МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ «ВИТЕБСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ П.М.МАШЕРОВА»

Факультет математики и информационных технологий

Кафедра математики

КУРСОВОЙ ПРОЕКТ

По дисциплине «Администрирование Linux»

РАЗРАБОТКА ПРИЛОЖЕНИЯ С ИСПОЛЬЗОВАНИЕМ СТЕКА MERN

Беляев Максим Сергеевич.

4 курс, 41 группа

Руководитель:

Александрович Татьяна Алиевна,

старший преподаватель кафедры

математики

Витебск, 2022

**Реферат**

Курсовая работа 24 страницы, 12 источников.

*REACTJS, PYTHON, DOCKER, NODEJS, EXPRESSJS, MONGOOSE (MONGODB), СИСТЕМА УПРАВЛЕНИЯ БАЗАМИ ДАННЫХ.*

Объект исследования – *fullstack-*приложение.

Предмет исследования – изучение языка программирования общего назначения *NodeJS, фреймворка ExpressJS, ODM-*библиотека *Mongoose (MongoDB),* библиотека *ReactJS.*

Цель работы – изучить методологию и технологии разработки *web*- сервера, и создать логическую структуру *web*-сервиса.

Методы исследования: описательно-аналитические, сравнительные.

Научная новизна: использование современных средств и методов разработки *fullstack*-приложений.

Теоретическая и практическая значимость: получение практического опыта в разработке, реализация бесплатного неограниченного доступа к медиа-ресурсам в сети интернет.

**Содержание**

[Введение 4](#_Toc122279861)

[2 Технологии разработки клиентского приложения 7](#_Toc122279862)

[3 Технологии разработки сервера 8](#_Toc122279863)

[4 База данных 11](#_Toc122279864)

[5 Разработка сервера 13](#_Toc122279865)

[5.1 Запуск и подключение к базе данных. 13](#_Toc122279866)

[6 Создание клиентского приложения 19](#_Toc122279875)

[6.1 Структура проекта. 19](#_Toc122279876)

[6.2 Создание компонента. 19](#_Toc122279881)

[Заключение 23](#_Toc122279888)

[Список использованных источников 24](#_Toc122279889)

# Введение

В настоящее время практически каждый сервис по предоставлению доступа к медиа-ресурсам в сети интернет использует монетизацию. Предоставление услуг доступа зачастую осуществляется по подписке. Нередко предлагаются бесплатные варианты доступа, которые влекут за собой некоторые ограничения.

Рассмотрим на примере сервис социальной сети *Вконтакте*, который позволяет прослушивать музыку пользователям, находящимся в сети. На мобильных устройствах существует лимит по времени прослушивания. Преодолеть данный лимит возможно используя сторонние приложения.

Целью данной курсовой работы является разработка   
*fullstack*-приложения, которое будет предоставлять неограниченный доступ к медиа-ресурсам с *web-*платформ с открытым *API*

Для достижения данной цели необходимо решить следующие задачи:

* изучить технологии и методы разработки *fullstack*-приложения;
* разработать прототип клиентского приложения;
* спроектировать базу данных;
* разработать сервер для работы с БД;
* организовать взаимодействие сервера с *API* *web*-сервиса.

Объект исследования – *fullstack-*приложение.

Предмет исследования – изучение языка программирования общего назначения *NodeJS, фреймворка ExpressJS, ODM-*библиотека *Mongoose (MongoDB),* библиотека *ReactJS.*

Теоретическая и практическая значимость: получение практического опыта в разработке, реализация бесплатного неограниченного доступа к медиа-ресурсам в сети интернет.

* + 1. **Проектирование схемы взаимодействия компонентов приложения**
       1. **Взаимодействие компонентов.**

Перед началом проектирования схемы взаимодействия компонентов необходимо определить принцип работы всех частей приложения. Приложение будет разделено на 4 части:

* Клиентское приложения
* Сервер для работы с БД
* База данных
* Сервер хранения и обработки медиа-файлов

Рассмотрим схему взаимодействия модулей.

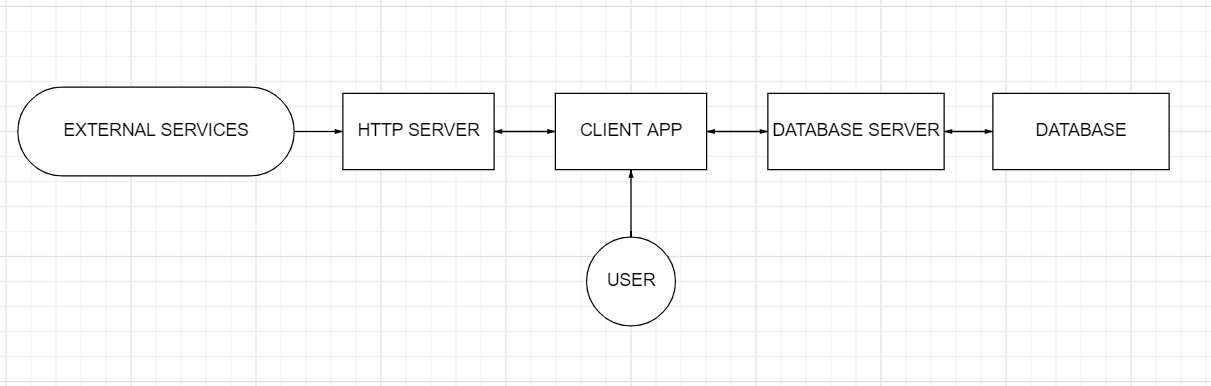


Рисунок 2.1 – Схема взаимодействия компонентов приложения

Клиентское приложение *(CLIENT APP)* воспроизводит медиа-файлы на устройствах пользователей, позволяет осуществлять управление пользовательским интерфейсом. Может размещаться на устройствах пользователей, не хранит информации о пользователях или медиа-файлах.

Сервер для работы с БД *(DATABASE SERVER)* взаимодействует непосредственно с клиентским приложением. В нём хранятся данные о медиа-файлах, служебная информация, информация о пользователях, взаимодействующих с клиентским приложением.

База данных *(DATABASE)* хранит записи о треках, плейлистах, данных авторизации, логах системы, настройках сервера.

Ввиду того, что БД не способна хранить и обрабатывать медиа-файлы появляется необходимость в разработке отдельного сервера *(HTTP SERVER)*, который будет хранить, обрабатывать, получать данные файлы, взаимодействуя со сторонними сервисами *(EXTERNAL SERVICES)*. В целях безопасности и сохранности данных пользователей, принято решение вынести данный модуль отдельный блок.

К серверу обработки и хранения файлов имеет доступ только клиентское приложение, что обеспечивает дополнительный слой защиты данных, хранящихся в БД. Ввиду того, что данный сервер напрямую взаимодействует с внешними ресурсами, доступ к серверу БД из этого модуля также следует ограничить.

# Технологии разработки клиентского приложения

Перед проектированием интерфейса приложения необходимо определить задачи, которые будет решать конечный продукт. Определить минимальные требования и инструменты разработки.

Клиентское приложение служит для воспроизведения медиа-контента на устройствах пользователей. Перед написанием данной курсовой работы принято решение написать в составе приложения *mp3-*плеер, способный воспроизводить треки, получаемые с сервера разрабатываемого приложения.

Приложение должно быть работоспособно с персональных компьютеров и мобильных устройств. Для решения данной задачи в качестве средства разработки была выбрана библиотека *ReactJS.*

Библиотека *ReactJS* использует компонентный подход, что позволяет упростить разработку, путём переиспользования созданных компонентов.

Данная библиотека, в отличие от *Angular*, не является компилируемой, что упрощает взаимодействие с разрабатываем продуктом. Также *React* использует функциональный подход, избавляя от необходимости использовать классы, что в конечном итоге не приводит к усложнению кода при разработке небольших или средних проектов.

Для разработки приложения не используются библиотеки управления состоянием объектов *(Redux, Mobx),* т.к. изучение дополнительных библиотек экспоненциально увеличивает время разработки проекта.

В перспективе, разработку мобильных версий приложения можно осуществить с помощью библиотеки *React Native,* позволяющей создавать приложения с использованием средств *frontend*-разработки.

# 3 Технологии разработки сервера

**3.1 Серверный язык программирования.**

В качестве серверного языка программирования выбран язык общего назначения *NodeJS.*

Преимущества:

***Асинхронность***. Все *API*-интерфейсы библиотеки *NodeJS* являются асинхронными, то есть не блокирующими загрузку. По сути, это означает, что сервер на основе *NodeJS* никогда не ожидает возврата данных от *API*. После вызова он переходит к следующему *API*, а механизм уведомлений *NodeJS* *Events* помогает серверу получить ответ от предыдущего вызова;

***Скорость***. Это одна из основных привлекательных особенностей. *JavaScript*-код, выполняемый в среде *NodeJS*, может быть в два раза быстрее, чем код, написанный на компилируемых языках, вроде *C* или *Java*, и на порядок быстрее интерпретируемых языков, наподобие *Python* или *Ruby*. Причиной подобного является неблокирующая архитектура платформы, а конкретные результаты зависят от используемых тестов производительности, но, в целом, *NodeJS* – это очень быстрая платформа;

***Простота***. *NodeJS* обладает достаточно простым синтаксисом, с помощью которого можно создавать легко масштабируемые приложения.

Использование в качестве клиентского и серверного взаимодействия средств разработки единого языка для всего приложения *(JavaScript)* так же способствует упрощению разработки.

* 1. **Используемые библиотеки.**

*NodeJS* имеет достаточно обширный набор библиотек. Для увеличения общей функциональности языка, в контексте разработки данного приложения используются следующие библиотеки:

* *Mongoose*
* *Express*
* *Cors*
* *Node-config*
* *BcrypsJS*
* *Jsonwebtoken*
* *Express-validator*

*Mongoose* является *ODM*-библиотекой, которая используется в качестве взаимодействия сервера с базой данных. Использование данной библиотеки позволяет избежать необходимости написания запросов к БД для работы с данными. Все манипуляции с данными, в таком случае, производятся средствами *NodeJS*.

*Express –* это[фреймворк *web*-приложений](https://ru.wikipedia.org/wiki/%D0%A4%D1%80%D0%B5%D0%B9%D0%BC%D0%B2%D0%BE%D1%80%D0%BA) для *NodeJS*, реализованный как [свободное и открытое программное обеспечение](https://ru.wikipedia.org/wiki/%D0%A1%D0%B2%D0%BE%D0%B1%D0%BE%D0%B4%D0%BD%D0%BE%D0%B5_%D0%B8_%D0%BE%D1%82%D0%BA%D1%80%D1%8B%D1%82%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5) под [лицензией *MIT*](https://ru.wikipedia.org/wiki/%D0%9B%D0%B8%D1%86%D0%B5%D0%BD%D0%B7%D0%B8%D1%8F_MIT). Он спроектирован для создания [веб-приложений](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D0%BF%D1%80%D0%B8%D0%BB%D0%BE%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5) и [*API*](https://ru.wikipedia.org/wiki/API)*.*

***CORS (Cross-Origin Resource Sharing****)* - это механизм, основанный на *HTTP*-заголовке, реализованный браузером, который позволяет серверу или *API* указывать любые источники (отличающиеся с точки зрения протокола, имени хоста или порта), отличные от источника, из которого неизвестный источник получает разрешение на доступ и загрузку ресурсов. Пакет ***CORS***, доступный в реестре *npm*, используется для устранения ошибок *CORS* в приложении *NodeJS*.

*Node-config* организует иерархические конфигурации для развертывания приложения. Это позволяет вам определить набор параметров по умолчанию и расширить их для различных сред развертывания (разработка, контроль качества, промежуточный, производственный и т. Д.). Конфигурации хранятся в [файлах конфигурации](https://github.com/node-config/node-config/wiki/Configuration-Files) в приложении и могут быть переопределены и расширены с помощью [переменных среды](https://github.com/lorenwest/node-config/wiki/Environment-Variables), [параметров командной строки](https://github.com/node-config/node-config/wiki/Command-Line-Overrides) или [внешних источников](https://github.com/lorenwest/node-config/wiki/Configuring-from-an-External-Source).

*BcrypsJS.* При отправке формы содержатся некоторые конфиденциальные данные (например, пароли), которые не должны быть видны никому, даже администратору базы данных. Чтобы конфиденциальные данные были защищены, в процессе разработки использована библиотека *bcrypsjs.* Этот модуль позволяет хранить пароли в виде хэшированных паролей вместо открытого текста.

*Jsonwebtoken – э*то открытый стандарт для безопасной передачи данных внутри сторон с использованием объекта JSON. JWT используется для механизмов аутентификации без состояния для пользователей и поставщиков, это означает, что сеанс поддерживается на стороне клиента вместо хранения сеансов на сервере.

*Express-validator.* При работе с формами, отправляющими пользовательские данные, перед сохранением данных их необходимо проверить и очистить:

Проверка позволяет утвердить, что введённые значения являются подходящими для каждого поля (расположены в правильном диапазоне, формат и т.д.) и что значения были предоставлены для всех обязательных полей.

Очистка удаляет или заменяет символы в данных, которые потенциально могут использоваться для отправки вредоносного содержимого на сервер.

# 4 База данных

В качестве системы управления базами данных выбрана *NoSQL DB* – *MongoDB.*

*MongoDB* — документоориентированная система управления базами данных с открытым исходным кодом. Для хранения данных используется *JSON*-подобный формат.

*NoSQL*-базы стали популярными благодаря быстродействию и хорошей масштабируемости

Преимущества *MongoDB:*

***Документоориентированная база*** — сохранение данных в формате документов вместо формата реляционного типа, это делает *MongoDB* очень гибкой и адаптируемой к бизнес-требованиям. Возможность хранения разных типов данных особенно важна при работе с большими данными, которые собираются из разных источников и не ложатся в одну структуру.

***Специальные запросы*** — *MongoDB* поддерживает поиск по полям, диапазонные запросы и поиск по регулярным выражениям. Могут быть сделаны запросы для возврата определенных полей в документах.

***Индексация*** — можно создать индексы для улучшения производительности поиска в *MongoDB*. Любое поле в документе может быть проиндексировано. Это обеспечивает высокую скорость работы *СУБД*.

***Репликация*** — эта *СУБД* может обеспечить высокую доступность с помощью наборов реплик. Набор реплик состоит из двух или более экземпляров *MongoDB*. Каждая реплика набора может выступать в роли первичной или вторичной. Первичная реплика — главный сервер, который взаимодействует с клиентом и выполняет все операции чтения/записи. Вторичные реплики сохраняют копию данных первичной реплики с помощью встроенной репликации. Если с первичной репликой что-то случилось, происходит автоматическое переключение на вторичную реплику, затем она становится основным сервером.

***Балансировка нагрузки*** — *MongoDB* использует концепцию шардинга для горизонтального масштабирования с помощью разделения данных между несколькими экземплярами БД. Она может работать на нескольких серверах, балансируя нагрузку и/или дублируя данные, чтобы поддерживать работоспособность системы в случае аппаратного сбоя.

***Доступность*** — *MongoDB* поддерживает все популярные языки программирования, ее можно использовать бесплатно как open source решение.

# 5 Разработка сервера

# 5.1 Запуск и подключение к базе данных.

# Первым этапом разработки будет разработка *REST* *API* *(Representational State Transfer)* сервиса. Необходимо создать *web*-сервер, способный принимать и отвечать на множественные *http*-запросы за единицу времени.

# Перед началом работы следует импортировать модуль *express* и создать экземпляр сервера.

# Листинг 5.1:

|  |
| --- |
| const express = require('express') … const app = express() |

# Для работы сервера необходимо создать асинхронную функцию, которая будет принимать *http*-запросы и передавать их экземпляру сервера.

# Листинг 5.2:

|  |
| --- |
| async function start() {      try {          const url = 'mongodb://127.0.0.1:27017/LynxDB'          mongoose.set('strictQuery', false)          await mongoose.connect(url, {useNewUrlParser: true})          app.listen(PORT,() => {              console.log(`Server is running on port ${PORT}`)          })      }      catch(err) {console.log(err)}  }  start() |

Создана функция *start(),* в которой запускается web-сервер на порту, указанном в переменной *PORT.*

В строке *mongoose.connect(…)* задаётся строка подключения к *СУБД.* [3]

Для работы данного модуля импортируется библиотека *mongoose*. Перед строкой подключения указывается ключевое слово *await,* что означает асинхронность операции, т.е. сервер не будет находиться в ожидании ответа от базы данных, и продолжит работу.

**5.2 Создание моделей.**

Ввиду того, что база данных хранит информацию о медиа-файлах, необходимо организовать взаимодействие сервера и объектов, хранящихся в БД. Это можно сделать при помощи выполнения запросов к базе данных.

Библиотека mongoose позволяет вызывать методы, выполняющие запросы к объектам БД. Для того чтобы сервер мог взаимодействовать с документами, хранящимися в коллекциях *MongoDB,* можно использовать модели – прототипы документов в базе данных.

# Листинг 5.3:

|  |
| --- |
| const {Schema, model} = require('mongoose');  const Tracks = new Schema({    track\_id: {      type: Number,    },    name: {      type: String,    },    artist: {      type: String,    },    img: {      type: String,    },    src: {      type: String,    }})  module.exports = model('Tracks', Tracks) |

Выше описан пример создания модели для сущности *tracks.*

Из библиотеки *mongose* импортируются объекты *Schema* и *model*.

Schema позволяет создать схему модели, с указанием полей, типов данных, дополнительных модификаторов.

Model служит для преобразования описанной схемы в объект модели и последующего экспортирования. Логика работы сервера, взаимодействие c другими компонентами путём выполнения *http*-запросов осуществляется на уровне работы с моделями.

**5.3 Создание маршрутов. *Router*.**

Для взаимодействия сервера с другими компонентами приложения следует обозначить способы и механизм взаимодействия с внешними программными интерфейсами. Этому будет способствовать объект *router*, который поставляется в библиотеке *express.* [4]

Данный объект позволяет создать маршруты, по которым внешние компоненты могут отправлять http-запрос. К каждому маршруту привязывается определённая логика, которая должна быть выполнена на сервере, либо передана на выполнение базе данных.

Рассмотрим маршрут */api/auth*/*register*, который будет принимать некоторые данные с клиентского приложения, а именно данные для регистрации – логин и пароль. После получения данных, их необходимо провалидировать – проверить на пригодность к записи в базу данных, проверить базу данных на наличие уже существующих идентичных записей, чтобы избежать дублирования данных. Перед записью необходимо захешировать пароль, чтобы не хранить его в открытом виде, что следует из соображений безопасности и сохранности личных данных пользователей.

Название (путь маршрута) задаётся произвольно. Впоследствии сам маршрут согласуется с клиентским приложением. [5]

# Листинг 5.4:

|  |
| --- |
| const Router = require('express')  const router = Router()  router.post(    '/register',    [      check('email', 'Email is uncorrect!').isEmail(),      check('password', 'Minimum password lenght = 8').isLength( {min : 8})    ],    async (req,res) => {    try {      /////////////////////////////////////////      //////// REGISTER VALIDATION      /////////////////////////////////////////      const errors = validationResult(req)      if (!errors.isEmpty()) {        return res.status(400).json({          errors: errors.array(),          message: 'Uncorrect register data!'        })      }      /////////////////////////////////////////      //////// USER EXISTING CHECK      /////////////////////////////////////////      const {email, password} = req.body      const candidate = await Users.findOne({email})      if(candidate) {        res.status(400).json({message : 'User already exists!' })      }      /////////////////////////////////////////      //////// PASSWORD HASHING      /////////////////////////////////////////      const hashed\_password = await bcrypt.hash(password, 12)      const User = new Users({email, password: hashed\_password})      await User.save()      res.status(201).json({message : 'Ok!'})    }catch (error) {res.status(500).json({message: "Something wrong!"})}}) |

Класс router экспортируется из *express*, создаётся экземпляр роутера. У объекта router вызывается встроенный метод *post(),* который будет срабатывать при отправке на сервер http-метода *POST.* В данном методе указывается сам маршрут, описывается исполняемая логика при срабатывании данного метода.

Блок *REGISTER VALIDATION* проверяет правильность полученных данных.

*USER EXISTING CHECK* проверяет существование пользователя в базе данных во избежание записи дубликата.

В блоке *PASSWORD HASHING* выполняется хеширование пароля перед записью в БД.

Далее создаётся один объект модели, в который записываются обработанные сервером данные, модель записывается в документ, расположенный в базе данных.

Таким образом осуществляется базовая реализация работы сервера, принимающего запросы и взаимодействующего с базой данных.

# 6 Создание клиентского приложения

# 6.1 Структура проекта.

# Использование библиотекой React компонентного подхода задаёт некоторую структуру проекта. Для облегчения масштабирования приложения, в процессе разработки принято решение разделить проект на 3 основные сущности. Первая – *views.* Компоненты, которые будут рендериться на устройствах пользователей.

# *Views* делятся на *pages* и *components*. В *pages* находятся компоненты, которые будут являться прототипами *web*-страниц. В *components* хранятся переиспользуемые компоненты, которые впоследствии будут частями *pages.*

# Вторая сущность – *hooks.* Это перегруженные функции состоящие из логики работы клиенского приложения, использующие React hooks, которые поставляются по умолчанию в React. В данной курсовой работе избегается рассмотрение React hooks ввиду объёмности данной темы.

# Последним блоком в проекте является *context*. Это шаблоны структур данных, которые будут передаваться между компонентами. Используется для синхронизации состояний данных для всего клиентского приложения.

# 6.2 Создание компонента.

# Ниже приведён пример создания компонента для страницы /login.

# Листинг 6.1:

|  |
| --- |
| import React, { useContext, useEffect, useState } from 'react'  import './login.scss'  import { useHttp } from '../../../hooks/http.hook'  import { useMessage } from '../../../hooks/message.hook'  import { authContext } from '../../../context/auth\_context'  export const LoginPage = () => {    const message = useMessage()    const auth = useContext(authContext)    const { loading, request, error, clear\_error } = useHttp()    const [ form, set\_form ] = useState({ email: '', password: '' })    const change\_handler = event => {      set\_form({ ...form, [event.target.name]: event.target.value })    }    ///////////// ERRORS VALIDATION /////////////    useEffect( () => {      message(error)      clear\_error()    }, [ error, message, clear\_error ])      const register\_handler = async () => {      try {        const data = await request('/api/auth/register', 'POST', {...form})        console.log('DATA', data)      } catch (error) { }    }    const login\_handler = async () => {      try {        const data = await request('/api/auth/login', 'POST', {...form})        auth.login(data.token, data.id)      } catch (error) { }    } |

# Сперва импортируются необходимые модули. Это делается перед написанием компонента, чтобы избежать блокировки работы компонента.

# Далее создаётся сам компонент, который экспортируется из файла. Внутри компонента описывается логика работы блоков данного компонента. Здесь вызываются используемые hooks и context. Логика работы страницы *login* предусматривает получение данных от пользователя, их валидацию и последующая отправка на сервер по маршруту */api/auth/login*. Получаемые сервером данные будут обработаты, сверены с данными в базе данных, и в качестве ответа на устройство клиента вернётся токен, который в свою очередь будет хранить время сессии, для избежания повторной авторизации после перезагрузки страницы клиентского приложения.

# После написания логики описывается сам объект рендеринга. Это единственный *html*-блок, содержащий вложенные блоки. Внутри него так же указываются функции *handler*, которые вызываются в случае воздействия пользователем на привязанный *html*-блок.

# Листинг 6.2:

|  |
| --- |
| return(      <div id="login\_box">        <div className="login\_form">          <span id="login\_title">LYNX<br/></span>          <input  className="login\_content" type="text" name="email" placeholder="Username" onChange={change\_handler} required/>          <input className="login\_content" type="password" name="password" placeholder="Password" onChange={change\_handler} required/>          <button className="login\_content" id="login\_btn" type="submit" value="Login" onClick={login\_handler}>Login</button>          <button className="login\_content" id="login\_btn" type="submit" value="Register" onClick={register\_handler} disabled={loading}>Register</button>        </div>      </div>    )} |

# Заключение

В результате работы были выбраны и изучены технологии разработки *web*-приложения, такие как *NodeJS, фреймворка ExpressJS, ODM-*библиотеки *Mongoose (MongoDB).*

По разработанным моделям созданы документы в базе данных.

Создан *web*-сервер, реализовано взаимодействие *web*-сервера с базой данных.

Создано клиентское приложение, реализован механизм воспроизведения медиа-файлов формата *mp3*. Реализовано взаимодействие клиентского приложения с вторичным http-сервером.

Все поставленные цели и задачи достигнуты.

# Список использованных источников

1. Голицына О. Л. Основы проектирования баз данных; Форум - Москва, 2012. - 416 c.
2. Free Database Designer – [Электронный ресурс] – Режим доступа: <https://app.dbdesigner.net/>

Дата доступа: 12.02.2022

1. Mongoose– [Электронный ресурс] – Режим доступа: <https://mongoosejs.com/docs/guide.html>

Дата доступа: 12.12.2022

1. Web-разработка – [Электронный ресурс] – Режим доступа: <https://www.youtube.com/c/VladilenMinin>

Дата доступа: 10.12.2022

1. NodeJS – [Электронный ресурс] – Режим доступа: <https://nodejs.org/docs/latest-v17.x/api/>

Дата доступа: 12.12.22