An Internship Course Report on

MERN DEVELOPMENT INTERNSHIP

Department of Computer Science and Engineering

Anil Neerukonda Institute of Technology and Sciences

(Affiliated to Andhra University) Sangivalasa, Visakhapatnam-531162

Submitted by

D. Sai Venkata Chaitanya - A22126510144

Internship Reviewer
Dr. S. Anusha
M.Tech
(Asst. Professor)

Summer Internship Coordinator
Mrs. G. Gowripushpa
M.Tech, Ph.D
(Assistant professor)

ANIL NEERUKONDA INSTITUTE OF TECHNOLOGY AND SCIENCES

SANGIVALASA, VISAKHAPATNAM – 531162

2022-2026

BONAFIDE CERTIFICATE

This is to certify that the course training on Web Development, work entitled "MERN STACK DEVELOPER INTERNSHIP BY CODEC TECHNOLOGIES" which is a bonafide work carried out by Dubakula Sai Venkata Chaitanya bearing Register No. A22126510134 respectively in fulfilment of Summer Internship Training in IV/IV B. Tech, 1st Semester in Computer Science Engineering of the ANITS, Visakhapatnam during the year 2022-2026. It is certified that all corrections indicated for internal assessment have been incorporated to account.

Dr. G. Srinivas
(Head of the Department)
Computer Science and Engineering
ANITS

COURSE COMPLETION CERTIFICATE

DECLARATION

D. Chaitanya, bearing College Register No. A22126510144 respectively, do hereby declare that this course training on “MERN Stack Development” is carried out by me. I hereby declare that the original summer internship is done by me as per the requirements of the training.

Station: Sangivalasa
Date: 18-08-2025

D. Sai Venkata Chaitanya
(A22126510134)
(IV/IV CSE-C)

ACKNOWLEDGMENT

An endeavour over a long period can be successful with the advice and support of many well-wishers. We take this opportunity to express our gratitude and appreciation to all of them.

We owe our tributes to the Head of the Department, Computer Science & Engineering, ANITS, for his valuable support and guidance during the period of project implementation.

We express our warm and sincere thanks for the encouragement, untiring guidance and the confidence they had shown in us. We are immensely indebted for their valuable guidance throughout our project.

We also thank all the staff members of the CSE department for their valuable advice. We also thank the Supporting staff for providing resources as and when required.

D. Sai Venkata Chaitanya
A22126510134
(IV/IV CSE-C)

TABLE OF CONTENTS

1. Introduction

1.1. What is the MERN stack?

The MERN stack is a popular web development framework that combines four powerful technologies: MongoDB, Express.js, React.js, and Node.js. This stack allows developers to build full-stack applications using JavaScript for both the client-side and server-side, providing a seamless development experience.

1.2. Benefits of TypeScript in Full-Stack Development

TypeScript is a superset of JavaScript that adds static typing, which helps catch errors at compile time rather than runtime. This leads to more robust and maintainable code, especially in large applications. TypeScript's features like interfaces, enums, and type aliases enhance code readability and developer productivity. Additionally, TypeScript's integration with modern IDEs provides powerful autocompletion and refactoring tools, making it easier to work with complex codebases.

1.3. Database Agnostic Architecture Overview

A database agnostic architecture allows developers to design applications that can work with multiple database systems without being tightly coupled to a specific one. This approach provides flexibility in choosing the best database for the application's needs, whether it's a NoSQL database like MongoDB or a relational database like PostgreSQL. By abstracting database interactions, the application can easily switch between different databases or support multiple databases simultaneously. This is particularly useful in full-stack development, where the backend can handle various data storage solutions while maintaining a consistent API for the frontend.

1.4. Project Overview and Shared Codebase

This report outlines a comprehensive internship course on MERN stack development with TypeScript, focusing on building a shared codebase that can be reused across multiple projects. The course covers the setup of a TypeScript environment, implementation of a database abstraction layer, and development of a shared authentication system. It also includes the creation of reusable UI components and styling, enabling developers to efficiently build and maintain applications.

The projects include an e-commerce platform, a mini social media application, and a LLM chat aggregator with web scraping capabilities. Each project demonstrates the practical application of the concepts learned throughout the course, showcasing the benefits of using TypeScript and the MERN stack in full-stack development.

2. Shared Architecture and Common Components

This section outlines the shared architecture and common components used across the projects in this internship course. The focus is on creating a reusable codebase that can be leveraged in multiple applications, ensuring consistency and efficiency in development.

2.1. TypeScript Setup for MERN Stack

To set up TypeScript for a MERN stack project, follow these steps:

1. **Initialize the project**:

```
mkdir mern-stack-project
cd mern-stack-project
npm init -y
```

2. **Install TypeScript and necessary dependencies**:

```
npm install typescript ts-node @types/node --save-dev
```

3. **Create a `tsconfig.json` file**:

```
{
  "compilerOptions": {
    "target": "ES6",
    "module": "commonjs",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true,
    "outDir": "./dist",
    "rootDir": "./src"
  },
  "include": ["src/**/*"],
  "exclude": ["node_modules", "dist"]
}
```

2.2. Database Abstraction Layer

The abstractions is simply an interface that defines the methods for database operations, relied on or expected by the application. This allows the application to interact with the database without being tightly coupled to a specific database implementation. The actual implementation of the database operations is done in a class that implements the interface.

Database abstraction in db.ts file:

```ts
// Import the appropriate database drivers
import sqlite, { ISqlite, open } from "sqlite";
import sqlite3 from "sqlite3";

// Application specific schema imports
import {
  Comment,
  // Follow,
  // Like,
  Post,
  Session,
  User,
  UserProfile,
} from "./schema.ts";

// Minimal transaction wrapper for the `sqlite` + `sqlite3`
driver, since they
// do not have built-in transaction support.
class Transaction {
  private active = false;
  constructor(
    private db: sqlite.Database<sqlite3.Database,
sqlite3.Statement>,
  ) {}

  get in_transaction(): boolean {
    return this.active;
  }

  async begin(): Promise<void> {
    if (this.active) throw new Error("Transaction already
started");
    await this.db.exec("BEGIN IMMEDIATE");
    this.active = true;
  }

  async run(sql: string, params?: any[]):
Promise<ISqlite.RunResult> {
    if (!this.active) {
      throw new Error("Cannot run SQL command outside of a
transaction");
    }
    return this.db.run(sql, params);
```

```
  }

  async get(sql: string, params?: any[]): Promise<any> {
    if (!this.active) {
      throw new Error("Cannot run SQL command outside of a
transaction");
    }
    return this.db.get(sql, params);
  }

  async all(sql: string, params?: any[]): Promise<any[]> {
    if (!this.active) {
      throw new Error("Cannot run SQL command outside of a
transaction");
    }
    return this.db.all(sql, params);
  }

  async commit(): Promise<void> {
    if (!this.active) throw new Error("No transaction to
commit");
    console.log("Committing transaction");
    await this.db.exec("COMMIT");
    this.active = false;
  }

  async rollback(): Promise<void> {
    if (!this.active) return;
    await this.db.exec("ROLLBACK");
    this.active = false;
  }
}

// DatabaseAdapter interface for database operations
export interface DatabaseAdapter {
  connect(): Promise<void>;
  disconnect(): Promise<void>;
  transaction(): Promise<Transaction>;

  // User methods
  createUser(
    username: string,
    email: string,
```

```
    passwordHash: string,
  ): Promise<User>;
  getUserByUsername(username: string): Promise<User | null>;
  getUserById(id: number): Promise<User | null>;
  updateUserProfile(
    userId: number,
    bio?: string,
    avatarUrl?: string,
    displayName?: string,
  ): Promise<User>;
  getUserProfile(
    userId: number,
    currentUserId?: number,
  ): Promise<UserProfile | null>;

  // Session methods
  createSession(
    userId: number,
    sessionId: string,
    expiresAt: Date,
  ): Promise<void>;
  getSession(sessionId: string): Promise<Session | null>;
  deleteSession(sessionId: string): Promise<void>;
  deleteExpiredSessions(): Promise<void>;

  // JWT methods
  isTokenRevoked(jti: string): Promise<boolean>;
  revokeToken(jti: string, expiresAt: Date): Promise<void>;
  cleanupExpiredTokens(): Promise<void>;

  // Application methods
  // Post methods
  createPost(userId: number, content: string, imageUrl?:
string): Promise<Post>;
  getPost(postId: number, currentUserId?: number): Promise<Post
| null>;
  updatePost(
    postId: number,
    userId: number,
    content?: string,
    imageUrl?: string | null,
  ): Promise<Post | null>;
  deletePost(postId: number, userId: number): Promise<boolean>;
```

```
  getPosts(
    limit?: number,
    offset?: number,
    currentUserId?: number,
  ): Promise<Post[]>;
  getUserPosts(
    userId: number,
    limit?: number,
    offset?: number,
    currentUserId?: number,
  ): Promise<Post[]>;
  getFollowingPosts(
    userId: number,
    limit?: number,
    offset?: number,
  ): Promise<Post[]>;

  // Comment methods
  createComment(
    userId: number,
    postId: number,
    content: string,
  ): Promise<Comment>;
  getPostComments(
    postId: number,
    limit?: number,
    offset?: number,
    currentUserId?: number,
  ): Promise<Comment[]>;
  updateComment(
    commentId: number,
    userId: number,
    content: string,
  ): Promise<Comment | null>;
  deleteComment(commentId: number, userId: number):
Promise<boolean>;

  // Like methods
  likePost(userId: number, postId: number): Promise<void>;
  unlikePost(userId: number, postId: number): Promise<void>;
  likeComment(userId: number, commentId: number): Promise<void>;
  unlikeComment(userId: number, commentId: number):
Promise<void>;
```

```
  isPostLiked(userId: number, postId: number): Promise<boolean>;
  isCommentLiked(userId: number, commentId: number):
Promise<boolean>;

  // Follow methods
  followUser(followerId: number, followingId: number):
Promise<void>;
  unfollowUser(followerId: number, followingId: number):
Promise<void>;
  isFollowing(followerId: number, followingId: number):
Promise<boolean>;
  getFollowers(
    userId: number,
    limit?: number,
    offset?: number,
  ): Promise<UserProfile[]>;
  getFollowing(
    userId: number,
    limit?: number,
    offset?: number,
  ): Promise<UserProfile[]>;
  searchUsers(query: string): Promise<{ users: UserProfile[] }>;
  searchPosts(query: string, currentUserId?: number): Promise<{
    posts: Post[];
  }>;
}

// DatabaseAdapter implementation for SQLite
export class SqliteAdapter implements DatabaseAdapter {
  private db: sqlite.Database<sqlite3.Database,
sqlite3.Statement>;

  constructor(private dbPath: string = "./social_media.db") {
    this.db = {} as sqlite.Database<sqlite3.Database,
sqlite3.Statement>;
  }

  async connect(): Promise<void> {
    this.db = await open({ filename: this.dbPath, driver:
sqlite3.Database });

    // Create tables
    await this.createTables();
```

```
    await this.createIndexes();

    console.log("Connected to SQLite database:", this.dbPath);
  }

  private async createTables(): Promise<void> {
    // Users table (extended)
    await this.db.run(`
      CREATE TABLE IF NOT EXISTS users (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        username TEXT UNIQUE NOT NULL,
        email TEXT UNIQUE NOT NULL,
        password_hash TEXT NOT NULL,
        display_name TEXT,
        bio TEXT,
        avatar_url TEXT,
        created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
        role TEXT DEFAULT 'user' CHECK (role IN ('user',
'admin', 'guest'))
      )
    `);

    // Sessions table
    await this.db.run(`
      CREATE TABLE IF NOT EXISTS sessions (
        id TEXT PRIMARY KEY,
        user_id INTEGER NOT NULL,
        created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
        expires_at DATETIME NOT NULL,
        FOREIGN KEY (user_id) REFERENCES users (id) ON DELETE
CASCADE
      )
    `);

    // Revoked tokens table
    await this.db.run(`
      CREATE TABLE IF NOT EXISTS revoked_tokens (
        jti TEXT PRIMARY KEY,
        revoked_at DATETIME DEFAULT CURRENT_TIMESTAMP,
        expires_at DATETIME NOT NULL
      )
    `);
```

```
    // Posts table
    await this.db.run(`
      CREATE TABLE IF NOT EXISTS posts (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        user_id INTEGER NOT NULL,
        content TEXT NOT NULL,
        image_url TEXT,
        created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
        updated_at DATETIME DEFAULT CURRENT_TIMESTAMP,
        FOREIGN KEY (user_id) REFERENCES users (id) ON DELETE
CASCADE
      )
    `);

    // Comments table
    await this.db.run(`
      CREATE TABLE IF NOT EXISTS comments (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        post_id INTEGER NOT NULL,
        user_id INTEGER NOT NULL,
        content TEXT NOT NULL,
        created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
        updated_at DATETIME DEFAULT CURRENT_TIMESTAMP,
        FOREIGN KEY (post_id) REFERENCES posts (id) ON DELETE
CASCADE,
        FOREIGN KEY (user_id) REFERENCES users (id) ON DELETE
CASCADE
      )
    `);

    // Likes table
    await this.db.run(`
      CREATE TABLE IF NOT EXISTS likes (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        user_id INTEGER NOT NULL,
        post_id INTEGER,
        comment_id INTEGER,
        created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
        FOREIGN KEY (user_id) REFERENCES users (id) ON DELETE
CASCADE,
        FOREIGN KEY (post_id) REFERENCES posts (id) ON DELETE
CASCADE,
        FOREIGN KEY (comment_id) REFERENCES comments (id) ON
```

```
DELETE CASCADE,
        UNIQUE(user_id, post_id),
        UNIQUE(user_id, comment_id),
        CHECK ((post_id IS NOT NULL AND comment_id IS NULL) OR
(post_id IS NULL AND comment_id IS NOT NULL))
      )
    `);

    // Follows table
    await this.db.run(`
      CREATE TABLE IF NOT EXISTS follows (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        follower_id INTEGER NOT NULL,
        following_id INTEGER NOT NULL,
        created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
        FOREIGN KEY (follower_id) REFERENCES users (id) ON
DELETE CASCADE,
        FOREIGN KEY (following_id) REFERENCES users (id) ON
DELETE CASCADE,
        UNIQUE(follower_id, following_id),
        CHECK (follower_id != following_id)
      )
    `);
  }

  private async createIndexes(): Promise<void> {
    // Existing indexes
    await this.db.run(
      `CREATE INDEX IF NOT EXISTS idx_sessions_expires_at ON
sessions(expires_at)`,
    );
    await this.db.run(
      `CREATE INDEX IF NOT EXISTS idx_sessions_user_id ON
sessions(user_id)`,
    );
    await this.db.run(
      `CREATE INDEX IF NOT EXISTS idx_revoked_tokens_expires ON
revoked_tokens(expires_at)`,
    );

    // New indexes for social media features
    await this.db.run(
      `CREATE INDEX IF NOT EXISTS idx_posts_user_id ON
```

```
posts(user_id)`,
    );
    await this.db.run(
      `CREATE INDEX IF NOT EXISTS idx_posts_created_at ON
posts(created_at DESC)`,
    );
    await this.db.run(
      `CREATE INDEX IF NOT EXISTS idx_comments_post_id ON
comments(post_id)`,
    );
    await this.db.run(
      `CREATE INDEX IF NOT EXISTS idx_comments_user_id ON
comments(user_id)`,
    );
    await this.db.run(
      `CREATE INDEX IF NOT EXISTS idx_likes_post_id ON
likes(post_id)`,
    );
    await this.db.run(
      `CREATE INDEX IF NOT EXISTS idx_likes_comment_id ON
likes(comment_id)`,
    );
    await this.db.run(
      `CREATE INDEX IF NOT EXISTS idx_likes_user_id ON
likes(user_id)`,
    );
    await this.db.run(
      `CREATE INDEX IF NOT EXISTS idx_follows_follower_id ON
follows(follower_id)`,
    );
    await this.db.run(
      `CREATE INDEX IF NOT EXISTS idx_follows_following_id ON
follows(following_id)`,
    );
  }

  async disconnect(): Promise<void> {
    await this.db.close();
  }

  async transaction(): Promise<Transaction> {
    const tx = new Transaction(this.db);
    await tx.begin();
```

```
    return tx;
  }

  ...
}
```

2.3. Authentication System Implementation

The authentication system is designed to handle user registration, login, and session management. It uses Express middleware for handling authentication requests and protecting routes. It also uses JWT (JSON Web Tokens) for secure authentication and session handling for API requests and maintaining flexibility.

```
import { NextFunction, Request, Response } from "express";
import bcrypt from "bcrypt";

import { LoginRequest, RegisterRequest, User } from "./
schema.ts";
import { DatabaseAdapter } from "./db.ts";
import { JWTService } from "./jwt.ts";
import { SessionManager } from "./session.ts";
import process from "node:process";

//
============================================================================
// AUTH SERVICE
//
============================================================================
export class AuthService {
  constructor(
    private db: DatabaseAdapter,
    private sessionManager: SessionManager,
    private jwtService: JWTService,
  ) {}

  async register(
    data: RegisterRequest,
  ): Promise<{ user: Omit<User, "password_hash"> }> {
    const existingUser = await
this.db.getUserByUsername(data.username);
    if (existingUser) {
      throw new Error("Username already exists");
    }
```

```
    const passwordHash = await bcrypt.hash(data.password, 10);
    const user = await this.db.createUser(
      data.username,
      data.email,
      passwordHash,
    );

    return {
      user: {
        id: user.id,
        username: user.username,
        email: user.email,
        created_at: user.created_at,
        role: user.role || "user",
      },
    };
  }

  // Traditional session login
  async loginWithSession(data: LoginRequest): Promise<{
    user: Omit<User, "password_hash">;
    sessionId: string;
  }> {
    const user = await this.authenticateUser(data);
    const sessionId = await
this.sessionManager.createSession(user.id);

    return {
      user: this.sanitizeUser(user),
      sessionId,
    };
  }

  // JWT login for APIs/mobile
  async loginWithJWT(data: LoginRequest): Promise<{
    user: Omit<User, "password_hash">;
    accessToken: string;
    refreshToken: string;
  }> {
    const user = await this.authenticateUser(data);
    const tokens = this.sessionManager.createJWTSession(user);

    return {
```

```
      user: this.sanitizeUser(user),
      ...tokens,
    };
  }

  private async authenticateUser(data: LoginRequest):
Promise<User> {
    const user = await this.db.getUserByUsername(data.username);
    if (!user) {
      throw new Error("Invalid credentials");
    }

    const isValidPassword = await bcrypt.compare(
      data.password,
      user.password_hash,
    );
    if (!isValidPassword) {
      throw new Error("Invalid credentials");
    }

    return user;
  }

  private sanitizeUser(user: User): Omit<User, "password_hash">
{
    return {
      id: user.id,
      username: user.username,
      email: user.email,
      created_at: user.created_at,
      role: user.role || "user",
    };
  }

  async logout(sessionId: string): Promise<void> {
    await this.sessionManager.destroySession(sessionId);
  }

  async logoutJWT(token: string): Promise<void> {
    await this.sessionManager.destroyJWTSession(token);
  }

  async validateSession(sessionId: string): Promise<User | null>
```

```
{
    return await this.sessionManager.validateSession(sessionId);
  }

  async validateJWTSession(
    token: string,
    requireDbCheck = false,
  ): Promise<User | null> {
    return await this.sessionManager.validateJWTSession(token,
requireDbCheck);
  }

  async refreshTokens(
    refreshToken: string,
  ): Promise<{ accessToken: string; refreshToken: string } |
null> {
    return await
this.sessionManager.refreshJWTTokens(refreshToken);
  }
}

//
================================================================================
// FLEXIBLE MIDDLEWARE
//
================================================================================
export function AuthMiddleware(
  authService: AuthService,
  options: {
    preferJWT?: boolean;
    requireDbCheck?: boolean;
  } = {},
) {
  return async (req: Request, res: Response, next: NextFunction)
=> {
    let user: User | null = null;
    let sessionId: string | undefined;

    // Try JWT first (for API/mobile)
    const authHeader = req.headers.authorization;
    if (authHeader?.startsWith("Bearer ")) {
      const token = authHeader.substring(7);
      user = await authService.validateJWTSession(
```

```
        token,
        options.requireDbCheck,
      );
      req.authType = "jwt";
      req.accessToken = token;
    }

    // Fallback to cookie session (for web)
    if (!user && !options.preferJWT) {
      sessionId = req.cookies.session_id;
      if (sessionId) {
        user = await authService.validateSession(sessionId);
        req.authType = "session";
        req.sessionId = sessionId;
      }
    }

    if (user) {
      req.user = user;
    }

    next();
  };
}

// Middleware for microservices (JWT only, no DB checks)
export function microserviceAuthMiddleware(jwtService:
JWTService) {
  return async (req: Request, res: Response, next: NextFunction)
=> {
    const authHeader = req.headers.authorization;
    if (!authHeader?.startsWith("Bearer ")) {
      return res.status(401).json({ error: "No token
provided" });
    }

    const token = authHeader.substring(7);
    const payload = jwtService.verifyTokenStateless(token);

    if (!payload) {
      return res.status(401).json({ error: "Invalid token" });
    }
```

```
    // Create user object from token data
    req.user = {
      id: payload.userId,
      username: payload.username,
      role: payload.roles || "user",
      email: "", // Not included in stateless token
      created_at: new Date().toDateString(),
      password_hash: "", // Not included in stateless token
    };

    next();
  };
}

// API-only middleware (JWT required)
export function apiAuthMiddleware(authService: AuthService) {
  return async (req: Request, res: Response, next: NextFunction)
=> {
    const authHeader = req.headers.authorization;
    if (!authHeader?.startsWith("Bearer ")) {
      return res.status(401).json({ error: "Bearer token
required" });
    }

    const token = authHeader.substring(7);
    const user = await authService.validateJWTSession(token);

    if (!user) {
      return res.status(401).json({ error: "Invalid or expired
token" });
    }

    req.user = user;
    req.accessToken = token;
    next();
  };
}

export function requireAuth(req: Request, res: Response, next:
NextFunction) {
  if (!req.user) {
    // Determine if this is an API call. When used inside a
router (e.g. app.use('/api', router)),
```

```
    // req.path is relative (e.g. '/orders'), so use
originalUrl/baseUrl checks instead.
    const originalUrl = req.originalUrl || "";
    const baseUrl = req.baseUrl || "";
    const acceptsJson = (req.headers.accept ||
"").includes("application/json");
    const isApi = originalUrl.startsWith("/api") ||
      baseUrl.startsWith("/api") || acceptsJson;

    if (isApi) {
      return res.status(401).json({
        success: false,
        error: "Authentication required",
      });
    }

    // Non-API browser request: redirect to your Next.js app's
login page
    const nextBase = process.env.NEXT_BASE_URL || "http://
localhost:3000"; // e.g. https://app.example.com
    return res.redirect(302, `${nextBase}/login`);
  }
  next();
}
```

2.4. Shared UI Components and Styling

The shared UI components are designed to be reusable across different projects, ensuring consistency in design and functionality. These components are built using React and styled with CSS-in-JS or a similar styling solution.

One of the key components that is used across all projects is the authContext component. Used to manage user authentication state, it provides methods for login, registration, logout, and user session management. This component is integrated into the main application layout, allowing for easy access to user information and authentication status throughout the application.

```
"use client";

import React, { createContext, useContext, useEffect, useState }
from "react";
import { User } from "../../schema.ts";
```

```tsx
interface AuthContextType {
  user: User | null;
  loading: boolean;
  login: (username: string, password: string) => Promise<void>;
  register: (
    username: string,
    email: string,
    password: string,
  ) => Promise<void>;
  logout: () => Promise<void>;
  refreshUser: () => Promise<void>;
}

const AuthContext = createContext<AuthContextType |
undefined>(undefined);

export function AuthProvider({ children }: { children:
React.ReactNode }) {
  const [user, setUser] = useState<User | null>(null);
  const [loading, setLoading] = useState(true);

  const fetchUser = async () => {
    try {
      const response = await fetch("/api/me", {
        method: "GET",
        credentials: "include",
        headers: { "Content-Type": "application/json" },
      });
      if (response.ok) {
        const result = await response.json();
        if (result.success) {
          setUser(result.data);
        }
      }
    } catch (error) {
      console.error("Failed to fetch user:", error);
    } finally {
      setLoading(false);
    }
  };

  const login = async (username: string, password: string) => {
    const response = await fetch("/api/login", {
```

```
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({ username, password }),
      credentials: "include",
    });

    const result = await response.json();
    if (!result.success) {
      throw new Error(result.error || "Login failed");
    }

    setUser(result.data.user);
  };

  const register = async (
    username: string,
    email: string,
    password: string,
  ) => {
    const response = await fetch("/api/register", {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({ username, email, password }),
      credentials: "include",
    });

    const result = await response.json();
    if (!result.success) {
      throw new Error(result.error || "Registration failed");
    }
  };

  const logout = async () => {
    try {
      await fetch("/api/logout", { method: "POST" ,
      credentials: "include",
      });
    } catch (error) {
      console.error("Logout error:", error);
    } finally {
      setUser(null);
    }
  };
```

```
  const refreshUser = async () => {
    setLoading(true);
    await fetchUser();
  };

  useEffect(() => {
    fetchUser();
  }, []);

  return (
    <AuthContext.Provider
      value={{ user, loading, login, register, logout,
refreshUser }}
    >
      {children}
    </AuthContext.Provider>
  );
}

export function useAuth() {
  const context = useContext(AuthContext);
  if (context === undefined) {
    throw new Error("useAuth must be used within an
AuthProvider");
  }
  return context;
}
```

3. Frontend Development with Next.js and React

This section covers the frontend development using Next.js and React, focusing on building a responsive and interactive user interface. The frontend is designed to work seamlessly with the backend services, providing a smooth user experience.

3.1. Next.js Framework with TypeScript

Next.js is a powerful React framework that enables server-side rendering, static site generation, and API routes. It provides a robust development environment with built-in support for TypeScript, making it easier to build type-safe applications as well as deploy them efficiently through its tight integration with Vercel Platform.

3.2. React Components and State Management

React components are the building blocks of the user interface in Next.js applications. They allow developers to create reusable UI elements that can manage their own state and lifecycle. State management can be handled using React's built-in hooks like `useState` and `useEffect`, or with more advanced libraries like Redux or Zustand for larger applications. This enables developers to create dynamic and interactive user interfaces that respond to user actions and application state changes.

The core idea of React is to break down the UI into smaller, manageable components that can be composed together. Each component can maintain its own local state, and when the state changes, React efficiently updates the UI to reflect those changes. This declarative approach simplifies the development process and makes it easier to reason about the application.

3.3. API Integration and Data Fetching

The Next.js front-end can easily integrate with backend APIs to fetch and display data. This is typically done using the `fetch` API or libraries like Axios. Next.js also provides built-in data fetching methods like `getStaticProps` and `getServerSideProps`, which allow developers to fetch data at build time or request time, respectively. This flexibility enables developers to choose the best approach for their specific use case, whether it's static site generation, server-side rendering, or client-side fetching.

Here, the chosen approach is to simply proxy fetch calls to the backend Express server to handle API requests. This allows the frontend to remain

decoupled from the backend implementation while still being able to access the necessary data and functionality.

3.4. Responsive Design Implementation

Responsive design is crucial for ensuring that web applications work well on various devices and screen sizes. In Next.js, responsive design can be achieved using CSS media queries, CSS-in-JS libraries like styled-components or Emotion, or utility-first CSS frameworks like Tailwind CSS. These tools allow developers to create flexible layouts that adapt to different screen sizes and orientations, providing a consistent user experience across devices.

4. Backend Development with Express.js

This section focuses on the backend development using Express.js, which serves as the web server for handling API requests and serving the frontend application.

The backend is responsible for processing requests, interacting with the database, and returning responses to the frontend. It also handles authentication and authorization, ensuring that only authorized users can access certain resources.

4.1. Express.js Server Setup with TypeScript

To set up an Express.js server with TypeScript, follow these steps:

1. **Initialize the project**:

```
mkdir express-server
cd express-server
npm init -y
```

2. **Install TypeScript and necessary dependencies**:

```
npm install express body-parser cors
npm install typescript ts-node @types/node @types/express --
save-dev
```

3. **Create a `tsconfig.json` file**:

```
{
  "compilerOptions": {
    "target": "ES6",
    "module": "commonjs",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true,
    "outDir": "./dist",
    "rootDir": "./src"
  },
  "include": ["src/**/*"],
  "exclude": ["node_modules", "dist"]
}
```

4.2. RESTful API Design and Implementation

RESTful API design is a common architectural style for building web services. It emphasizes stateless communication, resource-based URLs, and standard

HTTP methods (GET, POST, PUT, DELETE). In Express.js, you can define routes that correspond to these methods and handle requests accordingly.

```
async function startServer() {
  const db = new SqliteAdapter();
  await db.connect();

  // JWT configuration
  const JWT_SECRET = Deno.env.get("JWT_SECRET") ||
    "your-super-secret-jwt-key-change-in-production";
  const JWT_REFRESH_SECRET = Deno.env.get("JWT_REFRESH_SECRET")
||
    "your-super-secret-refresh-key-change-in-production";

  // Initialize services
  const jwtService = new JWTService(JWT_SECRET,
JWT_REFRESH_SECRET, db);
  const sessionManager = new SessionManager(db, jwtService);
  const authService = new AuthService(db, sessionManager,
jwtService);

  const chatService = new ChatService(db);
  // chatService.registerSource(new ScraperSource());

  // Cleanup expired sessions
  setInterval(() => {
    sessionManager.deleteExpiredSessions().catch(console.error);
  }, 60 * 60 * 1000);

  const PORT = process.env.PORT || 8000;
  const UI_ORIGIN = Deno.env.get("UI_ORIGIN") || "http://
localhost:3000";

  const app = express();
  const apiRoutes = express.Router();
  const mobileApiRoutes = express.Router();

  app.use(express.json());
  app.use(cookieParser());

  app.use(
    cors({
      origin: [UI_ORIGIN, "http://localhost:3000", "http://
```

```
localhost:3001"], // Support multiple origins for development
      credentials: true,
      methods: ["GET", "POST", "PUT", "PATCH", "DELETE",
"OPTIONS"],
      allowedHeaders: ["Content-Type", "Authorization"],
    }),
  );

  //
================================================================================
  // WEB API ROUTES (Cookie-based sessions + JWT support)
  //
================================================================================

  // Authentication middleware for web routes
  apiRoutes.use(AuthMiddleware(authService, { preferJWT:
false }));

  // Auth routes
  apiRoutes.post("/register", async (req: Request, res:
Response) => {
    try {
      const data = RegisterSchema.parse(req.body);
      const result = await authService.register(data);
      res.status(201).json({ success: true, data: result });
    } catch (error) {
      const err = error as Error;
      res.status(400).json({ success: false, error:
err.message });
    }
  });

  apiRoutes.post("/login", async (req: Request, res: Response)
=> {
    try {
      const data = LoginSchema.parse(req.body);
      const result = await authService.loginWithSession(data);
      res.cookie("session_id", result.sessionId, {
        httpOnly: true,
        secure: process.env.NODE_ENV === "production",
        sameSite: "strict",
        maxAge: 24 * 60 * 60 * 1000,
      });
```

```
      res.status(200).json({
        success: true,
        data: { user: result.user },
      });
    } catch (error) {
      const err = error as Error;
      res.status(401).json({ success: false, error:
err.message });
    }
  });

  apiRoutes.post("/logout", async (req: Request, res: Response)
=> {
    const sessionId = req.sessionId;
    if (sessionId) {
      await authService.logout(sessionId);
    }
    res.clearCookie("session_id");
    res.json({ success: true });
  });

  apiRoutes.get("/me", requireAuth, async (req: Request, res:
Response) => {
    console.log("/api/me endpoint hit");
    const user = req.user;
    if (!user) {
      return res.status(401).json({ success: false, error:
"Unauthorized" });
    }
    res.json({
      success: true,
      data: {
        id: user.id,
        username: user.username,
        email: user.email,
        created_at: user.created_at,
        display_name: user.display_name,
        bio: user.bio,
        avatar_url: user.avatar_url,
      },
    });
  });
```

```
  // Application specific routes
  // Get all chats with filtering
  apiRoutes.get("/chats", async (req: Request, res: Response) =>
{
    try {
      const {
        platform,
        is_favorite,
        is_archived,
        tags,
        limit = "50",
        offset = "0",
      } = req.query;

      const filters = {
        platform: platform as string,
        is_favorite: is_favorite === "true"
          ? true
          : is_favorite === "false"
          ? false
          : undefined,
        is_archived: is_archived === "true"
          ? true
          : is_archived === "false"
          ? false
          : undefined,
        tags: tags ? (tags as string).split(",") : undefined,
        limit: parseInt(limit as string),
        offset: parseInt(offset as string),
      };

      const chats = await chatService.getChats(filters);
      res.json({ success: true, data: chats });
    } catch (error) {
      res.status(500).json({ success: false, error: (error as
Error).message });
    }
  });

  // Get chat by ID with messages
  apiRoutes.get("/chats/:id", async (req: Request, res:
Response) => {
    try {
```

```
      const id = parseInt(req.params.id);
      const chat = await chatService.getChatWithMessages(id);

      if (!chat) {
        return res.status(404).json({
          success: false,
          error: "Chat not found",
        });
      }

      res.json({ success: true, data: chat });
    } catch (error) {
      res.status(500).json({ success: false, error: (error as
Error).message });
    }
  });

  ...

  // Mount routes
  app.use("/api", apiRoutes);

  // Health check endpoint
  app.get("/health", (req: Request, res: Response) => {
    res.json({
      success: true,
      status: "healthy",
      timestamp: new Date(),
    });
  });

  // Start server
  // Start server
  app.listen(PORT, () => {
    console.log(`📡 Server: http://localhost:${PORT}`);
    console.log(`🔍 Health: http://localhost:${PORT}/health`);
    console.log("");
    console.log("📋 Available Endpoints:");
    ...
  });
}
```

4.3. Database Integration Across Multiple Types

Here we use the zod library to define the database schema and data validation. This allows us to ensure that the data being sent to and from the database is in the correct format, reducing the risk of errors and improving the overall reliability of the application.

```
//
=============================================================================
// TYPES AND SCHEMAS
//
=============================================================================

import { z } from "zod";

export const LoginSchema = z.object({
    username: z.string().min(3).max(50),
    password: z.string().min(6).max(100),
  });

export const RegisterSchema = z.object({
  username: z.string().min(3).max(50),
  password: z.string().min(6).max(100),
  email: z.email(),
});

export type LoginRequest = z.infer<typeof LoginSchema>;
export type RegisterRequest = z.infer<typeof RegisterSchema>;

export interface User {
  id: number;
  username: string;
  email: string;
  password_hash: string;
  created_at: string;
  role: "user" | "admin" | "guest";
  bio?: string;
  avatar_url?: string;
  display_name?: string;
  followers_count?: number;
  following_count?: number;
  posts_count?: number;
}
```

```
export interface Session {
  id: string;
  user_id: number;
  created_at: string;
  expires_at: string;
}

// Application-specific types
export const CreatePostSchema = z.object({
  content: z.string().min(1).max(1000),
  image_url: z.optional(z.url()),
});

export const UpdatePostSchema = z.object({
  content: z.string().min(1).max(1000).optional(),
  image_url: z.optional(z.url()).nullable(),
});

export const CreateCommentSchema = z.object({
  content: z.string().min(1).max(500),
});

export const UpdateProfileSchema = z.object({
  bio: z.string().max(500).optional(),
  avatar_url: z.optional(z.url()),
  display_name: z.string().min(1).max(100).optional(),
});

export type CreatePostRequest = z.infer<typeof
CreatePostSchema>;
export type UpdatePostRequest = z.infer<typeof
UpdatePostSchema>;
export type CreateCommentRequest = z.infer<typeof
CreateCommentSchema>;
export type UpdateProfileRequest = z.infer<typeof
UpdateProfileSchema>;
```

4.4. Authentication and Authorization

The authentication and authorization system is built using a combination of JWT (JSON Web Tokens) and session management. This allows for secure user authentication and session handling, ensuring that only authorized users can access certain resources and perform specific actions.

We simply apply the middleware to the API routes to protect them. The middleware checks for the presence of a valid JWT token or session cookie, validates the user, and attaches the user information to the request object.

```
  const app = express();
  const apiRoutes = express.Router();

  app.use(express.json());
  app.use(cookieParser());

  app.use(
    cors({
      origin: [UI_ORIGIN, "http://localhost:3000", "http://
localhost:3001"], // Support multiple origins for development
      credentials: true,
      methods: ["GET", "POST", "PUT", "PATCH", "DELETE",
"OPTIONS"],
      allowedHeaders: ["Content-Type", "Authorization"],
    }),
  );

  apiRoutes.use(AuthMiddleware(authService, { preferJWT:
false }));
```

5. Project 1: E-commerce Platform

5.1. Project Requirements and Features

The e-commerce platform is a full-stack web application built using the MERN stack with TypeScript. The project demonstrates the implementation of a complete online shopping system with the following core features:

- **User Authentication**: Secure registration and login system with session management
- **Product Catalog**: Display of products with details, pricing, and inventory management
- **Shopping Cart**: Add/remove items, quantity management, and cart persistence
- **Order Management**: Order creation, tracking, and history for users
- **Responsive Design**: Mobile-friendly interface using Next.js and Tailwind CSS
- **API Architecture**: RESTful API design with both web and mobile endpoints

The platform supports both traditional session-based authentication for web users and JWT-based authentication for mobile applications, demonstrating flexibility in authentication strategies.

5.2. Product Management and Shopping Cart

The product management system is implemented with a SQLite database containing a products table with the following schema:

```
CREATE TABLE products (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  name TEXT NOT NULL,
  description TEXT,
  price REAL NOT NULL,
  stock_quantity INTEGER NOT NULL,
  image_url TEXT
);
```

The shopping cart functionality includes:

- **Cart Persistence**: Cart items are stored in the database and linked to user sessions
- **Inventory Validation**: Stock quantity checks before order placement

- **Price Calculation**: Real-time total calculation with tax and shipping considerations
- **Order Processing**: Transaction-based order creation ensuring data consistency

Key API endpoints for cart management:
- `GET /api/products` - Retrieve all products
- `GET /api/products/:id` - Get specific product details
- `POST /api/orders` - Create new order with cart items
- `GET /api/orders` - Retrieve user's order history

5.3. User Interface and Shopping Experience

The frontend is built with Next.js 15 and React, featuring:

Page Structure:
- Landing page with hero section and product highlights
- Product listing page with filtering and search capabilities
- Individual product detail pages with image galleries
- Shopping cart page with item management
- User dashboard for order tracking and profile management
- Responsive login/registration forms

User Experience Features:
- Automatic authentication state management using React Context
- Real-time cart updates without page refreshes
- Loading states and error handling for all API interactions
- Mobile-responsive design using Tailwind CSS
- Protected routes requiring authentication for checkout

The authentication context provides seamless user state management:

```
const { user, loading, login, register, logout } = useAuth();
```

5.4. Code Implementation and Database Schema

The e-commerce platform uses a modular architecture with clear separation of concerns:

Backend Structure:
- `main.ts` - Express server setup with CORS and middleware configuration
- `db.ts` - Database abstraction layer with SQLite implementation
- `auth.ts` - Authentication service with session and JWT support

- `schema.ts` - TypeScript interfaces and Zod validation schemas

Database Schema:

```
-- Core tables
users (id, username, email, password_hash, created_at)
sessions (id, user_id, expires_at, created_at)
products (id, name, description, price, stock_quantity,
image_url)
orders (id, user_id, total_amount, status, created_at)
order_items (id, order_id, product_id, quantity, price)
revoked_tokens (jti, revoked_at, expires_at)
```

Key Implementation Features:

- Transaction-based order processing to ensure data consistency
- Dual authentication system (sessions for web, JWT for mobile)
- Input validation using Zod schemas
- Error handling with proper HTTP status codes
- Database connection pooling and query optimization

The platform demonstrates production-ready practices including proper error handling, input validation, and secure authentication mechanisms.

6. Project 2: Mini Social Media Application

6.1. Social Media Features and Requirements

The mini social media application is a comprehensive platform that replicates core social networking features. Built with the same shared architecture as the e-commerce platform, it demonstrates advanced social interaction capabilities:

Core Features:

- **User Profiles**: Customizable profiles with bio, avatar, and display names
- **Post Creation**: Text and image posts with rich content support
- **Social Interactions**: Like, comment, and share functionality
- **Follow System**: User following/follower relationships
- **News Feed**: Personalized content feed based on followed users
- **Search Functionality**: Search for users and posts with real-time results
- **Content Management**: Edit and delete posts and comments
- **Real-time Updates**: Live interaction counts and notifications

Advanced Features:

- Content moderation capabilities
- Privacy settings for posts and profiles
- Hashtag support for content discovery
- User engagement analytics
- Mobile-responsive design with touch-friendly interactions

6.2. User Profiles and Content Management

The user profile system extends the basic authentication with rich social features:

Profile Management:

```
interface UserProfile {
  id: number;
  username: string;
  display_name?: string;
  bio?: string;
  avatar_url?: string;
  created_at: string;
  followers_count: number;
  following_count: number;
  posts_count: number;
```

```
  is_following?: boolean;
  is_own_profile?: boolean;
}
```

Content Creation and Management:

- Rich text post creation with image upload support
- Post editing with version history
- Content validation using Zod schemas
- Automatic content statistics (likes, comments, shares)
- User-generated content moderation

API Endpoints for Content Management:

- `POST /api/posts` - Create new post
- `GET /api/posts` - Retrieve paginated posts
- `PUT /api/posts/:id` - Update existing post
- `DELETE /api/posts/:id` - Delete post
- `GET /api/users/:id/posts` - Get user's posts
- `PATCH /api/profile` - Update user profile

6.3. Feed and Interaction System

The social interaction system implements a sophisticated engagement model:

Feed Algorithm:

- Chronological feed of followed users' posts
- Engagement-based content ranking
- Real-time updates for new content
- Infinite scroll pagination for smooth user experience

Interaction Features:

```
// Like system for posts and comments
POST /api/posts/:id/like
POST /api/comments/:id/like

// Comment system with nested replies
POST /api/posts/:id/comments
GET /api/posts/:id/comments
PUT /api/comments/:id
DELETE /api/comments/:id

// Follow system
POST /api/users/:id/follow
```

```
GET /api/users/:id/followers
GET /api/users/:id/following
```

Database Schema for Social Features:

```
-- Extended user table with social features
users (id, username, email, password_hash, display_name, bio,
avatar_url, created_at)

-- Posts and content
posts (id, user_id, content, image_url, created_at, updated_at)
comments (id, post_id, user_id, content, created_at, updated_at)

-- Social interactions
likes (id, user_id, post_id, comment_id, created_at)
follows (id, follower_id, following_id, created_at)

-- Indexes for performance
CREATE INDEX idx_posts_user_id ON posts(user_id);
CREATE INDEX idx_posts_created_at ON posts(created_at DESC);
CREATE INDEX idx_follows_follower_id ON follows(follower_id);
```

6.4. Implementation Details and User Interface

The social media application showcases advanced React patterns and state management:

Frontend Architecture:

- Component-based architecture with reusable UI elements
- Context API for global state management (auth, notifications)
- Custom hooks for data fetching and caching
- Optimistic UI updates for immediate user feedback
- Real-time interaction counters

Key React Components:

```
// Post component with interaction features
<PostCard
  post={post}
  onLike={handleLike}
  onComment={handleComment}
  onShare={handleShare}
/>

// User profile with follow functionality
```

```
<UserProfile
  user={user}
  isFollowing={isFollowing}
  onFollow={handleFollow}
/>

// Feed with infinite scroll
<InfiniteFeed
  posts={posts}
  onLoadMore={loadMorePosts}
  loading={loading}
/>
```

Backend Service Layer: The `SocialMediaService` class encapsulates all social media business logic:

```
export class SocialMediaService {
  // Profile management
  async updateProfile(userId: number, data:
UpdateProfileRequest)
  async getUserProfile(userId: number, currentUserId?: number)

  // Post management
  async createPost(userId: number, data: CreatePostRequest)
  async getPosts(limit: number, offset: number, currentUserId?:
number)
  async getFeed(userId: number, limit: number, offset: number)

  // Social interactions
  async togglePostLike(userId: number, postId: number)
  async toggleFollow(followerId: number, followingId: number)

  // Search functionality
  async searchUsers(query: string)
  async searchPosts(query: string, limit: number,
currentUserId?: number)
}
```

Performance Optimizations:

- Database indexing for fast queries
- Pagination for large datasets
- Caching strategies for frequently accessed data
- Optimized SQL queries with proper joins

- Image optimization and lazy loading

The application demonstrates production-ready social media features with scalable architecture and responsive design.

7. Project 3: LLM Chat Aggregator with Web Scraping

7.1. Project Overview and LLM Integration

The LLM Chat Aggregator is the most sophisticated project in the internship portfolio, demonstrating advanced web scraping, data aggregation, and multi-platform integration capabilities. This application serves as a centralized hub for managing conversations across multiple AI platforms:

Supported Platforms:

- **ChatGPT** (OpenAI) - GPT-3.5 and GPT-4 conversations
- **Claude** (Anthropic) - Claude 3 and Claude 2 chat sessions
- **Grok** (X/Twitter) - Grok AI conversations
- **Gemini** (Google) - Bard/Gemini chat interactions

Core Capabilities:

- Automated chat history synchronization from multiple platforms
- Unified chat interface for viewing conversations across platforms
- Advanced search and filtering across all imported chats
- Chat organization with tags, favorites, and archiving
- Platform account management with secure cookie storage
- Real-time synchronization with external AI platforms
- Export functionality for chat backups and analysis

Technical Innovation:

- Browser automation using Playwright for web scraping
- Cookie-based authentication for platform access
- Robust error handling for unreliable web scraping
- Scalable architecture supporting multiple concurrent sources
- Database optimization for large chat datasets

7.2. Web Scraping with Playwright

The web scraping implementation uses Playwright for browser automation, providing reliable access to AI platforms that don't offer public APIs:

Playwright Implementation Architecture:

```
export class ChatGPTSource implements ChatSource {
  name = "ChatGPT";
  platform = "chatgpt" as const;
  private browser: any | null = null;
```

```
  private context: any | null = null;
  private page: Page | null = null;

  constructor(private cookies: {
    name: string; value: string; domain?: string; path?: string;
  }[]) {}

  private async initBrowser(): Promise<void> {
    this.browser = await chromium.launch({ headless: true });
    this.context = await this.browser.newContext();
    await this.context.addCookies(this.cookies.map(cookie => ({
      ...cookie,
      domain: cookie.domain || ".openai.com",
      path: cookie.path || "/"
    })));
    this.page = await this.context.newPage();
  }
}
```

Scraping Strategies by Platform:

1. **ChatGPT Scraping:**
   - Navigate to `https://chat.openai.com/`
   - Extract chat list from navigation sidebar
   - Parse individual conversations using DOM selectors
   - Handle dynamic loading and infinite scroll
2. **Claude Scraping:**
   - Access `https://claude.ai/chats`
   - Extract conversation metadata from chat list
   - Navigate to individual chats for message extraction
   - Handle Anthropic's specific DOM structure
3. **Grok Scraping:**
   - Integrate with X/Twitter's Grok interface
   - Handle OAuth-based authentication flow
   - Extract conversations from embedded chat interface
4. **Gemini Scraping:**
   - Access Google's Gemini interface
   - Handle Google account authentication
   - Parse Material Design components for chat data

Error Handling and Reliability:

```
async isAvailable(): Promise<boolean> {
  try {
    await this.page.goto("https://chat.openai.com/", {
      waitUntil: "networkidle"
    });
    const isLoggedIn = await this.page.evaluate(() =>
      !document.querySelector('[data-testid="login-button"]')
    );
    return isLoggedIn;
  } catch {
    return false;
  }
}
```

7.3. Chat Interface and Multi-Provider Support

The unified chat interface provides a seamless experience for managing conversations across platforms:

Chat Management Features:

```
interface Chat {
  id: number;
  title: string;
  platform: "chatgpt" | "claude" | "grok" | "gemini";
  platform_chat_id: string;
  url: string;
  tags: string[];
  is_favorite: boolean;
  is_archived: boolean;
  message_count: number;
  created_at: string;
  updated_at: string;
  account_id?: number; // Links to platform account
}

interface Message {
  id: number;
  chat_id: number;
  role: "user" | "assistant" | "system";
  content: string;
  timestamp: Date;
  metadata?: Record<string, any>;
```

```
  created_at: string;
}
```

Platform Account Management: The system supports multiple accounts per platform, enabling users to aggregate chats from different accounts:

```
interface PlatformAccount {
  id: number;
  user_id: number;
  platform: string;
  account_name: string;
  cookies: Array<{name: string; value: string; domain?: string}
>;
  created_at: string;
}
```

API Endpoints for Chat Management:

- `GET /api/chats` - List all chats with filtering options
- `GET /api/chats/:id` - Get specific chat with messages
- `POST /api/chats` - Create new chat manually
- `PATCH /api/chats/:id` - Update chat metadata (tags, favorites)
- `DELETE /api/chats/:id` - Delete chat and messages
- `GET /api/chats/search/:query` - Search across all chats
- `POST /api/chats/sync` - Trigger synchronization from all sources
- `GET /api/chats/stats/overview` - Get aggregated statistics

Advanced Filtering and Search:

```
const filters = {
  platform: "chatgpt" | "claude" | "grok" | "gemini",
  is_favorite: boolean,
  is_archived: boolean,
  tags: string[],
  limit: number,
  offset: number,
  account_id: number
};
```

7.4. Backend Processing and Data Pipeline

The backend implements a sophisticated data processing pipeline for handling chat synchronization:

ChatService Architecture:

```
export class ChatService {
  private sources: Map<string, { source: ChatSource; accountId:
number }> = new Map();

  async loadUserSources(userId: number): Promise<void> {
    const accounts = await
this.db.getPlatformAccountsByUserId(userId);
    for (const account of accounts) {
      const source =
this.createSourceForPlatform(account.platform, account.cookies);
      this.sources.set(`${account.platform}-
${account.account_name}`,
                       { source, accountId: account.id });
    }
  }

  async syncAllSources(): Promise<{ success: number; errors:
string[] }> {
    const results = { success: 0, errors: [] as string[] };

    for (const [key, { source, accountId }] of
this.sources.entries()) {
      try {
        if (!(await source.isAvailable())) {
          results.errors.push(`${key}: Source not available`);
          continue;
        }

        const chats = await source.getChats();
        for (const chatData of chats) {
          const messages = await
source.getMessages(chatData.platform_chat_id);
          await this.db.createChatWithMessages(chatData,
messages, accountId);
          results.success++;
        }
      } catch (error) {
        results.errors.push(`${key}: ${error}`);
      } finally {
        await source.close();
      }
    }
```

```
    return results;
  }
}
```

Database Schema for Chat Aggregation:

```
-- Platform accounts for multi-account support
platform_accounts (
  id INTEGER PRIMARY KEY,
  user_id INTEGER,
  platform TEXT,
  account_name TEXT,
  cookies TEXT, -- JSON stored as text
  created_at DATETIME,
  UNIQUE(user_id, platform, account_name)
);

-- Chats with platform linking
chats (
  id INTEGER PRIMARY KEY,
  title TEXT,
  platform TEXT,
  platform_chat_id TEXT,
  url TEXT,
  tags TEXT, -- JSON array as text
  is_favorite BOOLEAN DEFAULT FALSE,
  is_archived BOOLEAN DEFAULT FALSE,
  message_count INTEGER DEFAULT 0,
  account_id INTEGER,
  created_at DATETIME,
  updated_at DATETIME,
  UNIQUE(platform, platform_chat_id, account_id)
);

-- Messages with rich metadata
messages (
  id INTEGER PRIMARY KEY,
  chat_id INTEGER,
  role TEXT CHECK (role IN ('user', 'assistant', 'system')),
  content TEXT,
  timestamp DATETIME,
  metadata TEXT, -- JSON metadata
  created_at DATETIME,
```

```
  FOREIGN KEY (chat_id) REFERENCES chats (id) ON DELETE CASCADE
);
```

Performance Optimizations:

- Batch processing for large chat imports
- Incremental synchronization to avoid duplicate data
- Database indexing for fast search across messages
- Connection pooling for concurrent scraping operations
- Rate limiting to respect platform usage policies
- Caching strategies for frequently accessed chats

Security Considerations:

- Secure cookie storage with encryption
- User isolation for multi-tenant support
- Input validation for all scraped content
- Error logging without exposing sensitive data
- Graceful handling of authentication failures

The Chat Aggregator demonstrates enterprise-level data processing capabilities with robust error handling, scalable architecture, and comprehensive platform integration.

8. Deployment and Production Setup

This section outlines the deployment and production setup for the projects, ensuring that they are ready for real-world use. It covers environment configuration, database migration, application deployment, and basic monitoring and maintenance practices.

The main goal is to ensure that the applications are secure, performant, and maintainable in a production environment. The platform chosen here is Vercel for its ease of use with Next.js applications, and Docker for the Express.js backend with MongoDB.

8.1. Environment Configuration

Environment configuration is crucial for ensuring that applications run correctly in different environments (development, staging, production). This includes setting environment variables for sensitive information like API keys, database connection strings, and other configuration settings.

In Next.js, environment variables can be defined in a `.env.local` file for local development and `.env.production` for production. These variables can be accessed using `process.env.VARIABLE_NAME`.

In the Express.js backend, environment variables can be set in a `.env` file and accessed using the `dotenv` package. This allows for secure storage of sensitive information without hardcoding it into the source code.

8.2. Application Deployment

Deployment involves taking the application from a development environment and making it available for users. This typically includes building the application, configuring the server, and deploying the code to a hosting platform.

In the case of Next.js applications, Vercel provides a seamless deployment experience. Developers can connect their GitHub repository to Vercel, and it will automatically build and deploy the application whenever changes are pushed to the main branch.

For the Express.js backend, Docker can be used to create a containerized application. This allows for consistent deployment across different environments. The Dockerfile can be configured to build the application, install dependencies, and run the server.

8.3. Basic Monitoring and Maintenance

Monitoring and maintenance are critical for ensuring the application runs smoothly in production. This includes tracking performance metrics, error logging, and regular updates.

Monitoring can be achieved using tools like Sentry for error tracking, and Prometheus or Grafana for performance monitoring. These tools can provide insights into application performance, user behavior, and potential issues.

What is also essential is the use of logging to capture important events and errors in the application. This can be done using libraries like Winston or Morgan in the Express.js backend, and the built-in logging capabilities of Next.js.

9. Conclusion and References

This internship project has provided a comprehensive overview of full-stack web development, covering a wide range of technologies and methodologies. From building a robust backend with Express.js and TypeScript to creating dynamic user interfaces with Next.js and React and finally high-level use of database integration with MongoDB and SQLite, the projects demonstrate the application of modern web development practices.

9.1. Project Outcomes and Applications

The projects developed during this internship showcase the ability to create scalable, maintainable, and user-friendly web applications. The e-commerce platform demonstrates core e-commerce functionalities, while the mini social media application highlights social interaction features.

The LLM Chat Aggregator is attempt to solve a personal problem of managing conversations across multiple AI platforms, showcasing advanced web scraping and data aggregation techniques. Each project has been designed with a focus on user experience, performance, and security, making them suitable for real-world applications.

9.2. References and Resources

No.	URL	Description
1	https://react.dev/learn	React Documentation
2	https://nodejs.org/docs/latest/api/	Node.js API Documentation
3	https://www.mongodb.com/docs/	MongoDB Documentation
4	https://www.w3schools.com/	W3Schools Web Development Tutorials
5	https://expressjs.com/en/5x/api.html	Express.js API Documentation
6	https://www.typescriptlang.org/docs/	TypeScript Documentation
7	https://playwright.dev/docs/intro	Playwright Documentation
8	https://deno.land/manual/getting_started/installation	Deno Installation Guide
9	https://www.zod.dev/	Zod Schema Validation Library
10	https://vercel.com/docs	Vercel Documentation
11	https://www.docker.com/	Docker Documentation

Table 1: References and Resources