|  |  |  |
| --- | --- | --- |
| **МИНИСТЕРСТВО НА ОБРАЗОВАНИЕТО И НАУКАТА** | | |
| **ПРОФЕСИОНАЛНА ГИМНАЗИЯ ПО ЕЛЕКТРОТЕХНИКА И ЕЛЕКТРОНИКА**  бул. Пещерско шосе № 26  4002 гр. Пловдив, България  тел. 032 / 643-657  info-1690174@edu.mon.bg |  | **VOCATIONAL SCHOOL OF ELECTRICAL ENGINEERING AND ELECTRONICS**  26 Peshtersko chaussee blvd.,  4002 Plovdiv, Bulgaria  Phone: 032 / 643-657  info-1690174@edu.mon.bg |

**професия код 481030 „Приложен програмист“**

**специалност код 4810301 „Приложно програмиране“**

**ДИПЛОМЕН ПРОЕКТ**

**за придобиване трета степен професионална квалификация**

**Тема: УЕБ ПРИЛОЖЕНИЕ ЗА РАЗГОВОРИ И ОБМЯНА НА СЪОБЩЕНИя**

**Дипломант**: …………… **Ръководител-консултант**: …………

/Симеон Бухалов/ /Катя М. Семкова/

Клас: 12а

e-mail: sbuhalov3@gmail.com0

Пловдив

2024 година

# СЪДЪРЖАНИЕ

[СЪДЪРЖАНИЕ 0](#_Toc163579017)

[УВОД 1](#_Toc163579018)

# УВОД

В днешната цифрова ера, комуникацията е сърцето на всяко социално и бизнес общество. От момента, в който Интернетът стана неизменна част от ежедневния живот, потребителите се нуждаят от инструменти, които да им осигурят бърза, ефективна и сигурна комуникация с техните колеги, приятели и близки.

Онлайн общуването позволява на потребителите да създават много повече връзки с други хора от различни места по-света, което би подпомогнало за развитието на човека. То преодолява ограниченията на далечните разстояния и несъответствия в дневния график на общуващите, като им предоставя способността до комуникират от където и да е стига да имат нужното устройство.

Целта на разработката е създаването на чат приложение, което предлага ползотворни услуги за потребителите си. С помощта на приложението, те могат да създават групи, където да споделят идеи, информация и преживявания с колеги, семейство или приятели.

За да бъде успешно реализирането на приложението, е необходимо да се изследват методите и технологиите, които ще бъдат използвани в процеса на разработка. Трябва да се представи как тези методи и технологии ще бъдат приложени и как ще се развиват функционалностите им. Това включва създаването на интерактивен уебсайт, който дава възможност на потребителите да създават акаунти и да комуникират един с друг.

В първа глава ……… ……………. ……………… ……….. …. …………. …… ………………………… ……………………….

Във втора глава ……… ……………. ……………… ……….. …. …………. …… ………………………… ……………………….

В трета глава ……… ……………. ……………… ……….. …. …………. …… ………………………… ……………………….

В четвърта глава ……… ……………. ……………… ……….. …. …………. …… ………………………… ……………………….

В пета глава ……… ……………. ……………… ……….. …. …………. …… ………………………… ……………………….

# ГЛАВА 1 – ТЕОРЕТИЧНИ ОСНОВИ И КОНЦЕПЦИИ ПО ТЕМАТА

## Обзор на технологични алтернативи

Текущата разработка изисква внимателно планиране и избор на подходящи технологии и стекове. При разглеждането на различни опции за стекове правят впечатление следните: RoR, MERN, SpringBoot, Django, Svelte и ASP.NET. Изборът на подходящи технологии е от съществено значение за успешното развитие на проекта и за крайния потребителски опит. С правилната комбинация от технологии може да се гарантира стабилност, производителност и сигурност на приложението, като същевременно се улеснява и ускорява процеса на разработка.

### RoR стек

**Ruby on Rails (RoR)** е популярен Full Stack фреймуърк, който използва Ruby за сървърната част и обработва всичко - от базата данни до интерфейса на потребителя. RoR е известен със своята простота, елегантност и способността си да ускори процеса на разработка.

**Ruby** е език за програмиране, който е създаден с акцент върху четимостта на кода. Ruby е динамичен, обектно-ориентиран език, който улеснява разработката.

**Ruby on Rails (RoR)** е уеб фреймуърк, който се базира на Ruby и предоставя множество вградени функции, които ускоряват процеса на разработка. Той следва принципите на Convention over Configuration и Don't Repeat Yourself (DRY), което улеснява разработчиците да се концентрират върху бизнес логиката на приложението, а не на конфигурацията и първоначалната настройка. RoR поддържа различни релационни бази от данни като PostgreSQL, MySQL и SQLite.

Предимства на този стек са, че има голяма колекция от компоненти, които са готови за ползване. Те могат да бъдат достъпени от пакетния мениджър на Ruby – RubyGems. Благодарение на тези готови компоненти, разработката става много по-бърза, поради по-малкото писане на код. Друго предимство е, че RoR има вградена защита срещу различни видове атаки, като някои от тях са SQL инжекции и CSRF.

Недостатъци могат да бъдат времето за научаване на Ruby, както и липсата на ресурси въпреки активната общност на RoR.

### MERN стек

MERN се състои от 4 технологии: MongoDB, Express.js, React и

Node.js. Известен е с това, че е гъвкав, което го прави удобен за работа без значение от изискванията на проекта.

**MongoDB** е NoSQL база данни, която предлага гъвкава и мащабируема структура. Тя използва JSON-подобни документи и е идеална за уеб приложения, които изискват скорост при обработка на данни.

**Express.js** е минималистичен уеб фреймуърк за Node.js, който предоставя удобен начин за създаване на уеб сървъри и управление на маршрутирането. Той също така улеснява работата с бази данни и взаимодействието с други уеб услуги.

**React** е библиотека за създаване на потребителски интерфейси, която предоставя голям набор от инструменти за създаване на динамични и интерактивни уеб приложения. Тя използва компонентен подход и виртуален DOM за оптимизация на производителността.

**Node.js** е сървърна платформа, която позволява на разработчиците да създават уеб приложения използвайки JavaScript както на клиентската, така и на сървърната страна. Тя е изключително гъвкава и мащабируема, което я прави идеална за разработка на съвременни уеб приложения.

MERN стека е много добър поради причината, че всички технологии, които се използват в него са на JavaScript, което улеснява връзката между фронтенда и бекенда.

Недостатъци на този стек са сложността му за неопитни разработчици. Поради това, че MongoDB е NoSQL база данни, разработчиците се нуждаят от повече опит, за да може да е напълно ефикасна базата данни. Друг такъв недостатък е това, че React е базиран на компоненти и библиотеки, което го прави много зависим от поддръжката на тези компоненти. Проблемът идва там, че библиотеките не са създадени от създателите на React и не винаги са добре оптимизирани.

### Spring Boot стек

**Spring Boot** е Full Stack фреймуърк за Java, който се използва за създаване на съвременни уеб приложения и микросервизи. Той е базиран на Spring Framework и предоставя бърз и лесен начин за стартиране на нови проекти с минимална конфигурация. Основни части от стека са: Spring Framework, Spring Boot, Spring Data, Spring Security, Spring Web MVC

**Spring Framework** е основният компонент на стека, който предоставя инфраструктурна поддръжка за разработка на Java приложения.

**Spring Boot** е слой над Spring Framework, който улеснява стартирането и конфигурацията на приложения. Spring Boot автоматизира много от конфигурационните задачи и така подпомага за бързо разработване на приложения.

**Spring Data** предоставя абстракция над различни видове бази данни и предоставя лесен начин за извършване на операции като CRUD (Create, Read, Update, Delete). Spring Data улеснява взаимодействието с базите данни и намалява необходимостта от писане на ръчен SQL код.

**Spring Security** отговаря за автентикацията и авторизацията в приложенията. Предоставя вградени начини за управление на потребителски роли, сесии, токени и други сигурностни аспекти.

**Spring Web MVC** предоставя възможност за създаване на уеб приложения със Spring. Spring Web MVC е традиционен модел-изглед-контролер (MVC) фреймуърк.

Предимства на Spring Boot са обширният набор от инструменти за разработка (Spring Cloud, Spring Data, Spring Security и др.) както и това, че автоматизираната конфигурация ускорява процеса на разработката.

Един от недостатъците е, че въпреки автоматизираната конфигурация, на разработчиците все пак може да им е трудно да научат самият стек, най-вече ако до сега не са работили с Java. Друг проблем е изискването на ресурси на стека, поради големият набор от инструменти – трябва да се подбере правилен хостинг, който може да издържи на по-голямо натоварване.

A green text on a black background

Description automatically generated

Фигура 1 – Лого на Spring Boot

### Django стек

**Django** е Full Stack уеб фреймуърк, базиран на Python, който улеснява и ускорява разработката на уеб приложения със своята гъвкавост и готовите за употреба компоненти. Основни компоненти за стека са: Django Framework, Django ORM, Django Templates, Django Forms, Django Admin.

**Django Framework**: Основният компонент на стека, който предоставя цялостно решение за уеб разработката. Django предлага вградени функционалности като ORM (Object-Relational Mapping), маршрутиране, форми, административен интерфейс и много други.

**Django ORM**: Обектно-релационната система на Django улеснява взаимодействието с бази данни чрез Python код, като автоматизира много от задачите, свързани с манипулирането на данни.

**Django Templates**: Предлагат лесен и гъвкав начин за генериране на динамичен HTML код. Шаблоните на Django позволяват лесно разделяне на логиката и представянето на уеб приложенията.

**Django Forms**: Улесняват създаването и валидацията на формуляри в уеб приложенията. Django Forms автоматизират процеса на обработка на формулярни данни и управление на взаимодействието с потребителите.

**Django Admin**: Вграденият административен интерфейс на Django предоставя готово за употреба решение за управление на данни, потребители и права в уеб приложенията.

Предимства са бързата разработка благодарение на компонентите, които предлага, безопасността – има вградена защита от CSRF и SQL инжекции и има активна общност, което предоставя различни библиотеки за ползване в проектите, както и ресурси за учене на стека.

Един голям недостатък е, че Django улеснява много от задачите на разработчиците, но въпреки това, той може да бъде по-сложен за научаване, особено за начинаещи в Python и уеб разработката.

### Svelte стек

**Svelte** е модерен JavaScript фреймуърк за създаване на интерактивни уеб приложения с прост и елегантен подход към разработката. Той предлага различни инструменти, улесняващи разработката като: Svelte Components, Svelte Stores, Svelte Routing и др. **SvelteKit** е официалният маркетингов фреймуърк на Svelte, който предоставя интегрирано и мощно решение за разработката на уеб приложения със Svelte. Вместо традиционния подход с виртуален DOM, Svelte прилага компилация на кода, генерирайки чист JavaScript, който работи директно с DOM-a. Тази архитектура е разработена с идеята да улесни процеса на създаване на уеб приложения, като премахва необходимостта от излишен повтарящ се код.

В основата на Svelte лежи чистият JavaScript, съчетан с функционалности, които автоматично въвеждат реактивност в кода. Например, променливите стават реактивни по подразбиране – актуализират DOM-a динамично в момента в който се променят.

**SvelteKit** допълва Svelte, предоставяйки интегрирана среда за разработка и богат набор от инструменти. Той е проектиран да улесни работата на разработчиците, като предлага автоматична маршрутизация, интеграция с Vite и опции за статично и динамично рендериране на уеб страниците.

**Svelte Components**: В Svelte, всичко е компонент. Компонентите са изградени от HTML, CSS и JavaScript код, които са организирани в еднофайлови компоненти (.svelte файлове). Това улеснява преизползването, тестването и поддръжката на кода.

Svelte Store играе централна роля в управлението на състоянието в Svelte стека то е прост и ефективен начин за управление на състоянието на приложението и комуникация между компонентите.

Предимства са прозиводителността, заради начина, по който се компилира кода, чистотата, заради компонентите, предлагани от Svelte, които предотвратяват повтарящият се код и гъвкавостта – поради чистият JavaScript код е много лесно интегрирането на библиотеки и инструменти.

Svelte и SvelteKit са сравнително нови технологии, затова един недостатък е, че хората, ползващи ги не са много и няма много официална документация.

### ASP.NET стек

**ASP.NET MVC** е популярен и мощен технологичен стек за разработка на уеб приложения, базиран на .NET платформата. Той предлага структуриран и модулен подход към създаването на уеб приложения, който разделя логиката за представяне, бизнес логиката и данните в различни компоненти на приложението. Основните му части са: Razor View Engine, Entity Framework, ASP.NET Identity и ASP.NET Web API, като най-предпочитаната база данни за ползване с него е SQL Server поради лесната комуникация с Entity Framework.

**Razor View Engine**: Разширение на ASP.NET MVC, което позволява създаването на динамични HTML страници, комбинирайки HTML и C# код в един и същи файл.

**Entity Framework (EF)**: ORM (Object-Relational Mapping) инструмент, който улеснява взаимодействието с бази данни чрез предоставяне на удобни API за операции с данни.

**ASP.NET Identity**: Библиотека за управление на идентичността, която предоставя функционалности като управление на потребители, автентикация и авторизация.

**ASP.NET Web API**: Фреймуърк за създаване на HTTP базирани услуги, което улеснява интеграцията на уеб приложения с различни клиентски платформи.

ASP.NET стекът е много добър с това, че благодарение на ASP.NET Identity сигурността се имплементира много по-лесно, има голяма общност, следователно има много ресурси, документацията е лесна за четене и интеграцията на други .NET инструменти и технологии е по-лесна.

Недостатък на този стек е зависимостта му от Windows – употребата му и работата с него е силно ограничена на различни операционни системи различни от Windows

## Избор на подходящи технологии за текущата разработка

При разработването на чат приложение, е важно да се изберат подходящите технологии, които да осигурят ефективна, скалируема и сигурна реализация на проекта. **Svelte** и **SvelteKit** представляват модерни и иновативни технологии за уеб разработка, които са идеални за създаване на динамични и интерактивни уеб приложения като чат системи. С тях управлението на състоянието на страниците става лесно и бързо, което е ключово за обмяната на съобщения между потребители.

**ASP.NET MVC** е утвърден и надежден бакенд стек, който предлага широк набор от функционалности и инструменти за разработване на уеб приложения. Той е добре поддържан, което гарантира сигурността на бакенда, както и данните на потребителите. Заради възможността за интеграция на SQL Server с Entity Framework, комуникацията с базата данни става много лесна, което осигурява ефективността на управлението на данните. Благодарение на механизмите за автентикация и авторизация на ASP.NET Identity потребителите могат да очакват високи стандарти за сигурност, което е критично за защитата на данните им в приложението.

## Специфики на ASP.NET – Svelte стека

Комбинирането на Svelte с ASP.NET създава архитектура за уеб приложения, комбинираща иновативен и бърз фронтенд с мощен и гъвкав бакенд.

### Svelte и SvelteKit за клиентска част

Svelte преобразува декларативния код в оптимизиран JavaScript преди зареждането на приложението, като осигурява по-бързо и плавно потребителско изживяване. Той се отличава с ясен и директен синтаксис, улесняващ разработката и поддържането на сложни интерфейси и предоставя компоненти за разработчиците. Чрез компонентите, програмистите могат да избегнат повтарянето на един и същ код, като ползват един компонент с различни параметри вместо да пренаписват кода всеки път. Това прави кодът по-чист и по-лесен за надграждане и работене върху него.

SvelteKit предоставя гъвкава разработческа среда и структура, ускоряващи началото на нови проекти. Интегрираните решения за маршрутизация и състояние улесняват създаването на уеб приложения с оптимизирано кеширане.



Фигура 2 – Лого/Банер на Svelte

### ASP.NET Core за сървърна част

ASP.NET Core осигурява мощна и безопасна платформа за разработка на уеб API. Разполага с много добра защита срещу различни видове атаки като SQL инжекции, като така данните на потребителите биват предпазени. Разработката на Web API става чрез задаване на Endpoint и създаване на функционалност за този Endpoint. Със своите инструменти и библиотеки, тя улеснява интеграцията и поддръжката на сигурни бакенд решения.

### EntityFrameworkCore и MSSQL

Entity Framework Core (EFC) представлява интегрирано средство за управление на бази данни, което улеснява връзкат между приложенията и базите данни във фреймуърка на ASP.NET Core. Този ORM инструмент преобразува сложните операции с база данни в прости програмни операции, което опростява CRUD операциите.

Съвместно с SQL Server Management Studio (SSMS) и Microsoft SQL Server (MSSQL), Entity Framework Core предлага мощна платформа за управление на релационни данни. MSSQL е високопроизводителна и надеждна система за управление на бази данни, която предлага богат набор от функционалности и възможности за управление на данните. Тази комбинация осигурява възможност за създаване на устойчиви и мащабируеми уеб приложения, които могат да се адаптират към различни бизнес изисквания и натоварване.

### ASP.NET Core Identity

ASP.NET Core Identity е интегриран модул в ASP.NET Core, предоставящ гъвкави и мощни инструменти за управление на потребителски идентичности в уеб приложенията. Този модул улеснява регистрацията, автентикацията, авторизацията и управлението на потребителски профили, като предоставя високо ниво на сигурност и защита на потребителските данни.

### Node.js и Node Package Manager

Node.js е мощна и гъвкава платформа за създаване на сървърни приложения, базирана на JavaScript. Тя пoмага на разработчиците да създадат ефикасно уеб приложение, което може да обработва заявки паралелно, без забавяне.

Една от ключовите характеристики на Node.js е неблокиращата I/O операция, която позволява оптимизиране на изпълнението на заявките и увеличаване на скоростта на приложението. Това прави Node.js идеален избор за приложения, които се използват в реално време като чат системи, стрийминг платформи и социални мрежи.

С Node.js, разработчиците имат достъп до голям брой модули и библиотеки чрез npm (Node Package Manager), което улеснява интеграцията с различни технологии и инструменти. Това поддържа ефективната разработка и ускорява процеса на създаване на функционалности и модули в сървърната част на приложението.



Фигура 3 – Лого на Node.js

## NuGet Package Manager

NuGet е инструмент, който подпомага процеса на добавяне, интегриране и поддръжка на разнообразни пакети и библиотеки в .NET приложения. Чрез интуитивен интерфейс и функционалност, NuGet дава възможност на разработчиците да избират и интегрират ключови компоненти и решения в техните проекти, осигурявайки така ускорен и оптимизиран процес на разработка. Той предоставя инструменти за лесно търсене, инсталация и актуализация на пакети, като така допринася за непрекъснатото развитие и поддържане на висококачествените .NET приложения.

### Vercel за Frontend хостинг

Vercel е платформа за разработка и хостинг, специализирана в оптимизацията и разгръщането на фронтенд приложения. Създадена с фокус върху производителността и удобството за разработчиците, Vercel предоставя едностепенно решение за създаване, тестване и публикуване на уеб приложения, което обединява всички етапи на разработката в една интуитивна и ефективна работна среда.

Vercel се интегрира напълно с популярни системи за управление на версии като GitHub, GitLab и Bitbucket, позволявайки автоматизирана актуализация при всяка промяна на кода.

### MyWindowsHosting за Backend хостинг

MyWindowsHosting е водеща платформа за уеб хостинг, специализирана в предоставянето на надеждни и мащабируеми услуги за хостинг на бакенд приложения, базирани на различни технологии и платформи. Със своя фокус върху Windows базирани решения, MyWindowsHosting предлага обширен набор от функции и инструменти, които улесняват процеса на разработка, разгръщане и управление на бакенд инфраструктурата. Платформата също така предлага уникалната възможност за 2 месеца безплатен пробен период, позволявайки на потребителите да изпробват и оценят качествата и предимствата ѝ преди да се ангажират с дългосрочен план.

MyWindowsHosting предоставя интегрирана поддръжка за популярни Microsoft технологии като ASP.NET, .NET Core, SQL Server и други, което осигурява съвместимост и оптимизирана производителност на бакенд приложенията.

Платформата поддържа WebDeploy, което позволява лесно и бързо качване на уеб приложения и обновления безпосредствено от разработческата среда, като гарантира синхронизирането на кода и конфигурациите между разработческата машина и хостинг сървъра.

MyWindowsHosting предлага цялоденонощна техническа поддръжка и екип от професионалисти, готови да помогнат при всякакви проблеми или въпроси, свързани с хостинга и инфраструктурата.

### SignalR за комуникация в реално време

SignalR е библиотека за ASP.NET Core, която предоставя прост и мощен начин за реализация на реално-временна комуникация между уеб клиенти и сървъри. Този инструмент улеснява създаването на интерактивни и динамични уеб приложения, които могат да обменят данни в реално време, без необходимостта от постоянни заявки към сървъра.

SignalR позволява моменталното изпращане на съобщения и обновления от сървъра към уеб клиентите, като гарантира актуалност и динамичност на предоставяната информация.

A blue circle with a black arrow in it

Description automatically generated

Фигура 4 – Схема на структурата на проекта

### Други технологии

CORS (Cross-Origin Resource Sharing) – стандартна техника за контролиране на достъпа до ресурсите на уеб приложения, позволявайки ограничен достъп до източници на различни домейни. Този механизъм е особено полезен при разработването на уеб приложения с архитектура, където ресурсите се предоставят от различни домейни или услуги.

JasonWebToken – популярен механизъм за автентикация в ASP.NET приложенията, използващ JSON Web Tokens (JWT) за предоставяне на сигурни и компактни токени за удостоверяване на потребителите. Този подход осигурява сигурен и ефективен начин за проверка на идентичността на потребителите и управление на достъпа до ресурсите в уеб приложенията.

Bootstrap – edin от най-популярните и използвани фронтенд фреймуърци в света на уеб разработката. Този мощен инструмент предоставя обширен набор от предварително дефинирани стилове, компоненти и скриптове, които улесняват и ускоряват процеса на създаване на съвременни и атрактивни уеб приложения.

FontAwesome – набор от икони, предоставящ широка гама от векторни изображения, които могат да бъдат лесно интегрирани в уеб приложенията. Тези икони добавят допълнителен естетически елемент и функционалност към интерфейса, без да се налага да се използват сложни графични редактори или специализирани изображения.

Cropper.js – мощна и гъвкава JavaScript библиотека, която предоставя функционалност за обработване и редактиране на изображенията директно в уеб браузъра. Тази библиотека е изключително полезна за разработчиците, които работят с уеб приложения, които изискват манипулиране на изображения, като профилни снимки, галерии или редактори за изображения.

# ГЛАВА 2 – СТРУКТУРА НА РАЗРАБОТКАТА

## 2.1. Обща схема

Проектът се разработва с цел създаване на съвременно и функционално уеб приложение, което комбинира най-добрите практики и технологии от фронтенд и бакенд разработката. Разделя се на две части – Frontend и Backend, които са хостнати с различни услуги. Frontend-а е хостнат на Node.js сървър чрез Vercel, а Backend-a е хостнат на KESTREL сървър чрез MyWindowsHosting, като базата данни е част от backend-a.

A diagram of a computer server

Description automatically generated

Фигура 5 – Схема на структурата на проекта

### 2.1.1. Клиентска част (Frontend)

Фронтендът на проекта е базиран на модерния Svelte фреймуърк, с поддръжка от SvelteKit за улеснено разработване и мащабируемост. Основната му задача е да предоставя динамичен и интерактивен потребителски интерфейс, използвайки силата на Svelte's компилатор за генериране на ефективен JavaScript код.

Фронтендът е организиран чрез TypeScript Data Handlers, които са отговорни за обработката на HTTP заявките към бекенда. Тези хендлъри обработват информация в JSON формат и предоставят необходимите обектни интерфейси за конверсия на данни в TypeScript обекти. За сигурността на заявките, JWT токенът се добавя към всяка заявка след успешна аутентикация на потребителя. Обектите могат да бъдат пренесени към бекенд-а направо през URL-a на заявката като параметри или като се запишат като част към тялото ѝ.

Допълнително, Svelte хранилищата (stores) се използват за управление на реактивните данни между различните компоненти на приложението. Тези хранилища позволяват лесно обменяне на данни и обновяване на интерфейса при промяна на състоянието. Примери за използване на такива хранилища в проекта са:

* Информацията за сегашния потребител;
* Заявките за приятелство;
* Приятелите на текущият потребител и др.

За обмяната на съобщения в реално време отговарят Hubs (хъбовете) във фронтенда. Те създават връзка към хъб с определен еndpoint в бакенда. Всеки потребител е свързан с уникално ID на връзката си към хъба. След създаване на връзката биват дефинирани слушатели за събития /event listeners/ за различни действия, които могат да се случат, и при получаване на такова събитие /event/ от бакенда, функцията за съответния слушател /listener/ се изпълнява. Това предоставя двустранна комуникация между фронтенда и бакенда, като позволява бързо и ефективно обменяне на данни и съобщения без да има нужда от презареждане на страницата.

За визуалната част на приложението са включени векторни графики и икони от FontAwesome, а за стилизацията се използва частично Bootstrap и обикновен CSS.

### 2.1.2. Сървърна част (Backend)

Backend-ът на проекта е разработен с използване на ASP.NET Core Framework. Централната му роля е да предоставя API за взаимодействие с клиентската част на приложението. Вместо традиционните Views, Backend-ът работи като API, като контролерите са от типа ApiController, което позволява гъвкаво управление на URL пътеките за HTTP заявките от Frontend-а.

За създаване на сигурни и компактни токени за удостоверяване на потребителите, в проекта е интегриран JWTService. Той е специализиран за генериране на JWT токени при успешен вход или регистрация на потребителите. При заявки, изпратени от фронтенда бива прикачен този токен, и ако endpoint-а не получи токена, или токена е невалиден, то потребителят не бива авторизиран.

Backend-ът на разработката използва EntityFrameworkCore – ORM библиотеката на ASP.NET Core и MSSQL. Тази библиотека позволява създаването, модифицирането и изтриването на таблици или колони въз основа на дефинирани модели и DbContext-а на приложението. За управление на промените в схемата на базата от данни, EntityFrameworkCore използва миграции. Миграцията включва необходимия код за отразяване на новите промени в MSSQL, както и еквивалентния код за премахване на тези промени.

Моделите в проекта дефинират структурата на данните, с които работи приложението. Чрез EntityFrameworkCore, разработчиците могат да управляват тези модели и да синхронизират промените с базата данни. Освен основните модели, са създадени и Input модели. Те служат за конвертиране на JSON данни от Frontend-а в структурирани обекти, готови за обработка от Backend-а.

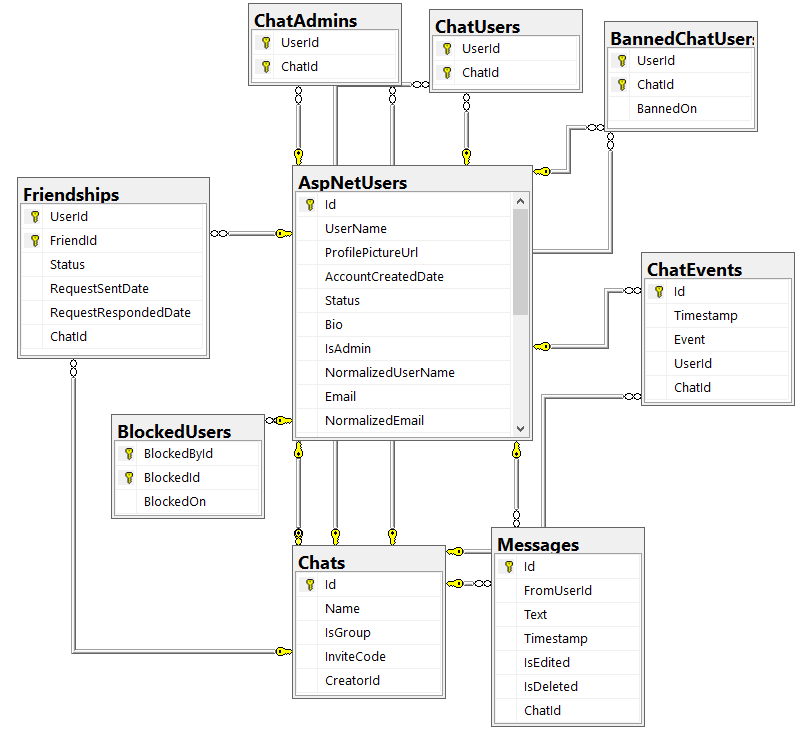
OutputModel-ите са още една част от архитектурата, които представят специфични данни, извлечени от създадените модели, оптимизирани за нуждите на различните изгледи и операции в приложението.

### 2.2. Структура на Базата от данни

Проектът използва релационна SQL база данни с MSSQL като система за управление. Базата данни е организирана в различни таблици, свързани чрез полета за вторичен ключ, които сочат към първичните ключове на други таблици.

Базата съдържа различни таблици, които могат да бъдат свързани помежду си. Тази връзка се извършва посредством първичните ключове на редовете в таблиците. Връзките от тип many-to-many (един запис от таблицата може да е свързан с много от друга или обратното) се извършват чрез създаване на отделна таблица за връзка. Тази трета таблица не се нуждае от първичен ключ, а съдържа само две полета – външни ключове към обектите от другите таблици. Пример за такава „посредническа“ таблица може да е UserRoles таблицата, която съдържа външни ключове към потребител и роля.

Някои от таблиците са ръчно създадени, а другите - автоматично от ASP.NET Identity. Допълнително създадените таблици са генерирани чрез модели и миграции с помощта на Entity Framework Core (EF Core). EF Core позволява разработчиците да дефинират модели за техните данни, които автоматично се превръщат в SQL таблици.



Фигура 6 – Схема на базата данни

### EFC и Identity таблици

Таблиците от ASP.NET Identity управляват потребителската идентичност и авторизация в приложението. Те включват **AspNetUsers**, **AspNetRoles**, **AspNetUserRoles**, **AspNetUserClaims**, **AspNetUserLogins** и **AspNetUserTokens**. Въпреки това, че AspNetUsers е генерирана автоматично, тя все пак наследява допълнителни полета от ApplicationUser модела, като те са: ProfilePictureUrl; AccountCreatedDate; Status, Bio и IsAdmin. Повечето таблици разчитат на външни ключове.

### 2.2.2. Таблици създадени чрез модели

Таблицата **AspNetUsers** съхранява информация за потребителите на системата. Колоните, добавени чрез модел включват потребителско име (**UserName**), URL на профилна снимка (**ProfilePictureUrl**), дата на създаване на акаунта (**AccountCreatedDate**), статус на активност на потребителя (**Status**), биография (**Bio**), флаг за администраторски права (**IsAdmin**). На тази таблица се базират всички останали, поради факта, че приложението не може да функционира без потребители.

Таблицата **Chats** съхранява информация за чатовете в системата. Включва колони за уникален идентификатор (**Id**), име на чата (**Name**), флаг за групов чат (**IsGroup**), код за покана, чрез който потребителите могат да се присъединят към чата (**InviteCode**) и идентификатор на създателя на чата (**CreatorId**). В случай, че флага за групов чат е със стойност false, не се генерира код за покана, тъй като това означава че чата е приятелски – само между двама потребители.

Таблицата **Messages** съхранява съобщенията в чатовете. Колоните включват уникален идентификатор (**Id**), идентификатор на потребителя, изпратил съобщението (**FromUserId**), текст на съобщението (**Text**), времева маркировка (**Timestamp**), флагове за редакция и изтриване (**IsEdited**, **IsDeleted**) и идентификатор на чат, с който се определя към кой чат принадлежи съобщението (**ChatId**).

Таблицата **ChatEvents** записва събитията в чатовете. Колоните включват уникален идентификатор (**Id**), времеви маркировки (**Timestamp**), тип на събитието (**Event**), идентификатор на потребителя, за когото се отнася събитието (**UserId**) и идентификатор на чата, за който се отнася събитието (**ChatId**). Чрез тези събития биват изписани съобщения в чата, които дават информация за състоянието на потребителите.

Таблицата **ChatUsers** управлява участниците в чатовете. Съдържа колони за идентификатор на потребителя (**UserId**) и идентификатор на чат (**ChatId**).

Таблицата **ChatAdmins** управлява администраторските права в чатовете. Съдържа колони за идентификатор на потребителя (**UserId**) и идентификатор на чат (**ChatId**).

Таблицата **BannedChatUsers** поддържа списък на потребителите, чиито достъп до определени групови чатове е забранен. Съдържа колони за идентификатор на потребителя (**UserId**), идентификатор на чат (**ChatId**) и дата на забрана (**BannedOn**).

Таблицата **Friendships** съхранява информация за приятелските връзки между потребителите. Включва колони за идентификатор на потребителя, който първоначално е изпратил заявката за приятелсто (**UserId**), идентификатор на приятеля (**FriendId**), статус на заявката (**Status**), дата на изпращане на заявката (**RequestSentDate**), дата на отговор на заявката (**RequestRespondedDate**) и идентификатор на чата, който бива създаден когато заявката е прието (**ChatId**).

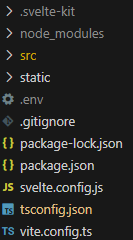
Таблицата **BlockedUsers** съхранява информация за потребителите, които са блокирани от други потребители. Включва колони за идентификатор на потребителя, който блокира (**BlockedById**), идентификатор на блокирания потребител (**BlockedId**) и дата на блокиране (**BlockedOn**). Тази таблица е добавена с цел предпазване от нежелани контакти и съхраняване на личната свобода при общуване.

### 2.3. Структура на файловете

При разработката на софтуерни приложения е от съществено значение файловата организация да е структурирана. Това не само подобрява четимостта и поддръжката на кода, но и улеснява екипната работа и разширяемостта на проекта.

### 2.3.1. Файлова структура на клиентска част

В папката на фронтенда са разположени всички директории и файлове, които се отнасят за клиентската част на проекта.



Фигура 7 – Структура на файловете в клиентската част

В папката **.svelte-kit** в проекта се съдържат конфигурационни файлове и настройки, които управляват поведението и настройките на SvelteKit приложението.

node\_modules се създава автоматично се създава автоматично, при инсталиране на зависимости за Node.js приложения чрез npm. Тази директория съдържа всички пакети (библиотеки, модули и зависимости), които проектът използва, както и техните зависимости. Примери за такиа инсталирани пакети в проекта са SignalR, cropper.js, emoji-picker-element и др.

**src** е една от основните директории в проекта. Тя съдържа всички компоненти, маршрути, стилове и допълнителни функционалности като скриптове, които са създадени от разработчика.

В static се съхраняват файлове, като стилове и снимки. Те могат да бъдат достъпени директно без да се обработват допълнително.

.env се използва за съхранение и управление на чувствителни променливи, които биха улеснили конфигурацията на проекта.

.gitignore е текстови файл, който се използва от системата за контрол на версиите Git за указване на файлове и директории, които не следва да бъдат проследявани или включвани в репозитория.

.npmrc е конфигурационен файл за npm (Node Package Manager), който позволява задаването на различни настройки за npm.

package-lock.json е автоматично генериран файл от npm (Node Package Manager), който съдържа точна информация за всички зависимости на проекта.

package.json e основен файл за Node.js проектите. Той съдържа метаинформация за проекта, като име, версия, зависимости, скриптове и други конфигурационни настройки.

svelte.config.js е конфигурационен файл за Svelte проекти. В него се определят различни настройки, които SvelteKit ползва при компилиране на кода.

tsconfig.js е подобен на svelte.config.js поради това, че и той е конфигурационен файл с настройки, но те се ползват при компилиране на TypeScript код.

vite.config.js отговаря за конфигурацията на Vite – инструмент за създаване на проекти.

A screenshot of a computer

Description automatically generated

Фигура 8 – Изглед на „routes”

На фигура 8 са представени маршрутите в папка routes, както и някои от файловете там.

app.html служи като основен шаблон на интерфейса на приложението. В него се задават всички HTML метатагове, които трябва да оказват влияние на останалите компоненти, както и HTML линкове за различни библиотеки и скриптове. Пример за това е интегрирането на bootstrap чрез линк за стилове.

app.d.ts подпомага за връзката между Svelte и TypeScript. В него могат да се дефинират интерфейси, методи и други зависимости, които биха се ползвали навсякъде в приложението.

Папката routes е основна част на приложението. В нея се събират и определят абсолютно всички маршрути за разработката. Съдържа различни .svelte файлове.

+layout.svelte служи като основното оформление на уеб страницата. В него има HTML, както може да има и JavaScript или TypeScript код, въведен чрез <script> таг и различни стилове въведени чрез <style> таг. Когато е в основата на routes папката той ще бъде приложен върху всеки един маршрут от уеб страницата.

+page.svelte е компонентен файл и представлява една от основните уеб страниците в приложението. В него се дефинира уникални логика и функционалности в зависимост от изискванията на уеб страницата.

Аналогично, останалите папки са други маршрути, които отговарят за други уеб страници в приложението и съдържат още такива .svelte файлове.

A screenshot of a computer

Description automatically generated

Фигура 9 – Изглед на „lib”

На фигура 9 са представени директориите в папка lib. Папката се ползва за съхраняване на библиотеки, функции, компоненти и други ресурси, които биха се ползвали многократно в приложението.

В components се съдържат, компоненти, които могат да се използват повече от веднъж. Главната им функция е да предотвратят повтарянето на един и същ код, както и да направят кода по лесно четим.

css папката служи са съхраняване на основни стилове в приложението. Те описват как ще изглежда потребителският интерфейс и предоставят лесното му редактиране. Разширенията на файловете вътре са .css.

Папката handlers съдържа .ts файлове, които служат за обработка на заявките изпратени към бакенда и за управление на бизнес логиката на приложението. Пример за такива handler-и в проекта са chatHandler.ts, който отговаря за извличането на съобщенията за определен чат от базата данни, за изпращане на съобщение и запазването му, както и за дефиниране на интерфейси.

Папката hubs съдържа функции, които свързват клиента с определен хъб към бакенда. Това е нужно за да се осъществи комуникацията в реално време между клиентската и сървърната част, без да има нужда от презареждане на страницата.

stores служи за съхраняване на данни, които се използват за управляване на състоянието на уеб приложението. Полезни са, поради причината, че могат да се ползват между всички компоненти в приложението и чрез тях да се управляват реактивни променливи, които подпомагат за динамичния изглед на сайта.

### 2.3.2 Файлова структура на сървърната част

В папката на бакенда са разположени всички файлове, които отговарят за логиката на сървърната част, с изключение на базата от данни. Тъй като вече е разработен фронтенд, папката Views е премахната и бакенда служи само като API.

A screenshot of a computer

Description automatically generated

Фигура 10 – Структура на файловете в сървърната част

Папката wwwroot е стандартна папка в ASP.NET, която служи за съхранение на статични файлове, като css, изображения, JavaScript кодове. В текущата разработка папката се използва за съхранение на профилните снимки на потребителите. Почти аналогична е на static папката в клиентската част.

Controllers играе ключова роля за проекта. Там се намират класове, които обработват конкретни видове заявки изпратени от клиентската част.

В папката Data се съхранява ApplicationDbContext.cs - централният клас за управление на данните в приложението. Този клас наследява **DbContext** от Entity Framework Core и осъществява връзката между сървърната част и базата данни.

Hubs съдържа класовете, които отговарят за SignalR хъбовете. Те служат за управление на взаимодействието между сървъра и клиентите в реално време.

Папката Migrations служи за съхранение на миграциите на базата данни. Миграциите са автоматизирани версии на схемата на базата данни, които позволяват на разработчиците да управляват и актуализират структурата на базата данни чрез код. При нужда за актуализиране на базата данни, след промяна на моделите, отразяващи съответната таблица, разработчикът трябва да създаде нова миграция ръчно или с команда преди да актуализира базата.

Models съхранява класовете, които служат за формиране на таблиците в базата данни. Те подпомагат за манипулирането на данните в базата.

В Models има две подпапки – InputModels и OutputModels. InputModels служи за съхранение на класове, които представят моделите за входящи данни при обработката на конкретни заявки. Пример за текущата разработка са LoginInput.cs, който съдържа полета за потребителско име и имейл. Класът се използва, когато е нужно потребител да бъде автентикиран, като подпомага за обработване на заявката. OutputModels съдържа класове, които представят моделите за изходящи данни, които приложението връща в отговор на заявки. Пример за това е ChatItem.cs. Този клас е нужен, тъй като съобщенията между потребителите са или изпратени от човек или са събитие, което трябва да бъде рендерирано по конкретен начин.

Папката Services служи за съхранение на класове, които съдържат бизнес логиката на приложението. Целта им е да централизират логиката и да предоставят ясен и структуриран начин за управление на приложението. Примери за разработката са ITokenService.cs и TokenService.cs. Те са два свързани класа, които използват в за управление на генерирането и валидирането на токените, които биват използвани за автентикация и авторизация.

appsettings.json е конфигурационен файл, който съдържа различни настройки, използвани за конфигуриране на приложението. Пример за такава настройка е ConnectionStrings, предоставяща информация за връзката с базата от данни.

Program.cs е основният стартиращ файл приложението и играе ключова роля при инициализацията и конфигурацията му. В този файл се определя как приложението ще бъде стартирано и какви настройки и услуги ще бъдат конфигурирани. Някои от услугите, които са добавени в текущата разработка са CORS – механизъм, който позволява поискването и получаването на ресурси от друг домейн отворено; SignalR – библиотеката осъществяваща комуникацията в реално време между потребителите.

# ГЛАВА 3 – ПРОГРАМНА РЕАЛИЗАЦИЯ НА РАЗРАБОТКАТА

### 3.1. Реализация на Backend

Реализацията на бакендa е един от ключовите етапи в разработката на софтуерни приложения, която обхваща проектирането, разработването и тестването на сървърната част на приложението. В този раздел ще бъде разгледан процеса на създаване на бакенд, който поддържа и управлява основната логика на приложението.

Ще се разгледат архитектурата на приложението, включително моделите, контролерите, услугите, хъбовете и миграциите, както и връзката с базата данни.

### 3.1.1. Създаване на модел

В процеса на разработка на софтуерни приложения с ASP.NET, една от първите стъпки е дефинирането на моделите, които представляват структурата на данните, с които ще работи бакенд частта на приложението. Моделите служат за дефиниране на схемата на базата данни, както и за управление на бизнес логиката и валидацията на данните.

public class Chat

{

public Chat()

{

// Initialize collections to prevent NullReferenceException

Messages = new List<Message>();

Users = new List<ChatUser>();

Admins = new List<ChatAdmin>();

BannedChatUsers = new List<BannedChatUser>();

}

[Key]

public int Id { get; set; }

[Required]

public string Name { get; set; }

[Required]

public bool IsGroup { get; set; } = true;

public string? InviteCode { get; set; }

[Required]

public int CreatorId { get; set; }

[ForeignKey("CreatorId")]

// Ignore these properties when serializing to JSON

[JsonIgnore]

public virtual ApplicationUser Creator { get; set; }

[JsonIgnore]

public virtual ICollection<Message> Messages { get; set; }

[JsonIgnore]

public virtual ICollection<ChatUser> Users { get; set; }

[JsonIgnore]

public virtual ICollection<ChatAdmin> Admins { get; set; }

[JsonIgnore]

public virtual ICollection<BannedChatUser> BannedChatUsers { get; set; }

}

Моделът от извадката представлява основната структура за управление на чатовете в системата, включително информация за съобщенията, потребителите, администраторите и забранените потребители. Той използа Data Annotations – библиотека, която помага на компилаторът да разбере какви са условията за полетата когато извършва манипулации върху тях и когато ги създава за в базата данни. Примери за такива анотации са [Key] – обозначава първичният ключ таблицата, с който най-често се индексират записите; [Required] – прави полето задължително за попълване при създаване на нов обект или запис; [ForeignKey] – показва, че полето се използва като външен ключ към друг модел или таблица, както в случая CreatorId се използва за да се разбере кой потребител е създателят на текущият чат. Атрибутът [JsonIgnore] се използва в моделите на ASP.NET Core за игнориране на определени свойства при преобразуване на обект към JSON формат.

В разработката се използват и InputModels и OutputModels, които са специализирани модели, предназначени за обработване и представяне на данни в бакенда. Тези модели не са пряко свързани с таблици в базата данни, но служат за оформяне и валидация на информацията, която приложението обработва и връща към клиентската страна.

InputModel-ите се използват за приемане и валидация на входящите данни от клиентската страна преди да бъдат обработени.

public class LoginInput

{

public string UsernameEmail { get; set; }

public string Password { get; set; }

}

В извадката се представя InputModel, който отговаря за данните, които потребителя въвежда, когато си влиза в акаунта.

OutputModel-ите се използват за форматиране и представяне на данните, които приложението връща към клиентската страна. Те определят структурата и формата на данните, които се изпращат обратно към клиента след обработка в бакенда.

public class ChatItem

{

public int Id { get; set; }

public string Content { get; set; }

public DateTime Timestamp { get; set; }

public string UserName { get; set; }

public int UserId { get; set; }

public bool IsEvent { get; set; }

public string IsActive { get; set; }

public string EventType { get; set; }

public string UserPfp { get; set; }

public bool IsEdited { get; set; }

public bool IsDeleted { get; set; }

public bool WithDetails { get; set; }

}

В извадката се представя OutputModel, който отговаря за начина и структурата по който данните ще бъдат изпратени обратно към клиентската част.

### 3.1.2. Създаване на контролер

[Route("auth")]

[ApiController]

public class AuthenticationController : ControllerBase

{ //… }

Поради факта, че бакенда се използва като API, е необходимо да се добави атрибутът [ApiController]. Този атрибут информира ASP.NET Core, че контролерът трябва да работи като част от API и автоматично обработва някои стандартни задачи като валидация на моделите и управление на грешките. Друг важен атрибут е [Route("")], който определя базовия път за API ендпойнтите в контролера. В случая с "auth", всички ендпойнти, дефинирани в този контролер, ще бъдат достъпни под /auth пътя. Това помага за организацията и структурирането на API ендпойнтите, правейки ги по-лесни за навигация и управление. Контролерът наследява ControllerBase, което е базов клас за контролери без поддръжка на изгледи.

### 3.1.3. Създаване на хъб

[Route("chatHub")]

public class ChatHub : Hub

{

public string GetConnectionId()

{

return Context.ConnectionId;

}

}

Тук създаваме SignalR хъб с име ChatHub, към който клиентите ще се свързват. Атрибутът [Route("chatHub")] определя пътя, по който ще бъде достъпен хъбът. В този случай, всички взаимодействия с хъба ще се извършват чрез /chatHub.

Методът **GetConnectionId()** връща уникалния идентификатор на текущата връзка. Този идентификатор може да бъде използван за идентификация на клиентската връзка в рамките на хъба, което е полезно за управлението на връзките и комуникацията между клиент и сървър.

### 3.1.2. Важни методи за разработката

[HttpPost("register")]

public async Task<IActionResult> Register([FromBody] RegisterInput model)

{

var existingUser = await \_userManager.FindByNameAsync(model.UserName);

if (existingUser != null)

{

return new JsonResult ( new {userExists=true});

}

existingUser = await \_userManager.FindByEmailAsync(model.Email);

if (existingUser != null)

{

return new JsonResult ( new {emailExists=true});

}

var user = new ApplicationUser

{

UserName = model.UserName,

Email = model.Email,

ProfilePictureUrl = $"/images/profile\_pictures/placeholder.png",

AccountCreatedDate = DateTime.UtcNow,

};

var result = await \_userManager.CreateAsync(user, model.Password);

if (!result.Succeeded)

{

return BadRequest(result.Errors);

}

var token = await \_tokenService.GenerateToken(user);

var cookieOptions = new CookieOptions

{

HttpOnly = true,

Expires = DateTime.UtcNow.AddDays(365)

};

Response.Cookies.Append("token", token, cookieOptions);

return new JsonResult(new { accountRegistered = true });

}

Методът от извадката се намира в AccountController.cs и се използва, за да се извърши регистрация на нов потребител. Този метод приема RegisterInput модел като параметър, който съдържа информацията необходима за регистрацията на потребителя.

Първоначално проверява дали потребителя съществува, и ако да връща този резултат към клиентската част. Ако потребителят не съществува, то метода създава нов обект от типа на модела ApplicationUser, използвайки данните от RegisterInput модела, който получава. След това създава самият акаунт като го добавя в базата и при успешно добавяне в базата нов JWT токен бива създаден и закачен към бисквитките на браузъра.

[HttpGet("getMessages")]

public async Task<IActionResult> GetMessages([FromQuery] int chatId, [FromQuery] int page = 1, [FromQuery] int pageSize = 20)

{

var user = await \_userManager.GetUserAsync(User);

if (user == null || !(await IsUserInChat(user.Id, chatId)))

{

return Unauthorized();

}

var messages = await \_context.Messages

.Where(m => m.ChatId == chatId)

.Select(m => new ChatItem

{

Id = m.Id,

Content = m.Text,

Timestamp = m.Timestamp,

UserName = m.FromUser.UserName,

UserId = m.FromUser.Id,

UserPfp = m.FromUser.ProfilePictureUrl,

IsActive = m.FromUser.Status,

IsEdited = m.IsEdited,

IsDeleted = m.IsDeleted

})

.OrderByDescending(m => m.Timestamp) // Order by timestamp descending to get the latest messages first

.Skip((page - 1) \* pageSize) // Skip the messages that are before the current page

.Take(pageSize) // Take only the messages for the current page

.ToListAsync();

var events = await \_context.ChatEvents

.Where(e => e.ChatId == chatId)

.Select(e => new ChatItem

{

Id = e.Id,

Content = e.Event.ToString(),

Timestamp = e.Timestamp,

UserName = e.User.UserName,

EventType = e.Event.ToString(),

IsEvent = true

})

.OrderByDescending(e => e.Timestamp) // Order by timestamp descending to get the latest events first

.Skip((page - 1) \* pageSize) // Skip the events that are before the current page

.Take(pageSize) // Take only the events for the current page

.ToListAsync();

var chatItems = messages.Concat(events).OrderBy(i => i.Timestamp).ToList();

return new JsonResult(new { chatItems });

}

Горепоказаният метод може да бъде намерен в ChatController.cs. Той се използва за извличане на всички съобщения за даден чат, за да могат да бъдат показани на потребителя. Като параметри приема идентификатора на чата от URL заявката, текущата страница и последен параметър за това колко елемента да се покажат за една страница. Параметрите за страници позволяват пагинация, което ускорява бързината на сайта, поради факта, че не е нужно всеки път да бъдат изтеглени всички съобщения от базата а само първите 20 и да се теглят допълнителни само, когато е нужно. Методът извлича съобщенията спрямо филтри, които са зададени и ги комбинира в един ChatItem обект, тъй като съобщенията са или от вид Message или ChatEvent. След като ги извлече ги сортира спрямо маркировката за дата и ги връща към фронтенда като JSON резултат.

[HttpPost("sendMessage")]

public async Task<IActionResult> SendMessageToRoom([FromBody] SendMessageInput data)

{

var user = await \_userManager.GetUserAsync(User);

var chat = await \_context.Chats.FirstOrDefaultAsync(c => c.Id == Int32.Parse(data.chatId));

// Check if the chat exists, if the user is authenticated, and if the user is in the chat

if (chat == null || user == null || !(await IsUserInChat(user.Id, chat.Id)))

{

return Unauthorized();

}

var Message = new Message

{

ChatId = Int32.Parse(data.chatId),

Text = data.message,

FromUserId = user.Id,

Timestamp = DateTime.Now,

IsEdited = false,

FromUser = user,

IsDeleted = false

};

\_context.Messages.Add(Message);

await \_context.SaveChangesAsync();

// Send the new message to all clients in the chat group

await \_chat.Clients.Group(data.chatId).SendAsync("ReceiveMessage", new { Message, UserPfp = user.ProfilePictureUrl } );

return Ok();

}

Кодът горе е метод, който се използва за изпращане на съобщение в определен чат. Той приема параметрите message и chatId от тялото на заявката. Проверява дали потребителят е автентикиран, дали чатът съществува и дали потребителят а част от чата чрез отделен метод. Ако някоя от тези проверки се провали, то методът връща резултат, че потребителят не е авторизиран.

След това се създава ново съобщение по модел Message и се добавя в базата данни. Накрая се изпраща съобщението до всички клиенти, които са част от съответния чат и са свързани към него, използвайки SignalR.

### 3.2. Реализация на Frontend

След разработването на бакенд, следва създаването на фронтенд интерфейс, който предоставя удобен и интуитивен начин за потребителско взаимодействие. В този раздел ще бъдат разгледани архитектурата, компонентите и основните технологии, използвани в разработката на фронтенд частта със Svelte и SvelteKit. Ще се опише как се свързва фронтендът с бакенда чрез API заявки и как се управлява състоянието на приложението за оптимална производителност и потребителско изживяване.

### 3.2.1. Реализация на главното оформление на сайта

За да има консистентен и професионален вид разработката, то се изполва +layout.svelte файл. Този файл служи като основен шаблон за всички страници в приложението и в случая служи като хедър на сайта с името и логото както и включва останалото съдържание на другите страници. В този раздел бъде разгледано как се структурира и стилизира главното оформление на сайта, както и как се интегрират компоненти и функционалности от други части на приложението в този шаблон.

<div class="container-fluid">

<nav class="navbar navbar-expand-lg navbar-light">

. . .

<a class="navbar-brand" on:click={() => isLogged ? goto('/chat/home') : goto('/welcome')}>Boomerang</a>

. . .

</nav>

<main>

<slot/>

</main>

</div>

Този код представлява основната структура на +layout.svelte файлът, който служи като шаблон за главната страница на уебсайта.

Първо се задава контейнер с пълна широчина, който обгражда цялата страница. Чрез <nav> тага се задава навигационната лента на сайта, която включва линк към началната страница на приложението. Линкът на името на сайта реагира на кликване и пренасочва потребителя към началната страница или към „Добре дошли“ страницата в зависимост от това дали потребителят е автентикиран или не. **<main>** тагът съдържа основното съдържание на страницата, което ще бъде заменено или разширено от компонентите на конкретната страница, която използва този layout. **<slot/>** е специален таг в Svelte, който позволява да се замести съдържанието на компонента, в който се използва, със съдържанието, което се подава като дете на този компонент.

### 3.2.1. Реализация на логин формата

<form on:submit={userLogin}>

<div>

<div class="mb-4">

<div class="form-outline">

<input type="text" id="userEmail" bind:value={usernameEmail} class="form-control form-control-lg" />

<label class="form-label" for="userEmail">Потребителско име или имейл</label>

</div>

<div class="form-outline">

<input type="password" id="password" bind:value={password} class="form-control form-control-lg" />

<label class="form-label" for="password">Password</label>

</div>

</div>

{#if (loginError)}

<div class="alert alert-danger" role="alert">

{loginErrorMessage}

</div>

{/if}

<div class="mt-2 pt-2">

<input class="btn btn-primary btn-lg" type="submit" value="Вход"/>

</div>

</form>

Извадката горе представлява кода, отговарящ за логин формата на уебсайта, която потребителите използват за вписване в системата.

**<form on:submit={userLogin}>** определя формата и при кликване на бутона за вписване се изпълнява функцията **userLogin**, която извършва проверка дали полетата са правилни и изпраща заявка към сървъра за автентикация.

Има два input елемента – за въвеждане на потребителско име или имейл и за въвеждане на парола. Те са обвързани със съответните променливи usernameEmail и password.

След това има div елемент обгарден във {#if} условие. Целта на този елемент е да изписва грешка, когато потребителят не въведе правилни данни за вписване. {#if} условието рендерва div-а само когато променливата loginError е true и изписва стойността на loginErrorMessage променливата.

### 3.2.2. Реализация на контейнера за съобщения

<div class="messages-container" bind:this={scrollContainer} on:scroll={handleScroll}>

{#each chatItems as item (item.id)}

<div class="message" style="display: flex; align-items: start;">

{#if item.isEvent}

<EventMessage {item} />

{:else}

<UserMessage {groupInfo} {item} {imageUrl} {userInfo} {isEditingMessage} {deleteMessage} {confirmEdit} {cancelEdit} {isEditing} withoutUserDetails={false} />

{/if}

</div>

{/each}

</div>

Тази извадка представлява контейнера за съобщения в чата на уебсайта.

Първо се дефинира контейнера. Той е свързан с променливата scrollContainer, което позволява програмно да се управлява скролването му. След това чрез {#each} Svelte директива се обхожда масивът chatItems и рендерва всички елементи. За всяка итерация се проверява дали атрибута на елемента isEvent има стойност true, като в случай че стойността е true, то се рендерва компонента <EventMessage/>, който съобщава за събития, случили се в чата, а ако стойността е false се рендерва <UserMessage/> компонент, който представлява обикновено съобщение от потребител включващо името на потребителя, профилната му снимка, датата на съобщението и самото съобщение. Къдравите скоби се използват за предаване на properties или данни от родителския компонент към деца компоненти.

### 3.2.3. Реализация на HTTP заявка към сървърната част

export async function handleMessageSubmit(messageToSubmit: string, chatId: string){

let token = await getToken();

const requestBody = JSON.stringify({message: messageToSubmit, chatId: chatId});

const response = await fetch(`${backendUrl}/chat/sendMessage`, {

method: 'POST',

headers: {

'Authorization': `Bearer ${token}`,

'Content-Type': 'application/json'

},

credentials: 'include',

body: requestBody

});

if (!response.ok) {

console.error('Error sending message:', await response.text());

}

}

В този код се дефинира асинхронна функция **handleMessageSubmit**, която използва **fetch** за изпращане на HTTP POST заявка към сървърната част на приложението. Дефинира се променлива token, която получава стойността си от изпълняването на функцията getToken(), връщаща текущият JWT токен. Дефинира се константа requestBody, която преобразува параметрите на функцията във JSON обект. След това чрез fetch се изпраща HTTP заявката към бакенда, като в тялото се задава, че метода е POST, задават се хедърите, включващи авторизационния токен и се предава и тялото на заявката, което в случая е requestBody обекта. Накрая чрез if проверка се проверява дали отговорът не е със статус 200 (OK) и ако не е то се изписва съобщението за грешката в конзолата.

### 3.2.4. Реализация на връзка със SignalR хъб

Реализацията на връзка със SignalR хъб е ключова част от архитектурата на приложението, осигуряваща реално време на комуникацията между клиентската и сървърната част.

async function setupConnection() {

if (connection) {

ready=false;

await connection.stop();}

connection = new HubConnectionBuilder()

.withUrl(`${backendUrl}/chatHub`)

.build();

connection.on("ReceiveMessage", async function(data: any){

let chatItemToAdd = createChatItem(data, 'ReceiveMessage');

chatItems = [...chatItems, chatItemToAdd];

console.log("chatItem: ", chatItemToAdd.withoutDetails);

await tick();

scrollContainer.scrollTop = scrollContainer.scrollHeight;

});

. . .

await connection.start()

.then(function(){

connection.invoke('getConnectionId')

.then(async function(connectionId: string){

\_connectionId = connectionId;

await loadMessages();

await getGroupInfo();

await joinRoom();

await tick();

ready=true;

})

})

.catch(function(err: any){

console.error(err.toString());

});

}

Функцията setupConnection инициализира и управлява връзката с хъба, като следва определен пореден ред на операции. Тя се изпълнява при всяко отваряне на чат.

Преди да се създаде нова връзка, се проверява дали вече има активна връзка и я спира, ако е така. Използвайки HubConnectionBuilder класа, се конфигурира връзката с хъба, указвайки URL адреса към сървъра.

Чрез метода **connection.on**, се добавя listener за събитието "ReceiveMessage", който се извиква при получаване на ново съобщение от сървъра. Съобщението се обработва и добавя към списъка текущите съобщения.

След успешното създаване и конфигуриране на връзката, тя се стартира и извършва редица асинхронни операции като извличане на идентификационен номер, зареждане на съобщения, получаване на информация за групата и присъединяване към отделна стая за чата в хъба. Тези асинхронни операции гарантират, че страницата винаги ще е с най-новите съобщения и актуална информация, тъй като се осъществява динамично обновление на съдържанието чрез SignalR хъб. Това осигурява потребителско изживяване, което е непрекъснато и актуално, без нуждата от ръчно обновление или презареждане на страницата.

### 3.3. Начин за хостване

При разработката на приложението се използват специализирани платформи за хостинг. Vercel е избран за фронтенда, благодарение на своята бързина и интуитивна облачна инфраструктура. В същото време, MyWindowsHosting е идеалният избор за ASP.NET приложението, предлагайки надеждност и гъвкавост, без да изисква сложни настройки.

### 3.3.1. Backend хостинг

За хостване на бакенда на приложението, използваме MyWindowsHosting, като се възползваме от възможностите за WebDeploy. Този метод на разгръщане предлага удобство и ефективност, позволявайки качването на приложението да стане бързо и без проблеми. След вписване в хостинг акаунта в MyWindowsHosting се избира VS WebDeploy опцията, която е в Manage Website менюто. Отваря се прозорец на екрана и там има поле Publishing XML с бутон Get Publish Settings. След кликане на бутона на машината бива изтеглен .xml файл. Следващата стъпка е отваряне на проекта във Visual Studio и кликване с десен бутон на Application-a във Solution Explorer-а. От там следва да се избере Publish опцията, която отваря нов прозорец, предоставящ възможност за импортиране на профил. След избиране на опцията за импортиране на профил, разработчикът трябва да постави .xml файла от по-рано и да потвърди. Накрая остава да бъде натиснат Publish бутона, за да се хостне най-актуалната версия на сайта, като преди това разработчикът може да конфигурира допълнително профила за publish-ване.

### 3.3.2. Frontend хостинг

Vercel предоставя бързина и надеждност, които са от съществено значение за оптималното представяне и управление на фронтенд приложения, със почти изцяло автоматизирано хостване.

Първата стъпка е подготвяне на GitHub хранилище, в което ще бъде съхранена фронтенд частта. След това разработчикът трябва да качи проекта в това хранилище, използвайки Git команди. Следващата стъпка е отваряне на Vercel и натискане на Add New бутона, който препраща потребителя към нова страница. В новата страница трябва да се избере хранилището, в което е проекта и след това Vercel автоматично намира на какъв framework е разработката и задава настройките за хостинга. Последната стъпка е натискане на Deploy бутона, което финализира процеса на хостване.