет их на правильность. Этот класс является базовым для создания форм аутентификации и может быть расширен и кастомизирован под конкретные нужды приложения.

Создадим класс LoginUser для замены функции аторизации `login_user()` и унаследуем его от LoginView:

```
1. class LoginUser(LoginView):
2.     form_class = LoginUserForm
3.     template_name = 'users_app/login.html'
4.     extra_context = {
```

```
5.         'title': 'Авторизация'
6.     }
```

Класс `LoginView` работает корректно с переменной `form_class` если в нее присвоить значение класса `AuthenticationForm` который создает форму на уровне самого Django. Но мы хотим использовать свою собственную форму, которую создали ранее. Для этого нужно в нашей созданной форме изменить наследование с `forms.Form` на `AuthenticationForm`:

```
1. from django.contrib.auth.forms import AuthenticationForm
2.
3. class LoginUserForm(AuthenticationForm):
4.     username = forms.CharField(label='Логин',
5.     widget=forms.TextInput(attrs={'class': 'form-input'}))
6.     password = forms.CharField(label='Пароль',
7.     widget=forms.PasswordInput(attrs={'class': 'form-input'}))
8.     class Meta:
9.         model = get_user_model()
10.        fields = ['username', 'password']
```

В классе `Meta` получаем модель `User` с помощью стандартной функции `get_user_model()`. Это рекомендуемая практика на случай изменения модели. Тогда в программе ничего дополнительно менять не придется. Также указали отображать в форме поля `username` и `password`. Именно они необходимы для аутентификации пользователя по БД.

6.10.3 Декоратор `login_required()` и класс `LoginRequiredMixin`

Декоратор `login_required()` является удобным инструментом в Django, который позволяет ограничить доступ к определенным функциям представлениям (views) только для аутентифицированных пользователей. Если пользователь не авторизован и пытается получить доступ к представлению, защищенному этим декоратором, он будет перенаправлен на страницу входа.

Реализация декоратора `login_required()` в коде использует другой декоратор `user_passes_test()`. Он проверяет, является ли пользователь аутентифицированным с помощью лямбда-функции `lambda u: u.is_authenticated`.

Если пользователь не авторизован, то декоратор перенаправляет его на страницу входа, указанную в `settings.LOGIN_URL`. При этом текущий абсолютный путь передается в строке запроса в параметре `next`. Например: `/accounts/login/?next=/polls/3/`.

Класс `LoginRequiredMixin` является миксином в Django, который обеспечивает функциональность, аналогичную декоратору `login_required()`, для классов-представлений (class-based views).

Он позволяет ограничить доступ к определенным представлениям только для аутентифицированных пользователей. Если пользователь не авторизован и пытается получить доступ к представлению, использующему этот миксин, он будет перенаправлен на страницу входа или получит ошибку HTTP 403 Forbidden, в зависимости от значения параметра `raise_exception`.

В нашем проекте реализуем возможность добавлять фильм только авторизованным пользователем. Для этого применим в класс `AddFilm` наследование класса `LoginRequiredMixin`:

```
1. class AddFilm(LoginRequiredMixin, DataMixin, CreateView):
2.     form_class = AddFilmsForm
3.     template_name = 'cinema_app/add_films.html'
4.     success_url = reverse_lazy('home')
5.     title_page = 'Добавление фильма'
6.     login_url = 'users:login'
```

Следует заменить что в переменную `login_url` мы прописываем маршрут перенаправления в случае, если пользователь не авторизован на сайте.

6.10.4 Функционал реализации отзывов

Так как теперь мы можем работать с авторизованными пользователями, реализуем добавления отзыва и редактирования. Для этого создадим в приложении `cinema_app` еще одну модель (таблицу в БД):

```
1. class FeedBack(models.Model):
2.     comment = models.TextField(blank=True, verbose_name='Отзыв')
3.     cinema = models.ForeignKey('Cinema', on_delete=models.CASCADE,
4.                                related_name='feedbackcinema')
5.     author = models.ForeignKey(get_user_model(),
6.                                on_delete=models.SET_NULL, related_name='feedbackauthor', null=True)
7.     def __str__(self):
8.         return self.comment
```

Далее опишем класс для добавления отзыва к фильму:

```
1. class AddFeedBack(LoginRequiredMixin, DataMixin, CreateView):
2.     model = FeedBack
3.     fields = ['comment']
4.     template_name = 'cinema_app/addfeedback.html'
5.     title_page = 'Добавить комментарий'
6.     login_url = 'users:login'
7.
8.     def form_valid(self, form):
9.         feedback = form.save(commit=False)
10.        feedback.author = self.request.user
11.        feedback.cinema = Cinema.objects.get(pk=self.kwargs['pk'])
12.        return super().form_valid(form)
13.
14.     def get_success_url(self):
15.         film_slug = self.object.cinema.slug
16.         return reverse('film', kwargs={'film_slug': film_slug})
```

Но редактирования отзыва сделаем так, что возможность данного функционала возможна только у пользователя, который оставил этот отзыв. Если у пользователя нет еще отзыв к данному фильму, то он отобразим кнопку «оставить отзыв», если же уже есть отзыв, то эта кнопка отображается не будет. Для этого при отображении фильма с отзывами нам нужно,

задать переменную, назовем ее `feedback_exist`, которая будет возвращать `True` или `False` в зависимости есть ли отзыв данного пользователя у данного фильма.

```
1. class ShowFilm(DataMixin, DetailView):
2.     template_name = 'cinema_app/film.html'
3.     slug_url_kwarg = 'film_slug'
4.     context_object_name = 'film'
5.
6.     def get_object(self, queryset=None):
7.         return get_object_or_404(Cinema.active,
            slug=self.kwargs[self.slug_url_kwarg])
8.
9.     def get_context_data(self, **kwargs):
10.         context = super().get_context_data(**kwargs)
11.         film = context['film']
12.         if self.request.user.is_authenticated:
13.             feedback_exist =
                FeedBack.objects.filter(author=self.request.user, cinema=film).exists()
14.             context['feedback_exist'] = feedback_exist
15.         return self.get_mixin_context(context,
            title=context['film'].title)
```

Передадим переменную в шаблон и реализуем условие, если переменная существует, то отобразим кнопку:

```
1. {% if not feedback_exist %}
2. <p class="link-add-feedback"><a href="{% url 'add_feed' film.pk
    %}">Оставить отзыв</a></p>
3. {% endif %}
```

Таким образом мы реализовали функционал с отзывами от авторизованных пользователей к фильму.

6.10.5 Регистрация пользователей с классом UserCreationForm

Для регистрации пользователей на сайте создадим форму RegisterUserForm и унаследуем ее от базовой формы из коробки Django UserCreationForm:

```
1. class RegisterUserForm(UserCreationForm):
2.     username = forms.CharField(label='Логин',
3.     widget=forms.TextInput(attrs={'class': 'form-input'}))
4.     password1 = forms.CharField(label='Пароль',
5.     widget=forms.PasswordInput(attrs={'class': 'form-input'}))
6.     password2 = forms.CharField(label='Повторите пароль',
7.     widget=forms.PasswordInput(attrs={'class': 'form-input'}))
8.
9.     class Meta:
10.         model = get_user_model()
11.         fields = ['username', 'email', 'first_name', 'last_name',
12.         'password1', 'password2']
13.         labels = {
14.             'username': 'Логин',
15.             'email': 'E-mail',
16.             'first_name': 'Имя',
17.             'last_name': 'Фамилия',
18.         }
19.         widgets = {
20.             'email': forms.TextInput(attrs={'class': 'form-input'}),
21.             'first_name': forms.TextInput(attrs={'class': 'form-
22.             input'}),
23.             'last_name': forms.TextInput(attrs={'class': 'form-
24.             input'}),
25.         }
```

Для представления регистрации отдельного класса как такового нет, поэтому опишем его унаследую от класса CreateView:

```
1. class RegisterUser(CreateView):
2.     form_class = RegisterUserForm
3.     template_name = 'users_app/register.html'
4.     extra_context = {'title': 'Регистрация пользователя'}
5.     success_url = reverse_lazy('users:login')
```

Благодаря классу `UserCreationForm` при введении не корректных данных или при несовпадении вводимых паролей, будет всплывать ошибка. При успешной регистрации пользователя мы будем перенаправленный на страницу авторизации пользователя.

Заключение

В рамках данной дипломной работы был разработан и реализован веб-сайт на основе фреймворка Django. Целью проекта было создание функционального веб-приложения с возможностью авторизации пользователей, их регистрации, добавления новых данных в базу данных и отображения этих данных в браузере.

В процессе выполнения работы были реализованы следующие основные функциональности:

1. Реализация системы авторизации и аутентификации пользователей, обеспечивающая безопасность доступа к различным разделам сайта.
2. Разработка механизма регистрации новых пользователей с использованием формы для ввода необходимой информации.
3. Создание интерфейса для добавления новых данных в базу данных с последующим их отображением на страницах сайта.
4. Реализация механизма вывода данных из базы данных в браузере с возможностью фильтрации и сортировки.

Полученный результат в виде функционирующего веб-сайта демонстрирует успешную реализацию поставленных целей и задач проекта. Разработанное приложение предоставляет пользователям удобный и интуитивно понятный интерфейс для работы с данными и обеспечивает безопасность и защиту информации.

Этот проект не только расширил мои знания и навыки в области веб-разработки на основе Django, но и позволил применить их на практике для создания реального функционального веб-приложения. Дальнейшее развитие проекта может включать в себя расширение функциональности, оптимизацию работы приложения и улучшение пользовательского опыта.

В заключение, работа над данным проектом позволила мне глубже понять принципы работы с фреймворком Django, усовершенствовать навыки разработки веб-приложений и получить ценный опыт в создании функционального и привлекательного веб-сайта.

Список используемой литературы

1. Васильев А. Н. Программирование на Python в примерах и задачах. Москва: Эксмо, 2024.
2. Дронов В.А. Django 4. Практика создания веб-сайтов на Python. Санкт-Петербург: БХВ-Петербург, 2023.
3. Постолиит Анатолий. Python, Django и Bootstrap для начинающих. Санкт-Петербург: БХВ-Петербург, 2023.
4. Документация Django // Djangoproject.com: сайт. – URL: <https://docs.djangoproject.com/en/5.0/>
5. Инструменты разработчика // MDN Web Docs: сайт. – URL: https://developer.mozilla.org/ru/docs/Glossary/Developer_Tools
6. Исключения в python. Конструкция try - except для обработки исключений // Python 3 для начинающих: сайт. – URL: <https://pythonworld.ru/ipy-dannyx-v-python/isklyucheniya-v-python-konstrukciya-try-except-dlya-obrabotki-isklyuchenij.html>
7. Основы CSS // MDN Web Docs: сайт. – URL: https://developer.mozilla.org/ru/docs/Learn/Getting_started_with_the_web/CSS_basics
8. Теги HTML-форм // сайт. – URL: https://w3schools.com/html/html_forms.asp
9. Шпаргалка по принципам ООП // Tproger: сайт. – URL: <https://tproger.ru/translations/oop-principles-cheatsheet/>
10. Download PyCharm: Python IDE for Professional Developers by JetBrains: сайт. – URL: <https://www.jetbrains.com/pycharm/download/>
11. Download Python | Python.org: сайт. – URL: <https://www.python.org/downloads/> (дата обращения: 05.04.2023)
12. GitHub: сайт. – URL: <https://github.com/mozilla/geckodriver/releases>