CS 418 Course Project

February 28, 2024

Sean Baker

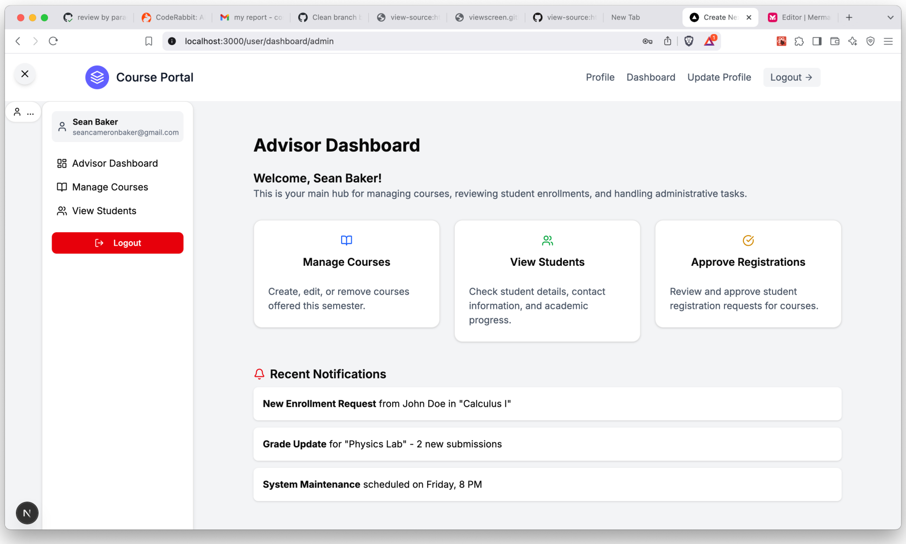
Table of Contents

Table of Figures

1. What is your website about?

My website is a **Course Advising Web Application** that helps students enroll in courses, track prerequisites, and request advising while allowing administrators to manage approvals. The tech stack includes **React.js for the frontend**, **Node.js with Express.js for the backend**, and **Mysql** for the database. I am building the backend on **Express.js** to handle authentication, advising workflows, and role-based access control. The authentication system includes **email verification, password encryption, and two-factor authentication (2FA)** for security. Below is a table picture of the admin dashboard

Figure 1 Screen capture of Admin Dashboard



Advisor Dashboard

Welcome, Sean Baker!
This is your main hub for managing courses, reviewing student enrollments, and handling administrative tasks.

Manage Courses
Create, edit, or remove courses offered this semester.

View Students
Check student details, contact information, and academic progress.

Approve Registrations
Review and approve student registration requests for courses.

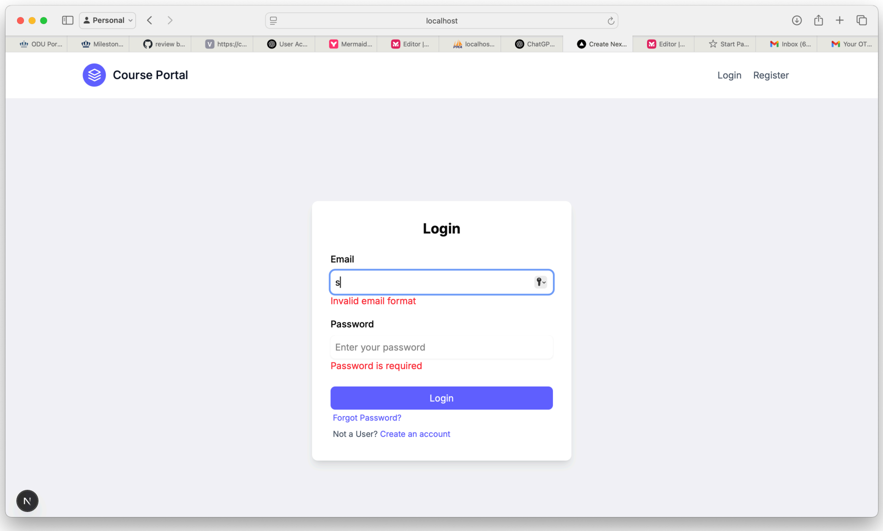
Recent Notifications

New Enrollment Request from John Doe in "Calculus I"

Grade Update for "Physics Lab" - 2 new submissions

System Maintenance scheduled on Friday, 8 PM

Figure 2 Login Form Screen



Invalid email format

Password is required

2. Milestone Accomplishments (10 points)

2.1 Implementation Status

Figure 3 Implementation Table

Fulfilled	Feature#	Specification
Yes	1	Users can register new accounts using email addresses
Yes	2	Users are identified by email address
Yes	3	Password is encrypted before storing in database (using bcrypt)
Yes	4	Users cannot register duplicate accounts (email uniqueness enforced)
Yes	5	User receives verification email upon registration
Yes	6	Users cannot log in until email is verified
Yes	7	Users can log into website using registered accounts
Yes	8	Users can reset passwords if forgotten
Yes	9	Users can change passwords after login
Yes	10	2-factor authentication implemented (email OTP)
Yes	11	Website has homepage for each user with profile and settings
Yes	12	Admin user created from backend
Yes	13	Admin user has different view from regular user

Figure 4 Feature Implementation Files

Feature	Implementation Status	Files Involved
User Authentication	Implemented	server/src/controllers/AuthController.js server/src/models/AuthModel.js

		server/src/middleware/AuthMiddleware.js client/src/hooks/useAuth.jsx
Email Verification	Implemented	server/src/utils/otpService.js server/src/services/emailService.js client/src/components/VerifyEmail.jsx
Course Management	Implemented	server/src/controllers/CourseController.js server/src/models/CoursesModel.js client/src/components/courses/courseList.jsx
User Profile	Implemented	server/src/controllers/UserController.js server/src/models/UserModel.js client/src/hooks/useProfile.jsx
Password Management	Implemented	server/src/controllers/PasswordController.js client/src/hooks/usePassword.jsx

Figure 5 Login Components Table

Component	Description	File Location
Login Form	Two-step login interface (password + OTP)	client/src/app/account/login/page.jsx
Authentication Hook	Client-side authentication logic	client/src/hooks/useAuth.jsx
JWT Validation	Token validation utility	client/src/utils/validJwt.js
Auth Controller	Server-side authentication logic	server/src/controllers/AuthController.js
Auth Middleware	Token verification middleware	server/src/middleware/AuthMiddleware.js
Password Controller	Password reset and verification	server/src/controllers/PasswordController.js
User Routes	Authentication and user API endpoints	server/src/routes/userRoutes.js
OTP Service	One-time password generation	server/src/controllers/AuthController.js
Email Service	Email delivery for verification	server/src/utils/sendEmailService.js

3. Project Architecture

Frontend:

- **Framework**: Next.js 15.2.0 with React 19
- **State Management**: Redux (Redux Toolkit)
- **UI Components**: Custom components with Tailwind CSS and Shadcn UI
- **Form Handling**: Formik with Yup validation
- **HTTP Client**: Axios for API calls
- **Authentication**: JWT token storage with HTTP-only cookies

Backend:

- **Runtime**: Node.js
- **Framework**: Express.js 4.21.2
- **Authentication**: JWT (jsonwebtoken) with secure cookie storage
- **Password Security**: Bcrypt for hashing
- **Email Services**: Nodemailer for verification, OTP, and password reset
- **Logging**: Winston logger

Database:

- **RDBMS**: MySQL (using mysql2 driver)
- **Schema**: Relational database with tables for users and courses

3.1 Frontend Components

1. **Authentication Module**
   - Registration page with validation
   - Login with two-factor authentication
   - Email verification flow
   - Password reset functionality
2. **User Dashboard**
   - Student view: Course browsing and enrollment
   - Admin view: User management and course administration
3. **Profile Management**
   - View and update personal information
   - Change password functionality
4. **Core Components**
   - Redux store for global state
   - API client for backend communication
   - Authentication context for session management
   - Form components with validation

3.2 Backend Components

1. **API Routes Layer**
   - User routes (authentication, profile)
   - Course routes (management, enrollment)
   - Admin-specific routes
2. **Controllers**
   - AuthController: Registration, login, verification
   - UserController: Profile management

- PasswordController: Reset and change functionality
- CourseController: Course operations

3. **Models**
   - UserModel: User data operations
   - AuthModel: Authentication-specific operations
   - CoursesModel: Course-related database operations
4. **Services**
   - Email service: Sending verification, OTP, and reset emails
   - OTP service: Generating and validating one-time passwords
   - Authentication service: Token generation and validation
5. **Middleware**
   - Authentication middleware: Verifying JWT tokens

3.3 Diagrams Of System Architecture

Figure 6 Diagram Of Program Components

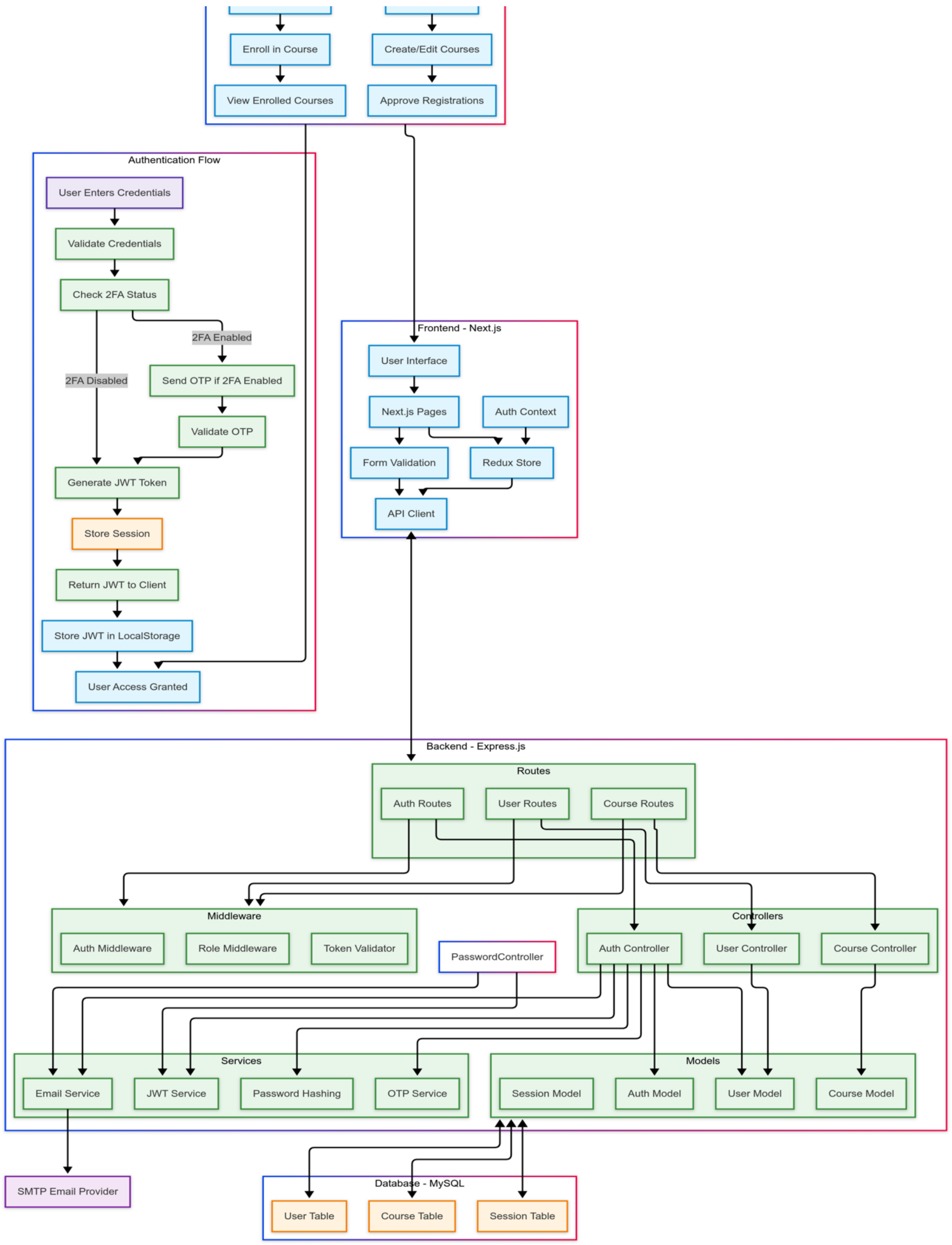
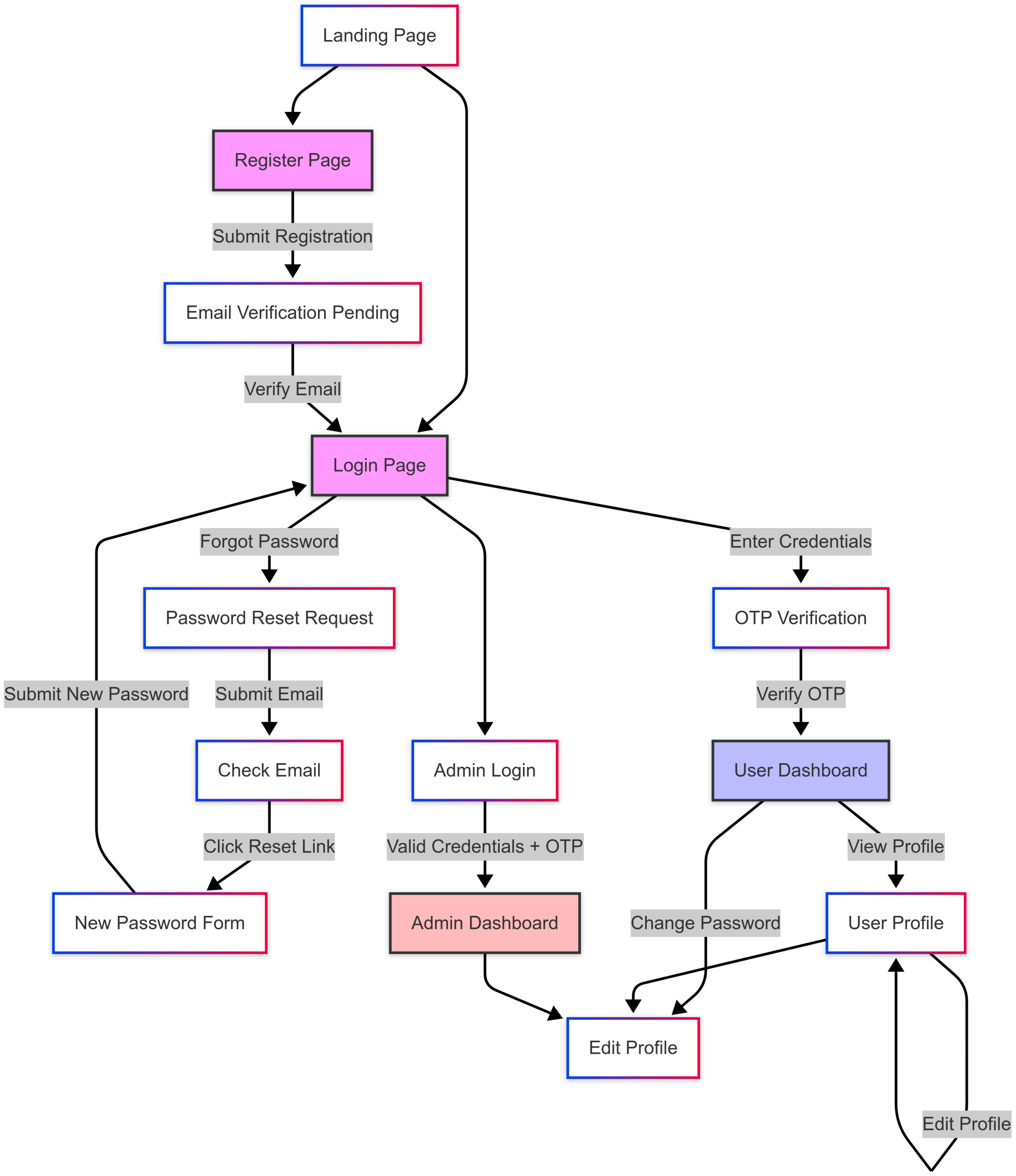


Figure 7 Authentication Flow Diagram


Landing Page
Register Page
Submit Registration
Email Verification Pending
Verify Email
Login Page
Forgot Password
Enter Credentials
Password Reset Request
OTP Verification
Submit New Password
Submit Email
Verify OTP
Check Email
Admin Login
User Dashboard
Click Reset Link
Valid Credentials + OTP
View Profile
New Password Form
Admin Dashboard
Change Password
User Profile
Edit Profile
Edit Profile

•

4. Database Design (20 points)

4.1 Overall Design

The **course_portal** database follows a **relational design** with a structured schema that manages users, courses, student enrollments, prerequisites, and course advising. The user **table** is the central entity, storing authentication data, user roles (admin vs. student), and verification statuses. It connects to the student_courses **table**, tracking course enrollments, statuses (Enrolled, Completed, Dropped), and grades. The courses **table** maintains course details and links to the course_prerequisites **table** to enforce prerequisite requirements. Additionally, the courseadvising **table** stores student advising requests, including prerequisites, GPA, planned courses, and approval status. The relationships enforce **referential integrity** with foreign key constraints, ensuring that students cannot enroll in courses without meeting prerequisite requirements and allowing administrators to manage course advising effectively.

Figure 8 Database Schema

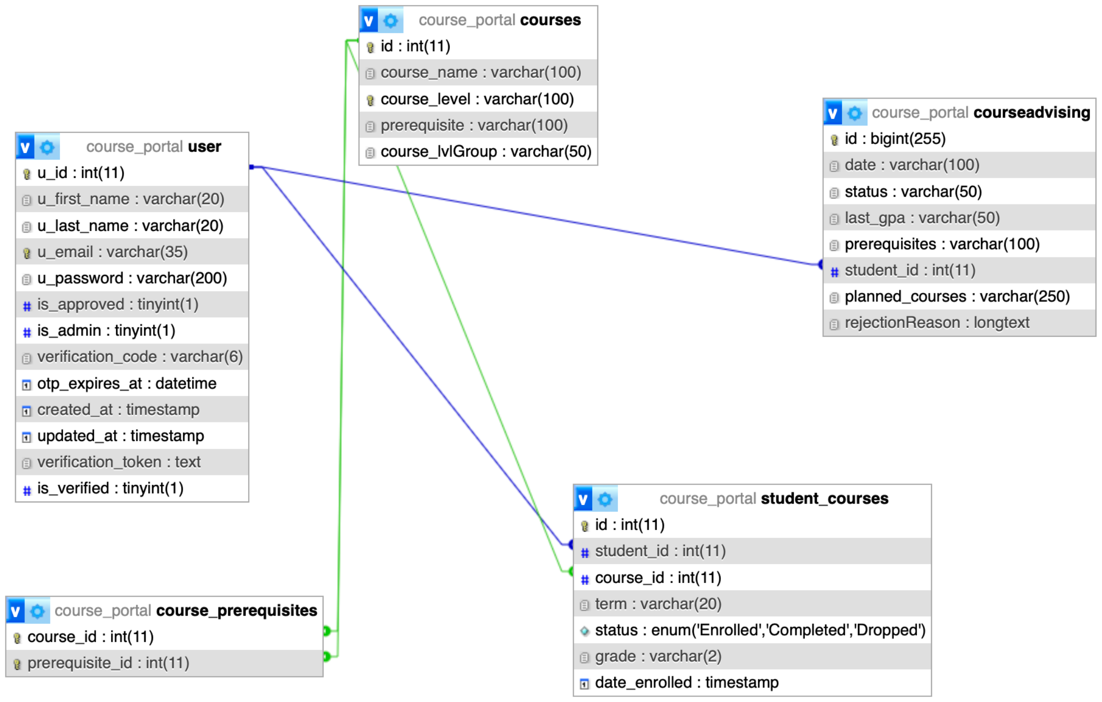
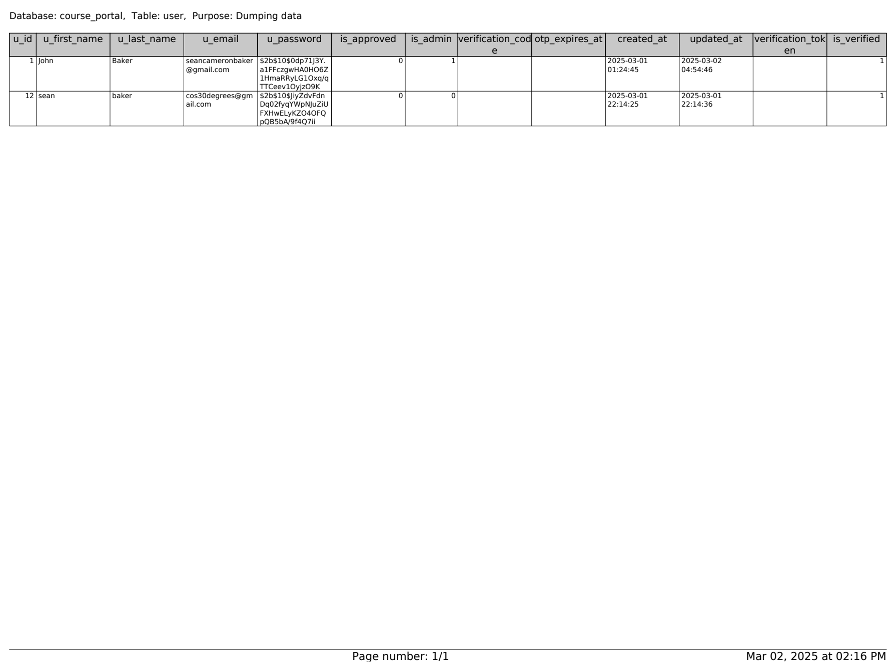

course_portal courses
id : int(11)
course_name : varchar(100)
course_level : varchar(100)
prerequisite : varchar(100)
course_lvlGroup : varchar(50)
course_portal courseadvising
id : bigint(255)
date : varchar(100)
status : varchar(50)
last_gpa : varchar(50)
prerequisites : varchar(100)
student_id : int(11)
planned_courses : varchar(250)
rejectionReason : longtext
course_portal user
u_id : int(11)
u_first_name : varchar(20)
u_last_name : varchar(20)
u_email : varchar(35)
u_password : varchar(200)
is_approved : tinyint(1)
is_admin : tinyint(1)
verification_code : varchar(6)
otp_expires_at : datetime
created_at : timestamp
updated_at : timestamp
verification_token : text
is_verified : tinyint(1)
course_portal student_courses
id : int(11)
student_id : int(11)
course_id : int(11)
term : varchar(20)
status : enum('Enrolled','Completed','Dropped')
grade : varchar(2)
date_enrolled : timestamp
course_portal course_prerequisites
course_id : int(11)
prerequisite_id : int(11)

Figure 9 Database Table With Data



Database: course_portal, Table: user, Purpose: Dumping data

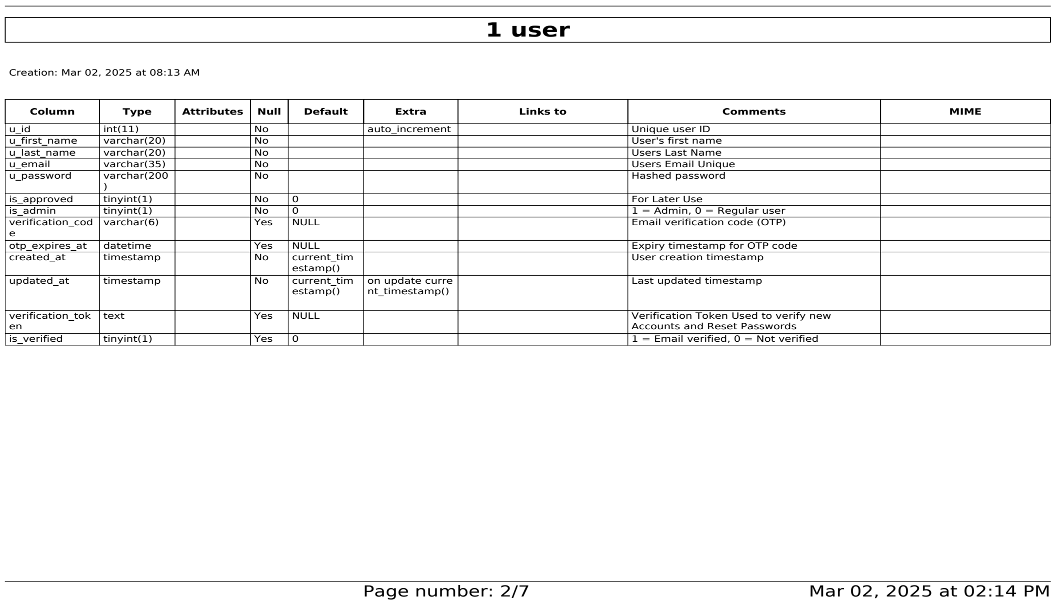
u_id	u_first_name	u_last_name	u_email	u_password	is_approved	is_admin	verification_code	otp_expires_at	created_at	updated_at	verification_token	is_verified
1	John	Baker	seancameronbaker@gmail.com	$2b$10$0dp71J3Y.a1FFczgwHA0HO6Z1HmaRRyLG1Oxq/qTTCeev1OyjzO9K	0	1			2025-03-01 01:24:45	2025-03-02 04:54:46		1
12	sean	baker	cos30degrees@gmail.com	$2b$10$JiyZdvFdnDq02fyqYWpNJuZiUFXHwELyKZO4OFQpQB5bA/9f4Q7ii	0	0			2025-03-01 22:14:25	2025-03-01 22:14:36		1

Page number: 1/1 Mar 02, 2025 at 02:16 PM

4.2 User table Design

The `user` table stores and manages user authentication, role-based access, and verification details for the course advising system. It includes fields for **user** identity (`u_id`, `u_first_name`, `u_last_name`, `u_email`), authentication (`u_password`, `verification_code`, `otp_expires_at`), and account status (`is_verified`, `is_approved`, `is_admin`). The table ensures security by storing encrypting passwords and enforcing email verification before login. Admin users are distinguished using the `is_admin` field, and timestamps (`created_at`, `updated_at`) track user activity. This table is central to managing students, advisors, and administrators in the system.

Figure 10 User table with comments



1 user

Creation: Mar 02, 2025 at 08:13 AM

Column	Type	Attributes	Null	Default	Extra	Links to	Comments	MIME
u_id	int(11)		No		auto_increment		Unique user ID	
u_first_name	varchar(20)		No				User's first name	
u_last_name	varchar(20)		No				Users Last Name	
u_email	varchar(35)		No				Users Email Unique	
u_password	varchar(200)		No				Hashed password	
is_approved	tinyint(1)		No	0			For Later Use	
is_admin	tinyint(1)		No	0			1 = Admin, 0 = Regular user	
verification_code	varchar(6)		Yes	NULL			Email verification code (OTP)	
otp_expires_at	datetime		Yes	NULL			Expiry timestamp for OTP code	
created_at	timestamp		No	current_timestamp()			User creation timestamp	
updated_at	timestamp		No	current_timestamp()	on update current_timestamp()		Last updated timestamp	
verification_token	text		Yes	NULL			Verification Token Used to verify new Accounts and Reset Passwords	
is_verified	tinyint(1)		Yes	0			1 = Email verified, 0 = Not verified	

Page number: 2/7 Mar 02, 2025 at 02:14 PM

5. Implementation (40 points)

5.1. Users should be able to register new accounts using email addresses

Implementation Files:
Frontend: *client/src/app/account/register/page.jsx*
Backend: *server/src/controllers/AuthController.js* (register method)
Validation: *client/src/validation/schemas.jsx* (registerSchema)

The registration page collects user information (first name, last name, email, password) through a form built with Formik for validation. When submitted, the form triggers the handleRegister function that sends a POST request to the server. The AuthController validates the data, checks for duplicate emails, hashes the password using bcrypt, and creates a new user record with a verification token.

5.2. Users are identified by email address

Implementation Files:
Database: *database/course_db.sql* (u_email field with UNIQUE constraint)

Backend: *server/src/models/UserModel.js* (findByEmail method)
The system uses email as the primary identifier for users. The database schema enforces email uniqueness with a UNIQUE constraint on the u_email field. The UserModel contains methods like findByEmail that retrieve user data based on email address:

```
static async findByEmail(email) {
  const [rows] = await pool.execute("SELECT * FROM user WHERE u_email = ?", [email]);
  return rows.length > 0 ? rows[0] : null;
}
```

5.3. Password must be encrypted before storing in the database

Implementation Files:
Backend: *server/src/controllers/AuthController*.js (register method)
Hashing Utility: *server/src/utils/authService.js* (hashPassword function)

Passwords are encrypted using bcrypt before storage. In the AuthController's register method:
const hashedPassword = await hashPassword(password);
The hashPassword function in authService.js uses bcrypt with a salt factor of 10 to create secure password hashes that are stored in the database instead of plaintext passwords.

5.4. Users cannot register duplicate accounts using the same email address

Implementation Files:
Database: database/course_db.sql (UNIQUE constraint on u_email)
Backend: server/src/controllers/AuthController.js (duplicate check)
This is implemented through both database constraints and application logic:

```
// In AuthController.js
const existingUser = await UserModel.findByEmail(email);
if (existingUser) {
  logger.warn(`Registration failed - Email already exists: ${email}`);
  return res.status(400).json({ status: "failed", message: "Email already exists" });
}
```

The database schema also enforces this with a unique constraint on the email field, providing a second layer of protection.

5.5. The user should receive a verification email upon successful registration

Implementation Files:
Backend Service: server/src/utils/emailService.js (sendVerificationEmail)

Controller: server/src/controllers/AuthController.js (register method)
After successful user creation, a verification email is sent:

```
// Generate verification token
const verificationToken = generateToken({ email }, "1d"); // 1-day expiration

// Create user in DB with token
await UserModel.createUser({
  firstName, lastName, email, hashedPassword, verificationToken
});
// Send verification email
await sendVerificationEmail(email, verificationToken);
```

The email contains a verification link with a JWT token that expires after 24 hours.

5.6. Users cannot log in to the system until their email has been verified

Implementation Files:
Database: *database/course_db.sql* (is_verified field)
Verification: *server/src/controllers/PasswordController.js* (verifyEmail method)
Login Check: *server/src/controllers/AuthController.js* (userLogin method)
The user table includes an is_verified boolean field that defaults to 0 (false). During login, the system checks this field and rejects login attempts if the email isn't verified. The verification process is handled by the PasswordController's verifyEmail method, which validates the token from the verification email and updates the user's verification status.

5.7 Users should be able to log into your website using the accounts they registered

Implementation Files:
Frontend: *client/src/app/account/login/page.jsx*
Backend: *server/src/controllers/AuthController.js* (userLogin method)
Hook: *client/src/hooks/useAuth.jsx*

The login page collects email and password, which are validated and sent to the backend. The AuthController verifies credentials, checks verification status, and then proceeds to the 2FA step by generating and sending an OTP. The login process is a two-step flow due to 2FA implementation.

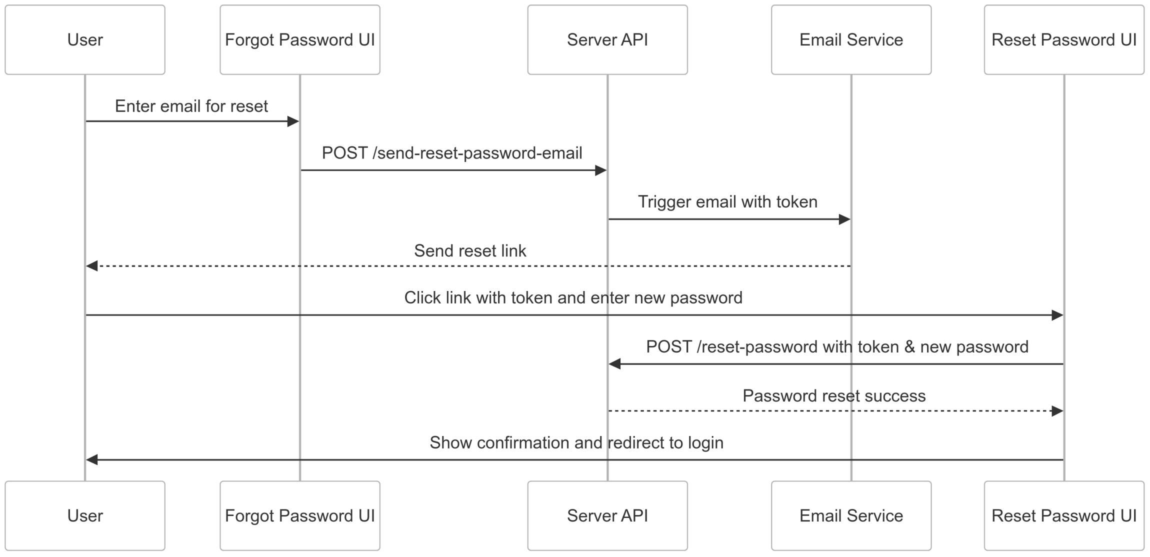
5.8. Users should be able to reset their passwords if they forget it

Implementation Files:
Frontend Request: *client/src/app/account/send-password-reset-email/page.jsx*
Frontend Reset: *client/src/app/account/reset-password/[token]/page.jsx*
Backend: *server/src/controllers/PasswordController.js*
Email Service: *server/src/utils/emailService.js* (sendResetPasswordEmail)

The password reset flow involves:
User requests a reset through the send-password-reset-email page
System generates a token and sends it via email
User clicks link in email, opening the reset-password page
New password is submitted and saved if the token is valid
The PasswordController handles token generation, validation, and password updates:

```
// Reset Password via Token
static async resetPassword(req, res) {
  // Verify token validity
  // Hash the new password
  // Update in database
}
```

Figure 11 Reset Password Authentication Flow


User
Forgot Password UI
Server API
Email Service
Reset Password UI
Enter email for reset
POST /send-reset-password-email
Trigger email with token
Send reset link
Click link with token and enter new password
POST /reset-password with token & new password
Password reset success
Show confirmation and redirect to login

5.9. Users should be able to change their passwords after they login

Implementation Files:
Backend: *server/src/controllers/UserController.js* (changeUserPassword method)
Frontend: *client/src/app/user/change-profile/page.jsx*
The UserController provides a changeUserPassword method for authenticated users:

```
static async changeUserPassword(req, res) {
const { password, password_confirmation } = req.body;
// Validate password match
// Hash new password
// Update in database
}
```

This endpoint is protected by authentication middleware to ensure only logged-in users can access it.

5.10. A 2-factor-authentication should be used when a user attempts to login

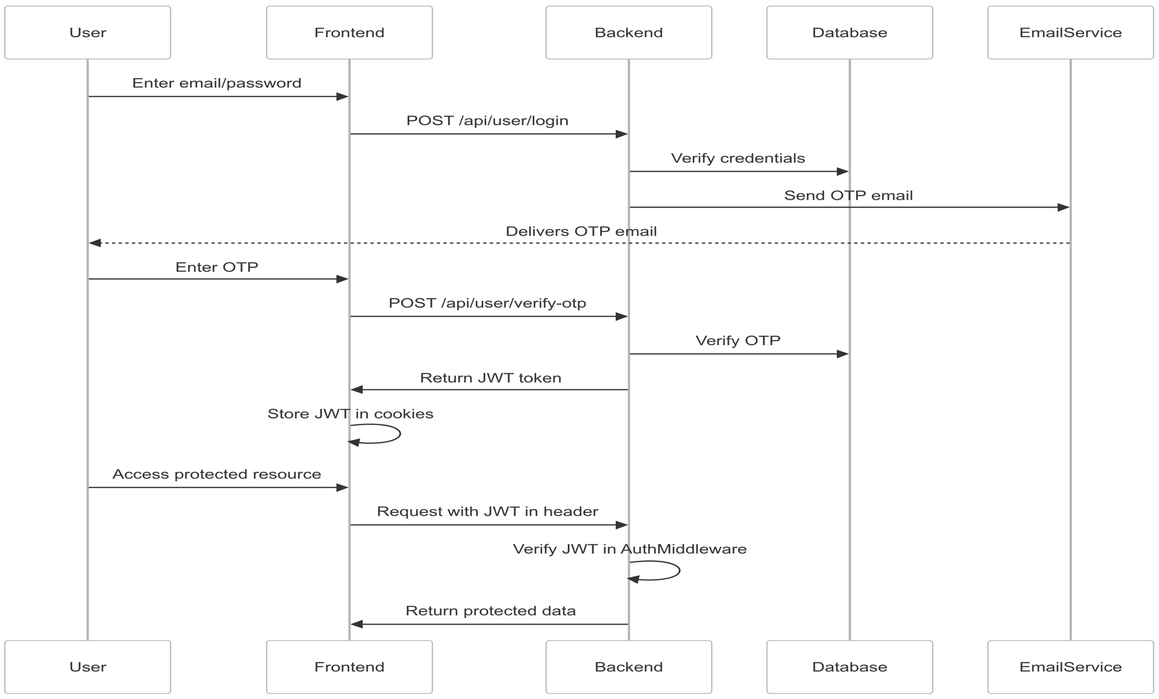
Implementation Files:
Frontend: *client/src/app/account/login/page.jsx* (two-step form)
Backend Generation: *server/src/utils/otpService.js*
Backend Verification: server/src/controllers/AuthController.js (verifyOTP method)

The system implements email-based OTP as the second factor:
After password verification, the system generates a 6-digit OTP
OTP is sent to the user's email via the sendOTPEmail function
User enters OTP in the login form's second step

AuthController.verifyOTP validates the OTP and completes the login process

```
// In otpService.js
export const generateOTP = () => {
  return Math.floor(100000 + Math.random() * 900000).toString();
};
```

Figure 12 User Registration and 2FA auth Flow



5.11. The website has a homepage for each user with profile management

Implementation Files:

Profile View: client/src/app/user/profile/page.jsx
Profile Update: client/src/app/user/update-profile/page.jsx
Backend: server/src/controllers/UserController.js (loggedUser and updateUserProfile methods)

The profile page displays user information, verification status, and admin status. It fetches data using the useProfile hook, which calls the backend's loggedUser endpoint. Users can update their profile information through the update-profile page, which submits changes to the updateUserProfile endpoint.

5.12. An admin user should be created from the backend

Implementation Files:
Database: *database/course_db.sql (is_admin field)*
Data: Sample admin user visible in SQL dump
The database schema includes an **is_admin** boolean field that defaults to 0 (false). Admin creation is not directly visible in the code examined, but the SQL dump shows an existing admin user: Admin users need to be created by directly setting this field to 1, which can only be done at the database level, not through the regular registration flow.

5.13. An admin user has a different view from a regular user

Implementation Files:
Admin Dashboard: client/src/app/user/dashboard/admin/page.jsx
Admin Layout: client/src/app/user/dashboard/admin/layout.jsx
Student Dashboard: client/src/app/user/dashboard/student/page.jsx
Admin users see a different dashboard with additional capabilities:
Manage Courses section for creating and editing courses
View Students section for user management
Approve Registrations section for approving user registration requests
The admin dashboard displays admin-specific navigation and functionality, while regular users see a student dashboard with more limited options. This differentiation is based on the user's is_admin status which is in dashboard/page.jsx and redirects the user to the correct dashboard.