

ICE4016 데이터베이스설계

<TermProject>

보고서 작성 서약서

1. 나는 타학생의 보고서를 베끼거나 여러 보고서의 내용을 짜집기하지 않겠습니다.

2. 나는 보고서의 주요 내용을 인터넷사이트 등을 통해 얻지 않겠습니다.

3. 나는 보고서의 내용을 조작하지 않겠습니다.

4. 나는 보고서 작성에 참고한 문헌의 출처를 밝히겠습니다.

5. 나는 나의 보고서를 제출 전에 타학생에게 보여주지 않겠습니다.

나는 보고서 작성시 윤리에 어긋난 행동을 하지 않고 정보통신공학인으로서 나의 명예를 지킬 것을 맹세합니다.

2023년 12 월 16 일

학부 정보통신공학

학년 3

성명 김지훈

학번 12191748



1. 개요

본 텀 프로젝트는 병원 관리 시스템의 데이터베이스와 웹 인터페이스를 설계하고 구현하는 것을 목표로 합니다. 이 시스템은 관리자, 의사, 간호사, 그리고 환자 페이지를 포함하여 다양한 사용자 역할에 맞춘 기능을 제공합니다. 주요 작업으로는 데이터베이스의 ERD 설계, 테이블 생성 및 정규화, 인덱스와 뷰의 구현, 데이터 삽입 스크립트 작성, 트랜잭션 관리, 그리고 세부적인 웹 페이지의 구현이 포함됩니다. 이 프로젝트는 병원 운영에 필수적인 데이터를 관리하고, 사용자가 효율적으로 데이터에 접근하고 조작할 수 있는 웹 기반 인터페이스를 목표로 합니다.

1. 상세 설계 내용

**<각 테이블 생성 및 역할, 이유 설명>**

**텍스트, 스크린샷, 도표, 폰트이(가) 표시된 사진

자동 생성된 설명**

**[DEPARTMENT 테이블]**

목적: 병원의 각 부서 정보를 담고 있습니다.

필드:

DepartmentID: 부서 고유 번호 (PK, AUTO\_INCREMENT).

DepartmentName: 부서의 이름 (NOT NULL).

PhoneNumber: 부서의 연락처.

**[USER 테이블]:** 이는 로그인할 때 user 테이블에서 Name, Address, PhoneNumber를 같이 관리하도록 했습니다.

목적: 시스템 사용자의 인증 정보를 관리합니다.

필드:

UserID: 사용자 고유 번호 (PK, AUTO\_INCREMENT).

Username: 사용자 이름.

Password: 사용자 비밀번호.

Role: 사용자 역할 ('Admin', 'Doctor', 'Nurse', 'Patient').

**[DOCTOR 테이블]**

목적: 병원 의사들에 대한 정보를 담고 있습니다.

필드:

DoctorID: 의사의 고유 번호 (PK, AUTO\_INCREMENT).

Name: 의사 이름.

Address: 주소를 담고 있습니다.

PhoneNumber: 전화번호.

DepartmentID: 속한 부서 번호 (FK).

UserID: 시스템 사용자 ID (FK).

**[PATIENT 테이블]**

목적: 환자의 개인 정보를 담고 있습니다.

필드:

PatientID: 환자의 고유 번호 (PK, AUTO\_INCREMENT).

SocialSecurityNumber, Gender, Address, BloodType, Height, Weight, PhoneNumber: 개인 정보.

UserID: 시스템 사용자 ID (FK).

**[EXAMINATION 테이블]**

목적: 환자의 검사 기록을 담고 있습니다.

필드:

ExaminationID: 검사 기록의 고유 번호 (PK, AUTO\_INCREMENT).

ExaminationDateTime: 검사 시간.

ExaminationDetails: 검사 세부 사항.

DoctorID: 담당 의사 번호 (FK).

PatientID: 환자 번호 (FK).

**[INPATIENT 테이블]**

목적: 입원 환자의 기록을 담고 있습니다.

필드:

InpatientID: 입원 기록의 고유 번호 (PK, AUTO\_INCREMENT).

PatientID: 환자 번호 (FK).

RoomInformation, AdmissionDateTime, DischargeDateTime: 입원 정보.

**[MEDICAL\_SPECIALTY 테이블]**

목적: 병원에서 제공하는 의료 전문 분야를 나타냅니다.

필드:

SpecialtyID: 전문 분야의 고유 번호 (PK, AUTO\_INCREMENT).

DepartmentID: 속한 부서 번호 (FK).

SpecialtyName: 전문 분야 이름.

**[NURSE 테이블]**

목적: 간호사에 대한 정보를 담고 있습니다.

필드:

NurseID: 간호사의 고유 번호 (PK, AUTO\_INCREMENT).

Name: 간호사 이름.

Address: 주소를 담고 있습니다.

PhoneNumber: 전화번호.

DepartmentID: 속한 부서 번호 (FK).

UserID: 시스템 사용자 ID (FK).

**[RESERVATION 테이블]**

목적: 환자의 예약 정보를 담고 있습니다.

필드:

ReservationNumber: 예약의 고유 번호 (PK, AUTO\_INCREMENT).

ReservationDateTime: 예약 날짜와 시간.

DepartmentID: 예약된 부서 번호 (FK).

PatientID: 예약한 환자 번호 (FK).

**[TREATMENT 테이블]**

목적: 환자의 치료 기록을 담고 있습니다.

필드:

TreatmentID: 치료 기록의 고유 번호 (PK, AUTO\_INCREMENT).

TreatmentDateTime: 치료 날짜와 시간.

TreatmentDetails: 치료 세부 사항.

NurseID: 담당 간호사 번호 (FK).

PatientID: 치료받은 환자 번호 (FK).

**[DEPARTMENT - DOCTOR 관계]**

DEPARTMENT 테이블과 DOCTOR 테이블 간의 관계는 1:N 관계로, 한 부서에 여러 명의 의사가 있을 수 있기 때문입니다. 이 관계는 실선으로 표현되는데, 이는 모든 의사가 반드시 하나의 부서에 속해야 하기 때문입니다. 각 의사는 고유한 DepartmentID를 외래 키로 가지고 있으며, 이는 의사가 부서에 종속되어 있다는 것을 나타냅니다.

**[DEPARTMENT - NURSE 관계]**

DEPARTMENT 테이블과 NURSE 테이블 간에도 1:N 관계가 존재합니다. 이는 한 부서에 여러 명의 간호사가 있을 수 있기 때문이며, 각 간호사는 특정 부서에 속해야 합니다. 실선은 간호사가 부서 없이는 존재할 수 없음을 나타내며, DepartmentID는 간호사 테이블에서 외래 키로 사용됩니다.

**[DOCTOR - PATIENT 관계]**

DOCTOR 테이블과 PATIENT 테이블은 N:M 관계를 가질 수 있습니다. 한 의사가 여러 환자를 담당할 수 있고, 한 환자도 여러 의사에게 진료를 받을 수 있기 때문입니다. 그러나 이 관계는 ERD에서 직접적으로 표현되지 않고, EXAMINATION 테이블을 통해 간접적으로 연결됩니다.

**[PATIENT - EXAMINATION 관계]**

PATIENT 테이블과 EXAMINATION 테이블은 1:N 관계입니다. 한 환자가 여러 번의 검사를 받을 수 있기 때문입니다. 이 관계는 점선으로 표현되며 각 검사는 환자 ID를 외래 키로 참조하여 환자에게 연결됩니다.

**[DOCTOR - EXAMINATION 관계]**

DOCTOR 테이블과 EXAMINATION 테이블도 1:N 관계입니다. 한 의사가 여러 검사를 진행할 수 있기 때문입니다. 점선은 의사가 검사 기록에 직접 연결되어야 함을 나타냅니다. 각 검사 기록은 의사 ID를 외래 키로 참조합니다.

**[NURSE - PATIENT 관계]**

NURSE 테이블과 PATIENT 테이블은 N:M 관계를 가질 수 있습니다. 한 간호사가 여러 환자를 담당할 수 있고, 한 환자도 여러 간호사에게 진료를 받을 수 있기 때문입니다. 그러나 이 관계는 ERD에서 직접적으로 표현되지 않고, EXAMINATION 테이블을 통해 간접적으로 연결됩니다.

**[NURSE - TREATMENT 관계]**

NURSE 테이블과 TREATMENT 테이블 간의 관계는 1:N입니다. 한 간호사가 여러 치료를 담당할 수 있기 때문입니다. 각 치료 기록은 간호사 ID를 외래 키로 참조하며, 이는 점선으로 표현됩니다.

**[PATIENT - TREATMENT 관계]**

PATIENT 테이블과 TREATMENT 테이블 사이의 관계는 1:N 관계입니다. 이는 한 명의 환자가 여러 번의 치료를 받을 수 있음을 의미합니다.

**[INPATIENT - PATIENT 관계]**

INPATIENT 테이블은 PATIENT 테이블과 1:1 관계를 가지며 실선으로 표현됩니다.

**[MEDICAL\_SPECIALTY - DEPARTMENT 관계]**

MEDICAL\_SPECIALTY 테이블은 DEPARTMENT 테이블과 1:N 관계를 가집니다. 각 전문 분야는 하나의 부서에만 속해야 하며, 부서는 여러 전문 분야를 만들어 낼 수 있습니다. 이 관계는 실선으로 표현되어, 각 전문 분야가 고유한 부서 ID를 가지고 있음을 나타냅니다.

**[USER - DOCTOR 관계]**

USER - DOCTOR 관계는 1:1 관계를 가집니다. 각 doctor 레코드는 하나의 user 레코드와 연결됩니다. 이는 의사가 시스템을 사용하는 데 필요한 인증 정보 (사용자 이름, 비밀번호, 역할)를 user 테이블에서 관리한다는 것을 의미합니다. doctor 테이블의 UserID 필드는 user 테이블의 UserID를 외래 키로 참조합니다.

**[USER - NURSE 관계]**

USER - NURSE 관계는 1:1 관계를 가집니다. 각 nurse 레코드는 하나의 user 레코드와 연결됩니다. 이는 간호사가 시스템 접근을 위한 인증 정보를 user 테이블에서 관리한다는 것을 의미합니다. nurse 테이블의 UserID 필드는 user 테이블의 UserID를 외래 키로 참조합니다.

**[USER - PATIENT 관계]**

USER - PATIENT 관계는 1:1 관계를 가집니다. 각 patient 레코드는 하나의 user 레코드와 연결됩니다. 이는 환자가 시스템에 접근하고 예약, 진료 기록 조회 등을 할 때 필요한 인증 정보를 user 테이블에서 관리한다는 것을 의미합니다. patient 테이블의 UserID 필드는 user 테이블의 UserID를 외래 키로 참조합니다.

**<전체 테이블에 대한 스키마 정규화>**

다치 속성 부재: 모든 테이블은 다치 속성을 갖지 않습니다. 즉, 각 필드는 단일 값을 가집니다.  
기본 키의 유일성: 어떤 테이블도 2개 이상의 기본 키를 갖지 않습니다. 각 테이블은 고유한 기본 키를 가집니다.  
기본 키 외 종속성 부재: 기본 키가 아닌 다른 속성이 기본 키를 제외한 다른 속성을 결정하지 않습니다.

**1NF (첫 번째 정규형):**

적용 테이블: Department, User, Doctor, Patient, Examination, Inpatient, Medical Specialty, Nurse, Reservation, Treatment.

설명: 모든 테이블이 1NF를 충족합니다. 각 테이블은 기본 키를 가지고 있으며, 모든 열 값이 원자적입니다 (반복 그룹이나 배열 없음).

**2NF (두 번째 정규형):**

적용 테이블: Department, User, Doctor, Patient, Examination, Inpatient, Medical Specialty, Nurse, Reservation, Treatment.

설명: 2NF는 모든 비주 키 속성이 기본 키 전체에 함수적으로 종속되는 것을 요구합니다. 즉, 기본 키의 일부분에 대한 부분 종속이 없어야 합니다. 복합 기본 키를 가진 테이블(예: Doctor, Nurse)은 각 필드가 전체 기본 키에 종속되어야 합니다. DoctorID와 NurseID는 각각의 테이블에서 고유하게 식별하며, DepartmentID와 UserID는 기본 키의 일부로서 각각의 테이블에서 부분 종속성이 없습니다. 다른 테이블들도 복합 키가 아니거나, 복합 키를 가진 경우에도 부분 종속성이 없어 2NF를 만족합니다.

**3NF (세 번째 정규형):**

적용 테이블: 모든 테이블.

설명: 3NF에서 이행적 종속이 없습니다.

**BCNF (보이스-코드 정규형):**

적용 테이블: 모든 테이블.

설명: 모든 테이블에서 비주 키 속성은 오직 기본 키, 즉 슈퍼키에만 종속되어 있습니다. 따라서 BCNF를 만족합니다.

**< 인덱스 생성>**

카디널리티가 높은 속성 후보 선정: (examination과 reservation, treatment, patient, inpatient, user)  
examination는 환자 별로 매번 검사 시 매번 데이터가 발생되기 때문에 선정하였습니다. reservation는 환자가 병원에 자주 예약 방문 한다면 데이터가 많이 발생하기 때문에 선정하였습니다. treatment는 환자는 주기적으로 계속 간호사의 치료를 받기 때문에 환자 당 치료 세부사항이 많이 발생하기 때문에 선정하였습니다. patient는 항상 환자는 새로 생겨나기 때문에 선정하였습니다. inpatient는 patient에 따라 똑같이 많을 것이라고 판단했습니다. 그럼 자연스럽게 user도 많아질 것이라 생각하여 user도 후보에 넣었습니다.

조회가 빈번한 속성 후보 선택: (examination과 reservation, treatment)

examination은 의사가 매번 환자 별로 조회해야되기 때문에 선정하였습니다. reservation은 환자 별로 매번 새로 생기는 reservation은 조회하고 관리해야 되기 때문에 선정하였습니다. treatment는 환자 별로 간호사가 매번 조회하고 기록해야 되기때문에 선정하였습니다.

허나 reservation은 수정 및 삭제 빈도가 높기 때문에 최종 탈락시켰습니다. 또한 인덱스는 검색 성능을 향상시키지만, 삽입, 삭제, 수정 작업에 부담을 줄 수 있습니다. 따라서 필요한 곳에만 인덱스를 생성합니다. Doctor와 nurse는 병원에서 근무하는 의사와 간호사가 퇴사를 잘 안하고 신입도 매번 들어오는 것이 아니라 판단했고 거의 고정이라 판단하여 인덱스를 사용하지 않아도 괜찮다고 생각했습니다.

**텍스트, 폰트, 번호, 라인이(가) 표시된 사진

자동 생성된 설명텍스트, 폰트, 번호, 스크린샷이(가) 표시된 사진

자동 생성된 설명**

최종적으로 카디널리티가 높고 조회가 빈번한 테이블 위주로 선택하여 examination과 treatment의 인덱스 테이블을 생성하였습니다. 실행 결과는 실행화면에서 설명하도록 하겠습니다.

**< VIEW >**

hospitalmanagement 데이터베이스의 두 가지 주요 기능인 검사(examination) 및 치료(treatment)에 대한 정보를 요약하고 효율적으로 조회하기 위해 생성된 두 개의 SQL 뷰(View)에 관한 내용을 다룹니다. 목적은 ExaminationSummary와 TreatmentSummary라는 두 뷰의 생성 과정과 구조를 설명하고, 이를 통해 사용자가 각각의 검사와 치료에 대한 종합 정보를 쉽게 조회할 수 있도록 하는 것입니다.

**[ExaminationSummary 뷰]**

목적: 환자의 검사 정보를 종합적으로 조회

구조:

환자 이름(PatientName)

검사 날짜 및 시간(ExaminationDateTime)

검사 세부 사항(ExaminationDetails)

검사를 담당한 부서 이름(DepartmentName)

SQL 쿼리:

CREATE VIEW ExaminationSummary AS

SELECT

p.Name AS PatientName,

e.ExaminationDateTime,

e.ExaminationDetails,

d.DepartmentName

FROM

examination e

JOIN

patient p ON e.PatientID = p.PatientID

JOIN

doctor doc ON e.DoctorID = doc.DoctorID

JOIN

department d ON doc.DepartmentID = d.DepartmentID;

**[TreatmentSummary 뷰]**

목적: 간호사에 의해 수행된 치료 정보를 종합적으로 조회

구조:

간호사 이름(NurseName)

환자 이름(PatientName)

치료 날짜 및 시간(TreatmentDateTime)

치료 세부 사항(TreatmentDetails)

치료가 이루어진 부서 이름(DepartmentName)

SQL 쿼리:

CREATE VIEW TreatmentSummary AS

SELECT

n.Name AS NurseName,

p.Name AS PatientName,

t.TreatmentDateTime,

t.TreatmentDetails,

d.DepartmentName

FROM

treatment t

JOIN

nurse n ON t.NurseID = n.NurseID

JOIN

patient p ON t.PatientID = p.PatientID

JOIN

department d ON n.DepartmentID = d.DepartmentID;

이러한 뷰는 복잡한 조인 쿼리를 간소화하여 사용자가 데이터베이스에서 필요한 정보를 더 쉽고 빠르게 검색할 수 있게 도와줍니다. ExaminationSummary와 TreatmentSummary 뷰는 특히 관리자, 의사, 간호사 등이 환자의 검사 및 치료 정보를 효율적으로 관리하고 조회하는 데 유용하게 사용될 수 있습니다. 실행결과에 대한 내용은 실행 화면 챕터에서 설명하도록 하겠습니다.

**<Transaction 설계>**

병원 시스템에서 환자의 예약 과정을 원활하고 안정적으로 관리하기 위해 트랜잭션 관리의 네 가지 주요 속성(원자성, 일관성, 격리성, 영속성)을 활용하는 것이 목표입니다.

원자성(Atomicity): 예약 과정에서 발생할 수 있는 모든 작업은 완전히 수행되거나 전혀 수행되지 않아야 합니다. 이를 위해 MySQL의 트랜잭션 관리 기능을 사용하여, 예약 과정 중 오류가 발생하면 모든 변경 사항을 롤백합니다.

일관성(Consistency): 예약 과정이 성공적으로 완료되면 데이터베이스는 항상 일관된 상태를 유지해야 합니다. 예약 데이터가 삽입되면 관련 테이블의 무결성이 유지되어야 합니다.

격리성(Isolation): 동시에 여러 환자가 예약을 시도할 때, 한 트랜잭션이 다른 트랜잭션의 작업에 영향을 주지 않아야 합니다. 이를 위해 SELECT ... FOR UPDATE 문을 사용하여 예약 시간과 부서에 대한 기존 예약을 잠그고, 다른 트랜잭션이 해당 데이터에 접근하지 못하도록 합니다.

영속성(Durability): 트랜잭션이 성공적으로 커밋되면, 그 결과는 데이터베이스에 영구적으로 반영되어야 합니다. 시스템이나 데이터베이스에 문제가 발생해도, 예약 정보는 안전하게 유지되어야 합니다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

환자가 예약을 요청하면, 시스템은 MySQL 데이터베이스에 연결을 맺고 트랜잭션을 시작합니다.

const connection = await promisePool.getConnection();

await connection.beginTransaction();

입력된 예약 날짜와 부서에 대해 기존 예약이 있는지 확인합니다.

const existingReservationQuery = `

SELECT \* FROM reservation

WHERE ReservationDateTime = ? AND DepartmentID = ?

`;

const [existingReservations] = await connection.query(

existingReservationQuery,

[ReservationDateTime, DepartmentID]

);

중복 예약이 없을 경우, 새로운 예약을 reservation 테이블에 추가합니다.

const addReservationQuery = `

INSERT INTO reservation (ReservationDateTime, DepartmentID, PatientID)

VALUES (?, ?, ?)

`;

await connection.query(addReservationQuery, [

ReservationDateTime, DepartmentID, PatientID

]);

예약 추가에 성공하면, 트랜잭션을 커밋하여 변경 사항을 확정합니다. 오류 발생 시, 롤백을 수행하여 모든 변경 사항을 취소합니다.

await connection.commit();

await connection.rollback();

예약 성공 시 "Transaction commit success" 콘솔 로그 출력. 오류 발생 시 적절한 오류 메시지 반환. 연결 해제.  
connection.release();

예약 성공: 중복되지 않는 예약 요청 시, 데이터베이스에 예약이 추가되고 "Transaction commit success" 콘솔 로그 출력.

중복 예약: 이미 예약된 시간대에 예약 시도 시 "This time slot is already reserved." 메시지 반환.

오류: 예약 중 오류 발생 시 "An error occurred while making the reservation." 메시지 반환.

텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명

트랜잭션 시작: START TRANSACTION; 명령어를 사용하여 트랜잭션을 시작합니다.

변수 설정: 예약에 필요한 PatientID, DepartmentID, ReservationDateTime 변수를 설정합니다.

기존 예약 확인: SELECT...FOR UPDATE 명령어를 사용하여 해당 시간과 부서에 이미 예약이 있는지 확인하고, 결과를 잠급니다. 이는 동시에 다른 트랜잭션이 같은 데이터를 수정하는 것을 방지합니다.

예약 추가: INSERT INTO...SELECT 명령어를 사용하여 중복 예약이 없을 경우에만 새 예약을 추가합니다.

영향받은 행 수 확인: ROW\_COUNT() 함수를 사용하여 새 예약이 추가되었는지 확인합니다.

결과에 따른 트랜잭션 처리: 영향받은 행이 있으면 트랜잭션을 커밋하고 성공 메시지를 출력합니다. 없으면 롤백하고 중복 예약 오류 메시지를 출력합니다.

**<웹페이지 생성 (.hbs+.js)>**

**[sql.js 설계]**

*import* mysql *from* "mysql2";

*// Database connection*

const pool = mysql.createPool({

  host: "localhost",

  user: "root", *// Change to your MySQL username*

  database: "hospitalmanagement", *// Adjust for your database name*

  password: "minji0715@", *// Change to your MySQL password*

  waitForConnections: true,

  connectionLimit: 10,

  queueLimit: 0,

});

const promisePool = pool.promise();

*export* { promisePool };

*// SELECT queries*

*export* const selectSql = {

  getUserByUsername: async (username) => {

    console.log(username);

    const [rows] = *await* promisePool.query(

      `SELECT \* FROM user WHERE Username = ?`,

      [username]

    );

    console.log(rows);

*return* rows[0];

  },

  getDoctors: async () => {

    const [rows] = *await* promisePool.query(`SELECT \* FROM doctor`);

*return* rows;

  },

  getNurses: async () => {

    const [rows] = *await* promisePool.query(`SELECT \* FROM nurse`);

*return* rows;

  },

  getExaminations: async (userId) => {

    const doctorQuery = `SELECT DoctorID FROM doctor WHERE UserID = ?`;

    const [doctorRows] = *await* promisePool.query(doctorQuery, [userId]);

*if* (doctorRows.length === 0) {

*throw* new Error("No doctor found with the given user ID.");

    }

    const doctorId = doctorRows[0].DoctorID;

    const examinationQuery = `SELECT \* FROM examination WHERE DoctorID = ?`;

    const [examinationRows] = *await* promisePool.query(examinationQuery, [

      doctorId,

    ]);

*return* examinationRows;

  },

*// Fetch TREATMENT info for NURSE*

  getTreatments: async (userId) => {

*// First, find the NurseID associated with the given UserID*

    const nurseQuery = `SELECT NurseID FROM nurse WHERE UserID = ?`;

    const [nurseRows] = *await* promisePool.query(nurseQuery, [userId]);

*if* (nurseRows.length === 0) {

*throw* new Error("No nurse found with the given user ID.");

    }

    const nurseId = nurseRows[0].NurseID;

*// Then, fetch treatments associated with the NurseID*

    const treatmentQuery = `SELECT \* FROM treatment WHERE NurseID = ?`;

    const [treatmentRows] = *await* promisePool.query(treatmentQuery, [nurseId]);

*return* treatmentRows;

  },

*// Fetch PATIENT info based on criteria*

  searchPatients: async (query) => {

*// This is a basic example. Modify the query for more complex search criteria.*

    const searchQuery = `

        SELECT \* FROM patient

        WHERE Name LIKE CONCAT('%', ?, '%') OR PatientID LIKE CONCAT('%', ?, '%')

    `;

    const [rows] = *await* promisePool.query(searchQuery, [query, query]);

*return* rows;

  },

  getReservationsByPatient: async (userId) => {

    const query = `

        SELECT r.ReservationNumber, r.ReservationDateTime, r.DepartmentID, d.DepartmentName

        FROM reservation r

        JOIN department d ON r.DepartmentID = d.DepartmentID

        WHERE r.PatientID = (SELECT PatientID FROM patient WHERE UserID = ?);

    `;

    const [rows] = *await* promisePool.query(query, [userId]);

*return* rows;

  },

*// ... additional SELECT queries*

};

*// INSERT queries*

*export* const insertSql = {

  addStaff: async (

    name,

    departmentId,

    address,

    phoneNumber,

    username,

    password,

    role

  ) => {

*// Insert into user table and get the inserted ID*

    const userQuery = `INSERT INTO user (Username, Password, Role) VALUES (?, ?, ?)`;

    const [userResult] = *await* promisePool.query(userQuery, [

      username,

      password,

      role,

    ]);

    const userId = userResult.insertId;

*// Insert into the appropriate staff table using the new UserId*

    const staffTable = role === "DOCTOR" ? "doctor" : "nurse";

    const staffQuery = `INSERT INTO ${staffTable} (Name, DepartmentID, Address, PhoneNumber, UserID) VALUES (?, ?, ?, ?, ?)`;

*await* promisePool.query(staffQuery, [

      name,

      departmentId,

      address,

      phoneNumber,

      userId,

    ]);

  },

  addExamination: async (

    ExaminationDateTime,

    ExaminationDetails,

    DoctorID,

    PatientID

  ) => {

    const query = `INSERT INTO examination (ExaminationDateTime, ExaminationDetails, DoctorID, PatientID) VALUES (?, ?, ?, ?)`;

*await* promisePool.query(query, [

      ExaminationDateTime,

      ExaminationDetails,

      DoctorID,

      PatientID,

    ]);

  },

  addTreatment: async (

    TreatmentDateTime,

    TreatmentDetails,

    NurseID,

    PatientID

  ) => {

    const query = `INSERT INTO treatment (TreatmentDateTime, TreatmentDetails, NurseID, PatientID) VALUES (?, ?, ?, ?)`;

*await* promisePool.query(query, [

      TreatmentDateTime,

      TreatmentDetails,

      NurseID,

      PatientID,

    ]);

  },

  makeReservation: async (ReservationDateTime, DepartmentID, userId) => {

    const patientQuery = `SELECT PatientID FROM patient WHERE UserID = ?`;

    const [patientRows] = *await* promisePool.query(patientQuery, [userId]);

*if* (patientRows.length === 0) {

*throw* new Error("No patient found with the given user ID.");

    }

    const patientId = patientRows[0].PatientID;

    const reservationQuery = `INSERT INTO reservation (ReservationDateTime, DepartmentID, PatientID) VALUES (?, ?, ?)`;

*await* promisePool.query(reservationQuery, [

      ReservationDateTime,

      DepartmentID,

      patientId,

    ]);

  },

};

*export* const updateSql = {

  updateStaff: async (staffId, updatedDetails, role) => {

    const staffTable = role === "DOCTOR" ? "doctor" : "nurse";

    const staffIdColumn = role === "DOCTOR" ? "DoctorID" : "NurseID";

*try* {

*// Update the staff details*

      const updateStaffQuery = `

        UPDATE ${staffTable}

        SET Name = ?, Address = ?, PhoneNumber = ?

        WHERE ${staffIdColumn} = ?;

      `;

*await* promisePool.query(updateStaffQuery, [

        updatedDetails.Name,

        updatedDetails.Address,

        updatedDetails.PhoneNumber,

        staffId,

      ]);

*// Update the user details if provided*

*if* (updatedDetails.Username || updatedDetails.Password) {

        const getUserQuery = `SELECT UserID FROM ${staffTable} WHERE ${staffIdColumn} = ?`;

        const [userRows] = *await* promisePool.query(getUserQuery, [staffId]);

*if* (userRows.length === 0) {

*throw* new Error("No user found with the given staff ID.");

        }

        const userId = userRows[0].UserID;

        const updateUserQuery = `

          UPDATE user

          SET Username = ?, Password = ?

          WHERE UserID = ?;

        `;

*await* promisePool.query(updateUserQuery, [

          updatedDetails.Username,

          updatedDetails.Password,

          userId,

        ]);

      }

    } *catch* (error) {

*throw* error; *// Rethrow the error after rollback*

    }

  },

*// ... additional SQL functions ...*

  updateExamination: async (

    examinationId,

    ExaminationDateTime,

    ExaminationDetails,

    PatientID

  ) => {

    const query = `UPDATE examination SET ExaminationDateTime = ?, ExaminationDetails = ?, PatientID = ? WHERE ExaminationID = ?`;

*await* promisePool.query(query, [

      ExaminationDateTime,

      ExaminationDetails,

      PatientID,

      examinationId,

    ]);

  },

  updateTreatment: async (

    treatmentId,

    TreatmentDateTime,

    TreatmentDetails,

    PatientID

  ) => {

    const query = `UPDATE treatment SET TreatmentDateTime = ?, TreatmentDetails = ?, PatientID = ? WHERE TreatmentID = ?`;

*await* promisePool.query(query, [

      TreatmentDateTime,

      TreatmentDetails,

      PatientID,

      treatmentId,

    ]);

  },

};

*export* const deleteSql = {

*// Delete DOCTOR or NURSE and the corresponding USER for Admin*

  deleteStaff: async (staffId, role) => {

    const staffTable = role === "DOCTOR" ? "doctor" : "nurse";

    const staffIdColumn = role === "DOCTOR" ? "DoctorID" : "NurseID";

*try* {

*// Get the UserID associated with the staff member*

      const selectQuery = `SELECT UserID FROM ${staffTable} WHERE ${staffIdColumn} = ?`;

      const [userRows] = *await* promisePool.query(selectQuery, [staffId]);

*if* (userRows.length === 0) {

*throw* new Error("No user found with the given staff ID.");

      }

      const userId = userRows[0].UserID;

      console.log(userId);

*// Delete the staff member*

      const deleteStaffQuery = `DELETE FROM ${staffTable} WHERE ${staffIdColumn} = ?`;

*await* promisePool.query(deleteStaffQuery, [staffId]);

*// Delete the associated user*

      const deleteUserQuery = "DELETE FROM user WHERE UserID = ?";

*await* promisePool.query(deleteUserQuery, [userId]);

    } *catch* (error) {

*throw* error; *// Rethrow the error after rollback*

    }

  },

*// ... additional DELETE queries*

  deleteExamination: async (examinationId) => {

    const query = `DELETE FROM examination WHERE ExaminationID = ?`;

*await* promisePool.query(query, [examinationId]);

  },

  deleteTreatment: async (treatmentId) => {

    const query = `DELETE FROM treatment WHERE TreatmentID = ?`;

*await* promisePool.query(query, [treatmentId]);

  },

  cancelReservation: async (reservationId) => {

    const query = `DELETE FROM reservation WHERE ReservationNumber = ?`;

*await* promisePool.query(query, [reservationId]);

  },

};

*// Export other necessary SQL functionalities*

원 관리 시스템을 위한 데이터베이스 설계의 주요 구성 요소와 SQL 작업 처리를 위한 JavaScript 코드 구현에 대해 설명합니다. 이 시스템은 Node.js 환경에서 MySQL 데이터베이스를 사용하여 구축되며, mysql2 라이브러리를 활용하여 비동기 방식으로 데이터베이스 작업을 수행합니다.

Node.js 환경에서 mysql2/promise 모듈을 사용하여 SQL 쿼리를 실행하며, 이를 통해 데이터베이스의 CRUD(Create, Read, Update, Delete) 작업을 관리합니다.

데이터베이스 연결 설정

데이터베이스와의 연결은 createPool 메소드를 통해 구성되며, 연결 풀을 생성하여 데이터베이스 요청을 관리합니다. 연결 풀은 동시에 여러 데이터베이스 연결을 유지하고, 필요에 따라 사용 가능한 연결을 재사용합니다. 이 방법은 서버에 대한 요청 처리를 최적화하고, 연결 생성과 해제에 소모되는 오버헤드를 줄입니다.

비동기 SQL 쿼리 실행

비동기적 SQL 쿼리 실행은 promisePool을 통해 수행됩니다. 이는 Node.js의 비동기 처리 패턴인 Promise를 사용하여, 쿼리 결과가 준비되면 해당 결과를 처리할 수 있도록 합니다. 따라서 서버가 다른 작업을 계속 처리할 수 있으며, 블로킹 없이 효율적으로 데이터베이스 작업을 수행할 수 있습니다.

SELECT 쿼리 처리

사용자 이름을 통한 사용자 정보 조회, 의사 및 간호사 정보 조회, 사용자 ID에 따른 검사 기록 조회 등을 비롯한 여러 조회 작업은 selectSql 객체에 메소드로 정의되어 있습니다. 이 메소드들은 필요한 파라미터를 받아 해당하는 SQL 쿼리를 실행하고, 결과를 반환합니다. 예를 들어, getUserByUsername 메소드는 주어진 사용자 이름으로 사용자 테이블을 조회하고, 해당 결과를 반환합니다.

INSERT 쿼리 처리

새로운 직원 추가, 검사 기록 추가, 치료 기록 추가 등의 작업은 insertSql 객체에 정의되어 있습니다. 이러한 작업은 새로운 데이터를 데이터베이스에 삽입하고, 필요한 경우 삽입된 레코드의 ID 값을 반환합니다. 예를 들어, addStaff 메소드는 새로운 직원 정보를 user 테이블과 doctor 또는 nurse 테이블에 추가합니다.

UPDATE 쿼리 처리

직원 정보 업데이트, 검사 기록 업데이트 등의 업데이트 작업은 updateSql 객체에 정의되어 있습니다. 이 메소드들은 주어진 파라미터를 사용하여 특정 레코드의 데이터를 업데이트합니다.

DELETE 쿼리 처리

직원 및 사용자 정보 삭제, 검사 기록 삭제, 예약 취소 등의 삭제 작업은 deleteSql 객체에 정의되어 있습니다. 이 메소드들은 주어진 ID 값을 사용하여 특정 레코드를 데이터베이스에서 제거합니다.

getUserByUsername(username)

목적: 주어진 사용자 이름으로 user 테이블에서 사용자를 검색합니다.

코드 설명: promisePool.query 함수를 사용하여 SQL 쿼리를 실행합니다. 쿼리문은 SELECT \* FROM user WHERE Username = ?로, ?에는 함수 인자로 받은 username이 바인딩됩니다. 조회 결과는 rows 배열에 저장되며, 첫 번째 원소 (rows[0])를 반환합니다.

getDoctors()

목적: 모든 의사의 정보를 doctor 테이블에서 조회합니다.

코드 설명: SELECT \* FROM doctor 쿼리를 실행하여 모든 의사의 데이터를 조회합니다. 조회된 결과는 rows 배열에 저장되며, 이 배열 전체를 반환합니다.

getNurses()

목적: 모든 간호사의 정보를 nurse 테이블에서 조회합니다.

코드 설명: SELECT \* FROM nurse 쿼리를 실행하여 모든 간호사의 데이터를 조회합니다. 조회된 결과는 rows 배열에 저장되며, 이 배열 전체를 반환합니다.

getExaminations(userId)

목적: 특정 의사의 모든 검사 기록을 조회합니다.

코드 설명: 먼저, SELECT DoctorID FROM doctor WHERE UserID = ? 쿼리를 실행하여 해당 의사의 DoctorID를 찾습니다. 이 DoctorID를 사용하여 SELECT \* FROM examination WHERE DoctorID = ? 쿼리를 실행, 해당 의사의 모든 검사 기록을 조회합니다. 조회된 검사 기록들은 examinationRows 배열에 저장되며, 이 배열 전체를 반환합니다.

getTreatments(userId)

목적: 특정 간호사의 모든 치료 기록을 조회합니다.

코드 설명: SELECT NurseID FROM nurse WHERE UserID = ? 쿼리를 실행하여 해당 간호사의 NurseID를 찾습니다. 이 NurseID를 사용하여 SELECT \* FROM treatment WHERE NurseID = ? 쿼리를 실행, 해당 간호사의 모든 치료 기록을 조회합니다. 조회된 치료 기록들은 treatmentRows 배열에 저장되며, 이 배열 전체를 반환합니다.

searchPatients(query)

목적: 환자 이름이나 ID를 기준으로 환자 정보를 검색합니다.

코드 설명: SELECT \* FROM patient WHERE Name LIKE CONCAT('%', ?, '%') OR PatientID LIKE CONCAT('%', ?, '%') 쿼리를 실행합니다. CONCAT 함수를 사용하여 %검색어% 형식으로 이름과 ID를 모두 검색 조건에 포함시킵니다. 조회된 환자 정보들은 rows 배열에 저장되며, 이 배열 전체를 반환합니다.

getReservationsByPatient(userId)

목적: 특정 환자의 모든 예약 정보를 조회합니다.

코드 설명: SELECT ... FROM reservation r JOIN department d ON r.DepartmentID = d.DepartmentID WHERE r.PatientID = (SELECT PatientID FROM patient WHERE UserID = ?) 쿼리를 실행합니다. 이 쿼리는 예약 테이블과 부서 테이블을 조인하고, 세션에서 받은 환자의 UserID에 해당하는 PatientID를 찾아 해당 환자의 모든 예약 정보를 조회합니다. 조회된 예약 정보들은 rows 배열에 저장되며, 이 배열 전체를 반환합니다.

addStaff(name, departmentId, address, phoneNumber, username, password, role)

목적: 새로운 직원(의사 또는 간호사)을 시스템에 추가합니다.

코드 설명: 먼저 user 테이블에 사용자의 아이디, 비밀번호, 역할을 추가합니다. INSERT INTO user (Username, Password, Role) VALUES (?, ?, ?) 쿼리를 사용하여 데이터를 삽입하고, 삽입된 레코드의 ID (insertId)를 가져옵니다. 이 userId를 사용하여 doctor 또는 nurse 테이블에 직원 정보를 추가합니다. 쿼리는 직원의 이름, 부서 ID, 주소, 전화번호 등을 포함합니다.

addExamination(ExaminationDateTime, ExaminationDetails, DoctorID, PatientID)

목적: 새로운 검사 기록을 추가합니다.

코드 설명: INSERT INTO examination (ExaminationDateTime, ExaminationDetails, DoctorID, PatientID) VALUES (?, ?, ?, ?) 쿼리를 사용하여 검사 날짜/시간, 세부사항, 의사 ID, 환자 ID를 포함하는 새로운 검사 기록을 추가합니다.

addTreatment(TreatmentDateTime, TreatmentDetails, NurseID, PatientID)

목적: 새로운 치료 기록을 추가합니다.

코드 설명: INSERT INTO treatment (TreatmentDateTime, TreatmentDetails, NurseID, PatientID) VALUES (?, ?, ?, ?) 쿼리를 사용하여 치료 날짜/시간, 세부사항, 간호사 ID, 환자 ID를 포함하는 새로운 치료 기록을 추가합니다.

makeReservation(ReservationDateTime, DepartmentID, userId)

목적: 환자에 의한 새로운 예약을 추가합니다.

코드 설명: 먼저 SELECT PatientID FROM patient WHERE UserID = ? 쿼리를 사용하여 주어진 사용자 ID에 해당하는 환자 ID를 찾습니다. INSERT INTO reservation (ReservationDateTime, DepartmentID, PatientID) VALUES (?, ?, ?) 쿼리를 사용하여 예약 날짜/시간, 부서 ID, 환자 ID를 포함하는 새로운 예약을 추가합니다.

updateStaff(staffId, updatedDetails, role)

목적: 직원의 정보를 업데이트합니다.

코드 설명: 직원의 역할에 따라 doctor 또는 nurse 테이블에서 해당 직원의 정보를 업데이트합니다. UPDATE ${staffTable} SET Name = ?, Address = ?, PhoneNumber = ? WHERE ${staffIdColumn} = ?; 쿼리를 사용하여 직원의 이름, 주소, 전화번호 등을 업데이트합니다. 사용자 이름이나 비밀번호가 제공되면, user 테이블에서도 해당 정보를 업데이트합니다.

updateExamination(examinationId, ExaminationDateTime, ExaminationDetails, PatientID)

목적: 검사 기록을 업데이트합니다.

코드 설명: UPDATE examination SET ExaminationDateTime = ?, ExaminationDetails = ?, PatientID = ? WHERE ExaminationID = ? 쿼리를 사용하여 주어진 ID의 검사 기록을 업데이트합니다.

updateTreatment(treatmentId, TreatmentDateTime, TreatmentDetails, PatientID)

목적: 치료 기록을 업데이트합니다.

코드 설명: UPDATE treatment SET TreatmentDateTime = ?, TreatmentDetails = ?, PatientID = ? WHERE TreatmentID = ? 쿼리를 사용하여 주어진 ID의 치료 기록을 업데이트합니다.

deleteStaff(staffId, role)

목적: 직원과 관련된 사용자 정보를 삭제합니다.

코드 설명: 먼저 직원의 역할에 따라 doctor 또는 nurse 테이블에서 해당 직원을 삭제합니다. 그 후 user 테이블에서도 해당 직원과 연결된 사용자 정보를 삭제합니다.

deleteExamination(examinationId)

목적: 검사 기록을 삭제합니다.

코드 설명: DELETE FROM examination WHERE ExaminationID = ? 쿼리를 사용하여 주어진 ID의 검사 기록을 삭제합니다.

deleteTreatment(treatmentId)

목적: 치료 기록을 삭제합니다.

코드 설명: DELETE FROM treatment WHERE TreatmentID = ? 쿼리를 사용하여 주어진 ID의 치료 기록을 삭제합니다.

cancelReservation(reservationId)

목적: 예약을 취소합니다.

코드 설명: DELETE FROM reservation WHERE ReservationNumber = ? 쿼리를 사용하여 주어진 예약 번호의 예약을 삭제합니다.

**[index.js]**

*import* express *from* "express";

*import* logger *from* "morgan";

*import* path *from* "path";

*import* expressSession *from* "express-session";

*import* loginRouter *from* "../routes/login";

*import* adminRouter *from* "../routes/admin"; *// For admin operations*

*import* doctorRouter *from* "../routes/doctor"; *// For staff (Doctor and Nurse) operations*

*import* nurseRouter *from* "../routes/nurse";

*import* patientRouter *from* "../routes/patient"; *// For patient operations*

const PORT = 3000;

const app = express();

app.use(express.static(path.join(\_\_dirname, "/public")));

app.use(express.urlencoded({ extended: false }));

app.use(express.json());

app.use(

  expressSession({

    secret: "hospital\_management\_secret",

    resave: true,

    saveUninitialized: true,

  })

);

app.set("views", path.join(\_\_dirname, "../views"));

app.set("view engine", "hbs");

app.use(logger("dev"));

*// Setting up routes for different user roles*

app.use("/", loginRouter);

app.use("/admin", adminRouter); *// Admin route for managing doctors and nurses*

app.use("/doctor", doctorRouter); *// Staff route for managing examinations and treatments*

app.use("/nurse", nurseRouter); *// Staff route for managing examinations and treatments*

app.use("/patient", patientRouter); *// Patient route for managing reservations*

app.listen(PORT, () => {

  console.log(`Server is running at http://localhost:${PORT}`);

});

병원 관리 시스템의 웹 서버 구성을 위한 주요 파일인 index.js에 대한 개발 내용을 기술합니다. 이 파일은 Node.js의 Express 프레임워크를 활용하여 서버를 설정하고, 라우팅, 세션 관리, 정적 파일 제공 등의 기능을 구현합니다.

웹 서버 설정

병원 관리 시스템의 웹 서버는 Express 미들웨어를 사용하여 구성됩니다. 먼저, express.static 미들웨어를 통해 정적 파일(예: CSS, JavaScript, 이미지 파일 등)을 제공합니다. 이러한 파일들은 서버의 /public 디렉토리에 저장되며, 클라이언트 요청에 따라 자동으로 제공됩니다.

데이터를 처리하기 위해, express.urlencoded와 express.json 미들웨어를 사용하여, 클라이언트로부터 전송된 폼 데이터와 JSON 데이터를 서버에서 쉽게 접근하고 파싱할 수 있도록 합니다.

세션 관리

express-session 미들웨어는 사용자의 세션을 관리하는데 사용됩니다. 이를 통해 사용자의 로그인 상태를 유지하고, 사용자가 서버에 요청을 할 때마다 사용자를 식별할 수 있습니다. 세션은 'hospital\_management\_secret'라는 비밀 키를 사용하여 암호화되어, 보안을 강화합니다.

템플릿 엔진 설정

서버는 Handlebars 템플릿 엔진을 사용하여 HTML 뷰를 렌더링합니다. views 디렉토리에 저장된 템플릿 파일들을 기반으로, 동적 웹 페이지를 생성하여 클라이언트에 제공합니다.

로깅

morgan 라이브러리는 로깅을 위해 사용됩니다. 개발 중에 서버에 대한 요청과 응답을 콘솔에 기록하여, 디버깅과 모니터링에 유용한 정보를 제공합니다.

라우팅 설정

서버는 다양한 사용자 역할에 따라 분리된 라우팅 경로를 제공합니다:

/: 로그인 페이지와 관련된 라우팅을 처리합니다.

/admin: 관리자 기능을 위한 라우팅으로, 의사와 간호사의 계정 관리 등의 작업을 처리합니다.

/doctor: 의료진(의사 및 간호사) 기능을 위한 라우팅으로, 검사와 치료 관리 등의 작업을 처리합니다.

/nurse: 간호사 기능을 위한 라우팅으로, 주로 치료 관리와 관련된 작업을 처리합니다.

/patient: 환자 기능을 위한 라우팅으로, 예약 관리 등의 작업을 처리합니다.

서버 실행

서버는 3000번 포트에서 실행됩니다. 서버가 시작되면, http://localhost:3000 주소를 통해 접근할 수 있으며, 콘솔에 서버 실행 메시지가 출력됩니다.

**[login.hbs+login.js 설계]**

=============================Login.js==================================

*import* express *from* "express";

*import* { selectSql } *from* "../Database/sql";

const router = express.Router();

*// Display login page*

router.get("/", (req, res) => {

  res.render("login");

});

*// Handle login post request*

router.post("/", async (req, res) => {

  const userdata = req.body;

  console.log(userdata.username);

*try* {

    const user = *await* selectSql.getUserByUsername(userdata.username);

    console.log("durl", user);

*if* (

      user.Username === userdata.username &&

      user.Password === userdata.password

    ) {

      console.log("로그인 성공");

      req.session.user = { UserID: user.UserID, Role: user.Role };

*switch* (user.Role) {

*case* "ADMIN":

          console.log("관리자 로그인 성공");

          res.redirect("/admin");

*break*;

*case* "DOCTOR":

          res.redirect("/doctor");

*break*;

*case* "NURSE":

          res.redirect("/nurse");

*break*;

*case* "PATIENT":

          res.redirect("/patient");

*break*;

*default*:

          res.redirect("/");

      }

    } *else* {

      res.send(

        `<script>alert('Login failed! Incorrect credentials.'); location.href='/';</script>`

      );

    }

  } *catch* (error) {

    console.error("Login error: ", error);

    res.status(500).send("Internal Server Error");

  }

});

*export* *default* router;

==============================Login.hbs================================

<!DOCTYPE html>

<head>

    <meta *charset=*"UTF-8">

    <meta *name=*"viewport" *content=*"width=device-width, initial-scale=1.0">

    <title>Login</title>

    <style>

        body {

            height: 75vh;

            display: flex;

            justify-content: center;

            align-items: center;

            flex-direction: column;

            background-color: #f5f5f5;

        }

*.frame* {

            border-radius: 15px;

            border: 1px solid #757575;

            padding: 20px;

            background-color: white;

            box-shadow: 0 4px 8px rgba(0, 0, 0, 0.1);

            width: 350px;

        }

        form {

            display: flex;

            flex-direction: column;

        }

        input, *.btn* {

            font-size: 16px;

            padding: 10px;

            margin: 10px 0;

        }

*.btn* {

            background-color: #4CAF50;

            color: white;

            border: none;

            border-radius: 4px;

            cursor: pointer;

        }

*.btn:hover* {

            background-color: #45a049;

        }

    </style>

</head>

<body>

    <div *class=*"frame">

        <h1>Login</h1>

        <form *method=*"post" *action=*'/'>

            <input *name=*"username" *type=*"text" required *placeholder=*"Username">

            <input *name=*'password' *type=*"password" required *placeholder=*"Password">

            <button *class=*'btn' *type=*"submit">Login</button>

        </form>

    </div>

</body>

</html>

병원 관리 시스템의 사용자 인증 과정에 대해 설명합니다. 이 과정은 웹 기반 인터페이스를 통해 제공되며, login.hbs 템플릿과 login.js 라우터 파일을 통해 구현됩니다.

login.hbs 파일 분석

페이지 구조: login.hbs는 HTML5 문서 구조를 따릅니다. <head> 섹션에는 문자셋, 뷰포트 설정, 페이지 제목 등이 정의됩니다.

CSS 스타일링: 로그인 폼의 스타일링은 사용자 인터페이스를 시각적으로 꾸미는 데 중점을 둡니다. 로그인 폼은 센터 정렬되며, 입력 필드와 버튼에 대해 특정 스타일이 적용됩니다.

폼 구성: 로그인 폼은 method="post"와 action='/' 속성을 가지며, 사용자로부터 아이디와 비밀번호를 입력받습니다. 이후, 정보는 서버로 전송됩니다.

login.js 파일 분석

라우터 설정: login.js 파일은 Express 라우터를 활용해 로그인 관련 HTTP 요청을 처리합니다.

로그인 페이지 렌더링: router.get() 메소드는 사용자가 처음 로그인 페이지를 방문할 때 호출되어 login 템플릿을 렌더링합니다.

로그인 데이터 처리: router.post() 메소드는 사용자가 로그인 폼을 제출할 때 실행됩니다. 이 과정은 다음과 같은 단계를 포함합니다:

req.body를 통해 사용자 입력 데이터를 받습니다.

selectSql.getUserByUsername 함수를 사용하여 데이터베이스에서 사용자 정보를 조회합니다.

사용자 데이터의 유효성을 검증합니다 (입력된 아이디와 비밀번호가 데이터베이스의 정보와 일치하는지 확인).

로그인 성공 시, req.session.user에 사용자 ID와 역할을 저장하고, 사용자 역할에 따라 적절한 페이지로 리디렉션합니다.

로그인 실패 시, 경고 메시지와 함께 로그인 페이지로 리디렉션합니다.

인증 과정

사용자가 로그인 폼에 정보를 입력하고 제출하면, login.js의 POST 핸들러가 실행됩니다. 사용자 데이터는 req.body를 통해 수신되며, 데이터베이스에서 해당 사용자의 정보를 조회합니다. 조회된 정보와 사용자가 입력한 정보가 일치하면 세션에 사용자 ID와 역할을 저장하고, 해당 사용자 역할에 맞는 라우트로 리디렉션합니다.

사용자 역할에 따른 리디렉션

인증 후 사용자는 그들의 역할(관리자, 의사, 간호사, 환자)에 따라 적절한 페이지로 리디렉션됩니다. 예를 들어, 관리자는 /admin으로, 의사는 /doctor로 리디렉션됩니다. 이러한 리디렉션은 사용자가 적절한 권한을 가진 페이지에 접근하도록 보장합니다.

에러 처리

로그인 과정에서 사용자 이름과 비밀번호가 일치하지 않거나 기타 오류가 발생하는 경우, 사용자에게 알림을 보내고 로그인 페이지로 다시 리디렉션합니다. 이는 JavaScript의 alert 함수를 통해 구현되며, 사용자에게 명확한 피드백을 제공합니다.

**[admin.js+admin.hbs 설계]**

==============================admin.js=================================

*import* express *from* "express";

*import* { selectSql, insertSql, updateSql, deleteSql } *from* "../Database/sql";

const router = express.Router();

*// Display admin dashboard with list of doctors and nurses*

router.get("/", async (req, res) => {

*if* (req.session.user && req.session.user.Role === "ADMIN") {

*try* {

      const doctors = *await* selectSql.getDoctors();

      const nurses = *await* selectSql.getNurses();

      res.render("admin", { doctors, nurses }); *// Assuming a Handlebars view named 'adminDashboard'*

    } *catch* (err) {

      console.error(err);

      res.status(500).send("Error retrieving staff data");

    }

  } *else* {

    res.redirect("/");

  }

});

*// Add a new doctor or nurse*

router.post("/add", async (req, res) => {

  const { Name, DepartmentId, Address, PhoneNumber, Username, Password, role } =

    req.body;

  console.log(

    Name,

    DepartmentId,

    Address,

    PhoneNumber,

    Username,

    Password,

    role

  );

*try* {

*await* insertSql.addStaff(

      Name,

      DepartmentId,

      Address,

      PhoneNumber,

      Username,

      Password,

      role

    );

    res.redirect("/admin");

  } *catch* (err) {

    console.error(err);

    res.status(500).send("Error adding staff");

  }

});

*// Update doctor or nurse details*

router.post("/update/:id", async (req, res) => {

  const staffId = req.params.id;

  const { role, Name, Address, PhoneNumber } = req.body;

  const updatedDetails = { Name, Address, PhoneNumber };

  console.log(staffId, role, Name, Address, PhoneNumber);

*try* {

*await* updateSql.updateStaff(staffId, updatedDetails, role);

    res.redirect("/admin");

  } *catch* (err) {

    console.error(err);

    res.status(500).send("Error updating staff details");

  }

});

*// Delete a doctor or nurse*

router.get("/delete/:id", async (req, res) => {

  const staffId = req.params.id;

  const { role } = req.query;

  console.log(staffId, role);

*try* {

*await* deleteSql.deleteStaff(staffId, role);

    res.redirect("/admin");

  } *catch* (err) {

    console.error(err);

    res.status(500).send("Error deleting staff");

  }

});

*export* *default* router;

==============================admin.hbs ===============================

<html *lang=*"en">

  <head>

    <meta *charset=*"UTF-8" />

    <meta *name=*"viewport" *content=*"width=device-width, initial-scale=1.0" />

    <title>Admin Dashboard</title>

*<!-- Add your CSS styling here -->*

  </head>

  <body>

    <h1>Admin Dashboard</h1>

    <h2>Manage Staff</h2>

*<!-- Add Staff Member -->*

    <h3>Add Staff Member</h3>

    <form *action=*"/admin/add" *method=*"post">

*<!-- Form fields for adding staff -->*

      <input *type=*"text" *name=*"Name" *placeholder=*"Name" required />

      <input

*type=*"number"

*name=*"DepartmentId"

*placeholder=*"DepartmentId"

        required

      />

      <input *type=*"text" *name=*"Address" *placeholder=*"Address" required />

      <input

*type=*"text"

*name=*"PhoneNumber"

*placeholder=*"PhoneNumber"

        required

      />

      <input *type=*"text" *name=*"Username" *placeholder=*"USERID" required />

      <input *type=*"text" *name=*"Password" *placeholder=*"Password" required />

      <select *name=*"role">

        <option *value=*"DOCTOR">DOCTOR</option>

        <option *value=*"NURSE">NURSE</option>

      </select>

*<!-- Additional fields as needed -->*

      <button *type=*"submit">Add Staff</button>

    </form>

*<!-- List of Doctors -->*

    <h3>Doctors DELETE</h3>

    {{*#each* doctors}}

      <p>{{this.Name}}

        -

        {{this.DepartmentID}}

        <a *href=*"/admin/delete/{{this.DoctorID}}?role=DOCTOR">Delete</a>

      </p>

    {{*/each*}}

    <h3>Doctors EDIT</h3>

    {{*#each* doctors}}

      <form *action=*"/admin/update/{{this.DoctorID}}?role=DOCTOR" *method=*"post">

        <input *type=*"hidden" *name=*"role" *value=*"DOCTOR" />

        <input *type=*"hidden" *name=*"DoctorID" *value=*"{{this.DoctorID}}" />

        <input *type=*"text" *name=*"Name" *value=*"{{this.Name}}" />

        <input *type=*"text" *name=*"Address" *value=*"{{this.Address}}" />

        <input *type=*"text" *name=*"PhoneNumber" *value=*"{{this.PhoneNumber}}" />

        <button *type=*"submit">Edit</button>

      </form>

    {{*/each*}}

*<!-- List of Nurses -->*

    <h3>Nurses DELETE</h3>

    {{*#each* nurses}}

      <p>{{this.Name}}

        -

        {{this.DepartmentID}}

        <a *href=*"/admin/delete/{{this.NurseID}}?role=NURSE">Delete</a>

      </p>

    {{*/each*}}

    <h3>Nurses EDIT</h3>

    {{*#each* nurses}}

      <form *action=*"/admin/update/{{this.NurseID}}?role=NURSE" *method=*"post">

        <input *type=*"hidden" *name=*"role" *value=*"NURSE" />

        <input *type=*"hidden" *name=*"NurseID" *value=*"{{this.NurseID}}" />

        <input *type=*"text" *name=*"Name" *value=*"{{this.Name}}" />

        <input *type=*"text" *name=*"Address" *value=*"{{this.Address}}" />

        <input *type=*"text" *name=*"PhoneNumber" *value=*"{{this.PhoneNumber}}" />

        <button *type=*"submit">Edit</button>

      </form>

    {{*/each*}}

*<!-- Include your JavaScript here -->*

  </body>

</html>

병원 관리 시스템 내에서 관리자가 의사 및 간호사를 관리하는 admin.js와 admin.hbs 파일의 설계 및 구현에 대해 설명하겠습니다. 이 기능은 관리자에게 병원 직원을 추가, 수정, 삭제하는 권한을 제공합니다.

admin.hbs 파일 분석

페이지 구조: admin.hbs는 관리자 대시보드의 HTML 구조를 정의합니다. 이 페이지는 관리자가 의사 및 간호사의 목록을 보고, 새로운 직원을 추가하거나 기존 직원의 정보를 수정, 삭제할 수 있는 인터페이스를 제공합니다.

직원 추가 폼: 사용자가 새로운 의사나 간호사를 추가할 수 있는 폼이 있습니다. 이 폼은 이름, 부서 ID, 주소, 전화번호, 사용자 이름, 비밀번호, 역할을 입력받아 /admin/add 경로로 POST 요청을 보냅니다.

직원 목록 표시: 의사와 간호사의 목록이 각각 표시됩니다. {{#each}} 블록은 Handlebars를 사용하여 서버로부터 받은 의사와 간호사 데이터를 반복적으로 렌더링합니다.

직원 삭제 및 수정 링크: 각 직원 옆에는 해당 직원을 삭제하는 링크가 제공됩니다. 또한, 직원 정보를 수정할 수 있는 폼이 제공되어, 변경된 데이터를 서버로 전송합니다.

admin.js 파일 분석

라우터 설정: admin.js는 Express 라우터를 사용하여 관리자 페이지 관련 HTTP 요청을 처리합니다.

대시보드 렌더링: GET 요청에 대해, 라우터는 관리자 대시보드 페이지를 렌더링합니다. 이때, 의사와 간호사의 목록을 데이터베이스에서 조회하여 페이지에 전달합니다.

직원 추가: POST /add 라우트는 새로운 직원을 추가하는 요청을 처리합니다. 폼 데이터를 받아 데이터베이스에 새로운 직원을 추가하는 insertSql.addStaff 함수를 호출합니다.

직원 정보 업데이트: POST /update/:id 라우트는 직원 정보를 업데이트하는 요청을 처리합니다. 수정된 직원 정보와 직원의 ID를 받아 updateSql.updateStaff 함수를 호출하여 데이터베이스에서 해당 직원의 정보를 업데이트합니다.

직원 삭제: GET /delete/:id 라우트는 직원을 삭제하는 요청을 처리합니다. 직원의 ID를 받아 deleteSql.deleteStaff 함수를 호출하여 데이터베이스에서 해당 직원을 삭제합니다.

전체적인 관리자 페이지 동작 과정

대시보드 페이지 렌더링

접근 권한 확인: admin.js의 GET / 라우트는 사용자의 세션을 확인하여 현재 사용자가 관리자 역할을 가지고 있는지 검사합니다. 관리자가 아닌 경우, 로그인 페이지로 리디렉션됩니다.

데이터 조회 및 페이지 렌더링: 관리자로 확인되면, selectSql.getDoctors()와 selectSql.getNurses() 함수를 사용하여 의사와 간호사의 데이터를 데이터베이스에서 조회합니다. 조회된 데이터는 admin 뷰로 전달되어, 사용자에게 대시보드 페이지가 렌더링됩니다.

직원 추가

폼 제출: admin.hbs에서 제공하는 '직원 추가' 폼을 통해 사용자는 새로운 의사나 간호사의 정보(이름, 부서 ID, 주소 등)를 입력하고 제출할 수 있습니다.

데이터 처리 및 저장: admin.js의 POST /add 라우트는 폼에서 제출된 데이터를 받아 insertSql.addStaff 함수를 호출합니다. 이 함수는 새로운 직원의 데이터를 데이터베이스에 추가합니다.

결과 반영: 직원 추가가 성공적으로 이루어지면, 사용자는 자동으로 대시보드 페이지로 리디렉션됩니다.

직원 정보 업데이트

업데이트 폼 제출: 각 직원 옆에 제공된 '편집' 폼을 통해 사용자는 직원의 정보를 수정하고 이를 제출할 수 있습니다.

데이터 업데이트: admin.js의 POST /update/:id 라우트는 수정된 정보를 받아 updateSql.updateStaff 함수를 호출합니다. 이 함수는 주어진 ID를 가진 직원의 정보를 데이터베이스에서 업데이트합니다.

결과 반영: 정보 업데이트가 성공적으로 이루어지면, 사용자는 대시보드 페이지로 리디렉션됩니다.

직원 삭제

삭제 링크 클릭: 각 직원 옆에 제공된 '삭제' 링크를 클릭하면, 사용자는 해당 직원을 삭제할 수 있습니다.

직원 삭제 처리: admin.js의 GET /delete/:id 라우트는 직원의 ID와 역할을 받아 deleteSql.deleteStaff 함수를 호출합니다. 이 함수는 해당 직원을 데이터베이스에서 삭제합니다.

결과 반영: 직원 삭제가 성공적으로 이루어지면, 사용자는 대시보드 페이지로 리디렉션됩니다.

**[doctor.js+doctor.hbs 설계]**

==============================doctor.js=================================

*import* express *from* "express";

*import* { selectSql, insertSql, updateSql, deleteSql } *from* "../Database/sql";

const router = express.Router();

*// Display doctor dashboard*

router.get("/", async (req, res) => {

*if* (req.session.user && req.session.user.Role === "DOCTOR") {

*try* {

      const userId = req.session.user.UserID; *// UserID from the session*

      let examinations = [];

      let searchResults = [];

      const searchQuery = req.query.searchQuery; *// Retrieve search query from request*

*if* (searchQuery) {

*// If there's a search query, perform the search*

        searchResults = *await* selectSql.searchPatients(searchQuery);

      } *else* {

*// Otherwise, retrieve the doctor's examinations*

        examinations = *await* selectSql.getExaminations(userId);

      }

      res.render("doctor", { examinations, searchResults, searchQuery }); *// Render with fetched data and search results*

    } *catch* (err) {

      console.error(err);

      res.status(500).send("Error retrieving data");

    }

  } *else* {

    res.redirect("/login");

  }

});

*// Add a new examination*

router.post("/add", async (req, res) => {

  const { ExaminationDateTime, ExaminationDetails, DoctorID, PatientID } =

    req.body;

*try* {

*await* insertSql.addExamination(

      ExaminationDateTime,

      ExaminationDetails,

      DoctorID,

      PatientID

    );

    res.redirect("/doctor");

  } *catch* (err) {

    console.error(err);

    res.status(500).send("Error adding examination");

  }

});

*// Update examination details*

router.post("/update/:id", async (req, res) => {

  const examinationId = req.params.id;

  const { ExaminationDateTime, ExaminationDetails, PatientID } = req.body;

*try* {

*await* updateSql.updateExamination(

      examinationId,

      ExaminationDateTime,

      ExaminationDetails,

      PatientID

    );

    res.redirect("/doctor");

  } *catch* (err) {

    console.error(err);

    res.status(500).send("Error updating examination");

  }

});

*// Delete an examination*

router.get("/delete/:id", async (req, res) => {

  const examinationId = req.params.id;

*try* {

*await* deleteSql.deleteExamination(examinationId);

    res.redirect("/doctor");

  } *catch* (err) {

    console.error(err);

    res.status(500).send("Error deleting examination");

  }

});

*export* *default* router;

==============================doctor.hbs================================

<html *lang=*"en">

  <head>

    <meta *charset=*"UTF-8" />

    <meta *name=*"viewport" *content=*"width=device-width, initial-scale=1.0" />

    <title>Doctor Dashboard</title>

*<!-- Add your CSS styling here -->*

  </head>

  <body>

    <h1>Doctor Dashboard</h1>

*<!-- Add Examination -->*

    <h3>Add EXAMINATION</h3>

    <form *action=*"/doctor/add" *method=*"post">

*<!-- Form fields for adding examination -->*

      <input *type=*"datetime-local" *name=*"ExaminationDateTime" required />

      <textarea

*name=*"ExaminationDetails"

*placeholder=*"Examination Details"

        required

      ></textarea>

      <input

*type=*"number"

*name=*"DoctorID"

*placeholder=*"Doctor ID"

*value=*"{{doctorId}}"

      />

*<!-- Assuming doctorId is passed from the server -->*

      <input *type=*"number" *name=*"PatientID" *placeholder=*"Patient ID" required />

      <button *type=*"submit">Add EXAMINATION</button>

    </form>

    <h3>Search Patients</h3>

    <form *action=*"/doctor" *method=*"get">

      <input

*type=*"text"

*name=*"searchQuery"

*placeholder=*"Enter patient name or ID"

        required

      />

      <button *type=*"submit">Search</button>

    </form>

*<!-- Display Search Results -->*

    {{*#if* searchResults}}

      <h2>Search Results</h2>

      <table>

        <tr>

          <th>Patient ID</th>

          <th>Name</th>

          <th>Social Security Number</th>

          <th>Gender</th>

          <th>Address</th>

          <th>Blood Type</th>

          <th>Height</th>

          <th>Weight</th>

          <th>Phone Number</th>

        </tr>

        {{*#each* searchResults}}

          <tr>

            <td>{{this.PatientID}}</td>

            <td>{{this.Name}}</td>

            <td>{{this.SocialSecurityNumber}}</td>

            <td>{{this.Gender}}</td>

            <td>{{this.Address}}</td>

            <td>{{this.BloodType}}</td>

            <td>{{this.Height}}</td>

            <td>{{this.Weight}}</td>

            <td>{{this.PhoneNumber}}</td>

          </tr>

        {{*/each*}}

      </table>

    {{*else*}}

      <p>No search results found.</p>

    {{*/if*}}

*<!-- Examination List for Deletion -->*

    <h3>EXAMINATION DELETE</h3>

    {{*#each* examinations}}

      <p>

        Examination ID:

        {{this.ExaminationID}}

        <a *href=*"/doctor/delete/{{this.ExaminationID}}">Delete</a>

      </p>

    {{*/each*}}

*<!-- Examination List for Editing -->*

    <h3>Edit EXAMINATION</h3>

    {{*#each* examinations}}

      <form *action=*"/doctor/update/{{this.ExaminationID}}" *method=*"post">

        Examination ID:

        {{this.ExaminationID}}

        <input

*type=*"datetime-local"

*name=*"ExaminationDateTime"

*value=*"{{this.ExaminationDateTime}}"

        />

        <textarea

*name=*"ExaminationDetails"

        >{{this.ExaminationDetails}}</textarea>

        <input *type=*"number" *name=*"DoctorID" *value=*"{{this.DoctorID}}" />

        <input *type=*"number" *name=*"PatientID" *value=*"{{this.PatientID}}" />

        <button *type=*"submit">Edit</button>

      </form>

    {{*/each*}}

  </body>

</html>

병원 관리 시스템에서 의사 역할을 위해 설계된 doctor.js (라우터)와 doctor.hbs (템플릿)의 코드 기반 기능과 동작 과정을 설명하겠습니다. 이 기능은 의사가 자신의 환자 검사 기록을 관리하고, 환자를 검색하는 데 필요한 인터페이스를 제공합니다.

doctor.js 파일 분석

대시보드 페이지 렌더링: router.get("/")는 세션 정보를 확인하여 현재 사용자가 의사 역할을 가지고 있는지 검증합니다. 그 후, 의사의 환자 검사 기록이나 검색된 환자 정보를 데이터베이스에서 조회하여 페이지에 전달하고 렌더링합니다.

검사 기록 추가: router.post("/add")는 새로운 검사 기록을 추가하기 위한 폼 데이터를 처리합니다. 이 데이터는 insertSql.addExamination 함수를 통해 데이터베이스에 저장됩니다.

검사 기록 수정: router.post("/update/:id")는 기존 검사 기록을 수정하기 위한 요청을 처리합니다. 수정된 데이터는 updateSql.updateExamination 함수를 통해 데이터베이스에서 업데이트됩니다.

검사 기록 삭제: router.get("/delete/:id")는 특정 검사 기록을 삭제하는 요청을 처리합니다. deleteSql.deleteExamination 함수를 호출하여 데이터베이스에서 해당 기록을 제거합니다.

doctor.hbs 파일 분석

검사 기록 추가 폼: 사용자가 새로운 검사 기록을 추가할 수 있는 폼을 제공합니다. 이 폼은 검사 날짜, 검사 세부사항, 의사 ID, 환자 ID를 입력받아 서버로 전송합니다.

환자 검색 기능: 환자 이름이나 ID를 기반으로 검색할 수 있는 폼을 제공합니다. 검색된 결과는 테이블 형태로 페이지에 표시됩니다.

검사 기록 목록: 기존에 기록된 검사들을 나열하며, 각 기록 옆에는 수정과 삭제 옵션이 제공됩니다. 수정 폼을 통해 사용자는 검사 기록을 업데이트할 수 있으며, 삭제 링크를 통해 기록을 제거할 수 있습니다.

대시보드 페이지 렌더링

접근 권한 검증: 사용자가 의사 대시보드에 접근하려 할 때, router.get("/")는 사용자 세션을 확인하여 사용자가 의사인지 확인합니다. 의사가 아닐 경우 로그인 페이지로 리디렉션됩니다.

검사 기록 및 환자 검색 데이터 로딩: 로그인한 의사의 ID를 세션에서 가져와서, selectSql.getExaminations 함수를 통해 해당 의사에게 할당된 검사 기록을 데이터베이스에서 조회합니다. 추가적으로, 환자 검색 쿼리가 있을 경우 selectSql.searchPatients 함수를 사용하여 검색 결과를 로드합니다.

페이지 렌더링: 조회된 검사 기록과 검색 결과는 doctor 뷰로 전달되어 사용자에게 대시보드 페이지로 렌더링됩니다.

검사 기록 추가

폼 제출: 의사는 doctor.hbs에 제공된 '검사 기록 추가' 폼을 통해 새로운 검사 기록 정보를 입력하고 제출합니다.

데이터 처리 및 저장: router.post("/add")는 폼 데이터를 받아 insertSql.addExamination 함수를 호출하여 새로운 검사 기록을 데이터베이스에 저장합니다.

결과 반영: 검사 기록 추가 후, 사용자는 대시보드 페이지로 리디렉션되어 새로운 기록을 확인할 수 있습니다.

검사 기록 수정

업데이트 폼 제출: 기존 검사 기록 옆에 있는 '편집' 링크를 통해 수정 폼이 표시됩니다. 사용자는 변경하고자 하는 정보를 입력하고 제출합니다.

데이터 업데이트 처리: router.post("/update/:id")는 수정된 검사 기록 정보를 받아 updateSql.updateExamination 함수를 호출하여 데이터베이스의 해당 기록을 업데이트합니다.

결과 반영: 수정이 완료되면, 사용자는 대시보드 페이지로 리디렉션되어 변경된 정보를 확인할 수 있습니다.

검사 기록 삭제

삭제 링크 클릭: 사용자가 검사 기록 옆에 있는 '삭제' 링크를 클릭하면 해당 기록을 삭제할 수 있습니다.

검사 기록 삭제 처리: router.get("/delete/:id")는 검사 기록의 ID를 받아 deleteSql.deleteExamination 함수를 호출하여 해당 기록을 데이터베이스에서 제거합니다.

결과 반영: 삭제 후, 사용자는 대시보드 페이지로 리디렉션되며, 삭제된 기록은 더 이상 목록에 표시되지 않습니다.

**[nurse.js+nurse.hbs 설계]**

==============================nurse.js=================================

*import* express *from* "express";

*import* { selectSql, insertSql, updateSql, deleteSql } *from* "../Database/sql";

const router = express.Router();

*// Display nurse dashboard*

*// Display nurse dashboard and handle patient search*

router.get("/", async (req, res) => {

*if* (req.session.user && req.session.user.Role === "NURSE") {

*try* {

      const searchQuery = req.query.searchQuery;

      let treatments = [];

      let searchResults = [];

*// If there is a search query, perform the search, otherwise get treatments*

*if* (searchQuery) {

        searchResults = *await* selectSql.searchPatients(searchQuery);

      } *else* {

*// Assuming you have a getTreatments function that retrieves treatments for the nurse*

        const nurseId = req.session.user.UserID; *// UserID from the session*

        treatments = *await* selectSql.getTreatments(nurseId);

      }

*// Render with fetched treatments and search results*

      res.render("nurse", { treatments, searchResults });

    } *catch* (err) {

      console.error(err);

      res.status(500).send("Error retrieving data");

    }

  } *else* {

    res.redirect("/login");

  }

});

*// Add a new treatment*

router.post("/add", async (req, res) => {

  const { TreatmentDateTime, TreatmentDetails, NurseID, PatientID } = req.body;

*try* {

*await* insertSql.addTreatment(

      TreatmentDateTime,

      TreatmentDetails,

      NurseID,

      PatientID

    );

    res.redirect("/nurse");

  } *catch* (err) {

    console.error(err);

    res.status(500).send("Error adding treatment");

  }

});

*// Update treatment details*

router.post("/update/:id", async (req, res) => {

  const treatmentId = req.params.id;

  const { TreatmentDateTime, TreatmentDetails, PatientID } = req.body;

*try* {

*await* updateSql.updateTreatment(

      treatmentId,

      TreatmentDateTime,

      TreatmentDetails,

      PatientID

    );

    res.redirect("/nurse");

  } *catch* (err) {

    console.error(err);

    res.status(500).send("Error updating treatment");

  }

});

*// Delete a treatment*

router.get("/delete/:id", async (req, res) => {

  const treatmentId = req.params.id;

*try* {

*await* deleteSql.deleteTreatment(treatmentId);

    res.redirect("/nurse");

  } *catch* (err) {

    console.error(err);

    res.status(500).send("Error deleting treatment");

  }

});

*export* *default* router;

==============================nurse.hbs================================

<html *lang=*"en">

  <head>

    <meta *charset=*"UTF-8" />

    <meta *name=*"viewport" *content=*"width=device-width, initial-scale=1.0" />

    <title>Nurse Dashboard</title>

*<!-- Add your CSS styling here -->*

  </head>

  <body>

    <h1>Nurse Dashboard</h1>

*<!-- Patient Search Form -->*

    <h3>Search Patients</h3>

    <form *action=*"/nurse" *method=*"get">

      <input

*type=*"text"

*name=*"searchQuery"

*placeholder=*"Enter patient name or ID"

        required

      />

      <button *type=*"submit">Search</button>

    </form>

*<!-- Search Results -->*

    <h3>Search Results</h3>

    {{*#if* searchResults}}

      <table>

        <tr>

          <th>Patient ID</th>

          <th>Name</th>

          <th>Social Security Number</th>

          <th>Gender</th>

          <th>Address</th>

          <th>Blood Type</th>

          <th>Height</th>

          <th>Weight</th>

          <th>Phone Number</th>

        </tr>

        {{*#each* searchResults}}

          <tr>

            <td>{{this.PatientID}}</td>

            <td>{{this.Name}}</td>

            <td>{{this.SocialSecurityNumber}}</td>

            <td>{{this.Gender}}</td>

            <td>{{this.Address}}</td>

            <td>{{this.BloodType}}</td>

            <td>{{this.Height}}</td>

            <td>{{this.Weight}}</td>

            <td>{{this.PhoneNumber}}</td>

          </tr>

        {{*/each*}}

      </table>

    {{*else*}}

      {{*#if* searchQuery}}

        <p>No search results found for "{{searchQuery}}".</p>

      {{*else*}}

        <p>Enter search criteria to find patients.</p>

      {{*/if*}}

    {{*/if*}}

*<!-- Add Treatment -->*

    <h3>Add TREATMENT</h3>

    <form *action=*"/nurse/add" *method=*"post">

*<!-- Form fields for adding treatment -->*

      <input *type=*"datetime-local" *name=*"TreatmentDateTime" required />

      <textarea

*name=*"TreatmentDetails"

*placeholder=*"Treatment Details"

        required

      ></textarea>

      <input *type=*"number" *name=*"NurseID" *value=*"{{nurseId}}" />

*<!-- Assuming nurseId is passed from the server -->*

      <input *type=*"number" *name=*"PatientID" *placeholder=*"Patient ID" required />

      <button *type=*"submit">Add TREATMENT</button>

    </form>

*<!-- Treatment List for Deletion -->*

    <h3>TREATMENT DELETE</h3>

    {{*#each* treatments}}

      <p>

        Treatment ID:

        {{this.TreatmentID}}

        <a *href=*"/nurse/delete/{{this.TreatmentID}}">Delete</a>

      </p>

    {{*/each*}}

*<!-- Treatment List for Editing -->*

    <h3>Edit TREATMENT</h3>

    {{*#each* treatments}}

      <form *action=*"/nurse/update/{{this.TreatmentID}}" *method=*"post">

        Treatment ID:

        {{this.TreatmentID}}

        <input

*type=*"datetime-local"

*name=*"TreatmentDateTime"

*value=*"{{this.TreatmentDateTime}}"

        />

        <textarea *name=*"TreatmentDetails">{{this.TreatmentDetails}}</textarea>

        <input *type=*"number" *name=*"NurseID" *value=*"{{this.NurseID}}" />

        <input *type=*"number" *name=*"PatientID" *value=*"{{this.PatientID}}" />

        <button *type=*"submit">Edit</button>

      </form>

    {{*/each*}}

  </body>

</html>

병원 관리 시스템의 간호사 대시보드 기능을 구현하는 nurse.js 및 nurse.hbs 파일의 코드 구조와 기능에 대해 분석합니다. 이 기능은 간호사가 환자 치료 기록을 관리하고, 환자 정보를 검색할 수 있는 인터페이스를 제공합니다.

nurse.js 파일 분석

대시보드 페이지 렌더링: router.get("/")는 간호사의 세션 정보를 확인하여 접근 권한을 검증합니다. 사용자가 간호사 역할을 가지고 있으면, 환자 검색 결과 또는 간호사에게 할당된 치료 기록을 데이터베이스에서 조회하여 렌더링합니다.

치료 기록 추가: router.post("/add")는 새로운 치료 기록을 추가하기 위한 폼 데이터를 처리합니다. 폼 데이터는 insertSql.addTreatment 함수를 호출하여 데이터베이스에 저장됩니다.

치료 기록 수정: router.post("/update/:id")는 기존 치료 기록을 수정하기 위한 요청을 처리합니다. 수정된 데이터는 updateSql.updateTreatment 함수를 호출하여 데이터베이스에서 업데이트됩니다.

치료 기록 삭제: router.get("/delete/:id")는 특정 치료 기록을 삭제하는 요청을 처리합니다. deleteSql.deleteTreatment 함수를 호출하여 해당 기록을 데이터베이스에서 제거합니다.

nurse.hbs 파일 분석

환자 검색 폼: 환자 이름 또는 ID로 환자를 검색할 수 있는 폼을 제공합니다. 검색 결과는 테이블 형태로 페이지에 표시됩니다.

치료 기록 추가 폼: 사용자가 새로운 치료 기록을 추가할 수 있는 폼을 제공합니다. 이 폼은 치료 날짜, 치료 세부사항, 간호사 ID, 환자 ID를 입력받아 서버로 전송합니다.

치료 기록 목록: 기존에 기록된 치료들을 나열하며, 각 기록 옆에는 수정과 삭제 옵션이 제공됩니다. 수정 폼을 통해 사용자는 치료 기록을 업데이트할 수 있으며, 삭제 링크를 통해 기록을 제거할 수 있습니다.

대시보드 페이지 렌더링

접근 권한 검증: 사용자가 간호사 대시보드에 접근하려 할 때, router.get("/")는 사용자 세션을 확인하여 사용자가 간호사인지 검증합니다. 간호사가 아닐 경우, 로그인 페이지로 리디렉션됩니다.

치료 기록 및 환자 검색 데이터 로딩: 로그인한 간호사의 ID를 세션에서 가져와서, selectSql.getTreatments 함수를 통해 해당 간호사에게 할당된 치료 기록을 데이터베이스에서 조회합니다. 환자 검색 쿼리가 있을 경우, selectSql.searchPatients 함수를 사용하여 검색 결과를 로드합니다.

페이지 렌더링: 조회된 치료 기록과 검색 결과는 nurse 뷰로 전달되어 사용자에게 대시보드 페이지로 렌더링됩니다.

치료 기록 추가

폼 제출: 간호사는 nurse.hbs에 제공된 '치료 기록 추가' 폼을 통해 새로운 치료 기록 정보를 입력하고 제출합니다.

데이터 처리 및 저장: router.post("/add")는 폼 데이터를 받아 insertSql.addTreatment 함수를 호출하여 새로운 치료 기록을 데이터베이스에 저장합니다.

결과 반영: 치료 기록 추가 후, 사용자는 대시보드 페이지로 리디렉션되어 새로운 기록을 확인할 수 있습니다.

치료 기록 수정

업데이트 폼 제출: 기존 치료 기록 옆에 있는 '편집' 링크를 통해 수정 폼이 표시됩니다. 사용자는 변경하고자 하는 정보를 입력하고 제출합니다.

데이터 업데이트 처리: router.post("/update/:id")는 수정된 치료 기록 정보를 받아 updateSql.updateTreatment 함수를 호출하여 데이터베이스에서 해당 기록을 업데이트합니다.

결과 반영: 수정이 완료되면, 사용자는 대시보드 페이지로 리디렉션되어 변경된 정보를 확인할 수 있습니다.

치료 기록 삭제

삭제 링크 클릭: 사용자가 치료 기록 옆에 있는 '삭제' 링크를 클릭하면 해당 기록을 삭제할 수 있습니다.

치료 기록 삭제 처리: router.get("/delete/:id")는 치료 기록의 ID를 받아 deleteSql.deleteTreatment 함수를 호출하여 해당 기록을 데이터베이스에서 제거합니다.

결과 반영: 삭제 후, 사용자는 대시보드 페이지로 리디렉션되며, 삭제된 기록은 더 이상 목록에 표시되지 않습니다.

**[patient.js+patient.hbs 설계]**

==============================patient.js================================

*import* express *from* "express";

*import* { promisePool } *from* "../Database/sql"; *// 이렇게 promisePool을 가져옵니다.*

*import* { selectSql, insertSql, deleteSql } *from* "../Database/sql";

const router = express.Router();

*// Display patient dashboard with reservations*

router.get("/", async (req, res) => {

*if* (req.session.user && req.session.user.Role === "PATIENT") {

*try* {

      const userId = req.session.user.UserID; *// UserID from the session*

      const reservations = *await* selectSql.getReservationsByPatient(userId);

      console.log(userId);

      res.render("patient", { reservations }); *// Render with fetched reservations*

    } *catch* (err) {

      console.error(err);

      res.status(500).send("Error retrieving reservation data");

    }

  } *else* {

    res.redirect("/login");

  }

});

*// // Handle new reservation*

*// router.post("/reserve", async (req, res) => {*

*//   const { ReservationDateTime, DepartmentID } = req.body;*

*//   const userId = req.session.user.UserID;*

*//   try {*

*//     await insertSql.makeReservation(ReservationDateTime, DepartmentID, userId);*

*//     res.redirect("/patient");*

*//   } catch (err) {*

*//     console.error(err);*

*//     res.status(500).send("Error making reservation");*

*//   }*

*// });*

*// Cancel a reservation*

router.post("/cancel/:id", async (req, res) => {

  const reservationId = req.params.id;

*try* {

*await* deleteSql.cancelReservation(reservationId);

    res.redirect("/patient");

  } *catch* (err) {

    console.error(err);

    res.status(500).send("Error cancelling reservation");

  }

});

router.post("/reserve", async (req, res) => {

  const { ReservationDateTime, DepartmentID, PatientID } = req.body;

  console.log(ReservationDateTime, DepartmentID, PatientID);

*// 데이터베이스 연결을 가져옵니다.*

  const connection = *await* promisePool.getConnection();

*try* {

*// 트랜잭션을 시작합니다.*

*await* connection.beginTransaction();

*// 기존에 같은 시간과 부서에 대한 예약이 있는지 확인합니다.*

    const existingReservationQuery = `

          SELECT \* FROM reservation

          WHERE ReservationDateTime = ? AND DepartmentID = ?

        `;

    const [existingReservations] = *await* connection.query(

      existingReservationQuery,

      [ReservationDateTime, DepartmentID]

    );

*if* (existingReservations.length > 0) {

*// 이미 예약이 존재한다면 롤백을 수행하고 오류 메시지를 반환합니다.*

*await* connection.rollback();

      connection.release();

*return* res.status(400).send("This time slot is already reserved.");

    }

*// 새 예약을 추가합니다.*

    const addReservationQuery = `

          INSERT INTO reservation (ReservationDateTime, DepartmentID, PatientID)

          VALUES (?, ?, ?)

        `;

    console.log(addReservationQuery); *// 올바른 변수 이름으로 변경*

*await* connection.query(addReservationQuery, [

      ReservationDateTime,

      DepartmentID,

      PatientID,

    ]);

*// 트랜잭션을 커밋합니다.*

*await* connection.commit();

    connection.release();

    console.log("Transaction commit success");

  } *catch* (error) {

*// 오류가 발생하면 롤백을 수행합니다.*

*await* connection.rollback();

    connection.release();

*// 오류 메시지를 반환합니다.*

    res.status(500).send("An error occurred while making the reservation.");

  }

});

*export* *default* router;

==============================patient.hbs===============================

<html *lang=*"en">

  <head>

    <meta *charset=*"UTF-8" />

    <meta *name=*"viewport" *content=*"width=device-width, initial-scale=1.0" />

    <title>Patient Dashboard</title>

*<!-- Add your CSS styling here -->*

  </head>

  <body>

    <h1>Patient Dashboard</h1>

*<!-- Existing Reservations -->*

    <h2>Your Reservations</h2>

    {{*#each* reservations}}

      <div>

        <p>Reservation ID: {{this.ReservationNumber}}</p>

        <p>Date/Time: {{this.ReservationDateTime}}</p>

        <p>Department ID: {{this.DepartmentID}}</p>

*<!-- Add other details as needed -->*

        <form *action=*"/patient/cancel/{{this.ReservationNumber}}" *method=*"post">

          <button *type=*"submit">Cancel Reservation</button>

        </form>

      </div>

    {{*/each*}}

*{{!-- <!-- Make a New Reservation -->*

*<h3>Make a New Reservation</h3>*

*<form action="/patient/reserve" method="post">*

*<label for="ReservationDateTime">Reservation Date and Time:</label>*

*<input*

*type="datetime-local"*

*id="ReservationDateTime"*

*name="ReservationDateTime"*

*required*

*/>*

*<label for="DepartmentID">Department:</label>*

*<input*

*type="number"*

*id="DepartmentID"*

*name="DepartmentID"*

*required*

*placeholder="Enter Department ID"*

*/>*

*<!-- Assuming PatientID is passed from the server and should not be changed by the patient -->*

*<input type="hidden" name="PatientID" value="{{patientId}}" />*

*<button type="submit">Reserve</button>*

*</form> --}}*

    <h2>Department you want to reserve</h2>

    <table>

      <tr>

        <th>Department ID</th>

        <th>Department Name</th>

        <th>Phone Number</th>

      </tr>

      <tr>

        <td>1</td>

        <td>Cardiology</td>

        <td>886-874-6319</td>

      </tr>

      <tr>

        <td>2</td>

        <td>Neurology</td>

        <td>001-108-153-7731</td>

      </tr>

      <tr>

        <td>3</td>

        <td>Pediatrics</td>

        <td>001-680-008-7640</td>

      </tr>

      <tr>

        <td>4</td>

        <td>Oncology</td>

        <td>6291608259</td>

      </tr>

      <tr>

        <td>5</td>

        <td>Orthopedics</td>

        <td>3701122628</td>

      </tr>

    </table>

    <h3>Transaction test Make Reservation</h3>

    <form *action=*"/patient/reserve" *method=*"post">

      <div>

        <label *for=*"ReservationDateTime">Reservation Date and Time:</label>

        <input

*type=*"datetime-local"

*id*="ReservationDateTime"

*name=*"ReservationDateTime"

          required

        />

      </div>

      <div>

        <label *for=*"DepartmentID">Department:</label>

        <input

*type=*"number"

*id*="DepartmentID"

*name=*"DepartmentID"

          required

*placeholder=*"Enter Department ID"

        />

      </div>

      <label *for=*"DepartmentID">Your patient ID:</label>

      <input *type=*"number" *name=*"PatientID" *value=*"{{patientId}}" />

      <button *type=*"submit">Reserve</button>

    </form>

  </body>

</html>

병원 관리 시스템에서 환자 대시보드 기능을 구현하는 patient.js 및 patient.hbs 파일에 대한 코드 기반 설명을 드리겠습니다. 이 기능은 환자가 자신의 예약을 관리하고 새로운 예약을 진행할 수 있는 인터페이스를 제공합니다.

patient.js 파일 분석

대시보드 페이지 렌더링: router.get("/")는 환자의 세션 정보를 확인하여 접근 권한을 검증합니다. 사용자가 환자 역할을 가지고 있으면, 환자의 예약 목록을 데이터베이스에서 조회하여 페이지에 렌더링합니다.

예약 취소 기능: router.post("/cancel/:id")는 특정 예약을 취소하는 요청을 처리합니다. deleteSql.cancelReservation 함수를 호출하여 데이터베이스에서 해당 예약을 삭제합니다.

새로운 예약 추가 기능: router.post("/reserve")는 새로운 예약을 추가하기 위한 요청을 처리합니다. 이 과정에는 데이터베이스 트랜잭션을 사용하여, 예약이 중복되지 않도록 검증하고 새로운 예약을 데이터베이스에 추가합니다.

patient.hbs 파일 분석

기존 예약 목록: 사용자의 기존 예약 목록이 표시됩니다. 각 예약에 대한 세부 정보와 '예약 취소' 버튼이 제공됩니다.

환자 검색 폼: 환자는 예약하고자 하는 부서 정보를 테이블 형태로 확인할 수 있습니다.

새로운 예약 폼: 사용자는 새로운 예약을 진행할 수 있는 폼을 통해 예약 날짜, 시간, 부서 ID, 환자 ID를 입력하고 제출할 수 있습니다.

대시보드 페이지 렌더링

접근 권한 검증: 사용자가 환자 대시보드에 접근할 때, router.get("/")는 사용자의 세션을 확인하여 환자 역할인지 검증합니다. 환자가 아닐 경우, 로그인 페이지로 리디렉션됩니다.

예약 데이터 로딩: 로그인한 환자의 ID를 세션에서 가져와서, selectSql.getReservationsByPatient 함수를 통해 해당 환자의 예약 목록을 데이터베이스에서 조회합니다.

페이지 렌더링: 조회된 예약 목록은 patient 뷰로 전달되어 사용자에게 대시보드 페이지로 렌더링됩니다.

예약 취소

예약 취소 요청: 환자는 대시보드에서 제공되는 '예약 취소' 버튼을 사용하여 특정 예약을 취소할 수 있습니다.

예약 취소 처리: router.post("/cancel/:id")는 취소할 예약의 ID를 받아 deleteSql.cancelReservation 함수를 호출하여 데이터베이스에서 해당 예약을 삭제합니다.

결과 반영: 예약 취소 후, 사용자는 대시보드 페이지로 리디렉션되어 변경된 예약 목록을 확인할 수 있습니다.

새로운 예약 진행

예약 폼 제출: 환자는 대시보드에 제공된 '새로운 예약' 폼을 통해 예약 날짜, 시간, 부서 ID를 입력하고 제출합니다.

예약 데이터 처리: router.post("/reserve")는 폼 데이터를 받아 데이터베이스 트랜잭션을 사용하여 새로운 예약을 진행합니다. 이 과정에는 예약 중복 검사가 포함되어 있습니다.

결과 반영: 새로운 예약이 성공적으로 추가되면, 사용자는 대시보드 페이지로 리디렉션되어 새로 추가된 예약을 확인할 수 있습니다.

**<HOSPITAL 데이타베이스 생성 스크립트 전체>**

[create.sql]

목적: 이 파일은 데이터베이스의 초기 구조를 설정하는 데 사용됩니다. 테이블 생성, 스키마 정의, 데이터베이스 관련 초기 설정과 같은 작업을 포함할 수 있습니다.

주요 기능: 데이터베이스의 테이블 생성, 관계 설정, 기본 키 및 외래 키 제약조건 설정 등을 담당합니다.

[insert.sql]

목적: 이 파일은 데이터베이스에 초기 데이터를 삽입하는 데 사용됩니다. 테스트 데이터 또는 시스템의 초기 실행을 위한 기본 데이터를 포함할 수 있습니다.

주요 기능: 테이블에 레코드를 추가하는 INSERT 문을 포함하며, 초기 데이터 설정에 필요합니다.

[index.sql]

목적: 이 파일은 데이터베이스 쿼리의 성능 최적화를 위해 인덱스를 생성하는 데 사용될 수 있습니다.

주요 기능: 특정 컬럼에 대한 인덱스 생성을 통해 데이터 검색 속도와 효율성을 향상시키는 데 중점을 둡니다.

[view.sql]

목적: 이 파일은 데이터베이스 내에서 뷰(view)를 생성하는 데 사용됩니다. 뷰는 하나 이상의 테이블에서 파생된 가상의 테이블로, 특정 데이터 검색 작업을 간소화하는 데 유용합니다.

주요 기능: 복잡한 쿼리를 단순화하고, 사용자에게 필요한 데이터만을 제공하는 데 사용되는 뷰를 정의합니다.

[insert\_Examination.sql]

목적: 이 파일은 특히 '검사(Examination)' 관련 데이터를 데이터베이스에 삽입하는 데 사용됩니다.

주요 기능: 검사 데이터의 INSERT 문을 포함하여 검사 관련 정보를 초기화하거나 업데이트합니다.

[insert\_treatment.sql]

목적: 이 파일은 '치료(Treatment)' 관련 데이터를 데이터베이스에 삽입하는 데 사용됩니다.

주요 기능: 치료 데이터에 관한 INSERT 문을 포함하여 치료 정보를 초기 설정하거나 업데이트합니다.

[transaction.sql]

목적: 이 파일은 데이터베이스 트랜잭션 관련 작업을 정의하는 데 사용됩니다. 트랜잭션은 여러 데이터베이스 작업을 하나의 단위로 묶어 일관성과 무결성을 보장하는 방법입니다.

주요 기능: 여러 단계의 데이터베이스 작업을 그룹화하여, 전체 작업이 성공적으로 완료되거나 실패할 경우 롤백하는 로직을 포함합니다.

1. 실행 화면

**<관리자 페이지>**

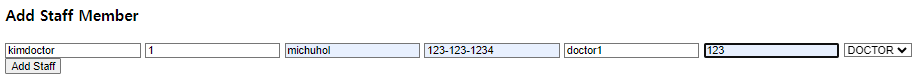
[의사 정보 입력]

관리자 admin1 | 123 | ADMIN 으로 로그인한 화면입니다.

텍스트, 스크린샷, 번호, 문서이(가) 표시된 사진

자동 생성된 설명

아래와 같이 Add staff Member에 데이터 값을 입력한 뒤 add staff 버튼을 누릅니다.



버튼을 누르면 아래의 화면과 같이 맨 밑에 kimdoctor와 관련된 데이터가 생성된 것을 확인할 수 있습니다.

텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

자동 생성된 설명

[의사 정보 수정]

텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

자동 생성된 설명

Kimdoctor의 이름을 kimdoctor123으로 바꾸고 주소를 donggu로 바꾸고 전화번호를 010-2400-8032로 수정 후 edit 버튼을 누른 결과입니다.

텍스트, 스크린샷, 번호, 평행이(가) 표시된 사진

자동 생성된 설명

정상적으로 의사 정보 수정이 이뤄진 것을 확인 할 수 있습니다. 아래의 데이터베이스에도 정상적으로 값이 바뀐 것을 확인할 수 있습니다.



[의사 정보 삭제]

텍스트, 스크린샷, 번호, 평행이(가) 표시된 사진

자동 생성된 설명

Doctor DELETE에서 맨 밑에 있는 kimdoctor123을 delete 버튼으로 삭제합니다. 아래와 같이 정상적으로 삭제되었음을 확인할 수 있습니다.

텍스트, 스크린샷, 문서, 폰트이(가) 표시된 사진

자동 생성된 설명

데이터베이스에서도 정상적으로 삭제된 것을 확인할 수 있습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

[간호사 정보 입력]

관리자 페이지에서 밑으로 화면을 내리게 되면 nurse와 관련된 값이 출력됩니다.

텍스트, 스크린샷, 문서, 디자인이(가) 표시된 사진

자동 생성된 설명

관리자 페이지 맨 위에 있는 add staff member에서 nurse로 목록칸을 바꿔주고 각각에 맞게 데이터를 입력 후 add staff 버튼을 누르면 됩니다.



그럼 아래와 같이 맨 밑에 kimnurse에 관한 정보가 출력되는 것을 확인할 수 있습니다.

텍스트, 스크린샷, 폰트, 문서이(가) 표시된 사진

자동 생성된 설명 텍스트, 번호, 문서, 스크린샷이(가) 표시된 사진

자동 생성된 설명

아래의 데이터베이스 상에서도 맨 밑을 보면 kimnurse가 제대로 저장된 것을 확인할 수 있습니다.

텍스트, 스크린샷, 메뉴, 폰트이(가) 표시된 사진

자동 생성된 설명

[간호사 정보 수정]

텍스트, 번호, 문서, 스크린샷이(가) 표시된 사진

자동 생성된 설명

Nurse edit로 이동하면 맨 밑에 kimnurse에 관한 정보가 있습니다. 그 칸에 다음과 같이 입력한 후 edit 버튼을 누릅니다.



Delete와 edit 칸 맨 밑을 보면 데이터가 변경된 것을 확인할 수 있습니다. 데이터베이스 상에서도 데이터가 수정된 것을 확인할 수 있습니다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명텍스트, 번호, 문서, 폰트이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 메뉴, 폰트이(가) 표시된 사진

자동 생성된 설명

[간호사 정보 삭제]

Nurse delete 에서 맨 밑 kimnurse123에 대한 delete 버튼을 누르도록 합니다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명텍스트, 번호, 문서, 폰트이(가) 표시된 사진

자동 생성된 설명

각화면과 데이터베이스 상에서 보면 kimnurse123에 대한 정보가 삭제되었음을 확인할 수 있습니다.

텍스트, 스크린샷, 폰트, 문서이(가) 표시된 사진

자동 생성된 설명텍스트, 번호, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 메뉴, 폰트이(가) 표시된 사진

자동 생성된 설명

**<직원(의사) 페이지>**

[EXAMINATION 정보 입력]

jennifer84 | 123 | DOCTOR로 로그인하면 보이는 화면입니다. 텍스트, 스크린샷, 번호, 문서이(가) 표시된 사진

자동 생성된 설명

2023-12-16 .오후 03:04 날짜와 시간을 선택한 후 검사 디테일에 knee injury를 입력하고 본인의 doctorID 를 입력시킨 후 해당 환자의 PatientID를 입력하도록 합니다. 그리고 add examination 버튼을 눌러 저장시킵니다.

스크린샷, 라인, 텍스트, 도표이(가) 표시된 사진

자동 생성된 설명

아래와 같이 Examination id 114번으로 생성된 것을 볼 수 있습니다.

텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

자동 생성된 설명

[EXAMINATION 정보 수정]

텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

자동 생성된 설명

Examination ID 114번에서 날짜와 시간 검사 디테일을 아래와 같이 바꿔보도록 하겠습니다.



아래의 화면과 같이 웹페이지 상으로나 데이터베이스 상으로나 데이터가 정상적으로 변경된 것을 확인할 수 있습니다.

텍스트, 번호, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

[EXAMINATION 정보 삭제]

텍스트, 번호, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

Examination delete 에서 examination 114에 대한 Delete 버튼을 눌러보도록 하겠습니다.

텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 폰트, 흑백이(가) 표시된 사진

자동 생성된 설명

웹페이지 상으로나 데이터베이스 상으로 114번에 대한 내용이 삭제되었음을 확인할 수 있습니다.

[조건에 맞는 patient 검색]

Search patients에서 2번 환자에 관한 정보를 검색해보도록 하겠습니다.

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

자동 생성된 설명

아래와 같이 새로운 페이지가 발생하면서 ID에 2가 포함된 환자들을 모두 출력시키는 것을 볼 수 있습니다.

텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

자동 생성된 설명

**<직원(간호사) 페이지>**

[treatment 정보 입력]

rowlandashley | 123 | NURSE로 로그인하면 보이는 화면입니다.

텍스트, 스크린샷, 번호, 평행이(가) 표시된 사진

자동 생성된 설명

아래와 같이 데이터를 입력합니다. 본인의 nurseID 1번을 입력하고 담당 환자의 PatientID를 입력한 후 add treatment 버튼을 누릅니다.

텍스트, 스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

아래와 같이 treatment id가 34번으로 생성된 것을 웹페이지 상으로나 데이터베이스 상으로 확인할 수 있습니다.

텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 폰트, 흑백이(가) 표시된 사진

자동 생성된 설명

[treatment 정보 수정]

텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

자동 생성된 설명

위 화면에서 34의 treatment id를 아래와 같이 수정한 뒤 edit 버튼을 눌러보도록 하겠습니다.



아래와 같이 데이터가 수정되었음을 알 수 있습니다.

텍스트, 스크린샷, 번호, 영수증이(가) 표시된 사진

자동 생성된 설명



[treatment 정보 삭제]

텍스트, 스크린샷, 번호, 영수증이(가) 표시된 사진

자동 생성된 설명

위 화면에서 treatment id 34번을삭제해 보도록 하겠습니다. Treatment delete에서 34번의 delete 버튼을 누른 결과입니다. 웹페이지나 데이터베이스에서 관련 데이터가 삭제되었음을 확인 할 수 있습니다.

텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

[조건에 맞는 patient 검색]

Search patients에서 patient ID에서 3이 포함된 모든 환자들을 조회해보도록 하겠습니다.

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

자동 생성된 설명

아래는 조회 결과입니다. 조회 결과 id에 3이 포함되어있으면 모두 출력되는 것을 볼 수 있습니다.

텍스트, 스크린샷, 문서, 폰트이(가) 표시된 사진

자동 생성된 설명

**<환자 페이지>**

[예약하기]

whitealexandria | 123 | PATIENT으로 환자로 로그인하면 보이면 화면입니다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

아래와 같이 transaction test make reservation에서 날짜와 시간을 선택하고 cardiology 부서인 1번을 입력하고 본인의 patientid를 입력한 후 reserve 버튼을 클릭합니다.

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

자동 생성된 설명

아래의 화면들을 통해 웹페이지나 데이터베이스 상에 제대로 데이터가 저장되었음을 확인할 수 있습니다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명 텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

[예약조회]

예약 조회는 기본적으로 본인 username과 password로 로그인하게 되면 해당 patient id가 예약한 예약을 조회할 수 있습니다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

[예약 취소]

아래의 your reservations에서 cancel reservation 버튼을 클릭하면 데이터가 삭제됩니다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

아래는 삭제한 결과 화면입니다. 웹페이지나 데이터베이스 상에서 데이터가 삭제되었음을 확인할 수 있습니다.

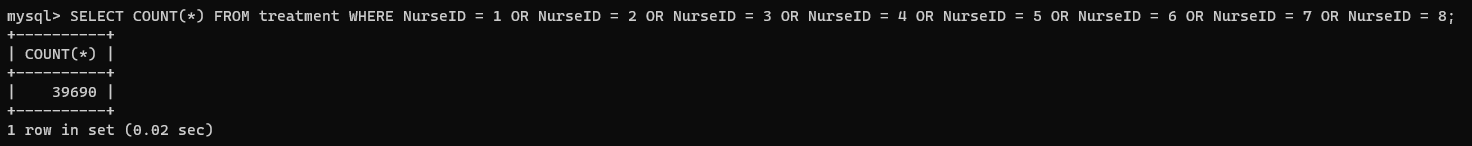
텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명**텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명**

**<INDEX 실행화면>**

**[인덱스 create 전 ]**

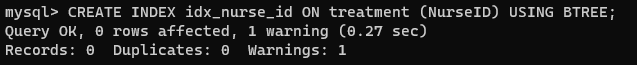
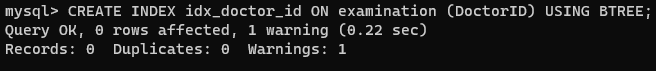
****

**텍스트, 스크린샷, 폰트, 블랙이(가) 표시된 사진

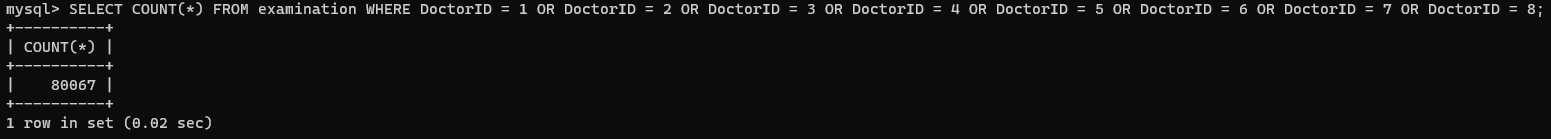
자동 생성된 설명** **텍스트, 스크린샷, 폰트, 블랙이(가) 표시된 사진

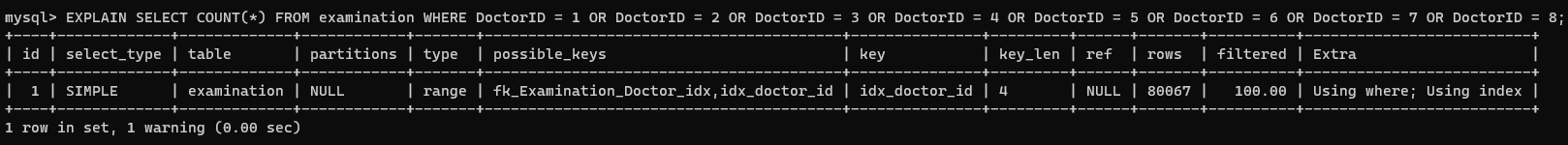
자동 생성된 설명**

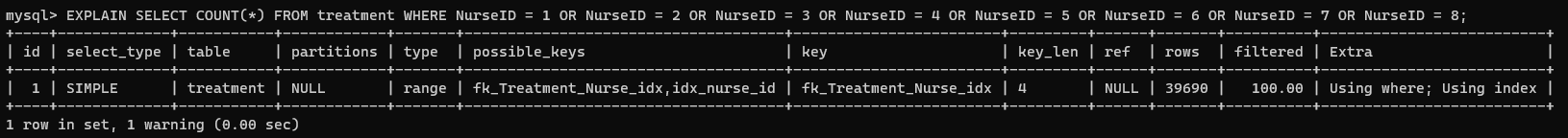
**[인덱스 create 후 ]**

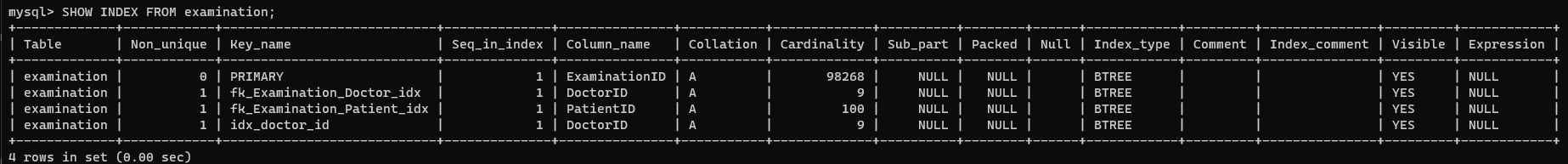
****

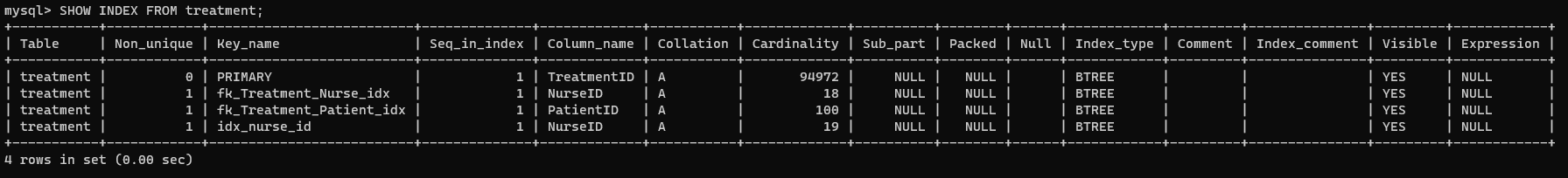
**텍스트, 스크린샷, 폰트, 블랙이(가) 표시된 사진

자동 생성된 설명**

****

****

****

****

examination 및 treatment 테이블에 대한 B-tree 인덱스 생성은 쿼리 실행 시간을 단축시키고, 실행 계획의 효율성을 향상시켰습니다. EXPLAIN의 결과에 따르면, 인덱스가 적용된 쿼리는 더 적은 행을 검색하여 실행 시간이 감소하였습니다. 이는 인덱스가 데이터 검색 성능을 크게 향상시킬 수 있음을 시사합니다.

**<VIEW 실행화면>**

**[ExaminationSummary 뷰]**

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명**

텍스트, 스크린샷, 메뉴이(가) 표시된 사진

자동 생성된 설명

…생략텍스트, 스크린샷, 메뉴, 폰트이(가) 표시된 사진

자동 생성된 설명

**[TreatmentSummary 뷰]**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 블랙이(가) 표시된 사진

자동 생성된 설명

…생략텍스트, 스크린샷, 메뉴, 블랙이(가) 표시된 사진

자동 생성된 설명

이러한 뷰는 복잡한 조인 쿼리를 간소화하여 사용자가 데이터베이스에서 필요한 정보를 더 쉽고 빠르게 검색할 수 있게 도와줍니다. ExaminationSummary와 TreatmentSummary 뷰는 특히 관리자, 의사, 간호사 등이 환자의 검사 및 치료 정보를 효율적으로 관리하고 조회하는 데 유용하게 사용될 수 있습니다. 본 보고서에서 제시된 ExaminationSummary 및 TreatmentSummary 뷰는 hospitalmanagement 데이터베이스의 사용성을 향상시키고, 관련 정보에 대한 접근성을 높이는 중요한 도구입니다. 이 뷰를 통해 사용자는 검사 및 치료에 관한 중요한 정보를 빠르고 정확하게 파악할 수 있습니다.

**<트랜젝션 실행화면>**

**[커밋 성공]**

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명



**[커밋 실패 후 롤백]**

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 폰트, 일렉트릭 블루이(가) 표시된 사진

자동 생성된 설명

트랜잭션 시작: 환자가 예약을 시도할 때, 서버는 데이터베이스 연결을 생성하고 트랜잭션을 시작합니다.

예약 가능 여부 확인: 시스템은 입력된 예약 날짜와 부서에 대해 기존 예약이 있는지 데이터베이스를 확인합니다. 이는 다른 트랜잭션에 의해 동시에 수정되지 않도록 잠금을 설정합니다(SELECT ... FOR UPDATE).

**[예약 처리]**

기존 예약이 없는 경우, 시스템은 새 예약을 데이터베이스에 추가하고 트랜잭션을 커밋합니다. 기존 예약이 있는 경우, 시스템은 트랜잭션을 롤백하고 환자에게 예약이 불가능하다는 메시지를 반환합니다.

**[테스트 결과]**

예약 성공: 환자가 겹치지 않는 시간과 부서를 선택하여 예약을 시도했을 때, 시스템은 예약을 성공적으로 처리하고 예약 완료 메시지를 반환했습니다.

중복 예약 시도: 두 환자가 동일한 시간과 부서에 대해 예약을 시도했을 때, 먼저 시도된 예약은 성공했으며, 두 번째 예약은 시스템이 롤백을 수행하고 예약 실패 메시지를 반환했습니다.

텍스트, 스크린샷, 폰트, 블랙이(가) 표시된 사진

자동 생성된 설명

위 화면 처럼 데이터베이스에도 제대로 저장된 것을 확인할 수 있습니다.

**[트랜젝션 스크립트 실행 화면]**

텍스트, 스크린샷, 메뉴, 폰트이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

데이터값들이 중복되지 않다면 저장에 성공하는 것을 볼 수 있습니다. Reservation successful! 메시지가 출력되며, 새로운 예약이 데이터베이스에 추가됩니다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

중복 예약 시 This time slot is already reserved. 메시지가 출력되며, 아무런 행이 추가되지 않습니다.

1. 결론

이 텀 프로젝트를 통해 데이터베이스 설계의 핵심 원칙과 실제 응용 프로그램 구현에 필요한 기술을 심도 있게 학습하였습니다. ERD 설계에서부터 테이블의 정규화, 인덱스와 뷰의 구현, 데이터 삽입, 그리고 트랜잭션 관리에 이르기까지, 이 프로젝트는 복잡한 데이터베이스 시스템의 설계 및 구현 과정에 대한 깊은 이해를 제공합니다. 또한, 웹 인터페이스의 구현을 통해 사용자 친화적인 환경을 조성하며, 실제 병원 관리 시스템의 요구 사항을 충족하는 실용적인 솔루션을 제공합니다. 프로젝트의 완성은 이론과 실습의 결합을 통해 효과적인 학습 경험을 제공하며, 데이터베이스 설계 및 개발에 대한 학생들의 실력을 한층 더 향상시켰습니다.

ERD 및 데이터베이스 설계: 병원 시스템의 복잡한 데이터 관계를 효과적으로 모델링하는 ERD를 설계하여, 데이터 무결성과 일관성을 보장하는 강력한 데이터베이스 구조를 구축했습니다.

데이터베이스 정규화와 인덱스 최적화: 데이터 중복을 최소화하고 쿼리 성능을 극대화하기 위해 테이블을 적절한 정규형으로 정규화하고, 인덱스를 신중하게 구현했습니다. 이를 통해 데이터 검색 속도와 시스템의 전반적인 효율성을 향상시켰습니다.

트랜잭션 관리 및 데이터 무결성 보장: 복잡한 데이터베이스 작업을 위한 트랜잭션 관리를 통해 시스템의 데이터 무결성을 보장했습니다. 이는 데이터 일관성 유지 및 복구 메커니즘에서 중요한 역할을 합니다.

웹 인터페이스 구현: 사용자 친화적인 웹 인터페이스를 통해 관리자, 의사, 간호사, 환자 등 다양한 사용자가 시스템과 효과적으로 상호작용할 수 있도록 했습니다. 이는 Handlebars 템플릿과 Express.js를 활용하여 구현되었습니다.

보안 및 접근 제어: 사용자 권한에 따른 접근 제어를 통해 시스템의 보안을 강화했습니다. 이는 세션 관리와 역할 기반의 접근 제어 로직을 통해 구현되었습니다.

응용 프로그램의 실제 요구사항 대응: 환자의 예약 시스템, 의료진의 검사 및 치료 기록 관리 등 병원 시스템의 핵심 요구사항을 충실히 반영했습니다.