# РЕФЕРАТ

Выпуская квалификационная работа бакалавра состоит из 53 с., 20 рис., 9 источ., 2 прил.

КЛИЕНТ-СЕРВЕРНАЯ АРХИТЕКТУРА, ВЕБ-ФРЕЙМВОРК, DJANGO, РЕЛЯЦИОННЫЕ БАЗЫ ДАННЫХ, КОНТЕЙНЕРИЗАЦИЯ, 1C.

Работа состоит из введения, двух глав и заключения. Темой этой работы является web-приложение для автоматизации документооборота. Целью этой работы является сокращение трудоемкости создания и хранения документов путем автоматической проверки целостности вводимых данных, добавления фильтров по полям документов при их выводе и создания отчетов по ним.

В первой главе описывается система 1С как одна из возможных систем для создания кроссплатформенного приложения, а также приведены причины, по которым она не была использована, и приводится описание и характеристика web-приложений в общем вместе с используемыми с ними технологиями, такими как базы данных и контейнеризация.

Во второй главе описывается процесс итоговой реализации web-приложения через Django и сущностей в привязанной к нему базе данных, и помещения всего приложения в легкое изолированное окружение.

# СОДЕРЖАНИЕ

[РЕФЕРАТ 2](#_Toc101647069)

[СОДЕРЖАНИЕ 3](#_Toc101647070)

[ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ 4](#_Toc101647071)

[ВВЕДЕНИЕ 5](#_Toc101647072)

[1 ТЕОРЕТИЧЕСКАЯ ЧАСТЬ 8](#_Toc101647073)

[1.1 Описание бизнес-процессов 8](#_Toc101647074)

[1.2 Система *1С* 10](#_Toc101647075)

[1.3 Web-приложение 13](#_Toc101647076)

[1.4 База данных 15](#_Toc101647077)

[1.5 Контейнеризация 20](#_Toc101647078)

[2 ПРАКТИЧЕСКАЯ ЧАСТЬ 23](#_Toc101647079)

[2.1 Структура приложения 23](#_Toc101647080)

[2.2 Django 23](#_Toc101647081)

[2.2.1 Описание 23](#_Toc101647082)

[2.2.2 Структура кода 25](#_Toc101647083)

[2.2.3 Перенаправление запросов на представления 27](#_Toc101647084)

[2.2.4 Обработка запросов 28](#_Toc101647085)

[2.2.5 Определение моделей данных 29](#_Toc101647086)

[2.2.6 Запрос информации из базы данных 30](#_Toc101647087)

[2.2.7 Генерация данных для ответа (HTML страниц) 31](#_Toc101647088)

[2.2.8 Web-сервер для *Django* 32](#_Toc101647089)

[2.3 Описание сущностей базы данных 34](#_Toc101647090)

[2.4 Интерфейс web-приложения 38](#_Toc101647091)

[2.5 Помещение в контейнер 39](#_Toc101647092)

[ЗАКЛЮЧЕНИЕ 44](#_Toc101647093)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 45](#_Toc101647094)

[ПРИЛОЖЕНИЯ 46](#_Toc101647095)

# ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ

|  |  |
| --- | --- |
| API | Application Programming Interface |
| ASGI | Asynchronous Server Gateway Interface |
| CPU | Central Processing Unit |
| CRUD | Create, Read, Update, Delete |
| CSS | Cascading Style Sheets |
| CSV | Comma-separated values |
| DBMS | Database Management System |
| DRY | Don’t Repeat Yourself |
| HTTP | Hypertext Transfer Protocol |
| Id | Identification |
| JSON | Javascript Object Notation |
| RegExp | Regular Expression |
| REST | Representational State Transfer |
| RPC | Remote procedure call |
| SQL | Structured Language Query |
| TCP | Transmission Control Protocol |
| UDP | User Datagram Protocol |
| URL | Uniform Resource Locator |
| WSGI | Web Server Gateway Interface |
| ООП | Объектно-ориентированное программирование |
| УПД | Универсальный передаточный документ |

# ВВЕДЕНИЕ

Некоторая компания занимается логистикой: перевозит под заказ клиента материалы из одного пункта в другую. При этом для каждого заказа оформляется ряд документов, такие как спецификация, сам заказ и УПД.

Внутри компании все эти данные в электронном виде хранятся в виде таблицы, которая заполняется сотрудниками вручную. Их хранение в таком виде изначально было удобным и выполняло все необходимые на тот момент функции, но в процессе заполнения этой таблицы и расширения количества данных, возникли следующие проблемы:

* Каждый раз при открытии этой таблицы, загружаются все данные, даже те, которые на тот момент не нужны. Это вызывает лишнюю нагрузку на диск, и, в зависимости от диска, а также от программы, в которой хранится и/или открывается эта таблицы, данные могут загружаться и обновляться очень медленно;
* Таблица хранится в виде единственного файла на одном компьютере, так что в случае, если по каким-то причинам появится еще одна таблица, например, в процессе создания резервных копий, то может оказаться, что таблицы отличаются и нет единственного источника истины;
* В таблице по умолчанию нет никаких механизмов проверки корректности введённых данных, так что если какая-либо строка заполнена данными, которые не соответствуют допустимым, то автоматической отметки не будет, и все зависит только от сотрудников;
* В таблице так же встречаются повторяющиеся данные, такие как названия компаний, с которыми идет сотрудничество, адреса из которых или в которые везут материал. При ручном вводе таких данных есть вероятность допущения сотрудником опечаток, которые могут казаться в этой таблице разными данными, например, при поиске по ним.

Целью этой работы является сокращение трудоемкости создания и хранения документов. Для достижения этой цели необходимо проверку целостности вводимых данных можно автоматизировать, и к выдаваемым документам добавить фильтры по определенным полям, чтобы выводить не все доступные документы, а только те, которые наиболее актуальны на момент запроса.

Тогда требуется разработать некоторую систему, реализующую следующий функционал:

* Может быть доступна с любого компьютера (т.е. быть кроссплатформенной): это необходимо, что всегда был один источник истины, и введенные сотрудниками данные были видны другим и не создавали противоречий;
* Для уменьшения времени ожидания и нагрузки на диск, загружала не все данные, а только необходимые;
* Имела возможность поиска по существующим данным либо для нахождения уже добавленной информации, либо, если её нужно использовать при формировании документов, проверки её наличия;
* Проверяла введенные данные на корректность заполнения, например, что не используется несуществующие данные при заполнении документов, или что заполненные поля документов не противоречат друг другу.

Дополнительно, для этой системы можно добавить формирование отчетов и разграничение прав на удаление.

Для реализации такой системы необходимо выполнить следующие задачи:

1. Описать сам бизнес-процесс: какие документы возможны для хранения, как они образуются и что в себе содержат;
2. По описанному бизнес-процессу спроектировать базу данных, её сущности и связи между ними;
3. Реализовать само web-приложение, взаимодействующее со спроектированной базой данных и выполняющее всю необходимую логику проверки вводимых данных, фильтрации документов и создания отчетов;
4. Поместить web-приложение и все его компоненты в легкую изолированные от окружения операционной системы оболочки, называемые контейнерами.

# ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

## Описание бизнес-процессов

К фирме обращаются клиенты – компании, которым нужен перевоз материала, например, грунта, щебеня, песка, и делают заказ по перевозке этого материала какой-то массы (в тоннах) или объема (в кубических метрах) из пункта A в пункт B. Фирма, в свою очередь, создает у себя создает:

* Спецификацию – описывает стоимость для клиента перевозки материала из пункта A в пункт B. В ней указывается номер спецификации, который получается после её создания в отдельной программе, перевозимый материал, адреса, откуда и куда доставляется материал, единицы измерения, в которых измеряется материал, и цена за эту единицу, а также сам клиент, к которому привязана спецификация.
* Описание пути – это цена перевозки из пункта A в пункт B, оплачиваемая фирмой. Перевозка также может осуществляться как своими водителями фирмы, так и внешними подрядчиками, при этом подрядчики включаются в описание пути, а водители нет. Кроме того, может существовать один и тот же путь с подрядчиком и без подрядчика, при этом цены могут (и должны, для получения прибыли) отличаться;
* Заказ – описывает количество перевозимого материала в единицах, указанных в спецификации, а также будет содержать в себе либо назначенного водителя фирмы, либо внешнего подрядчика. Из-за особенностей работы фирмы, в заказе может быть указан фактический путь, несовпадающий с тем, который указан в привязанной спецификации. Цена, оплачиваемая фирмой, определяется по фактическому пути.
* УПД – выставляемый счет за один или несколько заказов. Существует два формата УПД: один для фирмы, где указываются заказы, выполненные водителями фирмы; другой для подрядчика, где указываются выполненные им заказы. Один УПД оформляется на одного клиента. УПД создается отдельно, в системе должен указываться только его номер.

Визуально весь процесс описан на рис. 1.

Спецификация составляется только один раз, когда появляется новый клиент с указанным путем и материалом, после чего по этой спецификации можно создавать заказы неограниченное число раз. Заказы могут при создании не иметь исполняющего лица – водителя или подрядчика, они могут быть назначены в позднее время. После выполнения одного или нескольких заказов одного клиента, ему выставляется УПД на оплату, в котором перечисляются выполненные услуги либо водителями фирмы, либо внешними подрядчиками (два разных формата УПД).

Еще стоит заметить, что и у спецификации, и у заказа пути могут отличатся. Это не ошибка: в действительности одна из особенности фирмы – это то что она может выполнять перевозки материала по путям, фактически отличным от указанных в документах, но для правильной оплаты ставки водителям или внешним подрядчикам необходимо также вести учет о реальных перевозках.

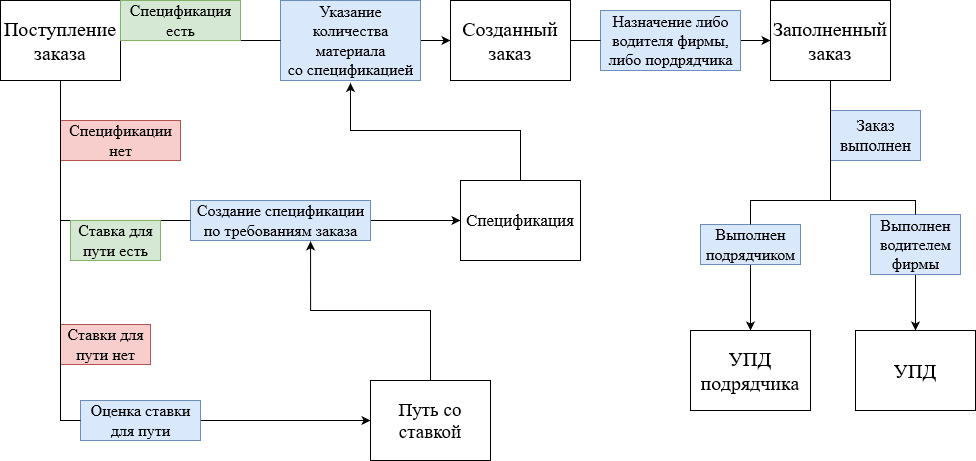


Рисунок 1 – Схема бизнес-процесса от поступления заказа до его закрытия

## Система *1С*

*1C* – разработка фирмы “1C”; программный продукт, предназначенный для автоматизации любого бизнес-процесса предприятия. Под собой *1С* может подразумевать целую развитую экосистему программ, но чаще всего, когда говорят о 1C, то смысл может меняться в зависимости от того, кто об этом говорит: бухгалтеры могут думать о введении бухгалтерского учета, кто-то будет это считать просто системой по автоматизации документооборота магазина.

На самом деле понятие о системе 1С может меняться в зависимости от того, кто с ней работает – программист, продавец, бухгалтер, бизнесмен и т.д. Это потому, что одним этим словом была названа каждая существующая часть этой системы, от языка программирования до конечного продукта. Всю систему можно разбить на компоненты:

* Платформа 1C – самостоятельная компонента, не несущая никакой практической ценности никому, кроме программистов, потому что сама она является средой разработки, и на ней опираются конфигурации;
* Конфигурации 1C – конечное программное решение, разработанное на основе платформы 1С. С конфигурациями работают уже конечные пользователи.

Компоненты показаны на рис. 2.

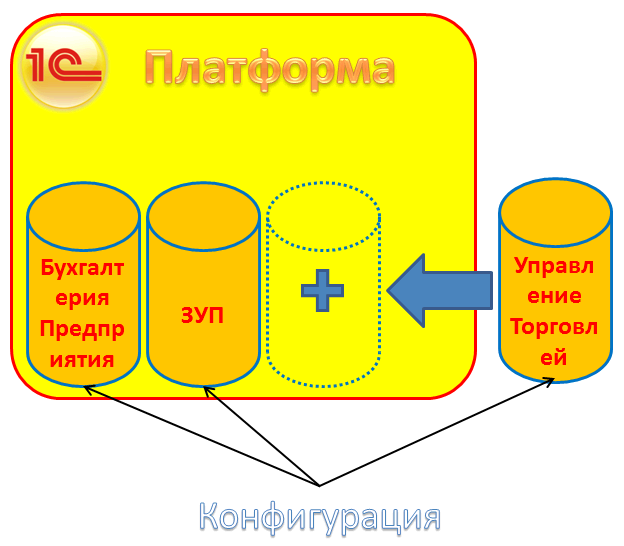


Рисунок 2 – Компоненты системы 1С [5]

Платформа 1С, в свою очередь, работает со своими базами данных в одном из двух форматах: в формате простого файла или в формате серверного решения.

В первом случае вся информация храниться в одном файле на одном компьютере во внутреннем формате CD. Такой формат на столько простой, что может быть установлен конечным пользователем самостоятельно, но страдает от низкой производительности: так как работа идет с одним файлом, то чтение из него и тем более запись в него затруднительна, если с этим файлом работает много человек, так как файл постоянно растет, а в случае с несколькими людьми еще необходимы синхронизация и, возможно, атомарные операции. Поэтому такое решение рекомендовано в том случае, если с этим файлом работает не более 4 человек.

Второй формат значительно сложнее первого и уже маловероятно, что может быть выстроен конечным пользователем без помощи специалиста. В этом случае вместо одного файла используются полноценные базы данных в таблицах на сервере, например, *MSSQL*. Очевидно, такой формат значительно быстрее одного файла из-за того, что реляционные базы данных оптимизированы для одновременной работы группы людей.

Еще у платформы 1С есть кроссплатформенность: чаще всего, конфигурации 1С используются на операционной системе *Windows*, поэтому наибольшая поддержка отдается этой версии. Но существуют и версии для других операционных систем, таких как *Linux* и *MacOS*. В случае, если последние две не являются удовлетворительными по каким-либо причинам, платформа 1С предлагает вариант Web-клиента, где работа с базами данных 1С идет через отрисовываемый в браузере интерфейс. У такого варианта есть свои ограничения, но по большей части работа в нем отлаженная и стабильная, и подходит в качестве кроссплатформенного решения.

Из отрицательных черт 1С нужно отметить запутанность экосистемы: у 1С существует множество разных форм лицензий и услуг, большинство из которых имеют достаточно абстрактные описания для неподготовленного человека. Если открыть список стоимостей услуг на сайте 1С, то можно обнаружить, что много услуг просто собраны в единую таблицу, некоторые из которых созвучны между собой, а отличаются только по цене. Сделано это, скорее всего, с целью толкнуть клиента на платные услуги консультирования специалиста.

1С не был выбран в качестве основы для системы введения документооборота по двум главным причинам:

1. По расчетам, покупка всех необходимых продуктов выходила на сумму в 50,000 рублей, что было не очень приемлемо для простого ведения учета для существующих документов;
2. Как раньше было сказано, путь, указанный в спецификации, может отличаться от реального пути заказа. Такую особенность нельзя вести в лицензированном программном обеспечении.

Требовалось альтернативное решение, которое выполняло все необходимые требования, в том числе кроссплатформенность.

## Web-приложение

Web-приложение – это прикладное программное обеспечение, которое, в отличие от запускаемых на локальной машине приложений, выполняется на сервере. Для доступа к web-приложению пользователь может использовать свой интернет браузер.

Web-приложения следуют строгой клиент-серверной архитектуре: сервер хранит в себе базу данных и всю бизнес-логику – backend, а клиенту предоставляет только возможности обращения к этой логике через построенные для отражения в браузере интерфейсы, а также отделенную от бизнеса логику, например, для динамического отображения контента без обновления страницы – frontend.

Особенность web-приложений в том, что, при условии, что браузер пользователя является одним из современных, оно может быть отображено вне зависимости от платформы, на которой стоит браузер, так как вся логика приложения лежит на сервере, а браузерам предоставляется только интерфейс к ней, поэтому с web-приложением можно работать с компьютера, со смартфона, а также и на устройствах без браузеров при правильной реализации приложения.

Web-приложения создают динамичные страницы web-сайта – страницы, которые меняют свое содержимое в зависимости от текущего состояния сессии с пользователем, например, содержимого базы данных, с которой взаимодействует backend, доступных прав пользователя. Противоположностью web-приложения является простые, статичные web-страницы, содержимое которых заложено в один файл, который просто отправляется при запросе. Естественно, статичные web-страницы быстрее динамических потому что они не создаются при каждом запросе и просто отправляются.

Тем не менее на момент написания диплома более популярно использовать динамические web-страницы даже для содержимого, который не меняется в зависимости от сессии: на web-сайтах со статьями, где web-страницы не отличаются за исключением статей (т.е. внешний вид одинаковый) вероятнее всего статьи содержаться не в виде файла, а в виде информации в базе данных, откуда она загружается в шаблон в web-страницы и отправляется пользователю. Такое решение имеет преимущества над статичной версией:

1. В случае, если меняется внешний вид сайта, то достаточно просто поменять шаблон, в который вставляется статичный контент из базы данных, вместо того чтобы менять каждую статичную web-страницу. Кроме того, его можно дополнять другой необходимой информацией, например, в шаблон можно добавить дату создания статьи, и она станет отобраться во всех статьях (если она сохранялась в базе данных);
2. Статичные файлы весят больше напрямую сохраненной информации в базе данных, так как в каждом файле хранится лишняя повторяющаяся информация, такая как расположение элементов в web-странице.

Для более сложных web-приложений принято добавлять API прослойку. Тогда пользовательские запросы через frontend будут общаться не напрямую с backend, а через предоставленный API. Добавление такой прослойки, очевидно, замедлит работу программы, так как каждый запрос будет проходить сначала через неё, а ей требуется дополнительное время на обработку запросов (поверх обработки backend'ом). Но это открывает новые возможности, такие как:

* Добавление нового контента страницы без обновления страницы – современные браузеры позволяют делать запросы к API с помощью Javascript, который также используется для добавления функционала страницы, в том числе модификации её содержимого;
* С доступным для пользователей API чаще всего можно общаться с помощью сообщений форматов, независящих от конкретных языков программирования, поэтому с API может общаться не только frontend, но и посторонние приложения. Это полезно в случае разработки нативных приложений, так как в таком случае не нужно поддерживать для каждого приложения свой backend.

Как и со всем остальным, в качестве основы web-приложения принято использовать готовые фреймворки – абстрагированная основа, в которой реализованы часто используемая функциональность, позволяющая конечному разработчику выбрать необходимый функционал и, возможно, дополнить его для создания конкретного приложения. Фреймворки могут включать себя инструменты для создания приложения, библиотеки с кодом, компиляторы и API для интеграции с другими системами и создания целых проектов. В частности, web-фреймворки могут включать в себя готовые решения, такие как маршрутизация HTTP запросов, ведение сессии через Cookie файлы, система аутентификации и т.п.

Фреймворки по умолчанию не реализуют свои механизмы хранения данных и перекладывают эту функцию на внешние базы данных, а сами лишь реализуют абстракцию, используемую для общения с базой данной через фреймворк, а не напрямую.

## База данных

Базой данных считается собрание связанных между собой данных в единое целое. Под данными принимаются факты, которые могут быть записаны и имеют не описанный явно смысл. Наиболее часто приводимый пример базы данных в программировании: объединение данных о книгах книжного магазина, таких как ISBN, название, описание, жанр, количество и так далее. Но и более простые форматы, например, телефонную книжку, можно назвать базой данных, так как там для каждого номера привязано ФИО.

Предыдущее определение получилось слишком обобщенным: приложения в книге являются связанными между собой данными, а книга получается их собранием. Но при этом книгу не называют базой данных. Для уточнения, к определению нужно привязать необходимые свойства:

* База данных описывает какую-то часть реального мира, называемую областью интерпретации (University of Disclosure). Изменения в этой области отражаются в базе данных;
* Данные в базе данных имеют между собой логическую связь без явно выраженного смысла. Например, строки телефонной книги содержат номер и ФИО, то есть номер и ФИО имеют логическую связь – у человека с таким ФИО есть такой номер. Соответственно, данные случайные и никак неупорядоченные данные не могут составлять базу данных;
* База данных проектируется, собирается и заполняется данными для определенного назначения. У неё есть определенная группа пользователей и предусмотренное прикладное назначение, в котором эти пользователи заинтересованы.

Если первые базы данных были достаточно маленькие и могли быть поддерживаемы вручную, то современные базы данных чаще всего неограниченны по размерам и количеству данных в них, а в более крупных базах данных еще требуется частое добавление или изменение данных, что невыполнимо вручную. Поэтому была создана DBMS – система управления базами данных – это система, способствующая определению, созданию, манипулированию и разделению базы данных. При определении базы данных задается структура, тип и ограничения данных, сохраняемых в ней. Определение базы данных и информация о ней сохранятся в виде директории или словаря данных, называемые мета-данными. При создании базы данных она сохраняется в доступный для DBMS носитель информации. Как хранятся данные зависит от самой DBMS, к примеру, данные могут храниться в B-tree. При манипуляции базой данных используются функции CRUD для добавления или обновления данных в базе данных, чтобы отобразить изменения в области интерпретации, или для чтения данных из ней и даже создания отчетов. При разделении базы контролируются одновременные обращения многих пользователей или программ для поддержания параллелизма и избегания столкновений при записи.

Программное обеспечение может обращаться к базе данных через DBMS, отправляя ей запросы. Обычно запросы к DBMS вызывают одну из CRUD операций. В случае, если требуется выполнить одновременно несколько зависимых друг от друга запросов, например, за нужно обновить счета двух банковских аккаунтов, то запросы объединяются в транзакции, в которых выполняются либо все запросы, либо никакие. Если транзакция отменена, то выполненные запросы откатываются.

Дополнительно DBMS имеет функционал по защите базы данных и её поддержке в длительное время. В защиту входит как защита от ошибок в аппаратном и программном обеспечении, так и защита самих данных путем создания пользователей, групп, а также систем аутентификации и авторизации. В поддержку входит способность системы развиваться вместе с изменением требований к ней, так как зачастую базы данных могут жить долгое время.

DBMS может также делиться на SQL и NOSQL. Под SQL DBMS понимают базы данных, чаще всего реляционные, которые способны принимать только SQL запросы. Такие DBMS хранят все данные в виде таблиц с предопределенной структурой, а для добавления данных вне этой структуры придется менять её саму. Второй вид, NOSQL, что означает Not Only SQL, используется в приложениях, где обычных SQL систем недостаточно. NOSQL представляют собой распределенные DBMS – они могут быть физически размещены не в одном месте, а в нескольких связанных между собой по сети. Для них характерны частичная упорядоченность данных, высокая производительность, доступность и реплицирование (копирование и резервирование) данных. NOSQL может хранить свои данных в виде:

* Документов – все данные сохраняются в виде документов в одном из распространённых форматов (например, JSON). Доступ к документам предоставляется через их ID, но можно настроить на поиск через другие индексы;
* Ключ-значение – простая модель быстрого доступа к значению по привязанному к нему ключу. Значением может быть запись, объект, документ или какая-либо еще более сложная структура данных;
* Семейства столбцов – в реляционных базах данных принято сохранять данные в виде строк. Здесь эти данные сохраняются в виде столбцов. Это полезно при агрегации определенных значений из разных данных (они будут записаны последовательно), но вредно при чтении всех значений некоторых данных, потому что, в отличие от реляционных таблиц, они записан на диске в разброс;
* Графов – все данные выражены в виде графов, а связи между ними в виде граней графов.

Существуют NOSQL DBMS, которые одновременно используют сразу две и более категорий. На рис. 3. показаны используемые структуры данных для обоих видов баз данных.

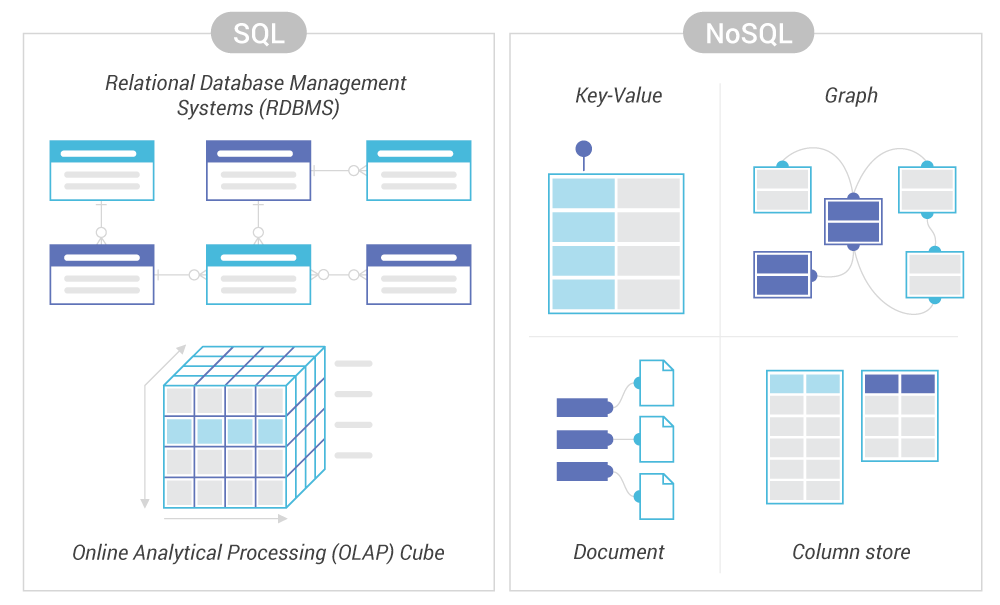


Рисунок 3 – Структуры данных в SQL и NOSQL [6]

SQL DBMS используется когда:

* У данных существует четкая структура;
* Требуется использование сложных запросов;
* К данным нужно применять определенные ограничения.
* Достаточно хранить данные на одном носителе информации, так как с SQL DBMS сложнее создать распределенные базы данных.

NOSQL DBMS используется когда:

* У данных нет четкой структуры;
* Нет нужды в сложных запросах;
* От базы данных требуется высокая производительность и адаптивность (например, когда на ранних стадиях разработки база данных еще проектируется);
* Требуется, чтобы система могла выполнять репликацию данных.

## Контейнеризация

Контейнеризация – помещение исходного кода в изолированное окружение с лишь необходимыми библиотеками операционной системы и зависимостями (посторонними библиотеками, используемыми в исходном коде) для создания контейнера – изолированный экземпляр с одним или несколькими процессами, содержащий в себе все необходимое для их запуска.

В реальном мире приложения могут запускаться на совершенно разных машинах, с разными окружениями, установленными библиотеками, конфигурациями и т.д. Причем на одной машине может оказаться, что одно приложение может иметь одни требования (специфичные версии библиотек), а другое может требовать совсем другие, уже не говоря о том, что даже если все зависимости присутствуют есть ненулевой шанс, что приложение не запустится из-за особенностей окружения. Чтобы эмулировать необходимое окружение для каждого приложения и поддерживать их постоянство используются контейнеры. Это особенно полезно в процессе самой разработки, так как в случае изменения требований разработчикам достаточно просто изменить содержимое своего контейнера.

Контейнеризация определением очень схожа с виртуализацией – разделение физических ресурсов для создания виртуальных версий чего-либо, например, серверов, носителей информации и, чаще всего, операционных систем. Но виртуализации необходимо создание гостевой операционной системы, чтобы была возможность запустить приложение на ней, а контейнеры запускаются на хостирующей операционной системе и пользуются её ядром, что делает контейнеры более легкой альтернативой виртуализации.

Для создания и настройки контейнера можно использовать посторонние программы. Современные инструменты могут собирать контейнеры по слоям – каждая выполняемая при создании контейнера команда образует свой слой, и если промежуточный контейнер уже существует, то он используется во всех последующих операциях, которые начинаются так же, – ограничивать используемые контейнерами ресурсы (например, только определенное количество времени на CPU или памяти на носителе информации), добавлять контейнеры в единую сеть, доступную только им, или перенаправлять порт с контейнера на хост машину, таким образом делая его доступным для локальной и, возможно даже, внешней сети.

По своей природе контейнеры имеют неограниченное количество применений, включая, но не ограничиваясь на: помещение стека (используемые при разработке и в окончательном продукте инструменты) в контейнеры для создания независимого и постоянного окружения; изолированного выполнения кода, например, для тестирования; помещения уже существующих приложений для увеличения их портабельности, улучшения поддержки микросервисов.

Преимущества контейнеризации:

* Портативность – исполняемый контейнер не зависит от операционной системы и окружения, на котором он запускается, поэтому он портативен и запускается одинаково что на сервере, что на облаке;
* Скорость – контейнеры часто называют легковесными, так как они запускаются на одной операционной системе и пользуются одним ядром без накладных расходов. Это уменьшает время и стоимость запуска сервера, так как не запускаются дополнительные операционные системы (виртуализация), и приводит к большей производительности;
* Поиск неисправностей – каждое из контейнеризованных приложений работают независимо друг от друга, поэтому в случае неисправности в одном контейнере все остальные продолжают работать. В это время разработчики могут найти и устранить неисправности в упавшем контейнере без вызова простоя в других;
* Простота контроля – кроме инструментов для создания контейнеров существуют инструменты для автоматизации создания и поддержания контейнеров, их обновления, мониторинга и отладки. Это нужно, так как в реальных приложениях, состоящих из микросервисов, количество контейнеров может достигать трехзначных значений;
* Безопасность – процессы, запущенные в контейнере, по умолчанию не могут взаимодействовать с другими контейнерами и хост системой, что дает дополнительный уровень безопасности и надежности, что контейнер не сможет повредить окружению.

Независимо от размера, компании, разрабатывающие приложения, начинают переходить на микросервисный формат как более совершенный подход к разработке и контролю, по сравнению с монолитным форматом, где само приложение, его интерфейс и используемая база данных вместе помещены на один сервер. С помощью микросервисов, сложное приложение может быть разбито на более мелкие сервисы, каждое из них специализирующееся на своей бизнес-логике и имеющее свою базу данных. Эти микросервисы могут общаться между собой с помощью RPC, REST API, очереди сообщений или другие возможные API. Используя микросервисы, разработчики могут работать над конкретными областями программы, не влияя на неё всю, что ускоряет процессы разработки, тестирования и развертывания.

# ПРАКТИЧЕСКАЯ ЧАСТЬ

## Структура приложения

В основе всего web-приложения лежит web-фреймворк *Django*, выполняющий большую часть функций: маршрутизация запросов, заполнение HTML шаблонов, поддерживание сессии с пользователем с помощью session cookie, общение с базой данных через абстракции. *Django* в свою очередь общается с отдельно работающей DBMS *PostgreSQL* – одной из наиболее известных систем реляционных баз данных. Хотя у *Django* есть встроенный сервер, он приложен для использования только в тестировании и не ориентирован на скорость и безопасность, поэтому необходимо использовать посторонний сервер, например, *Gunicorn*, так как у него уже есть встроенная настройка, ориентированная на *Django* приложения.

## Django

## Описание

*Django* - высокоуровневый web-фреймворк, написанный опытными разработчиками на *Python* для быстрой разработки защищенных и гибких web-сайтов. В *Django* присутствуют много часто используемых компонентов, избавляя разработчиков от необходимости заново их писать. Фреймворк имеет открытый исходный код, что дает возможность использующим его разработчикам самостоятельно добавлять функциональность и исправлять ошибки, а также большую документацию и активное сообщество.

*Django* помогает писать приложения, которые:

* Совершенные – *Django* следует философии "включая батарейки", т.е. большая часть того, что необходимо для разработки, уже включено в фреймворк, поэтому оно легко интегрируется в приложение с другими компонентами, следует шаблонам ООП и актуальную документацию;
* Универсальные – Фреймворк по умолчанию включает в себя всю функциональность, необходимую для создания приложений от систем управления содержимым и википедий, до целых новостных сайтов и социальных сетей, при этом доставляя содержимое в разных доступных форматах (HTML, JSON, XML, RSS и т.д.). В случае, если этого недостаточно, необходимые компоненты могут быть дописаны самостоятельно;
* Безопасные – *Django* включает в себя много мер, чтобы избежать наиболее частые ошибки, и старается максимально придерживаться их. Например, он безопасно хранит и управляет учетными записями, избегая таких ошибок, как хранение информации о сессии в ключе, где она может быть под угрозой изменения пользователем, и помещает туда только ключ от сессии, а информацию находит по ключу, а также не сохраняет напрямую пароль учетной записи, а прогоняет полученный от пользователя пароль через криптографическую хеш-функцию – одностороннюю функцию, легко переводящую пароль в хэш, который вычислительно невозможно перевести обратно. Таким образом, даже если атакующий сможет получить доступ к базе данным, он найдет только хэши паролей, восстановить которые практически невозможно. Другие меры защищают от таких уязвимостей, как SQL-инъекции (формирование атукающим специальным образом запросов, чтобы получить доступ к данным SQL базы данных), cross site scripting (помещение атакующим вредоносных скриптов в безопасный сайт), cross site request forgery (выполнение пользователем не желаемых действий в приложении, в котором они аутентифицированы, через сформированные атакующим запросы).
* Масштабируемые – из-за архитектуры кода, в которой каждый компонент ничем не связан с другим, каждый из них можно масштабировать вместе с увеличением трафика просто добавляя больше оборудования: серверов для кэша, баз данных, приложения;
* Поддерживаемые – код *Django* приложения пишется с использованием дизайн шаблонов и принципов, стимулирующих написание поддерживаемого и многоразового кода, в частности применяется DRY принцип, при котором избегается дублирующийся код, а изменения функций распространяется в соответствующие части кода. *Django* также способствует группировке схожей функциональности в многоразовые приложения, а также схожего кода в модули;
* Портабельные – *Python*, на котором написан *Django*, работает на разных платформах, что означает, что написанные на нем приложения не призваны к конкретным платформам, а могут работать на любых платформах, на которых доступен *Python*. Кроме того, *Django* поддерживается на многих хостинга, зачастую предоставляющие готовые инфраструктуры или документацию по хостингу *Django* приложений.

## Структура кода

В традиционном динамическом web-сайте, web-приложение ожидает HTTP запросы от браузера или любого другого клиента. После получения такого запроса, пытается определить необходимые требование из URL, по которому отправлен запрос, и, возможно, информации, приложенной к POST или GET данным. В зависимости от требований запроса, информация либо записывается в базу данных, либо считывается из неё, или даже выполнять другие действия для выполнения запроса. После этого, приложение возвращает ответ браузеру, чаще всего состоящий из HTML страницы, построенной путем вставления полученных данных в размеченные места HTML шаблона.

В Django приложениях обычно группируют весь код для каждого шага вместе в отдельные файлы:



Рисунок 4 – Обработка запроса в Django [7]

* URLs – описывают доступные приложению web-адреса и сопоставленные им представления. Отдельная функция использует таблицу для перенаправления HTTP запросов в необходимое представление в зависимости от URL, по которому отправлен запрос. Эта функция также может принимать шаблоны URL и, после подбора подходящего шаблона, добавить в запрос все предоставленные в шаблоне данные;
* Представление (View) – обрабатывающая запросы функция, получающая HTTP запросы и возвращающая HTTP ответы. Представления запрашивают необходимые данные или записывают их через запросы, а формирование HTML ответов оставляют шаблонам;
* Модели (Models) – определенные в виде Python классов сущности, описывающие структуру данных приложения и придающие возможность запрашивать записи в базе данных или менять их;
* Шаблоны (templates) – текстовые файлы, описывающие структуры конечного файла с техническими разметками там, где должны быть данные. Представление может динамически заполнить этот шаблон данными, полученными из модели, например, для создания HTML страницы. Шаблоны могут описывать структуру любого конечного файла, и с помощью них можно создавать JSON, CSV и другие файлы.

## Перенаправление запросов на представления

Таблица URL обычно содержится в файле url.py. На рисунке 5 таблица задается в переменной urlpatterns, где каждому маршруту (URL строка) сопоставляется представление. Если HTTP запрос получен по запросу, совпадающему с одним из маршрутов, то выполняется соответствующая функция представления, в которую передается информация о запросе:

|  |
| --- |
| urlpatterns = [      path('', TemplateView.as\_view(template\_name="base\_generic.html")),      path('specifications/', views.SpecificationsView.as\_view(), name='specification'),      path('specifications/new/', views.create\_specification, name='new\_specification'),      path('specifications/<int:pk>/', views.edit\_specification, name='edit\_specification'),      path('specifications/<int:pk>/delete/', views.delete\_specification,            name='delete\_specification'),  ] |

Рисунок 5 – Пример кода перенаправления запроса

Переменная urlpatterns содержит *Python* список функций path() и/или re\_path() функций, последние здесь не использованы.

В первый аргумент функций передается маршрут (шаблон), с которым сравнивается адрес запроса. В path() функции на месте угловых скобок описывается переменная, которая будет передана вместе запросом – до двоеточия стоит её тип, а после стоит название переменной, в которую передаются данные. Все, что стоит в маршруте запроса на месте угловых скобок, будет помещено в переменную, если оно совпадает типом. В функции re\_path() первым аргументом стоит не маршрут, а regexp шаблон.

Вторым аргументом функций передается представление, которое будет обрабатывать запрос. Оно может быть выражено как простой функцией, так и специальным классом, наследующим от базового *Django* класса с основным функционалом.

Аргумент name опционален. Он нужен для таких функций, как reverse() для восстановления шаблона по его названию. Это полезно, например, при задании представления, выполняющего перенаправление – тогда не нужно писать весь шаблон, а достаточно написать reverse(name).

## Обработка запросов

Функции и классы, обрабатывающие запросы, содержатся в файле view.py. Они принимают HTTP запросы и возвращают HTTP ответы, в промежутке обращаясь к другим компонентам фреймворка для доступа к базе данных, заполнения шаблонов и тому подобное.

На рисунке 6 приводится простейшее представление index. После вызова представления через таблицу URL, ему передается объект HttpRequest со всеми данными запроса. В данном случае, представление ничего не делает с запросом, а просто HTML страницу с единственной строкой “Hello World”:

|  |
| --- |
| from django.http import HttpResponse  def index(request):      return HttpResponse('Hello World') |

Рисунок 6 – Пример представления

## Определение моделей данных

Написанные на *Django* web-приложения управляют данными и запрашивают их через определенные *Python* объекты, называемые моделями. Модели определяют структуру сущностей в базе данных, включая типы и, возможно, размеры полей, значения по умолчанию, варианты списка выбора, подписи полей для форм. Определение модели не зависит от выбранной базе данных, на которой основывается приложение, модели – это абстрагированная сущность в понятном для *Django* формате, поэтому можно выбрать любую базу данных, возможно несколько. Так как реализации баз данных отличается, может оказаться, что некоторые конкретные особенности не работают с одной базой данных, но работают с другой. После выбора базы данных, общение с ней можно делать через предоставленные в Django методы модели, запросы к ней Django выполняет самостоятельно.

В приведенной модели на рисунке 7 представлена реализация описания транспорта. В ней есть поле для номера транспорта с типом char длиной 255 и поле для модели транспорта. У поля номера транспорта стоит ограничение, что все записи в нем уникальные (не может быть две машины с одинаковым номером). Так как у модели не указан первичный ключ, он создается по умолчанию. Во внутреннем классе Meta содержится общая информация о классе – его название в единственном и множественном числе, а также сортировка по умолчанию (когда не указана, сортирует по первичному ключу). Метод \_\_str\_\_ описывает запись при применении на объекте str().

|  |
| --- |
| class Vehicle(models.Model):      car\_id = models.CharField("Гос. номер", max\_length=255, unique=True)      model  = models.CharField("Модель", max\_length=255)      class Meta:          verbose\_name = "Транспорт"          verbose\_name\_plural = "Транспорты"          ordering = ['car\_id']      def \_\_str\_\_(self):          return f'"{self.car\_id}": {self.model}' |

Рисунок 7 – Пример модели

## Запрос информации из базы данных

Модели предоставляют простой и документированный API для поиска данных в базе данных. Можно искать и фильтровать записи по несколько полям одновременно и с разными критериями (полное совпадение, больше или меньше чем и т.д.), а также выполнять более сложные запросы, например, выполнять математические вычисления с отфильтрованными полями.

На рисунке 8 показано представление, фильтрующее выводимый список записей спецификаций в зависимости от данных GET запроса – пользователь вводит фильтры по из адреса и в адрес, материала, а представление фильтрует запрос по каждому из них. Стоит заметить, что запрос не выполняется сразу, а лишь заполняется необходимой информацией. Он выполняется только тогда, когда уже необходимы данных из базы.

|  |
| --- |
| class SpecificationsView(generic.ListView):      model = the\_models.Specification      def get\_queryset(self):          from\_addr = self.request.GET.get('from-addr', '')          to\_addr   = self.request.GET.get('to-addr', '')          material  = self.request.GET.get('material', '')          queryset = self.model.objects.all()          queryset = queryset.filter(from\_addr\_\_name\_\_icontains=from\_addr)          queryset = queryset.filter(to\_addr\_\_name\_\_icontains=to\_addr)          queryset = queryset.filter(material\_\_name\_\_icontains=material)          return queryset      def get\_context\_data(self, \*\*kwargs):          context = super().get\_context\_data(\*\*kwargs)          context['create\_path\_name'] = 'doc\_manager:new\_specification'          context['edit\_path\_name'] = 'doc\_manager:edit\_specification'          context['from\_addr'] = self.request.GET.get('from-addr', '')          context['to\_addr']   = self.request.GET.get('to-addr', '')          context['material']  = self.request.GET.get('material', '')          return context |

Рисунок 8 – Пример представления с фильтрами

## Генерация данных для ответа (HTML страниц)

Шаблонные системы позволяют создать шаблоны конечных страниц, помещая метки там, куда будут помещены данные после заполнения. Чаще всего, шаблоны используются для создания HTML страниц, но могут быть использованы для создания других документов. *Django* поддерживает как свою встроенную шаблонную систему, так и популярную *Python* библиотеку *Jinja2*.

На рисунке 9 показан упрощенный шаблон, который может быть использован представлением из прошлого примера – класс самостоятельно ищет файл под названием 'specification\_list.html' и передает шаблону список объектов object\_list, которые нужно отобразить. For цикл проходит по каждому объекту из этого списка и добавляет в HTML список поле doc\_no. Если список объектов пустой, то просто выводит, что спецификаций не найдено.

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>    <meta charset="utf-8">    <title>Документация</title>  </head>  <body>    {% if object\_list %}      <ul>        {% for object in object\_list %}          <li>{{ object.doc\_no }}</li>        {% endfor %}      </ul>    {% else %}      <p>Спецификаций не найдено</p>    {% endif %}  </body>  </html> |

Рисунок 9 – Пример HTML шаблона

В случае написания своей функции представления, шаблон можно генерировать с помощью функции render(request, template, context), где request – запрос, передаваемый в представление, template – путь к шаблону, context – Python словарь, в котором ключом является название переменной, а значением – её содержимым. Render в ответ возвращает HttpResponse, который нужно вернуть из представления.

## Web-сервер для *Django*

По умолчанию, *Django* способен принимать запросы и давать ответы из коробки, но в документации стоит предупреждение, что встроенный сервер нельзя использовать в конечном продукте, так как он не проходил проверки на безопасность и производительность (*Django* это web-фреймворк, а не сервер). Поэтому нужно рассмотреть альтернативы, которые уже можно применить с конечным продуктом.

Python серверов бывает два вида: WSGI и ASGI.

Web Server Gateway Interface (WSGI) – стандартный интерфейс для общения web-серверов с *Python* кодом. В *Python* есть большое количество web-фреймворков: *Django*, *Flask*, *Bottle*, *CherryPy* и т.д. Выбор одного из таких фреймворков ограничивал выбор серверов, так как не все из них имели реализацию для общения с выбранным фреймворком, и наоборот, выбор сервера ограничивает выбор фреймворков до тех, с которыми этот сервер может общаться.

*Java*, в свою очередь, использует сервлет API для того, чтобы web-сервер мог общаться с любым написанным на Java web-фреймворком, если он поддерживает этот API. WSGI следует той же методологии – избавить зависимость между конкретными серверами и фрейморками просто добавив прослойку в виде API

Asynchronous Server Gateway Interface (ASGI) – расширение WSGI, которое может поддерживать асинхронные (способные выполняться параллельно) сервера и фреймворки. ASGI имеет обратную совместимость и может поддерживать WSGI.

ASGI следует использовать для приложений, где производительность важна. Но, так как здесь она не критична, достаточно использовать WSGI, а в частности *Gunicorn* – простой, легкий и сравнительно быстрый WSGI HTTP сервер для *Python*, который имеет встроенную поддержку *Django* без дополнительных настроек.

Так как Gunicorn всего лишь WSGI HTTP сервер, то статичные файлы, такие как CSS для описания внешнего вида сайта и JS код для придания ему функциональности, им не передаются. Для решения этой проблемы в документации Gunicorn рекомендуется использовать Nginx – HTTP сервер, обратный прокси, почтовый сервер и в общем TCP/UDP сервер. Для этого проекта использована конфигурация, показанная на рисунке 10.

|  |
| --- |
| upstream backend {      server web:8000;  }  server {      listen 80;      location / {          proxy\_pass http://backend;          proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;          proxy\_set\_header Host $host;          proxy\_redirect off;      }      location /static/ {          alias /home/app/code/staticfiles/;      }  } |

Рисунок 10 – Конфигурация в Nginx

Обобщая, в конфигурации настраивается прослушивание порта 80, перенаправление запросов с него на сервер web с портом 8000. Web – это внутреннее обозначение в сети web-приложения. При обращении к статичным файлам, они будут браться из /home/app/code/staticfiles/. Такая простая конфигурация выполняет перенаправление запросов на приложение и отправку необходимых статичных файлов с ответом.

## Описание сущностей базы данных

Хотя *Django* абстрагируется от базы данных, выполняя её создание, управление и поддерживание самостоятельно, но её проектирование все ещё лежит на плечах разработчика. На рисунке 11 приведена диаграмма, визуально описывающая базу данных:

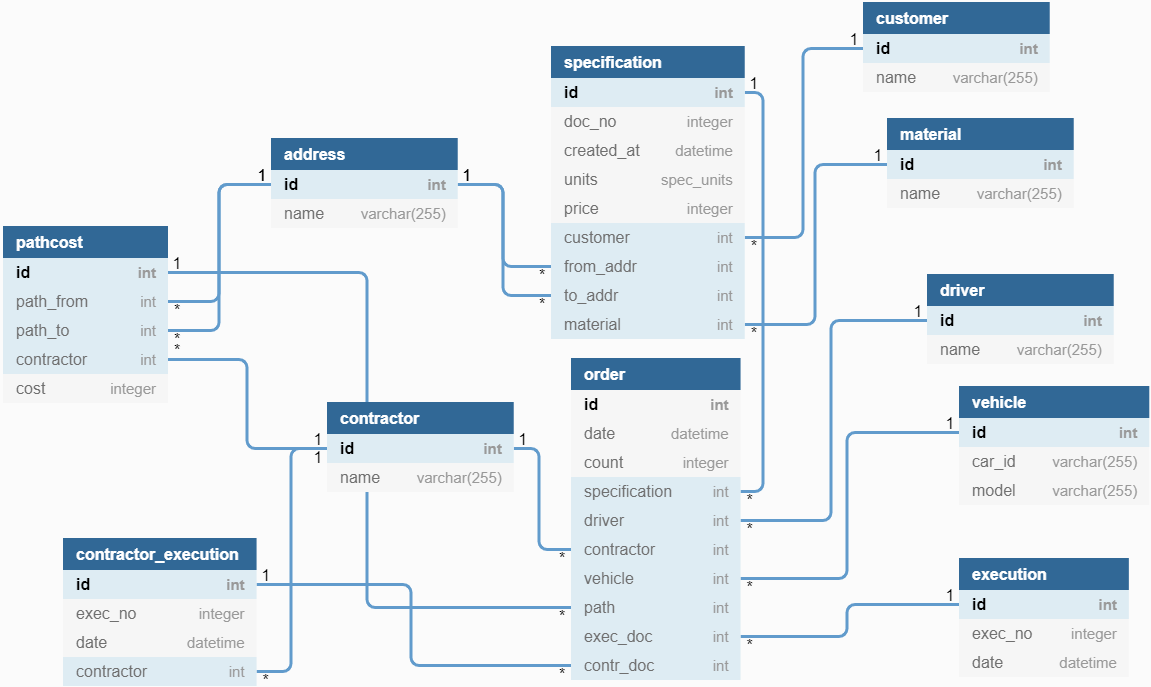


Рисунок 11 – Диаграмма базы данных.

База данных содержит 11 сущностей, большая часть из которых вспомогательная.

Сущности “адрес” (address), “подрядчик” (contractor), “заказчик” (customer), “материал” (material) и “водитель” (driver) имеют одинаковые структуры – одно поле для первичного ключа *pk* и одно поле для названия *name*. Они вынесены так как они не зависят от сущности, в которой они указываются внешним ключом, и чтобы их можно было дополнить при необходимости. За исключением водителя, там содержатся только уникальные значения

Остальные доступные сущности:

|  |  |
| --- | --- |
| Транспорт (vehicle) | |
| Описывает доступный водителям фирмы транспорт для перевозки груза | |
| pk | Первичный ключ |
| car\_id | Автомобильный номер |
| model | Модель транспорта |

Рисунок 12 – Сущность "Транспорт"

Поле car\_id содержит только уникальные значения.

|  |  |
| --- | --- |
| УПД (execution) | |
| Описывает выполненные фирмой заказы одного клиента. | |
| id | Первичный ключ |
| exec\_no | Номер УПД |
| date | Дата создания |

Рисунок 13 – Сущность "УПД"

Поле exec\_no содержит только уникальные значения. Само поле не содержит номера выполненных заказов, а, наоборот, номер УПД отмечается в выполненных заказах.

|  |  |
| --- | --- |
| УПД подрядчика (contractor\_execution) | |
| Описывает выполненные одним пордрядчиком заказы одного клиента. | |
| id | Первичный ключ |
| exec\_no | Номер УПД подрядчика |
| date | Дата создания |
| contractor | Подрядчик, кому принадлежит УПД |

Рисунок 14 – Сущность "УПД подрядчика"

Здесь поле exec\_no может иметь повторяющиеся значения, но пара exec\_no и contractor должна быть уникальной (у каждого подрядчика своя нумерация).

|  |  |
| --- | --- |
| Пути (pathcost) | |
| Описывает ставку пути и исполняющего подрядчика, если есть | |
| id | Первичный ключ |
| path\_from | Начало пути |
| path\_to | Конец пути |
| contractor | Исполняющий подрядчик |
| cost | Ставка по пути |

Рисунок 15 – Сущность "Пути"

Тройка path\_from, path\_to и contractor должна быть уникальной, в том числе, когда подрядчика нет (пустое значение все еще значение). Естественно, путь-цикл тоже недопустим.

|  |  |
| --- | --- |
| Спецификация (specification) | |
| Описывает стоимость перевозки одной указываемой единицы материала для конкретного клиента | |
| id | Первичный ключ |
| doc\_no | Номер спецификации |
| created\_at | Дата создания |
| units | Единица измерения (куб. метры или тонны) |
| price | Стоимость одной единицы |
| customer | Заказчик, с которым связана спецификация |
| from\_addr | Адрес погрузки материала |
| to\_addr | Адрес выгрузки материала |
| material | Название материала |

Рисунок 16 – Сущность "Спецификация"

Для спецификации свойственны следующие ограничения: номер спецификации может повторяться, но для каждого клиента он уникален; пятерка customer, from\_addr, to\_addr, material и units должна быть уникальна, то есть нельзя создать еще раз ту же самую спецификацию.

|  |  |
| --- | --- |
| Заказ (order) | |
| Созданный клиентом заказ на перевозку материала. | |
| id | Первичный ключ |
| date | Дата создания |
| count | Количество единиц материала для перевозки |
| specification | Спецификация, по которой создается заказ |
| driver | Исполняющий водитель |
| contractor | Исполняющий подрядчик |
| vehicle | Транспорт фирмы, используемый водителем |
| path | Путь, по которой осуществляется перевозка |
| exec\_doc | УПД, который закрывает заказ |
| contr\_doc | УПД подрядчика, который закрывает заказ |

Рисунок 17 – Сущность "Заказ"

Допустимо, что первоначально созданный заказ может не иметь исполняющего лица (водитель и подрядчик будут пустыми). После полного заполнения должны быть указаны путь, по которому выполняется фактическая перевозка path (может не совпадать со спецификацией), и исполняющее лицо – либо водитель с транспортом, либо подрядчик. После выполнения заказ можно закрыть созданием УПД, тип которого зависит от того, кто выполнял заказ. В соответствующие поля указываются УПД, закрывающие заказ.

## Интерфейс web-приложения

Реализованный интерфейс показан на рисунке 18:

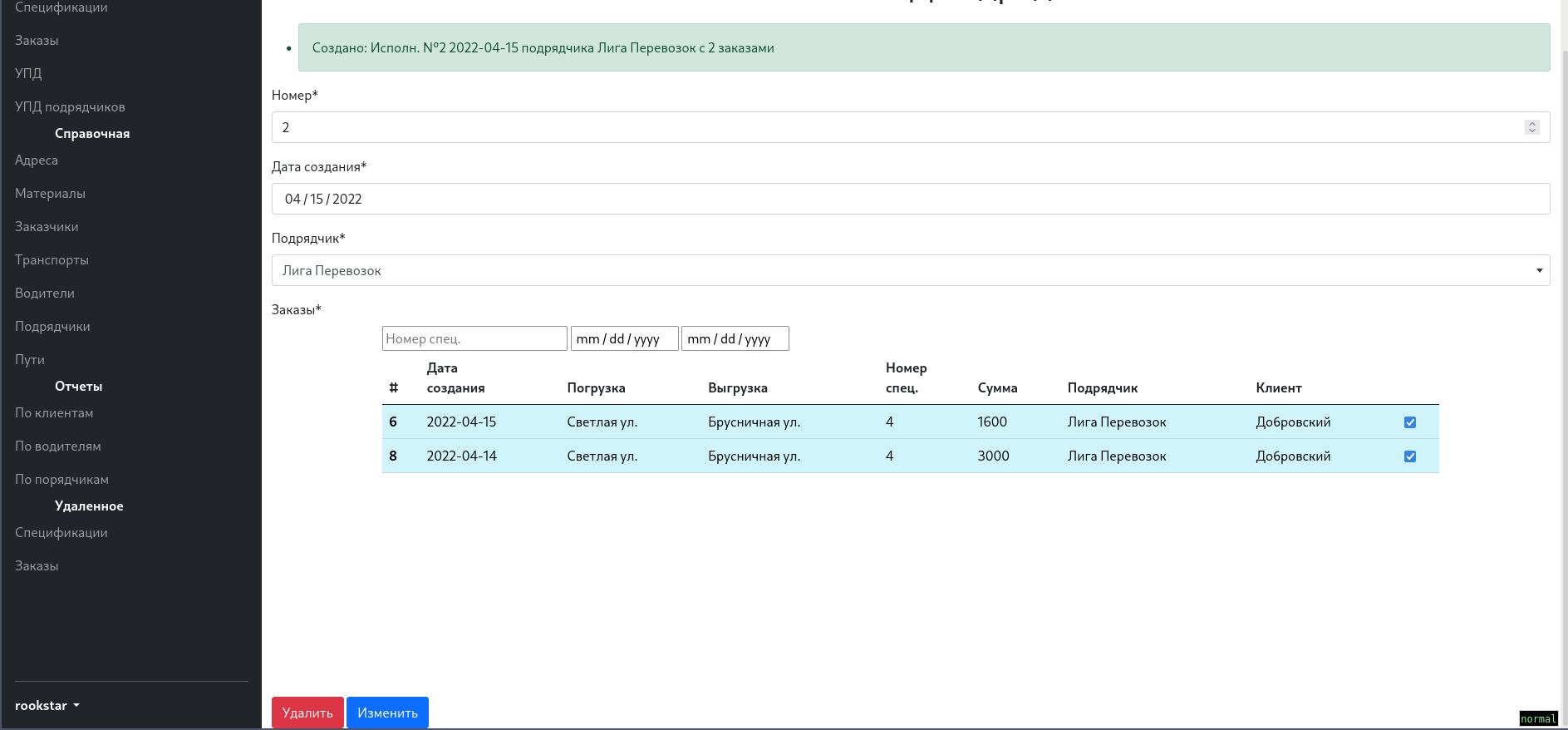


Рисунок 18 – Интерфейс web-приложения.

При реализации использовался *Bootstrap* – frontend-фреймворк с открытым исходным кодом. Он имеет реализованные CSS классы, применение к HTML элементов которых позволяет добавить не только единообразный внешний вид, но и добавить функциональность, привязанную к этим классам в Javascript части фреймворка. Например, сообщение и кнопки на рисунке 3 отрисованы с помощью bootstrap. Кроме CSS классов, Bootstrap имеет методологию блочного расположения элементов на странице. На рисунке 3 главный блок – белое полотно, на которое помещается всё основное содержимое. Вспомогательный блок – это боковая панель. Внешний вид надписей и их расположение на боковой панели также реализовано через Bootstrap.

## Помещение в контейнер

Все *Python* приложения по умолчанию зависят от окружения, в котором они запускаются – права, от которого они запущены, установленные модули (файл или директория, содержащая в себе *Python* код; *Python* библиотека), переменные окружающей среды, и иногда даже операционная система. Все это может отличаться между разными машинами и серверами, на которых они запускаются, и, естественно, результат их запуска может значительно отличаться - на одних серверах он может запускаться как нужно, на других приложение может вылетать из-за несовпадающих модулей или даже операционной системы. Также, окружение, на которой идет разработка, может не совпадать с окружением, куда выкладывается конечное приложение.

Чтобы создать единообразие запуска приложения, его можно поместить в контейнер. Для этого можно использовать *Docker* – платформа с открытым исходным кодом для создания, запуска приложений и их поставки, если они готовы. *Docker* помогает разделять приложения в инфраструктуре для их более быстрой поставки. С *Docker* инфраструктуру можно поддерживать также, как и отдельные приложения. Пользуюсь его методологиями для тестирования, поставки и развертки кода, можно значительно сократить задержку между написанием кода и его разверткой.

*Docker* использует архитектуру "клиент-сервер" – *Docker* клиент общается с *Docker* демоном (программа, которая выполняется на заднем плане и не работает с пользователем напрямую), который выполняет все основные задачи создания, запуска и распространения контейнеров. Клиент можно запускать на той же системе, на которой стоит демон, а можно подключать его к удаленному демону, на другой системе. *Docker* клиент общается с *Docker* демоном через REST API, UNIX сокет или сетевой интерфейс.

*Docker* разделяет понятия "изображение" (image) и "контейнер". Изображения – это закрытый для записи шаблон с инструкциями по созданию *Docker* контейнера и изолированной файловой системой, содержащей в себе все необходимые для его создания файлы (библиотеки, модули, код и/или бинарные файлы), а также метаинформацию, такую как переменные окружающей среды и стандартную команду запуска. Оно также может основываться на другом изображении.

Изображение создается по *Dockerfile* – простой текстовый файл с инструкциями по созданию изображения. Каждая новая инструкция создает новый слой изображения. Последний слой будет представлять конечное изображение. Пример *Dockerfile* на рисунке 19:

|  |
| --- |
| # syntax=docker/dockerfile:1  FROM python:3.8-slim-buster  ENV PYTHONUNBUFFERED=1  ENV USERNAME=app \      GROUPNAME=app \      APP\_HOME=/home/app/code/  RUN groupadd -r $USERNAME && \      useradd -g $GROUPNAME -r $USERNAME && \      mkdir -p $APP\_HOME  WORKDIR $APP\_HOME  COPY requirements.txt $APP\_HOME  RUN pip install -r requirements.txt  COPY . $APP\_HOME  RUN chown -R $USERNAME:$GROUPNAME $APP\_HOME  USER $USERNAME |

Рисунок 19 – Код для изображения в Docker

По порядку, изображение берет за основу другое изображение – легкую версию *Python* только с необходимыми пакетами, задает переменную окружения, отключающую буфферизацию вывода (все, что выводит *Python*, сразу выписывается в консоль), задает переменные с названием пользователя, группы и директории, под которой будет запускаться приложение, и создает их. Файл requirements.txt описывает все необходимые зависимости для приложения, он копируется и используется для установки. В конце копируется приложение и присваивается конечному пользователю, и выполняется вход под ним.

С помощью итогового изображения можно создавать ограниченное лишь техническими возможностями количество контейнеров, каждое из которых может работать независимо от других. Создание нескольких контейнеров из одного изображения называется горизонтальным масштабированием. Так как логика между контейнерами не отличается, то запросы к приложению можно балансировать между контейнерами.

Стандартный Docker клиент не умеет запускать одновременно несколько контейнеров, масштабировать их и вообще работать с группами связанных друг с другом контейнеров. Для этого существует отдельный клиент, называющийся docker-compose – инструмент для задания и запуска состоящих одновременно из нескольких контейнеров приложений. Для задания конфигурации каждого контейнера используется YAML файл. Пример YAML файла, используемого для запуска приложения, показан на рисунке 20:

|  |
| --- |
| version: "3.8"  services:    db:      image: postgres      volumes:        - ./data/db:/var/lib/postgresql/data      environment:        - POSTGRES\_DB=postgres        - POSTGRES\_USER=postgres        - POSTGRES\_PASSWORD=postgres    web:      build: .      command: gunicorn webdocs.wsgi:application --bind 0.0.0.0:8000      env\_file:        - key.env      depends\_on:        - db    nginx:      build: ./nginx      volumes:        - ./staticfiles:/home/app/code/staticfiles      ports:        - "80:80"      depends\_on:        - web |

Рисунок 20 - Код для docker-compose

Здесь создаются три сервиса – базы данных db, само web-приложение web и nginx с сервером. db пользуется изображением postgresql. Так как это контейнер, то все данные будут утеряны после его уничтожения. Чтобы их сохранить, дополнительно с помощью настройки volumes включается отражение внутреннего файла, содержащего базу данных, во внешней файл. web не пользуется никаким изображением, а строит его из находящегося в текущей папке Dockerfile'а. env\_file определяет загружаемый файл с переменными окружающей среды. Они нужны для хранения значений, которые должны храниться на конкретной машине и не должны быть в публичном доступе (например, закрытые ключи). depends\_on описывает зависимость web сервиса от db сервиса. command описывает стандартную команду, используемую при запуске контейнера.

# ЗАКЛЮЧЕНИЕ

В данной выпускной квалификационной работе бакалавра был форматизирован бизнес-процесс, описывающий последовательность действий от поступления заказа до его закрытия составлением УПД, спроектирована база данных, удовлетворяющая требованиям бизнес-процесса, в которой учтены все ограничения по данным в ней, и разработано приложение, значительно сокращающее трудоемкость создания и хранения документов путем автоматической проверки целостности вводимых данных, добавления фильтров по полям документов при их выводе и создания отчетов по ним.

Результатом этой работы является web-приложение, которое:

* Имеет систему авторизации и аутентификации пользователя для определения его ролей и прав;
* Может выдавать документы постранично, то есть на одну страницу выводится только определенной часть из всех имеющихся документов:
* Фильтрует выдаваемые документы или информацию, например, по определенному промежутку времени, или, в случае заказов, показывает только незаполненные заказы;
* Проверяет вводимые данные на целостность как на уровне базы данных, так и на прикладном уровне, во время ввода данных пользователем в формы;
* Формирует отчеты по заказам, высчитывая прибыль от них, по водителям и по подрядчикам, считая по каждым из них выплаты за выполненные работы, а также выводит эти отчеты в виде *excel* таблицы;
* Не удаляет сразу заказы и спецификации, а лишь помечает их на удаление, что дает возможность пользователям с более высокими правами перепроверить удаленные документы перед их полным удалением.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Elmasri R., Navathe S., Fundamentals of Database Systems 7th Edition // Pearson – 2016
2. W. Richard Stevens, TCP/IP Illustrated, Vol. 1: The Protocols // Addison–Wesley – 1994-96
3. Matthes E., Python Crash Course, 2nd Edition // No Starch Press – 2019
4. Кинзябулатов Р., Что такое 1С. О сложной системе простыми словами [Электронный ресурс] – URL: <https://habr.com/ru/company/trinion/blog/250893> (дата обращения 03.04.2022)
5. 1С Платформа и конфигурация: два продукта — одна задача [Электронный ресурс] – URL: <http://blog.it-terminal.ru/1c-admin/chto-takoe-platforma.html> (дата обращения 03.04.2022)
6. Differences Between SQL vs NOSQL [Электронный ресурс] – URL: <https://www.scylladb.com/learn/nosql/nosql-vs-sql/> (дата обращения 04.04.2022)
7. Mozilla, Resources for Developers – URL: <https://developer.mozilla.org/> (дата обращения 07.04.2022)
8. Containerization Explained – URL: <https://www.ibm.com/cloud/learn/containerization> (дата обращения 07.04.2022)
9. PEP 3333, Python Web Server Gateway Interface v1.0.1 – URL: <https://peps.python.org/pep-3333/> (дата обращения 12.04.2022)

# ПРИЛОЖЕНИЯ

Приложение 1. Модели *Django*.

|  |
| --- |
| from django.db         import models  from django.db.models  import Sum, F, Q  from safedelete.models import SafeDeleteModel  class Address(models.Model):      name = models.CharField("Полный адрес", max\_length=255, unique=True)      class Meta:          verbose\_name = "Адрес"          verbose\_name\_plural = "Адреса"          ordering = ['name']      def \_\_str\_\_(self):          return self.name  class Contractor(models.Model):      name = models.CharField("Подрядчик", max\_length=255)      class Meta:          verbose\_name = "Подрядчик"          verbose\_name\_plural = "Подрядчик"          ordering = ['name']      def \_\_str\_\_(self):          return self.name  class PathCost(models.Model):      path\_from = models.ForeignKey(                              Address,                              on\_delete=models.PROTECT,                              related\_name='path\_from',                              verbose\_name="Начало пути")      path\_to = models.ForeignKey(                              Address,                              on\_delete=models.PROTECT,                              related\_name='path\_to',                              verbose\_name="Конец пути")      contractor = models.ForeignKey(                              Contractor,                              on\_delete=models.PROTECT,                              null=True,                              blank=True,                              verbose\_name="Подрядчик")      cost = models.PositiveIntegerField("Ставка")      class Meta:          verbose\_name = "Путь со ставкой"          verbose\_name\_plural = "Пути со ставками"          ordering = ['path\_from', 'path\_to']          constraints = [              models.UniqueConstraint(                  fields=['path\_from', 'path\_to', 'contractor'],                  name='pathcost\_unique\_definition'              ),              models.CheckConstraint(                  check=~Q(path\_from=F('path\_to')),                  name='pathcost\_path\_not\_loop'              ),          ]      def clean(self):          # Using PostgreSQL, NULL is not equal NULL, so constraint          # with null will always pass. Manually check in this case          from django.core.exceptions import ValidationError          if self.contractor is None:              obj\_exists = PathCost.objects.filter(                      path\_from=self.path\_from,                      path\_to=self.path\_to,                      contractor\_\_isnull=True                  ).exists()              if obj\_exists:                  raise ValidationError('Путь с такими адресами уже существует')      def \_\_str\_\_(self):          string = f'Из "{self.path\_from}" в "{self.path\_to}" со стоимостью {self.cost}'          if self.contractor:              string += f' и подрядчиком {self.contractor}'          return string  class Customer(models.Model):      name = models.CharField("Клиент", max\_length=255, unique=True)      class Meta:          verbose\_name = "Клиент"          verbose\_name\_plural = "Клиенты"          ordering = ['name']      def \_\_str\_\_(self):          return self.name  class Material(models.Model):      name = models.CharField("Материал", max\_length=255, unique=True)      class Meta:          verbose\_name = "Материал"          verbose\_name\_plural = "Материалы"          ordering = ['name']      def \_\_str\_\_(self):          return self.name  class Vehicle(models.Model):      car\_id = models.CharField("Гос. номер", max\_length=255, unique=True)      model  = models.CharField("Модель", max\_length=255)      class Meta:          verbose\_name = "Транспорт"          verbose\_name\_plural = "Транспорты"          ordering = ['car\_id']      def \_\_str\_\_(self):          return f'"{self.car\_id}": {self.model}'  class Driver(models.Model):      name = models.CharField("ФИО", max\_length=255)      class Meta:          verbose\_name = "Водитель"          verbose\_name\_plural = "Водители"          ordering = ['name']      def \_\_str\_\_(self):          return self.name  class Specification(SafeDeleteModel, models.Model):      UNITS = (          ('m3', 'м3'),          ('t',  'т'),      )      doc\_no    = models.PositiveIntegerField("Номер")      date      = models.DateField("Дата создания")      units     = models.CharField("Ед. изм.", max\_length=3, choices=UNITS)      price     = models.PositiveIntegerField("Цена за ед.")      customer  = models.ForeignKey(                              Customer,                              on\_delete=models.PROTECT,                              verbose\_name="Клиент")      from\_addr = models.ForeignKey(                              Address,                              on\_delete=models.PROTECT,                              related\_name='spec\_from\_addr',                              verbose\_name="Начало пути")      to\_addr   = models.ForeignKey(                              Address,                              on\_delete=models.PROTECT,                              related\_name='spec\_to\_addr',                              verbose\_name="Конец пути")      material  = models.ForeignKey(                              Material,                              on\_delete=models.PROTECT,                              verbose\_name="Материал")      class Meta:          verbose\_name = "Спецификация"          verbose\_name\_plural = "Спецификации"          ordering = ['doc\_no']          permissions = [              ("undelete\_specification",    'Есть возможность восстанавливать спецификации, помеченные на удаление'),              ("hard\_delete\_specification", 'Есть возможность удалять спецификации, помеченные на удаление')          ]          constraints = [              models.UniqueConstraint(                  fields=['doc\_no', 'customer'],                  name='specification\_unique\_per\_customer'              ),              models.UniqueConstraint(                  fields=['customer', 'from\_addr', 'to\_addr', 'material', 'units'],                  name='specification\_unique\_definition'              ),              models.CheckConstraint(                  check=~Q(from\_addr=F('to\_addr')),                  name='specification\_path\_not\_loop'              ),          ]      def \_\_str\_\_(self):          return (f'Спец. №{self.doc\_no} {self.date} заказчика {self.customer}, '                  f'из {self.from\_addr} в {self.to\_addr}, '                  f'{self.material} {self.get\_units\_display()}')  class Execution(models.Model):      exec\_no  = models.PositiveIntegerField("Номер", unique=True)      date     = models.DateField("Дата создания")      class Meta:          verbose\_name = "УПД"          verbose\_name\_plural = "УПД"          ordering = ['date']      def \_\_str\_\_(self):          return (f'Исполн. №{self.exec\_no} {self.date} '                  f'с {self.order\_set.count()} заказами')  class ContractorExecution(models.Model):      exec\_no    = models.PositiveIntegerField("Номер")      date       = models.DateField("Дата создания")      contractor = models.ForeignKey(                              Contractor,                              on\_delete=models.PROTECT,                              verbose\_name="Подрядчик")      class Meta:          verbose\_name = "УПД подрядчика"          verbose\_name\_plural = "УПД подрядчика"          ordering = ['date', 'contractor']          constraints = [              models.UniqueConstraint(                  fields=['exec\_no', 'contractor'],                  name='cont\_exec\_unique\_definition'              ),          ]      def \_\_str\_\_(self):          return (f'Исполн. №{self.exec\_no} {self.date} подрядчика '                  f'{self.contractor} '                  f'с {self.order\_set.count()} заказами')  class Order(SafeDeleteModel, models.Model):      date          = models.DateField("Дата создания")      count         = models.PositiveIntegerField("Количество")      specification = models.ForeignKey(                              Specification,                              on\_delete=models.PROTECT,                              verbose\_name="Номер спец.")      driver     = models.ForeignKey(                              Driver,                              on\_delete=models.PROTECT,                              null=True,                              blank=True,                              verbose\_name="Водитель")      contractor = models.ForeignKey(                              Contractor,                              on\_delete=models.PROTECT,                              null=True,                              blank=True,                              verbose\_name="Подрядчик")      vehicle = models.ForeignKey(                              Vehicle,                              on\_delete=models.PROTECT,                              null=True,                              blank=True,                              verbose\_name="Транспорт")      path    = models.ForeignKey(                              PathCost,                              on\_delete=models.PROTECT,                              null=True,                              blank=True,                              verbose\_name="Путь")      exec\_doc  = models.ForeignKey(                              Execution,                              on\_delete=models.SET\_NULL,                              null=True,                              blank=True,                              verbose\_name="Номер УПД")      contr\_doc = models.ForeignKey(                              ContractorExecution,                              on\_delete=models.SET\_NULL,                              null=True,                              blank=True,                              verbose\_name="Номер УПД подрядчика")      @property      def price(self):          return self.specification.price \* self.count      class Meta:          verbose\_name = "Заказ"          verbose\_name\_plural = "Заказы"          ordering = ['id']          permissions = [              ("undelete\_order",    'Есть возможность восстанавливать заказы, помеченные на удаление'),              ("hard\_delete\_order", 'Есть возможность удалять заказы, помеченные на удаление')          ]          constraints = [              models.CheckConstraint(                  check=(Q(driver\_\_isnull=True) | Q(contractor\_\_isnull=True)),                  name="order\_not\_both\_drivers"              ),          ]      def \_\_str\_\_(self):          order\_str = f'Заказ с № спец. {self.specification.doc\_no}'          if self.path:              order\_str += f' по пути: {self.path}'          return order\_str |

Приложение 2. Базовый HTML шаблон base\_generic.html.

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>      <title>{% block title %}Реестр заказов{% endblock %}</title>      <meta charset="utf-8">      <meta name="viewport" content="width=device-width, initial-scale=1">        {% load static %}      <link rel="stylesheet" href="{% static 'css/bootstrap.min.css' %}" />      <link rel="stylesheet" href="{% static 'css/select2.min.css' %}" />      <link rel="stylesheet" href="{% static 'css/select2-bootstrap4.min.css' %}" />      {% block head\_addendum %}{% endblock %}  </head>  <body>      <div class="container-fluid">          <div class="row">              <div class="col-2 d-flex flex-column p-3 bg-dark text-white min-vh-100">              {% block sidebar %}                  <nav class="navbar navbar-dark mb-auto">                      <ul class="navbar-nav flex-column">                          <li class="nav-item ms-5">                              <strong>Данные</strong>                          </li>                          <li class="nav-item">                              <a class="nav-link" href="{% url 'doc\_manager:specification' %}">Спецификации</a>                          </li>                          <li class="nav-item">                              <a class="nav-link" href="{% url 'doc\_manager:order' %}">Заказы</a>                          </li>                          <li class="nav-item">                              <a class="nav-link" href="{% url 'doc\_manager:execution' %}">УПД</a>                          </li>                          <li class="nav-item">                              <a class="nav-link" href="{% url 'doc\_manager:contractor\_execution' %}">УПД подрядчиков</a>                          </li>                          <li class="nav-item ms-5">                              <strong>База знаний</strong>                          </li>                          <li class="nav-item">                              <a class="nav-link" href="{% url 'doc\_manager:address' %}">Адреса</a>                          </li>                          <li class="nav-item">                              <a class="nav-link" href="{% url 'doc\_manager:material' %}">Материалы</a>                          </li>                          <li class="nav-item">                              <a class="nav-link" href="{% url 'doc\_manager:customer' %}">Заказчики</a>                          </li>                          <li class="nav-item">                              <a class="nav-link" href="{% url 'doc\_manager:vehicle' %}">Транспорты</a>                          </li>                          <li class="nav-item">                              <a class="nav-link" href="{% url 'doc\_manager:driver' %}">Водители</a>                          </li>                          <li class="nav-item">                              <a class="nav-link" href="{% url 'doc\_manager:contractor' %}">Подрядчики</a>                          </li>                          <li class="nav-item">                              <a class="nav-link" href="{% url 'doc\_manager:pathcost' %}">Пути</a>                          </li>                          <li class="nav-item ms-5">                              <strong>Отчеты</strong>                          </li>                          <li class="nav-item">                              <a class="nav-link" href="{% url 'doc\_manager:order\_report' %}">По клиентам</a>                          </li>                          <li class="nav-item">                              <a class="nav-link" href="{% url 'doc\_manager:driver\_report' %}">По водителям</a>                          </li>                          <li class="nav-item">                              <a class="nav-link" href="{% url 'doc\_manager:contractor\_report' %}">По порядчикам</a>                          </li>                          <li class="nav-item ms-5">                              <strong>Удаленное</strong>                          </li>                          <li class="nav-item">                              <a class="nav-link" href="{% url 'doc\_manager:deleted\_specification' %}">Спецификации</a>                          </li>                          <li class="nav-item">                              <a class="nav-link" href="{% url 'doc\_manager:deleted\_order' %}">Заказы</a>                          </li>                      </ul>                  </nav>                  <hr>                  {% if user.is\_authenticated %}                  <div class="dropdown">                      <a id="dropdownProfile" class="text-white text-decoration-none dropdown-toggle" href="#" data-bs-toggle="dropdown">                          <strong>{{ user.get\_username }}</strong>                      </a>                      <ul class="dropdown-menu dropdown-menu-dark text-small shadow" aria-labelledby="dropdownProfile">                          <li><a class="dropdown-item" href="{% url 'logout'%}?next={{request.path}}">Выйти</a></li>                      </ul>                  </div>                  {% else %}                      <a class="text-white" href="{% url 'login'%}?next={{request.path}}">Войти</a>                  {% endif %}              {% endblock %}              </div>              <div class="col-10 ">{% block content %}{% endblock %}</div>          </div>      </div>      <script src="{% static 'js/jquery-3.6.0.min.js' %}"></script>      <script src="{% static 'js/popper.min.js' %}"></script>      <script src="{% static 'js/bootstrap.min.js' %}"></script>      <script src="{% static 'js/select2.min.js' %}"></script>      {% block body\_scripts %}{% endblock %}  </body>  </html> |