Министерство образования и науки Российской Федерации Федеральное государственное бюджетное образовательное учреждение высшего образования «Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»

ФАКУЛЬТЕТ *ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ*

КАФЕДРА *ТЕОРЕТИЧЕСКАЯ ИНФОРМАТИКА И КОМПЬЮТЕРНЫЕ ТЕХНОЛОГИИ*

Курсовая работа

по теме

«Приложение хранения библиографических ссылок».

Выполнил: Лапатин В.В.

Руководитель: Дубанов А.В.

Москва 2018

Содержание

Введение

В современном мире каждому приложению необходимо хранить и обрабатывать различную информацию, что делает чрезвычайно важным понимание разработчиком принципов работы с информацией. Для этого необходимо знать о различных средствах хранения информации и иметь представление о механизмах их работы. Также важным является умение разрабатывать серверные приложения, которые осуществляют взаимодействия с базами данных.

Таким образом, цель данной курсовой работы заключается в изучении различных средств хранения информации, создании базы данных и разработке приложений, предоставляющих инструменты для работы с сохраненными данными.

Постановку задачи можно сформулировать следующим образом.

Необходимо разработать приложение, которое выполняет генерацию списков литературы для учебных курсов в формате BibTeX. Для этого приложение должно иметь следующие функции:

1. Поиск информации о книгах.
2. Хранение и модификация информации о книгах.
3. Создание и модификация данных об учебных курсах.
4. Генерация списка литературы.

Обзор

Для любой программы интерфейс является одной из наиболее важных составляющих. Именно он определяет, как приложение будет взаимодействовать с другими программами и своими пользователями. Таким образом, можно ввести следующую классификацию интерфейсов:

1. Интерфейсы программирования приложений.
2. Графические.
3. Интерфейсы командной строки.

Несмотря на то, что эти интерфейсы имеют между собой мало общего, к ним предъявляется ряд общих требований:

1. Функциональность – интерфейс должен отвечать всем требованиям пользователя и соответствовать его задачам.
2. Логичность – интерфейс должен быть логичным и запоминающимся, чтобы взаимодействие пользователя с программой было как можно более простым и удобным.
3. Защищенность – интерфейс должен быть спроектирован таким образом, чтобы у пользователя не было возможности совершить ошибку.

После определения общих требований к любому интерфейсу, стоит рассмотреть перечисленные выше интерфейсы отдельно.

Интерфейс программирования приложений

К интерфейсам программирования приложений можно причислить любой интерфейс, который предназначен для использования разрабатываемой программы внешними приложениями. Обычно, в зависимости от используемых технологий, это набор классов, функций или методов, которые используются внешними программами. В случае данного приложения в качестве интерфейса взаимодействия было решено использовать веб-технологии. Это означает, что приложение будет взаимодействовать с любыми своими пользователями через протокол HTTP [1]. Таким образом, программный интерфейс приложения будет представлять из себя набор HTTP-методов.

В протоколе HTTP есть несколько видов методов, из которых

приложением будут использоваться следующие:

1. GET – это методы, которые запрашивают данные, они не предназначены для их записи.
2. POST – это методы, которые используются и для записи данных, и для их получения.

Говоря об HTTP интерфейсах, стоит отметить, что есть несколько различных подходов к их проектированию. Одним из наиболее общепринятых подходов является так называемый REST. Это можно расшифровать как Representel State Transfer [2]. Данная архитектура предлагает наложить на приложение ряд следующих ограничений:

1. Модель клиент-сервер – означает, что вся логика должна выполняться на удаленном сервере, а клиентское приложение должно исключительно предоставлять и получать данные.
2. Отсутствие состояния – сервер получает из запроса всю необходимую информацию и не хранит никакую информацию о сессии клиентов.
3. Кеширование – сервер сохраняет наиболее частые ответы, что позволяет не выполнять лишние запросы к базе данных и соответствующие вычисления, а сразу вернуть результат,
4. Единообразие интерфейса.
5. Уровни абстракции – сокрытие основного сервера за промежуточными. Например, без каких-либо изменений для пользователя можно внедрить между ним и сервером промежуточный сервер, который предназначен для хранения и возврата хэшированных данных пользователю. В случае отсутствия этих данных хэширующий сервер перенаправляет запрос исходному серверу.

Такой подход позволяет добиться лучшей производительности за счет отсутствия состояний между вызовами и кеширования, а архитектура становится более расширяемой из-за требований единообразия и выделения уровней абстракции.

Графический интерфейс пользователя

Графический интерфейс пользователя – это наиболее востребованные интерфейсы в современном мире, потому что пользователям намного более удобно пользоваться интерфейсами, которые для взаимодействия с пользователем могут использовать не только текст.

И, пожалуй, самый популярный вид графических приложений – это веб-сайты. Они получили такую популярность из-за своей универсальности: пользователь может открыть приложение на любом устройстве, на котором есть доступ в интернет и веб-браузер, в то время как любое другое приложение потребует установки на устройство пользователя. С другой стороны, веб-сайты очень удобны для разработчиков. Намного более выгодно разработать одно веб-приложение, чем создавать и поддерживать несколько программ для разных платформ.

Обычно разработку веб-приложения можно разделить на две части: «клиентская» и «серверная».

Клиентская – это та часть приложения, которая исполняется в браузере пользователя. Для написания веб-сайтов используются язык разметки HTML, язык описания стилей CSS и интерпретируемый язык программирования JavaScript [3].

Опишем роль каждого языка в веб-приложении:

1. HTML – это каркас всего приложения, который определяет, какие элементы будут использоваться и где они будут располагаться,
2. CSS определяет, как будут выглядеть элементы приложения. Сюда входят, например, внешний вид кнопок или вид шрифта отображаемого текста,
3. JavaScript – это интерпретируемый язык программирования, основная область применения которого заключается в придании интерактивности веб-страницам. Например, динамическую загрузку данных в таблицу можно реализовать только через JavaScript.

Серверной обычно называют ту часть приложения, которая стоит за клиентской. В основном это та программа, которая предоставляет данные, выполняет их хранение и агрегацию. Другими словами, сервер реализует всю «бизнес-логику», в то время как клиент нужен для предоставления

пользователям доступа к приложению. Более подробно об этом будет рассказано далее.

Интерфейс командной строки

Интерфейсы командной строки, в отличии от графических и программных интерфейсов, представляют наименее популярную группу приложений. Но, несмотря на это, они являются незаменимыми. Главное преимущество приложений командной строки заключается в том, что ими могут пользоваться в равной степени эффективно человек, и программа. Таким образом, интерфейсы командной строки являются сочетанием программных и графических интерфейсов: с одной стороны ими с определенной степенью удобства может пользоваться человек, а с другой стороны без каких-либо трудностей они могут использоваться для взаимодействия между приложениями. Также стоит отметить, что приложениям командной строки не нужен какой-либо графический интерфейс, только окно терминала.

Выбор СУБД

База данных является неотъемлемой частью любого приложения, которое выполняет хранение и обеспечивает работу с информацией. Так что выбор правильной технологии хранения данных является чрезвычайно важной задачей.

На текущий момент существует две различные ветви развития систем управления базами данных(СУБД) [4]: релационная и нереляционная.

Отличительной чертой реляционных баз данных является понятие отношения или таблицы. Каждая сущность, хранимая в базе данных, должна представлять собой строку таблицы со строго заданным типизированным набором столбцов. Также реляционные СУБД гарантируют выполнение так называемых свойств ACID к транзакционной системе, где под транзакцией понимается последовательность команд, представляющая логическую единицу работы с данными. Опишем свойства ACID:

1. Атомарность – транзация либо будет выполнена целиком, либо не выполнена совсем.
2. Согласованность – после выполнения транзакции в базе данных находятся корректные значения.
3. Изолированность – на транзакцию не могут оказать влияния другие транзакции, выполняемые параллельно.
4. Устойчивость – если транзакция была завершена, то даже при сбое системы изменения будут зафиксированы.

Данные свойства накладывают довольно серьезные ограничения на производительность, что послужило поводом для появления нереляционных СУБД. Их задачей было обеспечить хранение данных в высоконагруженных приложениях. В противовес свойствам ACID, нереляционные базы данных гарантируют выполнение свойств BASE:

1. Доступность – каждый запрос будет выполнен.
2. Гибкость – состояние системы может меняться со временем даже без ввода новых данных.
3. Согласованность в конечном счете – данные могут быть несогласованы в некоторые моменты времени, но в итоге приходят в согласованное состояние.

Стоит отметить, что существует множество видов нереляционных баз данных, перечислим основные:

1. Документоориентированные – хранят данные в документах.
2. Графовые – хранят данные в виде графа.
3. ключ-значение – хранят данные в виде пар вида "ключ-значение".

Так как данные, которые будут храниться в будущем приложении, достаточно просты по своей структуре, никакого выигрыша от использования графовых баз данных получить не получится.

Базы данных вида ключ-значение также не очень хорошо подходят для поставленной задачи, так как их структура является слишком простой, так что возникнут дополнительные трудности в работе с такой базой данных.

Таким образом, остаются документоориентированные базы данных. Было решено сравнить реляционные и нереляционные базы данных на примере PostreSQL и MongoDB.

В основе документоориентированной базы данных лежит понятие документа. В общем случае в качестве формата может быть множество различных стандартов: JSON, XML, YAML и так далее. В случае MongoDB для хранения используется BSON [5] – бинарное надмножество JSON. В данной СУБД документы группируются в так называемые коллекции. В отличии от реляционной модели, коллекция не имеет какой-либо строгой структуры. То есть в ней могут храниться абсолютно разные документы.

Данных подход имеет ряд своих преимуществ и недостатков. Перечислим положительные стороны:

1. Гибкость – так как коллекция не ограничена структурой хранимых данных, информация в ней может несколько отличаться от записи к записи. Это может быть полезно, если данные имеют в целом общий смысл, но в некоторых документах могут присутствовать какие-то особые поля.
2. Производительность – так как не происходит никаких проверок, запись и чтение работают существенно быстрее, чем в реляционном подходе.

Недостатки и ограничения:

1. Подобная организация плохо подходит для создание ссылок между объектами по, например, первичному ключу. Это связано с описанным выше отсутствием проверок. Канонический способ хранения данных – это использование вложенных документов. Но этот способ применим не везде из-за ограничения на размер документа в 16 МБ.
2. Как было описано в пункте выше, довольно сложно производить нормализацию данных, потому что все проверки необходимо производить не на уровне СУБД, а на уровне приложения.

Теперь стоит проанализировать данные, которые будут храниться в приложении.

Все данные имеют абсолютно строгую структуру. Если говорить о хранимых книгах, то для них есть набор полей, определенный стандартом BibTeX, который обязан быть у каждой записи. Остальные данные, такие как списки литературы и учебные курсы, также не имеют никакой вариативности. Таким образом, гибкость NoSQL подхода только добавит сложностей в связи с необходимостью ручной реализации множества проверок.

Конечная цель приложения – генерировать списки литературы для учебных курсов. И весьма логичным требованием будет то, что в любой момент времени приложение должно генерировать корректные отчеты. Таким образом, для данной задачи больше подходят требования ACID, чем BASE.

Как видно из рассуждений выше, несмотря на все свои преимущества, для поставленной задачи больше подходит реляционная модель.

Реализация

Модель сущность-связь

Первым делом необходимо формализовать таблицы базы данных. Опишем нужные таблицы:

- Textbook – таблица с книгами в формате BibTeX. Обладает несколькими UNIQUE столбцами:
  - ident – идентификатор книги, должен быть уникальным, так как именно он используется для идентификации библиографической ссылки в стандарте LaTeX.
  - isbn – уникальный для каждой книги ключ, позволяет исключить дублирование.
- LiteratureList – таблица, хранящая списки литературы. Имеет UNIQUE ограничение на пару из ID курса, которому присвоен список и года этого курса. Это необходимо для идентификации списка литературы.
- Literature – так называемая таблица пересечения, необходимая для создания связи многие-ко-многим между таблицами LiteratureList и Textbook.
- Course – задает учебный курс. Стоит отметить, что один учебный курс может иметь несколько списков литературы за разные года. Имеет ссылки на кафедру, к которой привязан и лектора, читающего курс. Обладает UNIQUE ограничением на тройку из названия, кафедры и семестра, в котором читается курс.
- Department – задает кафедру, имеет UNIQUE поле title, характеризующее название кафедры.
- Lecturer – хранит всех лекторов. Однозначно идентифицируется именем и датой рождения.

Все таблицы обладают суррогатными первичными ключами.

Стоит также отметить два важных решения, которые были приняты для всей базы данных.

Во-первых, каждая запись имеет поле timestamp – временную метку добавления или изменения записи. Это позволит иметь историю изменений. Также нужно уточнить, что данное поле имеет тип integer, что является более

общим решением, чем хранение метки во внутреннем формате СУБД.

Во-вторых, в базе данных будет отсутствовать возможность удаления записи. Для этого каждая запись имеет флаг isDeleted. При удалении пользователем записи она будет помечаться, как удаленная. У данного решения есть ряд преимуществ. Это позволяет обезопасить базу данных от ошибок пользователя и от потенциального взлома системы. И в том, и в другом случае никому не удастся нанести непоправимый ущерб данным. Причем блокировка будет осуществляться за счет того, что у учетной записи, через которую пользователь будет взаимодействовать с базой данных, не будет прав на удаление из базы данных. При этом все равно будет администратор, у которого данная возможность есть.

В итоге получаются диаграмма модели сущность-связь, показанная на рисунке 1.

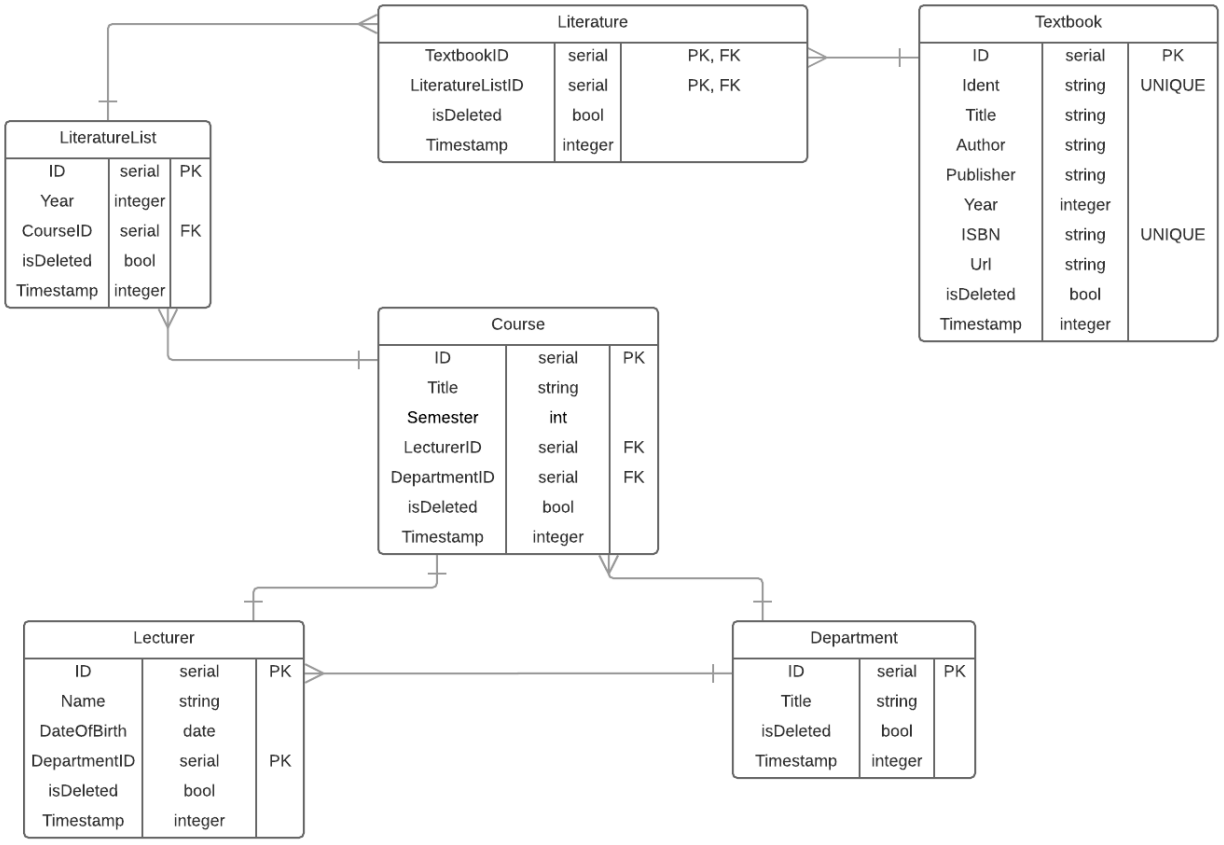

Рисунок 1: Диаграмма модели сущность-связь

Стоит отдельно отметить минимальные кардинальные связи в этой базе данных. Для упрощения конфигурации серверной части предполагается

запрет любым пользователям, кроме администратора, какого-либо удаления. Удаление предполагается каскадное, реализованное на серверной части с помощью обновления флага isDeleted. Таким образом, везде неявно предполагается минимальная кардинальная связь единица.

В качестве альтернативного решения можно было бы использовать триггеры, которые вместо удаления будут производить обновление поля. Но в таком случае пострадает переносимость. Дело в том, что не в каждой SQL базе данных присутствуют гибкие настройки прав и будет уже не так легко заменить базу данных. На уровне серверной части работа с базой данной реализована через стандартную библиотеку для работы с SQL, что дает возможность будущей миграции на любую SQL базу данных. Более подробно реализация будет описана ниже. В то время как предложенный подход не имеет такой завязки на настройку прав пользователей.

Выбор максимальных кардинальных связей основан исключительно на функциях таблиц, описанных выше и предполагаемой логики. Объясним выбор этих связей:

- Department-Lecturer – предполагается, что преподаватель может числиться только на одной кафедре, а у кафедры может быть много преподавателей,
- Department-Course – аналогично случаю Department-Lecturer,
- Lecturer-Course – обычно в университетских программах при наличии нескольких преподавателей в учебном курсе главным считается лектор, так как именно он определяет программу курса, а значит и его список литературы. Таким образом, для идентификации курса достаточно одного преподавателя – лектора. С другой стороны, лектор может вести несколько учебных курсов.
- Course-LiteratureList – у каждого курса может быть несколько разных списков литературы за разные года, но у каждого списка только один курс,
- LiteratureList-Literature-Textbook – связь многие-ко-многим между LiteratureList и Textbook, так как каждый список содержит множество книг, а учебник может быть использован в нескольких курсах.

Интерфейс программирования приложений

При рассмотрении интерфейсов программирования приложений была описана архитертура REST. Согласно этой архитектуре каждый метод должен принимать всю необходимую информацию для выполнения запроса. Таким образом, для каждого вида данных был реализован следующий набор методов:

- add – POST запрос, содержащий данные для добавления в теле,
- get – GET запрос, получающий данные от приложения,
- prototype – GET запрос, получающий прототип JSON-файла для искомой таблицы.

Также были реализованы две дополнительных команды: report и migrate.

Первая получает данные, необходимые для однозначной идентификации требуемого списка литературы и генерирует для этого списка литературы файл в формате BibTeX.

Команда migrate выполняет копирование списка литературы с одного года на другой. Это необходимо для того, чтобы облегчить работу пользователя, ведь обычно учебные программы слабо меняются от года к году.

Интерфейс командной строки

Как уже было сказано выше, для интерфейсов командной строки крайне важна автоматизируемость. Это делает невозможным использование так называемого интерактивного ввода. Другими словами, любое действие пользователя должно совершаться за один вызов приложения. В результате было решено использовать следующий подход к проектированию интерфейса: для каждого вида данных, с которыми будет работать приложение, будет реализована собственная команда. И для каждой такой команды будет реализован набор подкоманд, выполняющих необходимые действия. В общем случае команду приложения командной строки можно описать в следующем виде:

- prototype – данный метод добавляет новый файл в файловую систему, который содержит в себе прототип для нужного вида данных. После чего пользователь должен заполнить этот прототип информацией, которую

хочет добавить,

- add – сохраняет данные в приложение,
- get – позволяет получить данные из приложения, можно конфигурировать флагами.

Также отличительной чертой хорошего приложения командной строки является грамотно оформленная справка. Рисунок 2 показывает результат выполнения команды help.

```
~/C/C/B/s/w/C/bin ››› ./cli -h
Usage:
  cli [flags]
  cli [command]

Available Commands:
  book             Команда для работы с книгами
  course           Команда для работы с курсами
  department       Команда для работы с кафедрами
  help             Help about any command
  lecturer         Команда для работы с лекторами
  literature       Команда для работы с литературой
  literatureList   Команда для работы со списками литературы
  migrate          Выполнить копирование списка литературы с одного учебного года на другой.
  migratePrototype Получить заготовку JSON для миграции в файл, определяемый флагом.
  report           Выполнить создание списка литературы.
  search           Выполнить поиск книг в онлайн-источниках.

Flags:
  -h, --help   help for cli

Use "cli [command] --help" for more information about a command.
```

Рисунок 2: Пример работы команды help.

Для каждой отдельной команды реализован отдельный флаг –help, который показывает справку для конкретной команды, что можно видеть на рисунке 3.

Графический интерфейс

Как уже было сказано выше, графический интерфейс реализован в виде веб-сайта. Приложение было написано на языке TypeScript с использованием фреймворка Bootstrap. Выбор фреймворка обусловлен тем, что он предоставляет продвинутый набор стилей и HTML-элементов, что позволяет легко и быстро настроить внешний вид веб-приложения. В связи с тем, что единственный современный и поддерживаемый язык для написания веб-сайтов - это JavaScript, то из альтернатив ему можно

```
~/C/C/B/s/w/C/bin ❯❯❯ ./cli book help
Команда для работы с книгами

Usage:
  cli book [command]

Available Commands:
  add         Отправить книгу на сервер из файла, заданного флагом.
  get         Получить список книг в формате BibTeX, сохраненных в базе данных.
  prototype   Получить заготовку JSON для книги в файл, определяемый флагом.

Flags:
  -h, --help   help for book

Use "cli book [command] --help" for more information about a command.
```

Рисунок 3: Пример работы флага help для команды book

использовать только языки, компилируемые в JavaScript. В частности, к таким языкам можно отнести язык TypeScript компании Microsoft и язык Kotlin, разрабатываемый компанией JetBrains, который имеет отдельный плагин для компиляции в JS-код. К сожалению, Kotlin является очень молодым языком, который находится в активной разработке, так что было решено использовать TypeScript [7]. Преимуществом данного языка является то, что он является надмножеством JavaScript. Это означает, что любой код JavaScript является абсолютно корректным. Также важно отметить, что TypeScript поддерживает важные аспекты объектно-ориентированого программирования: по меньшей мере классы и инкапсуляцию. Но главным преимуществом этого языка является его строгая типизированность. Это облегчает разработку и позволяет писать более читаемый код по сравнению с JavaScript.

На рисунках 4 и 5 видно, приложение состоит из четырех логический частей.

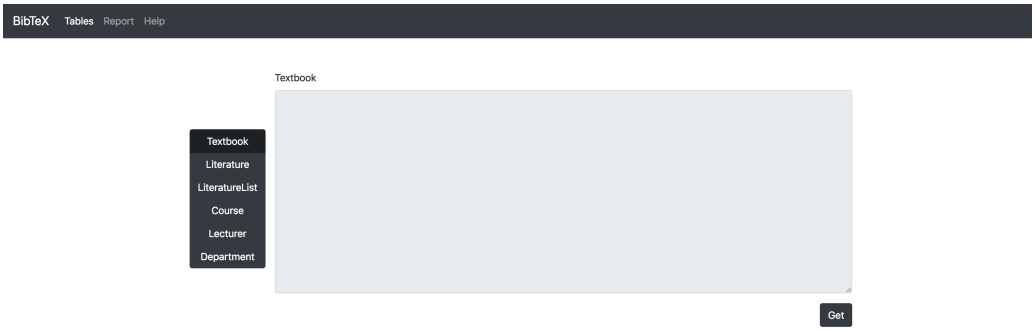

Рисунок 4: Верхняя часть веб-сайта

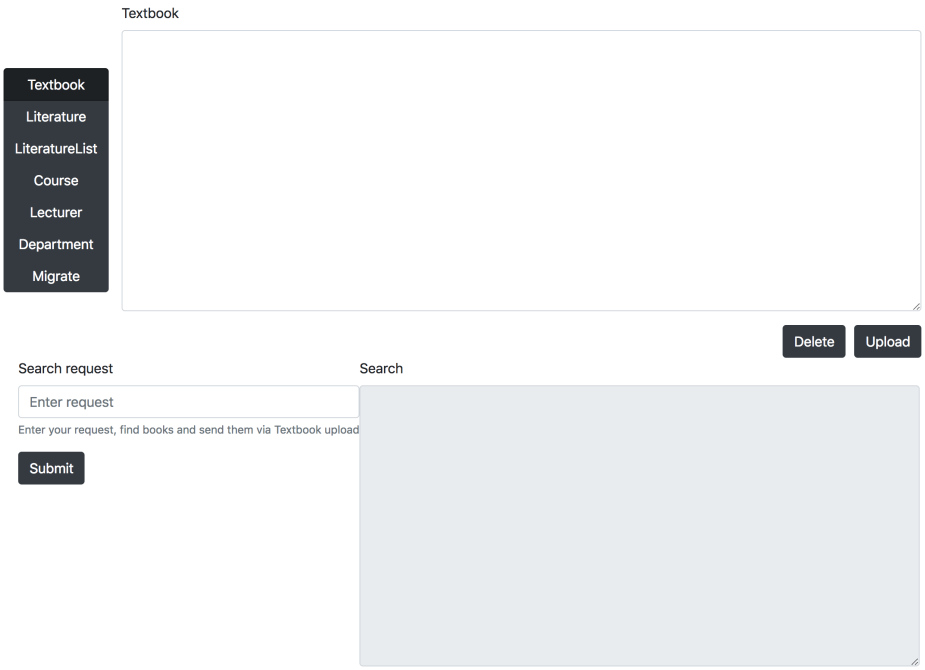

Рисунок 5: Нижняя часть веб-сайта

Первая - шапка сайта. Она содержит название приложения и ссылки на три страницы приложения: основную рабочую область, страницу генерации отчетов и страницу со справкой.

Вторая логическая часть - это большая недоступная для ввода область TextArea вместе с набором кнопок, отвечающих за переключение используемой таблицы. Эта часть нужна для получения данных, которые уже содержатся в базе данных для более удобного добавления новых записей в приложение.

Третья часть – доступная для ввода область TextArea, в которую автоматически загружаются JSON прототипы для текущего вида входных данных.

Заключительная область необходима для поиска необходимых книг в сервисе Google Books.

Серверная часть

Приложение написано на языке Go, его можно разделить на несколько модулей или пакетов, что и было сделано в исходном коде программы.

В первую очередь стоит отметить пакет `fetcher`, который отвечает за поиск книг в сервисе Google Books. Он формирует запрос с помощью API-

ключа и данных, введенных пользователем, после чего отправляет его и получает ответ, а дальше возвращает его в необходимом формате. Именно по этой причине для старта приложения необходимо вводить API-ключ.

Далее следует, пожалуй, самая крупная часть данного приложения: работа с базой данных. Как уже было упомянуто выше, работа с ней ведется с использованием связки из модуля стандартной библиотеки `database/sql` и драйвера `pq`. Пакет `database/sql` является исключительно интерфейсом, в то время как именно драйвер реализует всю необходимую логику. Для их взаимодействия нужно добавить в файл оба этих пакета, используя команду `import`.

Для начала будет правильным рассмотреть принципы работы с `database/sql`. Подключение к СУБД происходит с помощью метода `sql.Open`. Данный метод имеет два параметра:

- Имя драйвера – в данном случае это будет postgres.
- Конфигурационная строка – набор пар ключ-значение, определяющее все параметры подключения. Например, подключение к базе данных «bibtex» пользователя «username» к СУБД, находящейся на порту 8888, будет выглядеть следующим образом: `"user=username port=8888 dbname=bibtex"`.

Метод `sql.Open` вернет структуру `sql.DB`, через которую будут происходить все дальнейшие взаимодействия с СУБД, в частности, внесение и получение данных.

Следующим шагом стоит выполнить метод `DB.Ping`, который создает или восстанавливает подключение. Это позволяет удостовериться, что все работает корректно. После этого СУБД готова к использованию в приложении.

В первую очередь при разработке были реализованы методы `insert` и `select` для всех таблиц. Стоит отметить, что приложение рассчитано исключительно на использование из CLI и веб-приложения и не предусматривает вставки нескольких записей за раз. `Select` внутри запроса сразу выполняет необходимые объединения `join` для того, чтобы результат был максимально удобен для использования пользователем.

Следующим шагом были разработаны методы для удаления из таблиц.

Как уже было сказано выше, было решено использовать каскадное удаление на стороне сервера. Для этого были использованы транзакции. В пакете `database/sql` они представляются структурой `Tx`. Если начинать использовать транзакции в удалении, то было бы логично начать использовать их везде, в том числе и уже реализованных `insert` и `select`. Для этого был создан интерфейс языка Go, который можно увидеть в листинге 1.

```
type SQLExecutable interface {
        Exec(string, ...interface{}) (sql.Result, error)
        Query(string, ...interface{}) (*sql.Rows, error)
        QueryRow(string, ...interface{}) *sql.Row
}
```

Листинг 1: Интерфейс унификации транзакций.

Этот интерфейс необходим для унификации используемых методов структур `sql.DB` и `sql.Tx`. С использованием этого интерфейса стало возможно написание метода `getSQLExecutable`, который возвращает структуру транзакции, если она была иницилизирована и обычную `sql.DB` в ином случае.

После этого достаточно все использования `sql.DB` заменить на метод `getSQLExecutable`. Важно отметить, что для использования такого приема не требуется никакого изменения этих структур. Дело в том, что, в отличии от, например, языка Java, для имплементации интерфейса структуре не обязательно явно указывать список реализуемых интерфейсов. Компилятор способен сам определять, подходит ли структура под выбранный интерфейс. В случае неудачи при компиляции будет выведена ошибка.

Теперь можно перейти непосредственно к методам удаления. Принцип их работы прост: в рамках одной транзакции последовательно удалять все необходимые таблицы. Рассмотрим, например, последовательность действий при удалении таблицы `Course`:

1. удалить курс и получить список суррогатных ключей затронутых списков литературы,
2. по суррогатным ключам удалить списки литературы, получить список

суррогатных ключей для связи многие-ко-многим с учебниками,

3. по суррогатным ключам удалить все связи многие-ко-многим.

Аналогично будет происходить со всеми остальными удалениями.

Последней проблемой в этой части приложения стала реализация обновления. Самым логичным поведением тут является реализация политики «при `insert` добавить новую запись, если записи с таким первичным ключом нет, обновить в ином случае». Это позволяет свести любые обновления к добавлениям, что уменьшает количество кода и на сервере, и на клиентах. Но возникает проблема с тем, что такая возможность реализована во всех СУБД немного по разному. Например, в SQLite для таких целей есть отдельная команда `replace`. В PostgreSQL есть конструкция «ON CONFLICT». Добавление в таблицу `Lecturer` с учетом этой конструкции можно увидеть в листинге 2.

```
INSERT INTO schema.lecturer(
    lecturer_name,
    lecturer_date_of_birth,
    lecturer_department_id,
    lecturer_timestamp
    )
    VALUES ($1, $2, $3, $4)
        ON CONFLICT(lecturer_name, lecturer_date_of_birth)
        DO UPDATE SET
            lecturer_department_id=
                EXCLUDED.lecturer_department_id,
            lecturer_timestamp=
                EXCLUDED.lecturer_timestamp,
            lecturer_is_deleted=
                FALSE;
```

Листинг 2: SQL запрос добавления записи в таблицу Lecturer

При отладке этой части программы была обнаружена одна особенность языка Go, которая сильно усложняет тестирование работоспособности приложения. Дело в том, что тип error, являющийся стандартным типом для

ошибок, не предоставляет трассировку стека. Другими словами, по ошибке невозможно понять, в каком месте программы она была вызвана в случае, если обработка этой ошибки делегируется вызывающему методу. Дело в том, что тип error, которым представляются все ошибки в языке Go, является всего лишь интерфейсом с одним методом `Error() string`, возвращающим текст ошибки. Но, с другой стороны, такой подход позволяет реализовывать собственные типы ошибок, что и было сделано данном приложении. Ошибка представляется типом `Error`, который можно увидеть в листинге 3:

```
type Error struct {
        Message       string
        StackTrace    string
        DatabaseError *pq.Error
}
```

Листинг 3: Стурктура Error.

Перечислим назначение каждого поля.

1. `Message` – текстовое сообщение об ошибке,
2. `StackTrace` – трассировка ошибки, созданная на основе метода `runtime.Stack` пакета `runtime`,
3. `DatabaseError` – указатель на ошибку драйвера `pq`, работающего с PostgreSQL. Указатель равен `nil` в случае, если ошибка произошла не в драйвере.

Теперь следует описать принципы работы программного интерфейса. В разработанном приложении используется HTTP-сервер из стандартной библиотеки языка Go. Использовать этот пакет очень просто: необходимо добавить необходимый метод с нужной сигнатурой и URL-строку в сервер с помощью метода `http.HandleFunc`. В этой сигнатуре функция должна принимать два аргумента: `http.ResponseWriter`, который предназначен для отправки ответа клиенту и `http.Request`, содержащий информацию о поступившем запросе. В приложении на каждый запрос первым делом выполняется проверка входных данных, если таковые имеются. По большей части это проверка на наличие тела запроса, если оно должно присутствовать.

Также при обработке каждого запроса в ответ добавляется заголовок `Access-Control-Allow-Origin`, который необходим для того, чтобы при использовании этого API браузером все работало корректно. Без этого хэдера браузеры просто блокируют запросы к серверу. Согласно документации компании «Mozilla» [6], этот заголовок позволяет установить, с какими сайтами сервер умеет работать, а с какими не умеет. Например, в него можно записать `https://developer.mozilla.org`, что означает, что сервер умеет работать только с одним этим сайтом. Если значение будет `*`, тогда сервер будет работать с абсолютно любым клиентом.

Для каждой таблицы имеется следующий набор методов:

1. получение содержимого таблицы,
2. добавление записи в таблицу,
3. удаление из таблицы,
4. получение прототипа JSON-файла для записи таблицы.

Также сервер имеет три дополнительных метода помимо описанной структуры. Первый метод выполняет миграцию списка литературы с одного года на другой. Второй нужен для получения прототипа JSON-файла миграции, а третий выполняет генерацию отчета и записывает его на клиент в виде текстового файла. В случае веб-версии происходит создание текстового файла в формате `.bib`, который сохраняется в загрузки.

Тестирование

Тестирование выполнялось путем проверки работоспособности приложения на каждом этапе его использования.

Так как в проекте предусмотрено два различных и параллельных друг другу способа взаимодействия с приложением, ниже будет рассмотрен каждый из них. Для удобства работа с приложением командной строки будет приведена в листингах, а с веб-версией – снимками экрана. Причем в листингах знаком ">" будет обозначаться введенная bash-команда.

В качестве примера будет рассмотрен курс «Базы данных», преподаваемый на кафедре «Теоретическая информатика и компьютерные технологии» Вишняковым Игорем Эдуардовичем.

Первым делом необходимо создать кафедру, на которой работает пользователь.

Необходимые действия, которые нужно сделать в приложении командной строки, приведены в листинге 4.

```
> cli department prototype
Open department.txt and fill prototype struct with
correct data
> nano department.txt
> cli department add
```

Листинг 4: Добавление кафедры

В файле department.txt будет находиться JSON-объект, для которого будет необходимо заполнить поле `title` необходимым значением.

В случае работы с веб-приложением нужно выбрать необходимую таблицу, после чего в нее автоматически загрузится прототип искомой JSON-структуры, который следует модифицировать и отправить на сервер кнопкой `Upload`, расположенной справа снизу от поля ввода текста. Иллюстрацию можно видеть на рисунке 6.

Следующее действие – добавление лектора к только что введенной кафедре. Последовательность действий ничем не отличается от предыдущего шага и ее можно увидеть в листинге 5 и рисунке 7.

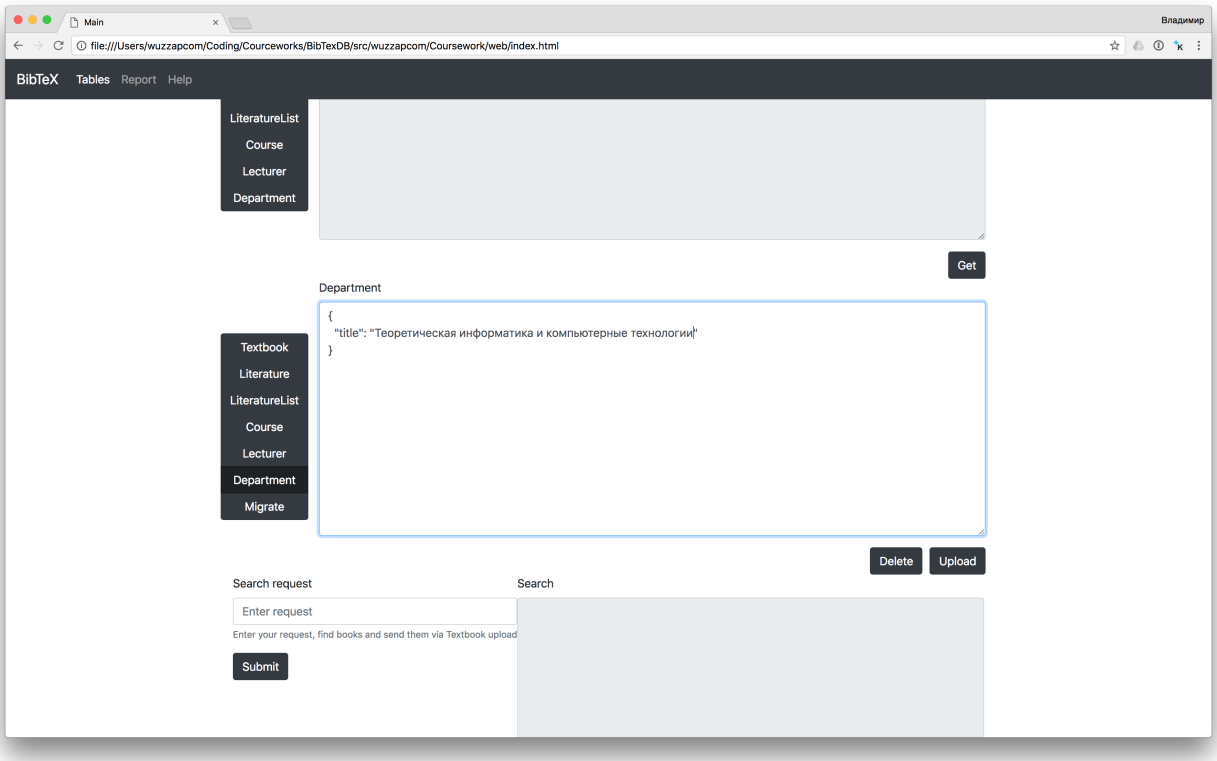

Рисунок 6: Добавление новой кафедры в веб-приложении

```
> cli lecturer prototype
Open lecturer.txt and fill prototype struct with correct data
> nano lecturer.txt
> cli lecturer add
```

Листинг 5: Добавление лектора

На шаге добавления курса становится видно преимущество веб-версии по сравнению с приложением командной строки. Оно позволяет в двух разных окнах вводить информацию и делать запросы к базе данных. Иллюстрацию можно видеть на рисунке 8. В этом примере можно держать перед глазами список добавленных лекторов чтобы, например, не забыть дату рождения, и одновременно вводить данные об учебном курсе.

В то время как в приложении командной строки такой возможности нет. Как вариант, можно отдельно запросить список лекторов командой `cli lector get`, чтобы открыть его в отдельном окне терминала или текстовом редакторе, но этот вариант явно проигрывает по удобству. Так что последовательность команд остается аналогичной и приведена в листинге 6.

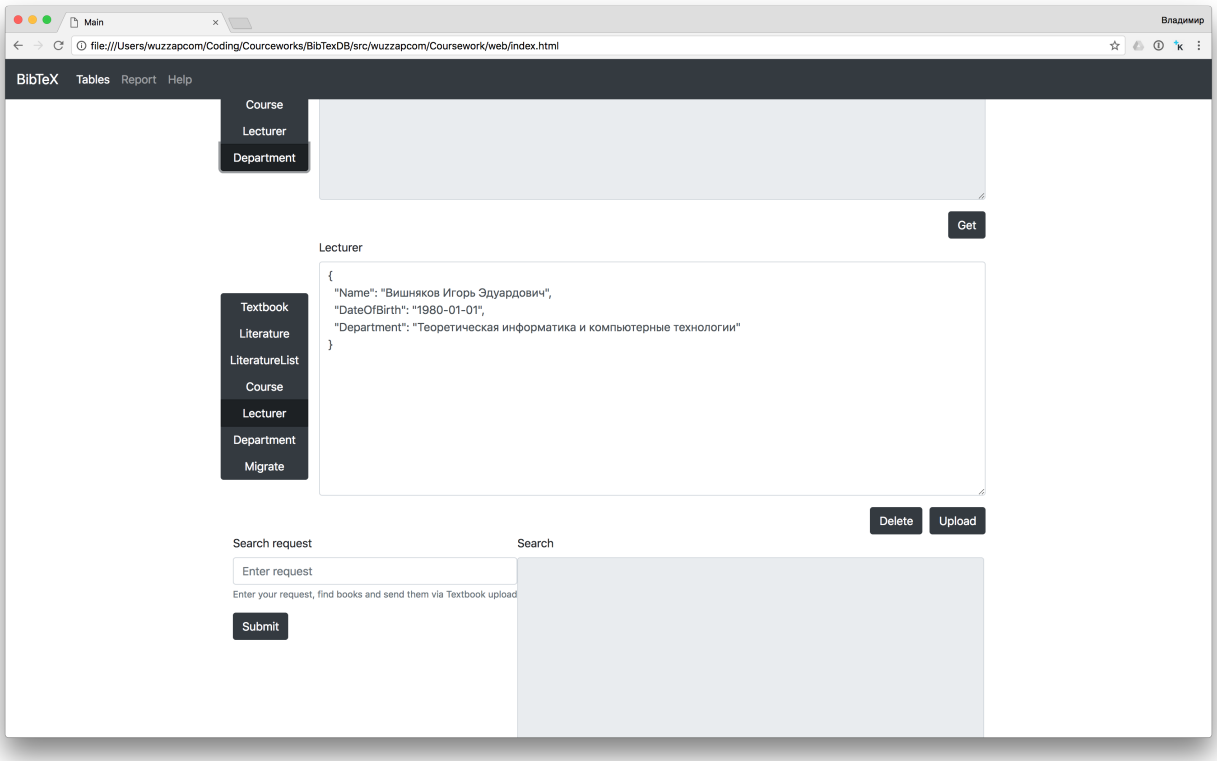

Рисунок 7: Добавление нового лектора в веб-приложении

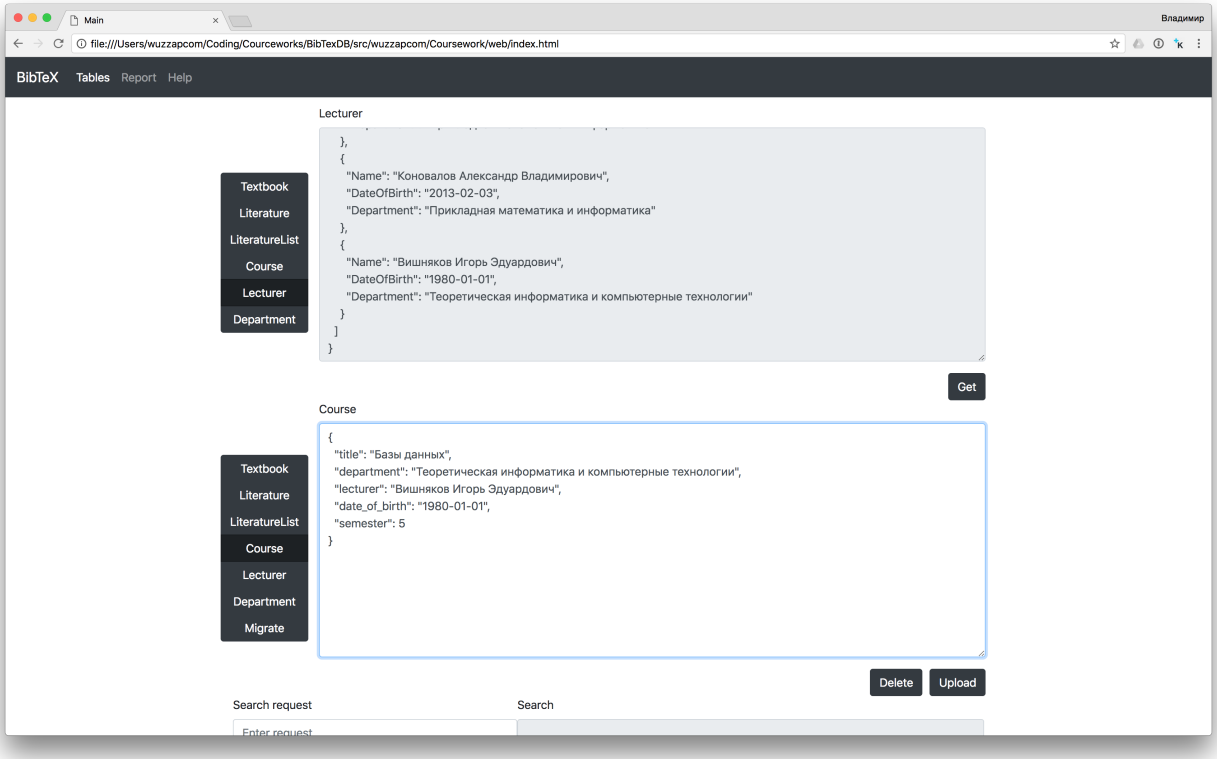

Рисунок 8: Добавление учебного курса в веб-приложении

```
> cli course prototype
Open course.txt and fill prototype struct with correct data
> nano course.txt
> cli course add
```

Листинг 6: Добавление лектора

Этап добавления нового учебного курса ничем не отличается от всех предыдущих, так что имеет смысл перейти к добавлению книг в приложение. Как уже было сказано выше, для этого предусмотрен модуль, позволяющий использовать сервис Google Books для поиска информации о книгах. Но возможно и добавление книг вручную, если нужного учебника не нашлось в базе сервиса. Для начала рассмотрим эту функцию в приложении командной строки из листинга 7.

```
> cli search —request="SQL"
Open searchResults.txt, view results, remove wrong
items and fix incorrect data.
> nano searchResults.txt
> cli book prototype
Open book.txt and fill prototype struct with correct data
> nano book.txt
> cli book add
```

Листинг 7: Поиск книг по запросу SQL в сервисе Google Books в приложении командной строки.

Другими словами, необходимо сначала выполнить поиск, получить результат в файл, после чего скопировать нужную книгу в `book.txt` и добавить ее командой `cli book add`.

Схожий принцип, но более удобный, используется и в веб-приложении, что можно увидеть на рисунке 9.

Добавление книги в список литературы также является набором тех же самых действий, так что имеет смысл перейти сразу к дополнительной функции: миграции списков литературы. По своей сути, эта функция является абсолютно тем же заполнением нужной JSON-структуры, так что будет удобно привести только вариант веб-приложения. Его можно видеть

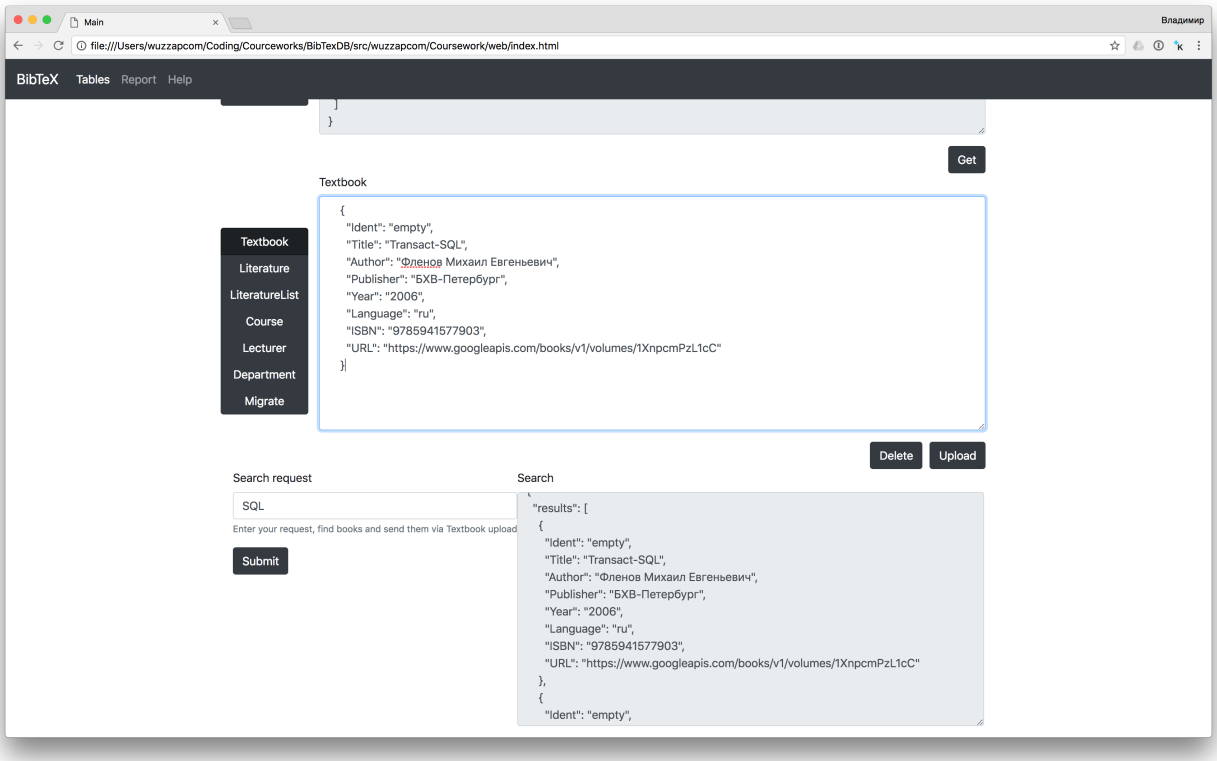

Рисунок 9: Поиск книг по запросу SQL в сервисе Google Books в веб-приложении.

на рисунке 10

Важной деталью, которую необходимо отметить и зафиксировать, является то, что необходимо создать заранее тот список литературы, в который будет производиться миграция.

Наконец стало возможным перейти к основному и самому важному пункту – генерации самого отчета.

Для начала рассмотрим, как происходит генерация в приложении командной строки. Запрос необходимого отчета происходит через отправку на сервер нужного списка литературы. Таким образом, запросить отчет можно последовательностью команд из листинга 8.

```
> cli literatureList prototype
Open literatureList.txt and fill prototype struct
with correct data
> nano literatureList.txt
```

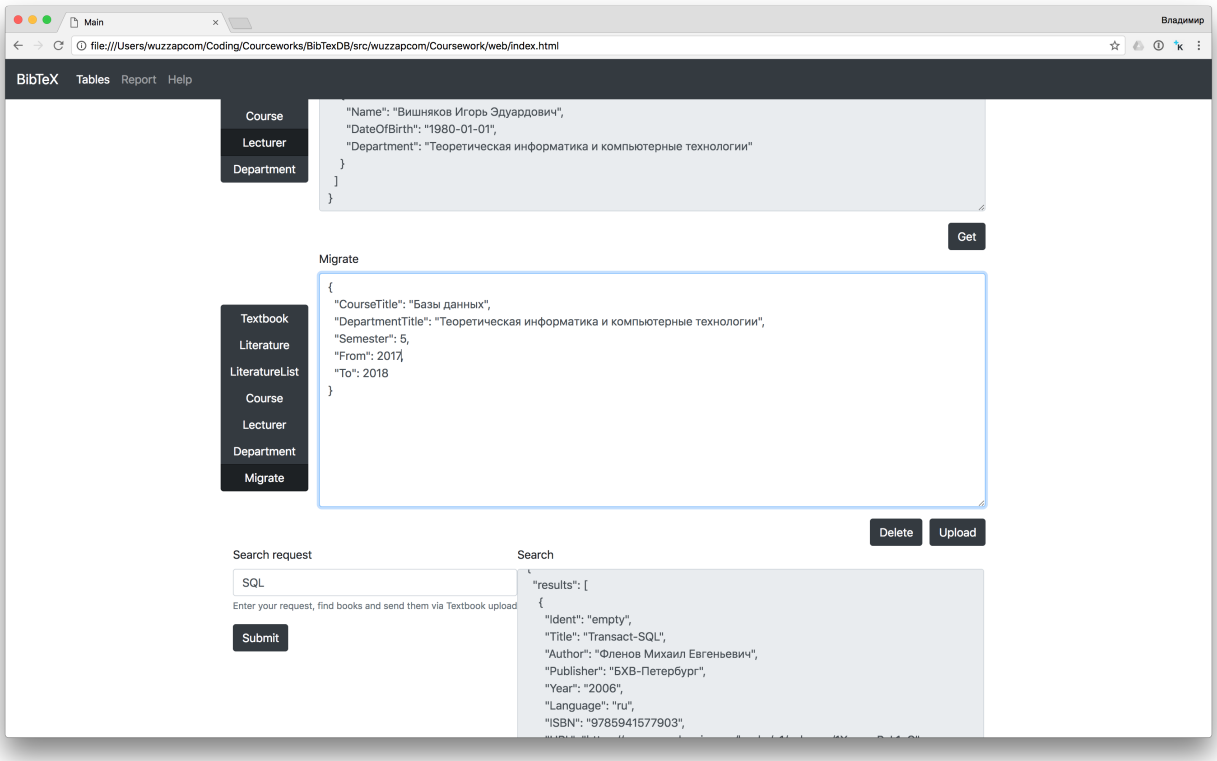

Рисунок 10: Демонстрация миграции списка литературы в веб-приложении.

```
> cli report --outputFile="report.bib"
```

Листинг 8: Запрос отчета в приложении командной строки.

Как можно видеть, в команде `cli report` явным образом указан файл, в который будет записан результат. В случае отсутствия этого флага отчет будет выведен в стандартный поток вывода, о чем сказано в справке данной команды.

Намного более удобно устроен весь процесс в веб-приложении. Специально для этого в нем сделано отдельное окно `Report`. В нем отображаются все списки литературы, по нажатию на которые браузером сохраняется текстовый файл `report.bib`. Рисунок 11 демонстрирует данную функцию.

В качестве заключения хочется отметить, что работоспособность веб-приложения была проверена в браузерах Safari и Chrome на ноутбуке под управлением операционной системы MacOS.

Таким образом, в данном разделе был рассмотрен полный цикл использования разработанного приложения, а также был продемонстрирован

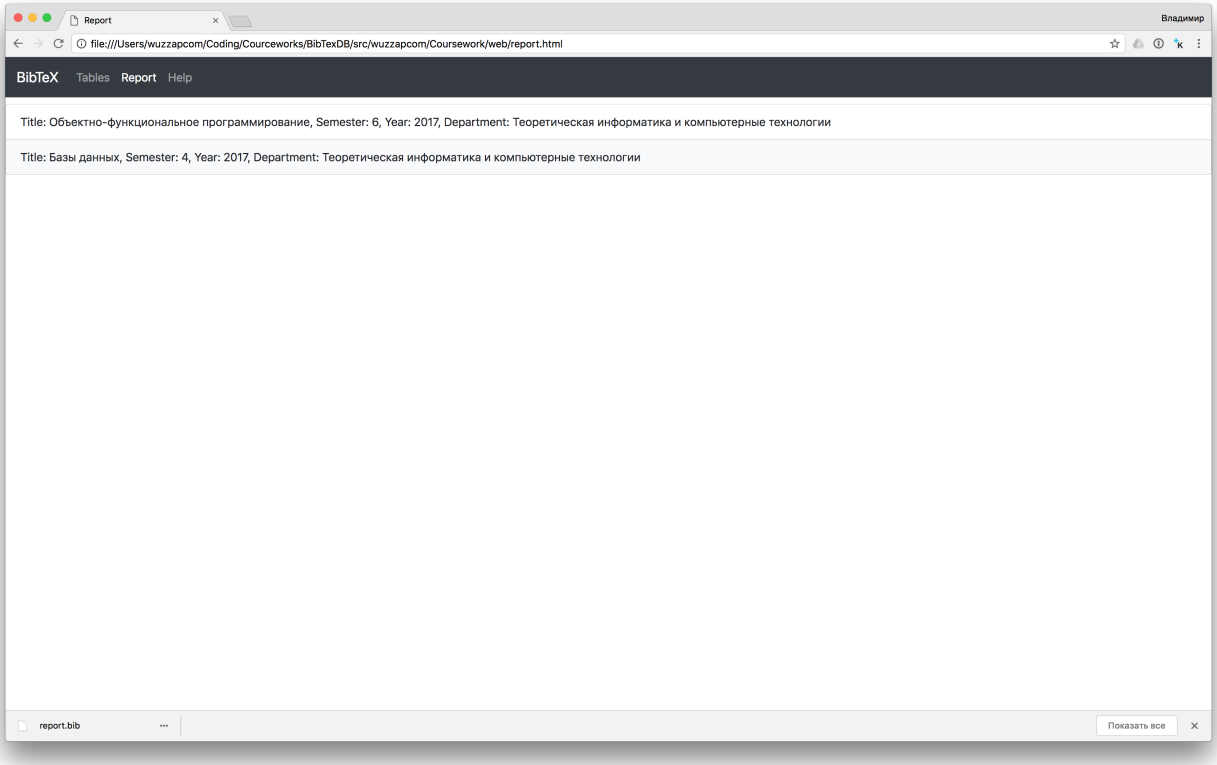

Рисунок 11: Демонстрация генерации отчета в веб-приложении.

принцип работы двух клиентских программ: веб-сайта и приложения командной строки. По результатам данного тестирования можно сказать, что приложение командной строки выигрывает в автоматизируемости, но является неудобным при использовании человеком, в то время как веб-сайт, наоборот, предоставляет пользователю дополнительные функции для улучшения качества работы с разработанным приложением.

Заключение

В ходе данной курсовой работы был выполнен анализ и сравнение двух различных ветвей развития систем управления базами данных: SQL и NoSQL, была произведена разработка серверного REST-приложения с HTTP-интерфейсом.

Структура разработанного приложения выглядит следующим образом:

1. База данных PostreSQL.
2. Серверное приложение на языке Go с использованием драйвера для PostreSQL.
3. Веб-сайт на языке TypeScript с использованием фреймворка Bootstrap.
4. Приложение командной строки на языке Go с использованием фреймворка Cobra.

Таким образом, было разработано приложение, выполняющее хранение и обработку информации о списках литературы учебных курсов и была продемонстрирована его работоспособность на примере курса под названием «Базы данных».

Список использованной литературы

[1] Стандарт RFC 2616 протокола HTTP[Электрон. ресурс] // Режим доступа: https://tools.ietf.org/html/rfc2616, свободный. – Загл. с экрана.(Дата обращения 16.06.2018)

[2] Р.Т. Филдинг. Representational state transfer[Электрон. ресурс] // Режим доступа: http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htmб свободный. – Загл. с экрана.(Дата обращения 22.05.2018).

[3] М. Макдональд. Веб-разработка. Исчерпывающее руководство – Санкт-Петербург: Питер, 2017. - 638 с.

[4] V. Gaurav. Getting Started with NoSQL - Birmingham: Packt Publishing Ltd, 2013. - 118 с.

[5] Документация СУБД MongoDB[Электрон. ресурс] // Режим доступа: https://docs.mongodb.com, свободный. – Загл. с экрана.(Дата обращения 16.06.2018).

[6] Документация для заголовка Access-Control-Allow-Origin[Электрон. ресурс] // Режим доступа: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Access-Control-Allow-Origin, свободный. – Загл. с экрана.(Дата обращения 22.05.2018).

[7] Документация языка программирования TypeScript[Электрон. ресурс] // Режим доступа: https://www.typescriptlang.org/docs/home.html, свободный. – Загл. с экрана.(Дата обращения 16.06.2018).