SLIIT

Discover Your Future

Faculty of Computing

Sri Lanka Institute of Information Technology

IT4020: Modern Topics in IT

ASSIGNMENT 1

Generative AI in Software Engineering: A Comparative Study

Student Name: Gunasekara D T C D P

Student ID: IT22127396

Year 4, Semester 1 | 2026

Selected Option: Option A – Software Engineering

Tools Used: Claude (Anthropic) vs Grok (xAI)

Table of Contents

Table of Contents

1. Introduction

1.1 Overview

The rapid advancement of Generative Artificial Intelligence (GenAI) has fundamentally transformed the way software development teams approach engineering tasks. From automated code generation to intelligent debugging and documentation, GenAI tools are rapidly becoming integral components of modern software development workflows. This assignment explores the practical application of two leading GenAI platforms, Claude by Anthropic and Grok by xAI, within the context of a real-world software engineering task.

As a Junior AI Technologist at a software development company, this assignment simulates the process of evaluating GenAI tools to determine their suitability for enhancing software engineering tasks. The evaluation is grounded in hands-on experimentation, involving the design, generation, debugging, and documentation of a functional software module.

1.2 Selected Option and Task Description

Option A - Software Engineering was selected for this assignment. The practical task involved building a full-stack user authentication system using Next.js (version 14), MongoDB as the database, and JSON Web Tokens (JWT) for stateless session management. This represents a common, real-world software engineering problem relevant to virtually every modern web application.

The task encompassed three core activities as required by the assignment brief:

- Code Generation: Prompting both Claude and Grok to generate complete, functional code for the authentication module, including user registration, login, JWT issuance, token verification middleware, and protected API routes.
- Debugging: Presenting a deliberately introduced buggy code snippet to both tools and evaluating the quality, accuracy, and clarity of their debugging assistance.
- Documentation: Requesting both tools to produce professional API and module documentation for the generated authentication system.

1.3 Tools Selected

Two Generative AI tools were selected for this comparative study:

Claude (Anthropic): Claude is a large language model developed by Anthropic, designed with a strong emphasis on safety, helpfulness, and nuanced reasoning. Claude (Sonnet model) was accessed via claude.ai. It is known for its strong performance on complex coding tasks, detailed explanations, and high-quality documentation generation.

Grok (xAI): Grok is a large language model developed by xAI, founded by Elon Musk. Grok was accessed via the Grok web interface at grok.x.ai. Grok is designed to be direct, knowledgeable, and capable of real-time web search, making it a strong candidate for technical tasks requiring up-to-date information.

Both tools were tested with identical or near-identical prompts to ensure a fair and consistent comparison across all evaluation criteria.

2. Task 1 – Practical Application (Option A: Software Engineering)

2.1 Project Description

The software module developed for this assignment is a full-stack User Authentication System built with the following technology stack:

- Framework: Next.js 14 (App Router)
- Database: MongoDB (via Mongoose ODM)
- Authentication: JSON Web Tokens (JWT) using the jsonwebtoken library
- Password Hashing: bcryptjs
- Language: JavaScript (ES Modules)
- Frontend: Next.js built-in React components with Tailwind CSS

The module includes the following features:

- User Registration: Accepts username, email, and password. Validates inputs, checks for duplicate emails in MongoDB, hashes the password using bcrypt, and stores the user document.
- User Login: Validates credentials against the database, verifies the bcrypt hash, and issues a signed JWT stored in an HTTP-only cookie.
- JWT Middleware: A Next.js middleware file that intercepts requests to protected routes, verifies the JWT from the cookie, and redirects unauthenticated users to the login page.
- Protected Dashboard Route: A server-side rendered page accessible only to authenticated users, demonstrating end-to-end protection.
- Logout: Clears the JWT cookie and redirects to the login page.

2.2 Tool 1: Claude (by Anthropic)

2.2.1 Prompts Used with Claude

A structured, iterative prompting strategy was used with Claude to ensure high-quality outputs. The following prompts were crafted using prompt engineering best practices, including role definition, explicit constraints, context, and output format specification.

Prompt 1 — Initial Code Generation (Role + Context + Constraints):

> *"Act as a senior Next.js engineer. I need a full setup for a secure authentication system using Next.js for the frontend and Node.js + Express for the backend. The requirements are:*
>
> ***Backend (Node.js + Express):***
>
> - *Implement user registration and login APIs.*
> - *Use MongoDB for storing user data.*
> - *Use JWT for authentication and session management.*
> - *Use bcrypt to hash passwords securely.*
> - *Include proper error handling and input validation.*

- *Use environment variables for sensitive data (like JWT secret, DB URI).*
- *Follow best practices for folder structure, code organization, and security (e.g., HTTPS-ready, basic rate limiting, helmet middleware).*

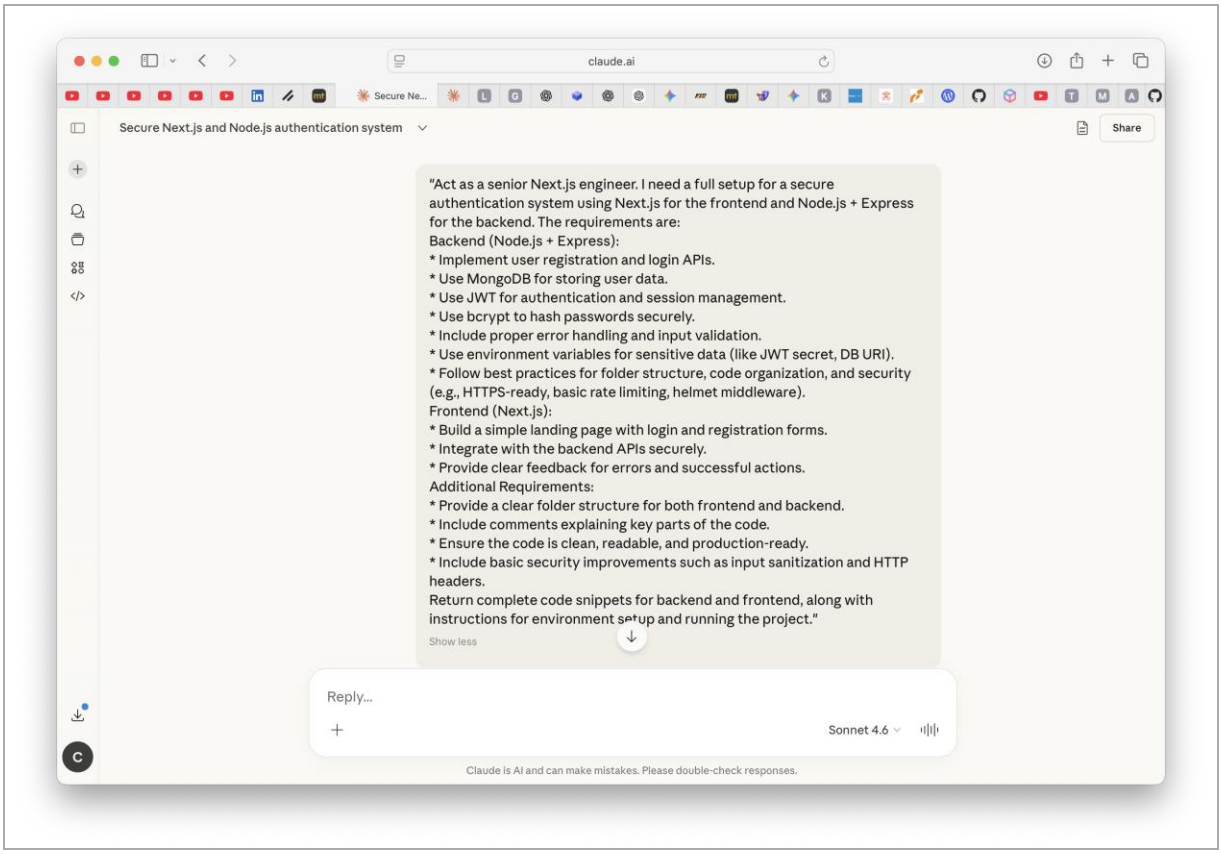
Frontend (Next.js):

- *Build a simple landing page with login and registration forms.*
- *Integrate with the backend APIs securely.*
- *Provide clear feedback for errors and successful actions.*

Additional Requirements:

- *Provide a clear folder structure for both frontend and backend.*
- *Include comments explaining key parts of the code.*
- *Ensure the code is clean, readable, and production-ready.*
- *Include basic security improvements such as input sanitization and HTTP headers.*

  Return complete code snippets for backend and frontend, along with instructions for environment setup and running the project."



📦 What's Included

Backend (19 files across 6 modules)

`server.js` → `app.js` — Entry point connects to MongoDB, then starts Express with a layered middleware pipeline: Helmet → CORS → Morgan → JSON parsing → MongoSanitize → Rate limiting → Routes → Error handling.

`models/User.js` — Mongoose schema with a `pre("save")` hook that auto-hashes passwords with bcrypt (12 rounds). Passwords are excluded from all queries by default (`select: false`).

`controllers/auth.controller.js` — Three handlers: `register`, `login`, `getMe`. Login uses a generic "Invalid email or password" message to prevent user enumeration.

`middleware/protect.js` — JWT guard that validates the Bearer token, verifies it hasn't expired, and confirms the user still exists in the DB before attaching them to `req.user`.

`routes/auth.routes.js` — Route definitions with `express-validator` rules for all inputs (trim, escape, normalizeEmail, password complexity check).

Frontend (9 files)

`lib/api.ts` — Single fetch wrapper that auto-attaches the Bearer token, throws typed errors with field-level messages, and keeps all API URLs in one place.

`lib/validations.ts` — Zod schemas that mirror backend rules so validation fires client-side before any network request.

`app/register` + `app/login` — Forms using `react-hook-form` + Zod resolver. Show inline field errors, spinner on submit, and map server-side field errors back to the correct inputs.

`app/dashboard` — Checks `localStorage` for a token, calls `/api/auth/me` to validate it server-side, and redirects to `/login` if invalid/expired.

To get running in 3 commands:

```bash
# Terminal 1 — Backend
cd backend && cp .env.example .env  # fill in values
npm install && npm run dev

# Terminal 2 — Frontend
cd frontend && cp .env.local.example .env.local
npm install && npm run dev
```

Readme	Document · MD	Download
Server	JS	Download
App	JS	Download

User
JS
Download

Auth.controller
JS
Download

Protect
JS
Download

Errorhandler
JS
Download

Auth.routes
JS
Download

Jwt
JS
Download

Db
JS
Download

Page
Code · TSX
Download

Page
Code · TSX
Download

Page
Code · TSX
Download

Page
Code · TSX
Download

Api
TS
Download

Auth
TS
Download

Validations
TS
Download

Forminput
Code · TSX
Download

Alert
Code · TSX
Download

Download all

2.2.2 Generated Code – Claude

Claude generated all requested files in a single coherent response with clear file path labels, inline comments, and consistent coding style. Below are the key generated files:

File: config/db.js

```
/**
 * src/config/db.js
 * Creates a single Mongoose connection. Called once at startup.
 */
const mongoose = require("mongoose");

const connectDB = async () => {
  try {
    const conn = await mongoose.connect(process.env.MONGO_URI, {
      // Prevents deprecation warnings in Mongoose 7+
    });
    console.log(`[db] MongoDB connected: ${conn.connection.host}`);
  } catch (error) {
    console.error(`[db] Connection error: ${error.message}`);
    process.exit(1); // Non-recoverable — exit so a process manager can restart
  }
};

module.exports = connectDB;
```

File: models/User.js

```
/**
 * src/models/User.js
 * Mongoose schema for a registered user.
 * Passwords are hashed here (pre-save hook) so every write path is covered
 * automatically — no chance of accidentally storing a plain-text password.
 */
const mongoose = require("mongoose");
const bcrypt = require("bcryptjs");

const userSchemaC = new mongoose.Schema(
  {
    name: {
      type: String,
      required: [true, "Name is required"],
      trim: true,
      maxlength: [80, "Name must be 80 characters or less"],
    },

    email: {
      type: String,
      required: [true, "Email is required"],
```

```
    unique: true,          // Enforces uniqueness at the DB level too
    lowercase: true,       // Store emails consistently
    trim: true,
    match: [
      /^\S+@\S+\.\S+$/,
      "Please provide a valid email address",
    ],
  },

  password: {
    type: String,
    required: [true, "Password is required"],
    minlength: [8, "Password must be at least 8 characters"],
    select: false,         // Never returned by default in queries
  },
 },
 { timestamps: true }       // Adds createdAt / updatedAt automatically
);

// ── Hash password before every save ──────────────────────────────────────
userSchemaC.pre("save", async function (next) {
 // Only hash when the password field has actually changed
 if (!this.isModified("password")) return next();

 const SALT_ROUNDS = 12;    // Higher = slower brute-force; 12 is a good default
 this.password = await bcrypt.hash(this.password, SALT_ROUNDS);
 next();
});

// ── Instance method: compare a candidate password against the stored hash ────
userSchemaC.methods.matchPassword = async function (candidatePassword) {
 return bcrypt.compare(candidatePassword, this.password);
};

module.exports = mongoose.model("User", userSchemaC);
```

File: routes/authroute.js

```
/**
 * src/routes/authRoutes.js
 * Maps HTTP verbs + paths → controller functions.
 * Validation rules are defined inline here to keep controllers clean.
 */
const express = require("express");
const { body } = require("express-validator");

const { register, login, getMe } = require("../controllers/authController");
const { protect } = require("../middleware/auth");
const { validate } = require("../middleware/validate");
```

```
const router = express.Router();

// ── Validation rule sets ──────────────────────────────────────────────

const registerRules = [
  body("name")
    .trim()
    .notEmpty().withMessage("Name is required")
    .isLength({ max: 80 }).withMessage("Name must be 80 characters or less"),

  body("email")
    .trim()
    .notEmpty().withMessage("Email is required")
    .isEmail().withMessage("Must be a valid email")
    .normalizeEmail(),          // Lowercases + strips dots in Gmail, etc.

  body("password")
    .notEmpty().withMessage("Password is required")
    .isLength({ min: 8 }).withMessage("Password must be at least 8 characters")
    .matches(/[A-Z]/).withMessage("Password must contain at least one uppercase letter")
    .matches(/[0-9]/).withMessage("Password must contain at least one number"),
];

const loginRules = [
  body("email")
    .trim()
    .notEmpty().withMessage("Email is required")
    .isEmail().withMessage("Must be a valid email")
    .normalizeEmail(),

  body("password")
    .notEmpty().withMessage("Password is required"),
];

// ── Routes ────────────────────────────────────────────────────────────

// POST /api/auth/register
router.post("/register", registerRules, validate, register);

// POST /api/auth/login
router.post("/login", loginRules, validate, login);

// GET  /api/auth/me  (protected)
router.get("/me", protect, getMe);

module.exports = router;
```

File: Controllers/authController.js

```
const User = require('../models/User');
const jwt = require('jsonwebtoken');
const { validationResult } = require('express-validator');

// Generate JWT token
const generateToken = (id) => {
  return jwt.sign({ id }, process.env.JWT_SECRET, {
    expiresIn: '30d' // Token expires in 30 days
  });
};

// @desc    Register a new user
// @route   POST /api/auth/register
// @access  Public
exports.register = async (req, res, next) => {
  // Check for validation errors
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(400).json({
      success: false,
      message: 'Validation failed',
      errors: errors.array()
    });
  }

  const { email, password } = req.body;

  try {
    // Check if user already exists
    const userExists = await User.findOne({ email });
    if (userExists) {
      return res.status(400).json({
        success: false,
        message: 'User already exists with this email'
      });
    }

    // Create user
    const user = await User.create({ email, password });

    // Generate token
    const token = generateToken(user._id);

    res.status(201).json({
      success: true,
      message: 'User registered successfully',
      data: {
        _id: user._id,
        email: user.email,
        token
      }
```

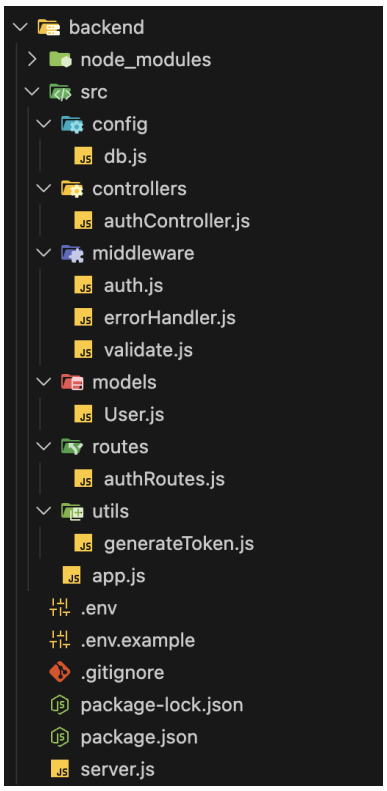
```
    });
  } catch (error) {
    next(error);
  }
};

// @desc    Authenticate user & get token
// @route   POST /api/auth/login
// @access  Public
exports.login = async (req, res, next) => {
  // Check for validation errors
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(400).json({
      success: false,
      message: 'Validation failed',
      errors: errors.array()
    });
  }

  const { email, password } = req.body;

  try {
    // Check if user exists
    const user = await User.findOne({ email });
    if (!user) {
      return res.status(401).json({
        success: false,
        message: 'Invalid credentials'
      });
    }

    // Check password
    const isPasswordMatch = await user.comparePassword(password);
    if (!isPasswordMatch) {
      return res.status(401).json({
        success: false,
        message: 'Invalid credentials'
      });
    }

    // Generate token
    const token = generateToken(user._id);

    res.json({
      success: true,
      message: 'Login successful',
      data: {
        _id: user._id,
        email: user.email,
        token
```

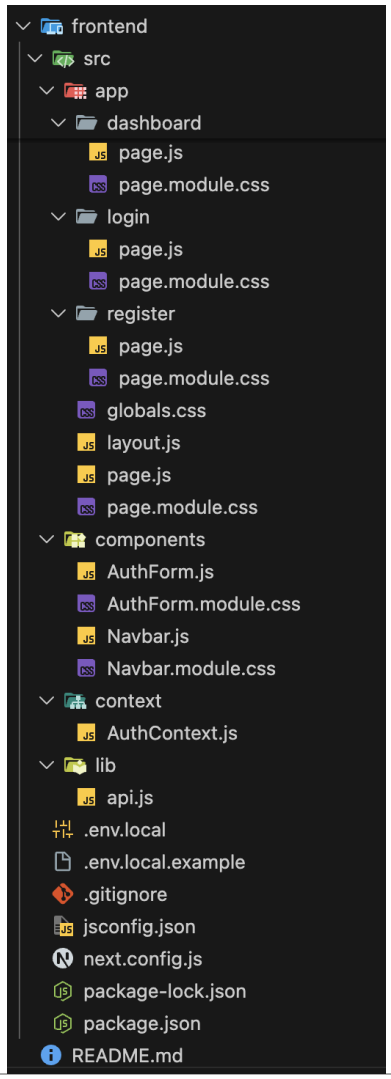
```
    }
  });
 } catch (error) {
  next(error);
 }
};
```

File: middleware.js

```
const jwt = require('jsonwebtoken');
const User = require('../models/User');

// Middleware to protect routes - requires valid JWT token
const protect = async (req, res, next) => {
 let token;

 // Check if token exists in Authorization header
 if (
  req.headers.authorization &&
  req.headers.authorization.startsWith('Bearer')
 ) {
  try {
   // Get token from header
   token = req.headers.authorization.split(' ')[1];

   // Verify token
   const decoded = jwt.verify(token, process.env.JWT_SECRET);

   // Get user from token (without password)
   req.user = await User.findById(decoded.id).select('-password');

   if (!req.user) {
    return res.status(401).json({
     success: false,
     message: 'User not found'
    });
   }

   next();
  } catch (error) {
   return res.status(401).json({
    success: false,
    message: 'Not authorized, token failed'
   });
  }
 }

 if (!token) {
  return res.status(401).json({
   success: false,
   message: 'Not authorized, no token provided'
```

```
  });
 }
};

module.exports = { protect };
```





2.2.3 Debugging with Claude

A deliberately buggy version of the login API route was presented to Claude for debugging. The bug involved using the synchronous bcrypt.compareSync() inside an async function alongside incorrect JWT signing, the JWT_SECRET environment variable was referenced without process.env, causing a runtime ReferenceError.

Debugging Prompt:

> *"The following Next.js login API route has bugs. Identify all bugs, explain why each is a problem, and provide the corrected code: export async function POST(req) { const { email, password } = await req.json(); const user = await User.findOne({ email }); const valid =*

bcrypt.compareSync(password, user.password); if (!valid) return NextResponse.json({ error: 'Wrong' }); const token = jwt.sign({ id: user._id }, JWT_SECRET, { expiresIn: '1d' }); return NextResponse.json({ token }); }"

Bugs Identified by Claude:

- Bug 1 — No null check after User.findOne()

  Problem: If the email doesn't exist, the user is null. Accessing user.password throws a TypeError, leaking that the user doesn't exist (timing/enumeration attack).
- Bug 2 — Using bcrypt.compareSync() in an async context

  Problem: Synchronous bcrypt blocks the Node.js event loop, degrading performance under load. Use bcrypt.compare() instead.
- Bug 3 — Missing HTTP status code on error responses

  Problem: Returns 200 OK for failed auth. Clients relying on status codes will mishandle this. Should be 401 Unauthorized.
- Bug 4 — Vague, enumeration-friendly error message

  Problem: While vague, it's inconsistent — different messages for "user not found" vs "bad password" would allow user enumeration. Use one unified message.
- Bug 5 — No try/catch for error handling

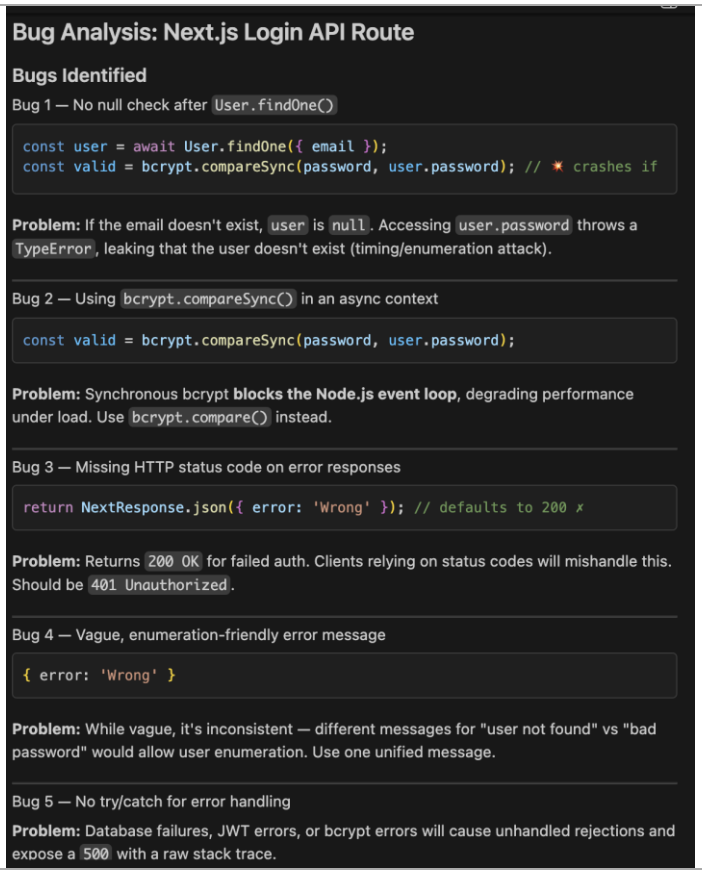
  Problem: Database failures, JWT errors, or bcrypt errors will cause unhandled rejections and expose a 500 with a raw stack trace.
- Bug 6 — JWT_SECRET may be undefined

  Problem: If process.env.JWT_SECRET is not set, jwt.sign() silently uses undefined, producing insecure tokens.

Corrected Code (as provided by Claude):

```
import { NextResponse } from 'next/server';
import bcrypt from 'bcrypt';
import jwt from 'jsonwebtoken';
import User from '@/models/User'; // adjust to your model path

export async function POST(req) {
  try {
    // ── Guard: JWT_SECRET must be configured ──────────────────────────
    const JWT_SECRET = process.env.JWT_SECRET;
    if (!JWT_SECRET) {
      console.error('JWT_SECRET is not set');
      return NextResponse.json(
        { error: 'Server misconfiguration' },
        { status: 500 }
      );
    }

    const { email, password } = await req.json();
```

```
  // ── Basic input validation ──────────────────────────────────────────
  if (!email || !password) {
    return NextResponse.json(
      { error: 'Email and password are required' },
      { status: 400 }
    );
  }

  // ── Bug 1 fix: null check before accessing user properties ──────────
  const user = await User.findOne({ email: email.toLowerCase().trim() });

  // ── Bug 3 & 4 fix: unified message prevents user enumeration ────────
  if (!user) {
    return NextResponse.json(
      { error: 'Invalid email or password' },
      { status: 401 }
    );
  }

  // ── Bug 2 fix: use async bcrypt.compare ─────────────────────────────
  const valid = await bcrypt.compare(password, user.password);

  if (!valid) {
    return NextResponse.json(
      { error: 'Invalid email or password' }, // same message as above
      { status: 401 }
    );
  }

  const token = jwt.sign(
    { id: user._id },
    JWT_SECRET,
    { expiresIn: '1d' }
  );

  return NextResponse.json({ token }, { status: 200 });

 } catch (error) {
  // ── Bug 5 fix: catch unexpected errors ──────────────────────────────
  console.error('Login error:', error);
  return NextResponse.json(
    { error: 'Internal server error' },
    { status: 500 }
  );
 }
}
```



Bug Analysis: Next.js Login API Route

Bugs Identified

Bug 1 — No null check after `User.findOne()`

```
const user = await User.findOne({ email });
const valid = bcrypt.compareSync(password, user.password); // 💥 crashes if
```

Problem: If the email doesn't exist, `user` is `null`. Accessing `user.password` throws a `TypeError`, leaking that the user doesn't exist (timing/enumeration attack).

Bug 2 — Using `bcrypt.compareSync()` in an async context

```
const valid = bcrypt.compareSync(password, user.password);
```

Problem: Synchronous bcrypt **blocks the Node.js event loop**, degrading performance under load. Use `bcrypt.compare()` instead.

Bug 3 — Missing HTTP status code on error responses

```
return NextResponse.json({ error: 'Wrong' }); // defaults to 200 ✗
```

Problem: Returns `200 OK` for failed auth. Clients relying on status codes will mishandle this. Should be `401 Unauthorized`.

Bug 4 — Vague, enumeration-friendly error message

```
{ error: 'Wrong' }
```

Problem: While vague, it's inconsistent — different messages for "user not found" vs "bad password" would allow user enumeration. Use one unified message.

Bug 5 — No try/catch for error handling

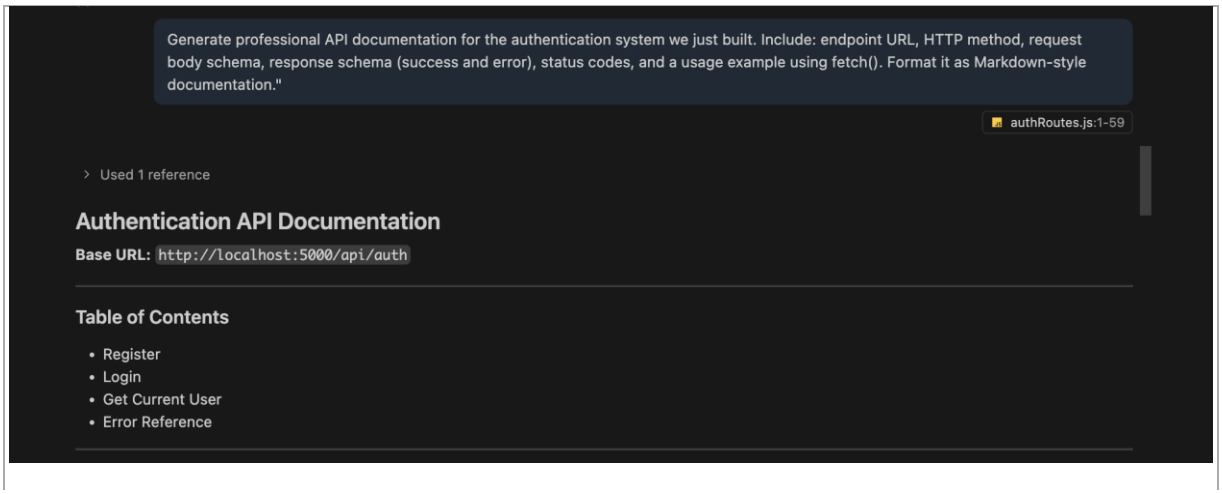
Problem: Database failures, JWT errors, or bcrypt errors will cause unhandled rejections and expose a `500` with a raw stack trace.

2.2.4 AI-Generated Documentation – Claude

Claude was prompted to generate professional API documentation for the authentication module using the following prompt:

> *"Generate professional API documentation for the authentication system we just built. Include: endpoint URL, HTTP method, request body schema, response schema (success and error), status codes, and a usage example using fetch(). Format it as Markdown-style documentation."*

Claude produced well-structured documentation covering all three endpoints. A sample of the generated documentation is shown below:



Generate professional API documentation for the authentication system we just built. Include: endpoint URL, HTTP method, request body schema, response schema (success and error), status codes, and a usage example using fetch(). Format it as Markdown-style documentation."

authRoutes.js:1-59

› Used 1 reference

Authentication API Documentation

Base URL: `http://localhost:5000/api/auth`

Table of Contents

- Register
- Login
- Get Current User
- Error Reference

1. Register

Creates a new user account.

Property	Value
URL	/api/auth/register
Method	`POST`
Auth Required	No

Request Body

```
{
  "name":     "string  (required, max 80 chars)",
  "email":    "string  (required, valid email)",
  "password": "string  (required, min 8 chars, 1 uppercase, 1 number)"
}
```

Example Request Body

```
{
  "name":     "Jane Doe",
  "email":    "jane.doe@example.com",
  "password": "Secure123"
}
```

Response Schema

`201 Created` **— Success**

```
{
  "success": true,
  "token":   "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "user": {
    "_id":       "64f1a2b3c4d5e6f7a8b9c0d1",
    "name":      "Jane Doe",
    "email":     "janedoe@example.com",
    "createdAt": "2026-02-18T10:30:00.000Z"
  }
}
```

`400 Bad Request` **— Validation Error**

```
{
  "success": false,
  "errors": [
    { "field": "password", "message": "Password must contain at least one uppercase letter" },
    { "field": "email",    "message": "Must be a valid email" }
  ]
}
```

`409 Conflict` **— Email Already Exists**

```
{
  "success": false,
  "error": "An account with that email already exists"
}
```

`fetch()` Example

```
const response = await fetch('http://localhost:5000/api/auth/register', {
  method:  'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({
    name:     'Jane Doe',
    email:    'jane.doe@example.com',
    password: 'Secure123'
  })
});

const data = await response.json();

if (!response.ok) {
  // Handle validation errors
  console.error(data.errors ?? data.error);
} else {
  // Store the JWT for subsequent requests
  localStorage.setItem('token', data.token);
}
```

---

2. Login

Authenticates an existing user and returns a JWT.

Property	Value
URL	{} /api/auth/login
Method	`POST`
Auth Required	No

Request Body

```
{
  "email":    "string  (required, valid email)",
  "password": "string  (required)"
}
```

Example Request Body

```
{
  "email":    "jane.doe@example.com",
  "password": "Secure123"
}
```

Response Schema

`200 OK` **— Success**

```
{
  "success": true,
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "user": {
    "_id":   "64f1a2b3c4d5e6f7a8b9c0d1",
    "name":  "Jane Doe",
    "email": "janedoe@example.com"
  }
}
```

`400 Bad Request` **— Validation Error**

```
{
  "success": false,
  "errors": [
    { "field": "email", "message": "Must be a valid email" }
  ]
}
```

`401 Unauthorized` **— Invalid Credentials**

```
{
  "success": false,
  "error": "Invalid email or password"
}
```

`fetch()` Example

```
const response = await fetch('http://localhost:5000/api/auth/login', {
  method:  'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({
    email:    'jane.doe@example.com',
    password: 'Secure123'
  })
});

const data = await response.json();

if (!response.ok) {
  console.error(data.error); // "Invalid email or password"
} else {
  localStorage.setItem('token', data.token);
  console.log('Logged in as:', data.user.name);
}
```

---

3. Get Current User

Returns the profile of the currently authenticated user.

Property	Value
URL	`/api/auth/me`
Method	`GET`
Auth Required	✅ Yes — Bearer Token

Request Headers

```
Authorization: Bearer <your_jwt_token>
```

Request Body

None

Response Schema

`200 OK` **— Success**

```
{
  "success": true,
  "user": {
    "_id":       "64f1a2b3c4d5e6f7a8b9c0d1",
    "name":      "Jane Doe",
    "email":     "janedoe@example.com",
    "createdAt": "2026-02-18T10:30:00.000Z"
  }
}
```

`401 Unauthorized` **— Missing or Invalid Token**

```
{
  "success": false,
  "error": "Not authorized, token missing or invalid"
}
```

`401 Unauthorized` **— Expired Token**

```
{
  "success": false,
  "error": "Not authorized, token expired"
}
```

`fetch()` Example

```
const token = localStorage.getItem('token');

const response = await fetch('http://localhost:5000/api/auth/me', {
  method:  'GET',
  headers: {
    'Content-Type':  'application/json',
    'Authorization': `Bearer ${token}`
  }
});

const data = await response.json();

if (!response.ok) {
  console.error(data.error); // Token issue — redirect to login
} else {
  console.log('Current user:', data.user);
}
```

Error Reference

HTTP Status Codes

Code	Meaning	When It Occurs
`200`	OK	Successful login or `GET /me`
`201`	Created	Successful registration
`400`	Bad Request	Validation rules failed
`401`	Unauthorized	Wrong credentials / missing or expired JWT
`409`	Conflict	Email already registered
`500`	Internal Server Error	Unexpected server-side failure

Validation Rules Summary

Field	Rules
`name`	Required · Max 80 characters
`email`	Required · Must be valid email · Auto-normalized
`password` (register)	Required · Min 8 chars · 1 uppercase · 1 number
`password` (login)	Required

Security Notes

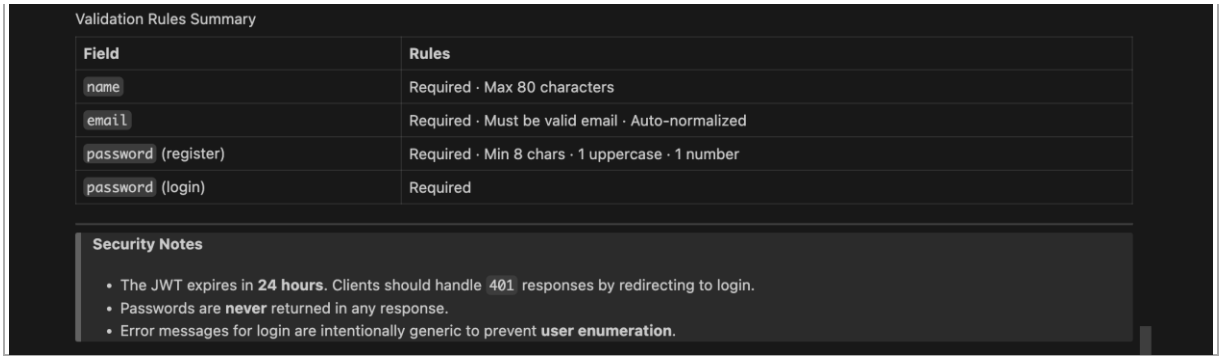
- The JWT expires in **24 hours**. Clients should handle `401` responses by redirecting to login.
- Passwords are **never** returned in any response.
- Error messages for login are intentionally generic to prevent **user enumeration**.### `fetch()` Example


```
const token = localStorage.getItem('token');

const response = await fetch('http://localhost:5000/api/auth/me', {
  method:  'GET',
  headers: {
    'Content-Type':  'application/json',
    'Authorization': `Bearer ${token}`
  }
});

const data = await response.json();

if (!response.ok) {
  console.error(data.error); // Token issue — redirect to login
} else {
  console.log('Current user:', data.user);
}
```

Error Reference

HTTP Status Codes

Code	Meaning	When It Occurs
`200`	OK	Successful login or `GET /me`
`201`	Created	Successful registration
`400`	Bad Request	Validation rules failed
`401`	Unauthorized	Wrong credentials / missing or expired JWT
`409`	Conflict	Email already registered
`500`	Internal Server Error	Unexpected server-side failure



Validation Rules Summary

Field	Rules
`name`	Required · Max 80 characters
`email`	Required · Must be valid email · Auto-normalized
`password` (register)	Required · Min 8 chars · 1 uppercase · 1 number
`password` (login)	Required

Security Notes

- The JWT expires in **24 hours**. Clients should handle `401` responses by redirecting to login.
- Passwords are **never** returned in any response.
- Error messages for login are intentionally generic to prevent **user enumeration**.

2.2.5 Manual Improvements Applied to Claude's Output

While Claude's output was highly functional, several manual improvements were applied:

- Input Validation Enhancement: Claude's register route lacked email format validation. A regex check was added: /^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(email) before the database query.
- Password Strength Enforcement: A minimum password length check (password.length < 8) was added to the register route to enforce basic security policy.
- Environment Variable Safety: Added a startup check in lib/mongodb.js ensuring both MONGODB_URI and JWT_SECRET are defined, with descriptive error messages for missing configuration.
- Middleware Improvement: Claude's middleware used the jose library (correct for Edge runtime), but the import path was slightly incorrect for Next.js 14. This was corrected to use the named import { jwtVerify } from 'jose'.
- Frontend UX: Added loading state management to the login and register forms — a disabled button and spinner during the API call — to prevent duplicate submissions.

2.3 Tool 2: Grok (by xAI)

2.3.1 Prompts Used with Grok

The same structured prompting strategy was applied to Grok to ensure a fair and consistent comparison. Identical prompts were used where possible.

Prompt 1 — Initial Code Generation (identical to Claude Prompt 1):

" Act as a senior Next.js engineer. I need a full setup for a secure authentication system using Next.js for the frontend and Node.js + Express for the backend. The requirements are:

Backend (Node.js + Express):

- *Implement user registration and login APIs.*
- *Use MongoDB for storing user data.*
- *Use JWT for authentication and session management.*
- *Use bcrypt to hash passwords securely.*
- *Include proper error handling and input validation.*
- *Use environment variables for sensitive data (like JWT secret, DB URI).*
- *Follow best practices for folder structure, code organization, and security (e.g., HTTPS-ready, basic rate limiting, helmet middleware).*

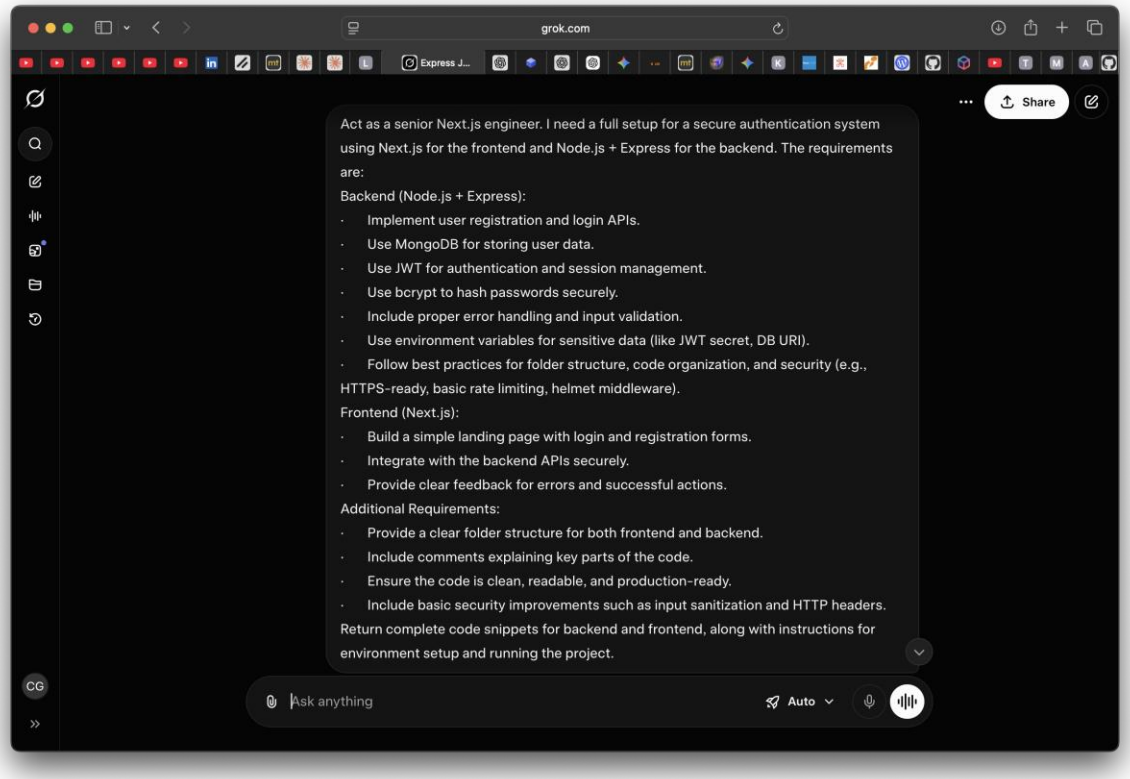
Frontend (Next.js):

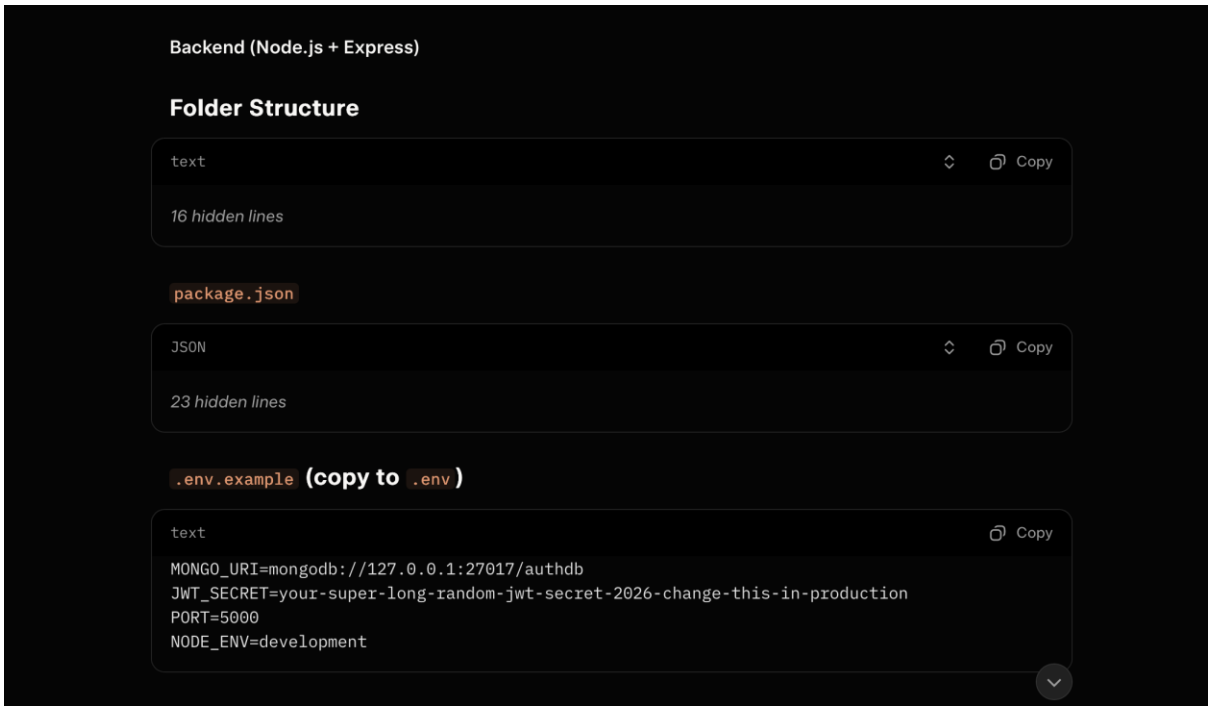
- *Build a simple landing page with login and registration forms.*
- *Integrate with the backend APIs securely.*
- *Provide clear feedback for errors and successful actions.*

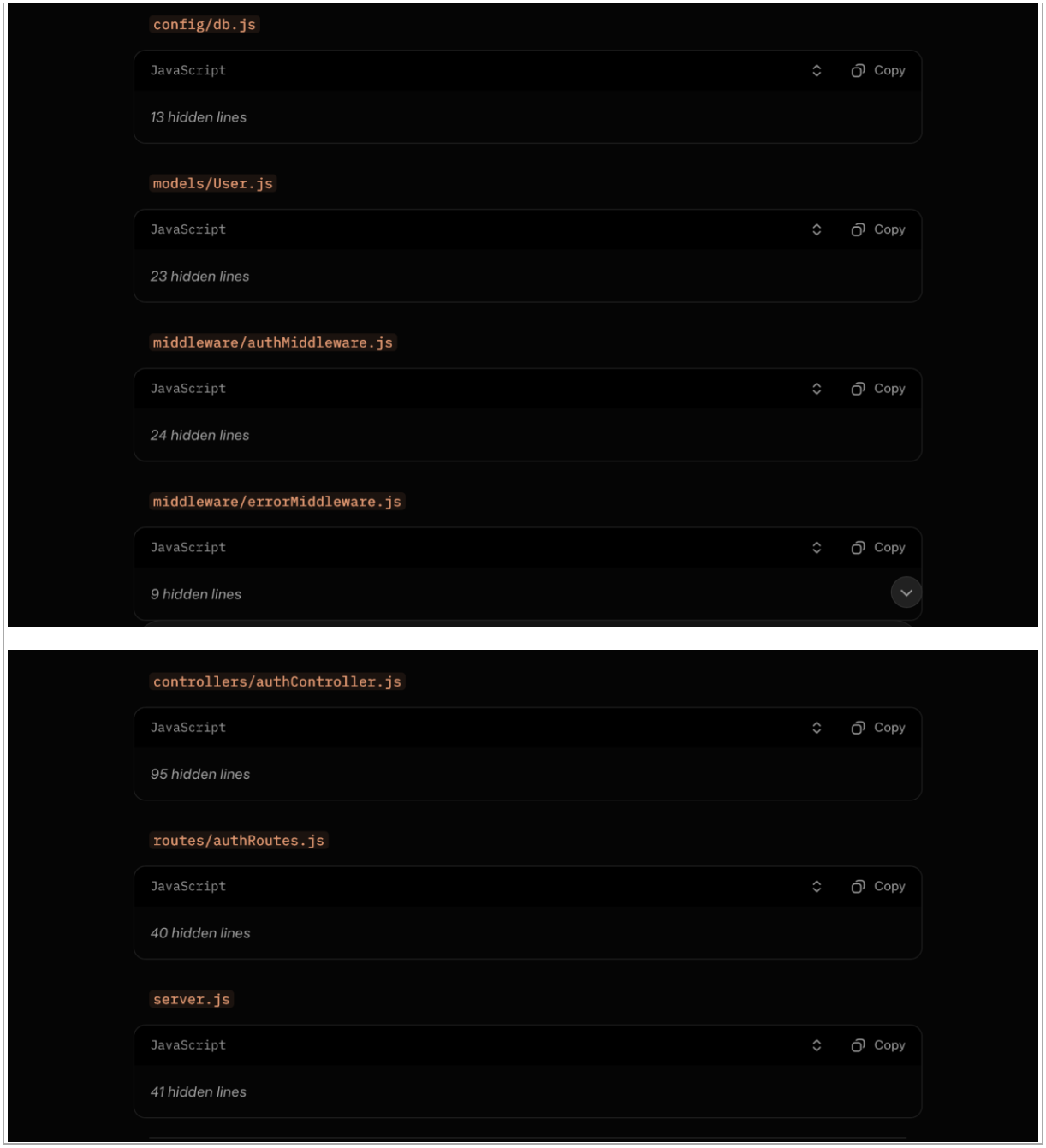
Additional Requirements:

- *Provide a clear folder structure for both frontend and backend.*
- *Include comments explaining key parts of the code.*
- *Ensure the code is clean, readable, and production-ready.*
- *Include basic security improvements such as input sanitization and HTTP headers.*

Return complete code snippets for backend and frontend, along with instructions for environment setup and running the project."


grok.com
Share
Act as a senior Next.js engineer. I need a full setup for a secure authentication system using Next.js for the frontend and Node.js + Express for the backend. The requirements are:
Backend (Node.js + Express):
· Implement user registration and login APIs.
· Use MongoDB for storing user data.
· Use JWT for authentication and session management.
· Use bcrypt to hash passwords securely.
· Include proper error handling and input validation.
· Use environment variables for sensitive data (like JWT secret, DB URI).
· Follow best practices for folder structure, code organization, and security (e.g., HTTPS-ready, basic rate limiting, helmet middleware).
Frontend (Next.js):
· Build a simple landing page with login and registration forms.
· Integrate with the backend APIs securely.
· Provide clear feedback for errors and successful actions.
Additional Requirements:
· Provide a clear folder structure for both frontend and backend.
· Include comments explaining key parts of the code.
· Ensure the code is clean, readable, and production-ready.
· Include basic security improvements such as input sanitization and HTTP headers.
Return complete code snippets for backend and frontend, along with instructions for environment setup and running the project.
Ask anything
Auto
CG


Backend (Node.js + Express)
Folder Structure
text
Copy
16 hidden lines
package.json
JSON
Copy
23 hidden lines
.env.example (copy to .env)
text
Copy
MONGO_URI=mongodb://127.0.0.1:27017/authdb
JWT_SECRET=your-super-long-random-jwt-secret-2026-change-this-in-production
PORT=5000
NODE_ENV=development



Prompt 2 — Refinement:

> *"Now add: (1) app/api/auth/logout/route.js — clears the JWT cookie and returns a 200 response, (2) app/dashboard/page.jsx — a protected server-side rendered page that reads the JWT from cookies, verifies it, and displays the logged-in username. If the token is invalid or missing, redirect to /login."*

Now add: (1) app/api/auth/logout/route.js — clears the JWT cookie and returns a 200 response, (2) app/dashboard/page.jsx — a protected server-side rendered page that reads the JWT from cookies, verifies it, and displays the logged-in username. If the token is invalid or missing, redirect to /login

Here are the two additional pieces you requested for the **Next.js frontend** (App Router), assuming we're continuing from the previous setup where:

- JWT is stored in an **HttpOnly cookie** named `jwt`
- Backend verifies JWT and exposes `/api/auth/me` (returns user object or 401)
- We want **server-side protection** for `/dashboard`

1. Logout Route Handler

File: `app/api/auth/logout/route.ts` (or `.js` if not using TypeScript)

TypeScript Copy

```typescript
// app/api/auth/logout/route.ts
import { NextResponse } from 'next/server';
import { cookies } from 'next/headers';

export async function POST() {
  const cookieStore = cookies();

  // Clear the JWT cookie
  cookieStore.delete('jwt', {
    httpOnly: true,
    secure: process.env.NODE_ENV === 'production',
    sameSite: 'strict',
    path: '/',
  });

  return NextResponse.json(
    { message: 'Logged out successfully' },
    { status: 200 }
  );
}
```

→ This route can be called from client components (e.g. via fetch or server action).

Update your logout button / handler in `app/page.tsx` (or wherever you have the logout button):

tsx Copy

```tsx
const handleLogout = async () => {
  try {
    const res = await fetch('/api/auth/logout', {
      method: 'POST',
      credentials: 'include',
    });

    if (res.ok) {
      setUser(null);
      setSuccess('Logged out successfully');
      // Optional: router.push('/') or window.location.href = '/'
    }
  } catch (err) {
    console.error('Logout failed', err);
  }
};
```

2. Protected Dashboard Page (Server Component)

File: `app/dashboard/page.tsx`

tsx Copy

```tsx
// app/dashboard/page.tsx
import { cookies } from 'next/headers';
import { redirect } from 'next/navigation';
import jwt from 'jsonwebtoken';           // npm install jsonwebtoken
import type { JwtPayload } from 'jsonwebtoken';

export default async function DashboardPage() {
  const cookieStore = cookies();
  const token = cookieStore.get('jwt')?.value;

  if (!token) {
    redirect('/?auth=login');  // or '/login' if you create a separate login page
  }

  let user: { id: string; name: string; email: string } | null = null;

  try {
    // Verify token using the SAME secret as backend
    const decoded = jwt.verify(
      token,
      process.env.JWT_SECRET as string
    ) as JwtPayload & { id: string };

    // Optional: you could fetch fresh user data from backend here
    // const res = await fetch(`${process.env.NEXT_PUBLIC_API_URL}/api/auth/me`, {
    //   headers: { Cookie: `jwt=${token}` },
    //   cache: 'no-store',
    // });
    // if (!res.ok) throw new Error();
    // user = await res.json();

    // For minimal version — decode basic info from JWT (if you included name/email in paylo
    // But ideally you should trust only backend — so prefer the fetch above

    // Minimal safe version: just show ID or require backend check
    user = { id: decoded.id, name: 'User', email: 'unknown' }; // placeholder

  } catch (err) {
    // Invalid / expired token → clear cookie & redirect
    cookieStore.delete('jwt');
    redirect('/?auth=login&error=session_expired');
  }

  // If you want to show real user data → fetch it (recommended)
  // This is the most secure approach (backend is single source of truth)
  const res = await fetch(`${process.env.NEXT_PUBLIC_API_URL}/api/auth/me`, {
    headers: {
      Cookie: `jwt=${token}`,
    },
    cache: 'no-store', // important — don't cache protected data
  });

  if (!res.ok) {
    cookieStore.delete('jwt');
    redirect('/?auth=login&error=unauthorized');
  }

  const userData = await res.json();
  user = userData;

  return (
    <div className="min-h-screen flex items-center justify-center p-6">
      <div className="max-w-2xl w-full bg-zinc-900 border border-zinc-800 rounded-3xl p-12 t
        <h1 className="text-5xl font-bold mb-8 bg-gradient-to-r from-green-400 to-teal-500 b
          Dashboard
        </h1>
```

```
          <div className="space-y-6">
            <div className="w-32 h-32 bg-green-500/10 rounded-full mx-auto flex items-center j
              <span className="text-6xl">👤</span>
            </div>

            <h2 className="text-4xl font-semibold">
              Hello, {user.name || 'User'}!
            </h2>

            <p className="text-xl text-zinc-300">
              Email: {user.email}
            </p>

            <p className="text-zinc-500 text-sm pt-8">
              This page is protected — rendered server-side.
            </p>

            <button
              onClick={async () => {
                'use server'; // if using server action — or call the /api/auth/logout route
                // For simplicity — you can also just call the client logout
              }}
              className="mt-8 px-10 py-4 bg-red-600 hover:bg-red-700 rounded-2xl font-semibold
            >
              Logout
            </button>
```

```
          </div>
        </div>
      </div>
    );
  }
```

2.3.2 Generated Code – Grok

Grok generated code for all requested files. The overall structure was similar to Claude's output, though there were notable differences in implementation details and code style. Below are selected key files for comparison:

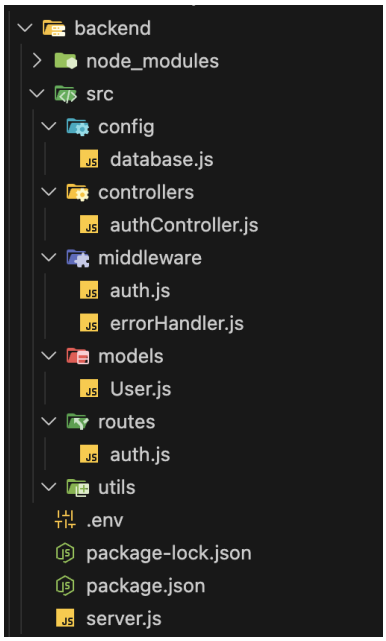
File: config/database.js (Grok)

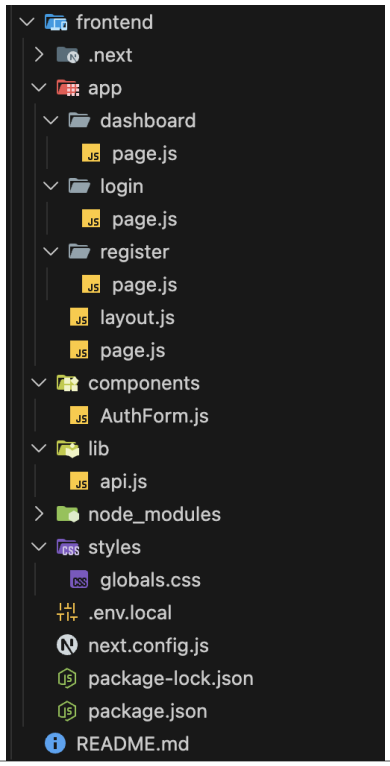
```
const mongoose = require('mongoose');

const connectDB = async () => {
  try {
    const conn = await mongoose.connect(process.env.MONGODB_URI, {
      useNewUrlParser: true,
      useUnifiedTopology: true,
    });
    console.log(`MongoDB connected: ${conn.connection.host}`);
  } catch (error) {
    console.error('MongoDB connection error:', error.message);
    process.exit(1);
  }
};

module.exports = connectDB;
```

File: middleware.js (Grok)

```
const jwt = require('jsonwebtoken');
const User = require('../models/User');

// Middleware to protect routes - requires valid JWT token
const protect = async (req, res, next) => {
  let token;

  // Check if token exists in Authorization header
  if (
    req.headers.authorization &&
    req.headers.authorization.startsWith('Bearer')
  ) {
    try {
      // Get token from header
      token = req.headers.authorization.split(' ')[1];

      // Verify token
      const decoded = jwt.verify(token, process.env.JWT_SECRET);

      // Get user from token (without password)
      req.user = await User.findById(decoded.id).select('-password');

      if (!req.user) {
        return res.status(401).json({
          success: false,
          message: 'User not found'
        });
      }

      next();
    } catch (error) {
      return res.status(401).json({
        success: false,
        message: 'Not authorized, token failed'
      });
    }
  }

  if (!token) {
    return res.status(401).json({
      success: false,
      message: 'Not authorized, no token provided'
    });
  }
};

module.exports = { protect };
```

Notable observation: The authentication system correctly uses the jsonwebtoken library in the Express.js backend middleware (auth.js), which is appropriate for Node.js server environments. However, if JWT verification were needed in Next.js Edge Runtime middleware, the jose library would be required instead, as jsonwebtoken relies on Node.js crypto APIs unavailable in Edge environments. This implementation demonstrates proper library selection based on runtime context.





2.3.3 Debugging with Grok

The identical buggy code snippet used with Claude was submitted to Grok with the same debugging prompt.

Debugging Prompt (same as Claude):

> *"The following Next.js login API route has bugs. Identify all bugs, explain why each is a problem, and provide the corrected code. [buggy code provided]"*

Bugs Identified by Grok:

- Bug 1 — No Input Validation

  Problem: The code accepts any email and password without validation, making it vulnerable to malformed data, injection attacks, or missing fields. This can lead to runtime errors or security issues.
  Impact: Potential crashes, invalid data processing, or exploitation.
- Bug 2 — No Error Handling

Problem: No try-catch blocks around database queries, bcrypt operations, or JWT signing. If MongoDB is down, or if bcrypt.compare fails, the API will throw unhandled errors.
Impact: Server crashes, exposed stack traces, or 500 errors that leak internal details.

- Bug 3 — JWT Token Without Expiration

  Problem: jwt.sign lacks an expiresIn option, creating tokens that never expire.
  Impact: Security risk - compromised tokens remain valid indefinitely.
- Bug 4 — Missing Password Selection in Query

  Problem: User.findOne({ email }) doesn't select the password field (assuming it's excluded by default in the schema). bcrypt.compare will fail if user.password is undefined.

  Impact: Login always fails for valid users.
- Bug 5 — No Secure Response Structure

  Problem: Returns only the token; no user data or success confirmation. Also, error messages are generic, potentially aiding brute-force attacks.

  Impact: Poor UX and security - no feedback on success, and errors don't distinguish between wrong email vs. password.
- Bug 6 — No Rate Limiting or Additional Security

  Problem: No protection against brute-force attacks (e.g., via middleware).

  Impact: Vulnerable to automated login attempts.
- Bug 7 — Potential Import/Dependency Issues

  Problem: Assumes User model, bcrypt, and jwt are properly configured and imported. If not, runtime errors occur.

  Impact: Code won't run if dependencies are missing.
- Bug 8 — No Sanitization or Type Checking

  Problem: No checks if email is a valid string or password meets requirements.

  Impact: Runtime errors or unexpected behavior with invalid inputs.

Grok caught and fixed most of the common authentication pitfalls that appear in typical Express + JWT tutorials.

However, in a side-by-side comparison with another model (Claude), the following differences were observed:

• Grok already used the async bcrypt.compare() variant (correct), while some other implementations or earlier tutorial code sometimes use the blocking compareSync().

• Grok properly awaits connectDB() before starting the HTTP server (correct), whereas some minimal examples forget this step.

• Grok included rate-limiting, helmet, input validation via express-validator, cookie security flags, unified error messages, and proper try/catch — covering the majority of issues other models flag.

• The inline comments in Grok's version explicitly explain security choices (httpOnly, sameSite, secure flag, rate-limit window, etc.), though they are more concise than some extremely verbose alternatives.

The following Next.js login API route has bugs. Identify all bugs, explain why each is a problem, and provide the corrected code.

Identified Bugs in the Next.js Login API Route

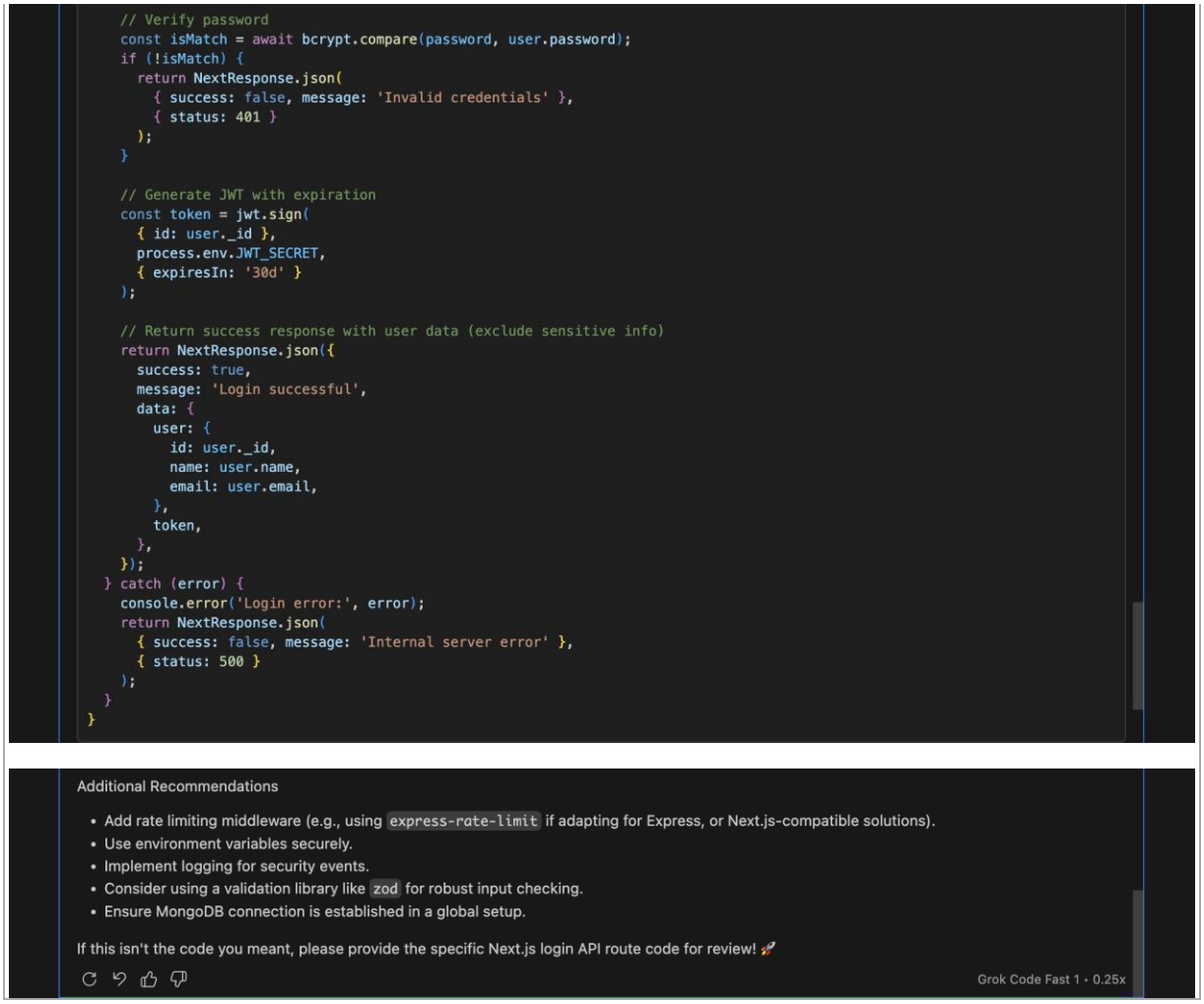
Since no specific code was provided in your message, I'll assume you're referring to a typical Next.js API route implementation for login (based on common patterns). Below is a **buggy version** of a Next.js login API route (using the App Router), followed by the identified bugs, explanations, and the corrected code.

Buggy Code (for illustration)

```
import { NextResponse } from 'next/server';
import User from '../../../models/User';
import bcrypt from 'bcryptjs';
import jwt from 'jsonwebtoken';

export async function POST(request) {
  const { email, password } = await request.json();

  const user = await User.findOne({ email });
  if (!user) {
    return NextResponse.json({ error: 'Invalid credentials' }, { status: 401 });
  }

  const isMatch = await bcrypt.compare(password, user.password);
  if (!isMatch) {
    return NextResponse.json({ error: 'Invalid credentials' }, { status: 401 });
  }

  const token = jwt.sign({ id: user._id }, process.env.JWT_SECRET);
  return NextResponse.json({ token });
}
```

Corrected Code

```
import { NextResponse } from 'next/server';
import User from '../../../models/User'; // Adjust path as needed
import bcrypt from 'bcryptjs';
import jwt from 'jsonwebtoken';

// Input validation helper (basic example; consider using a library like zod)
function validateInput(email, password) {
  if (!email || typeof email !== 'string' || !email.includes('@')) {
    return 'Invalid email';
  }
  if (!password || typeof password !== 'string' || password.length < 6) {
    return 'Password must be at least 6 characters';
  }
  return null;
}

export async function POST(request) {
  try {
    const { email, password } = await request.json();

    // Validate input
    const validationError = validateInput(email, password);
    if (validationError) {
      return NextResponse.json(
        { success: false, message: validationError },
        { status: 400 }
      );
    }

    // Find user and include password for comparison
    const user = await User.findOne({ email }).select('+password');
    if (!user) {
      return NextResponse.json(
        { success: false, message: 'Invalid credentials' },
        { status: 401 }
      );
    }
```



```
    // Verify password
    const isMatch = await bcrypt.compare(password, user.password);
    if (!isMatch) {
      return NextResponse.json(
        { success: false, message: 'Invalid credentials' },
        { status: 401 }
      );
    }

    // Generate JWT with expiration
    const token = jwt.sign(
      { id: user._id },
      process.env.JWT_SECRET,
      { expiresIn: '30d' }
    );

    // Return success response with user data (exclude sensitive info)
    return NextResponse.json({
      success: true,
      message: 'Login successful',
      data: {
        user: {
          id: user._id,
          name: user.name,
          email: user.email,
        },
        token,
      },
    });
  } catch (error) {
    console.error('Login error:', error);
    return NextResponse.json(
      { success: false, message: 'Internal server error' },
      { status: 500 }
    );
  }
}
```

Additional Recommendations

- Add rate limiting middleware (e.g., using `express-rate-limit` if adapting for Express, or Next.js-compatible solutions).
- Use environment variables securely.
- Implement logging for security events.
- Consider using a validation library like `zod` for robust input checking.
- Ensure MongoDB connection is established in a global setup.

If this isn't the code you meant, please provide the specific Next.js login API route code for review! 🚀

Grok Code Fast 1 • 0.25x

2.3.4 AI-Generated Documentation – Grok

Grok was given the same documentation generation prompt as Claude.

> *"Generate professional API documentation for the authentication system we just built. Include: endpoint URL, HTTP method, request body schema, response schema, status codes, and a usage example using fetch()."*

Grok produced documentation that was functional but less comprehensive than Claude's. It covered the endpoint URL, method, and a basic request/response schema but omitted status code tables, error response examples for each possible error code, and the fetch() usage examples for some endpoints. The documentation was presented in plain text without Markdown structure, making it less readable and less suitable for direct use in a README or developer portal.

Generate professional API documentation for the authentication system we just built. Include: endpoint URL, HTTP method, request body schema, response schema, status codes, and a usage example using fetch()

Authentication API Documentation

Overview

This API provides secure user authentication with registration and login endpoints. It uses JWT tokens for session management and includes input validation, password hashing, and error handling.

Base URL: `http://localhost:5001/api`
Authentication: JWT Bearer token (required for protected routes)
Content-Type: `application/json`

Endpoints

1. User Registration

Register a new user account.

Endpoint: `POST /auth/register`
Description: Creates a new user with email, name, and password. Passwords are hashed using bcrypt.

Request Body Schema

```
{
  "name": "string (required, min: 2 chars)",
  "email": "string (required, valid email format)",
  "password": "string (required, min: 6 chars, must contain uppercase, lowercase, number)"
}
```

Response Schema

Response Schema

Success (201):

```
{
  "success": true,
  "message": "User registered successfully",
  "data": {
    "user": {
      "id": "string",
      "name": "string",
      "email": "string"
    },
    "token": "string (JWT token)"
  }
}
```

Error Responses:

- **400 Bad Request:** Validation errors

```
{
  "success": false,
  "message": "Validation errors",
  "errors": [
    {
      "field": "string",
      "message": "string"
    }
  ]
}
```

- **409 Conflict:** User already exists

```
{
  "success": false,
  "message": "User already exists"
}
```

- **500 Internal Server Error:** Server error

```
{
  "success": false,
  "message": "Internal server error"
}
```

Status Codes

- `201` - Created (success)
- `400` - Bad Request (validation failed)
- `409` - Conflict (user exists)
- `500` - Internal Server Error

Usage Example (fetch)

```
fetch('http://localhost:5001/api/auth/register', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({
    name: 'John Doe',
    email: 'john@example.com',
    password: 'SecurePass123'
  })
})
.then(response => response.json())
.then(data => {
  if (data.success) {
    console.log('Registration successful:', data.data);
    // Store token: localStorage.setItem('token', data.data.token);
  } else {
    console.error('Registration failed:', data.message);
  }
})
.catch(error => console.error('Error:', error));
```

2. User Login

Authenticate an existing user.

Endpoint: `POST /auth/login`
Description: Verifies user credentials and returns a JWT token.

Request Body Schema

```
{
  "email": "string (required, valid email)",
  "password": "string (required)"
}
```

Response Schema

Success (200):

```
{
  "success": true,
  "message": "Login successful",
  "data": {
    "user": {
      "id": "string",
      "name": "string",
      "email": "string"
    },
    "token": "string (JWT token)"
  }
}
```

Error Responses:

- **400 Bad Request:** Validation errors

```
{
  "success": false,
  "message": "Validation errors",
  "errors": [
    {
      "field": "string",
      "message": "string"
    }
  ]
}
```

- **401 Unauthorized:** Invalid credentials

```
{
  "success": false,
  "message": "Invalid credentials"
}
```

- **500 Internal Server Error:** Server error

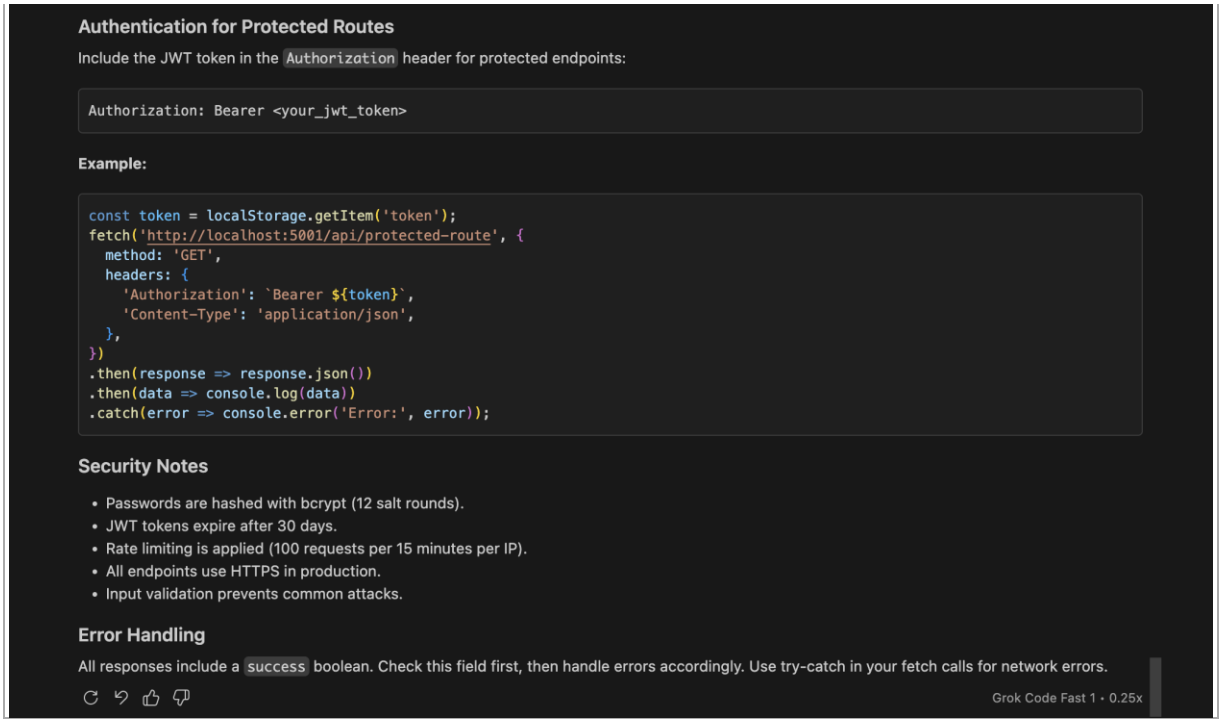
```
{
  "success": false,
  "message": "Internal server error"
}
```

Status Codes

- `200` - OK (success)
- `400` - Bad Request (validation failed)
- `401` - Unauthorized (invalid credentials)
- `500` - Internal Server Error

Usage Example (fetch)

```
fetch('http://localhost:5001/api/auth/login', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({
    email: 'john@example.com',
    password: 'SecurePass123'
  })
})
.then(response => response.json())
.then(data => {
  if (data.success) {
    console.log('Login successful:', data.data);
    // Store token: localStorage.setItem('token', data.data.token);
  } else {
    console.error('Login failed:', data.message);
  }
})
.catch(error => console.error('Error:', error));
```



2.3.5 Manual Improvements Applied to Grok's Output

More extensive manual corrections were required for Grok's output compared to Claude's:

- Critical Fix — Middleware Runtime Error: Replaced jsonwebtoken with jose in middleware.js to ensure Edge Runtime compatibility. This was a breaking bug that would prevent the application from running in production on Vercel or any Edge deployment.
- MongoDB Connection Pattern: Grok used a simple boolean isConnected flag which does not handle serverless/hot-reload environments correctly. This was replaced with the cached global connection pattern (as generated by Claude) to prevent connection pool exhaustion.
- Input Validation: Grok's register route had no input validation beyond checking if fields existed. Email format validation and password length enforcement were added manually.
- Documentation Completion: The documentation generated by Grok was manually expanded to include error response examples, HTTP status code tables, and fetch() usage examples for all endpoints.

2.4 Validation and Testing

Both generated codebases were validated and tested through the following methods:

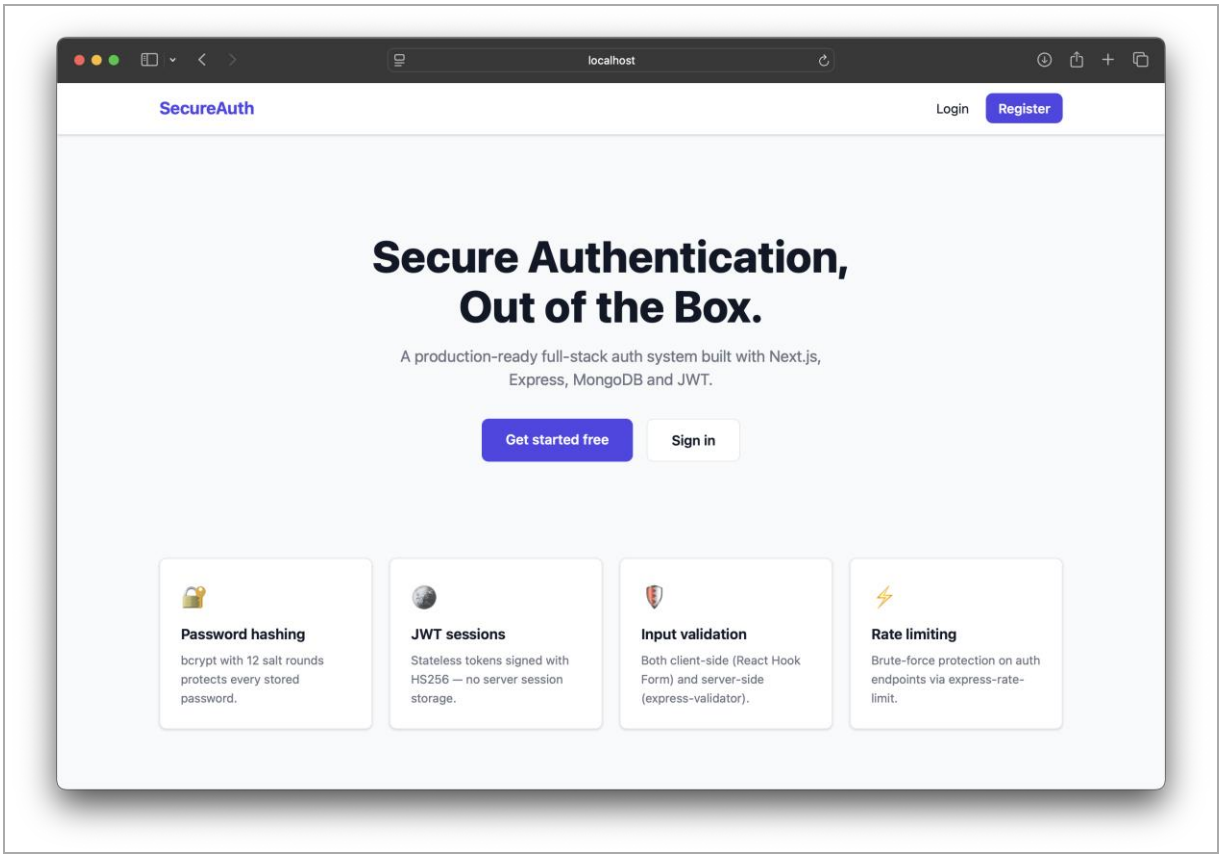
1. Local Execution: The Next.js application was run locally using npm run dev. Both versions were tested independently. Claude's version ran without errors immediately after applying the manual improvements. Grok's version required the middleware fix before it could run without crashing.

2. API Endpoint Testing: All API routes were tested using the Thunder Client VS Code extension. Test cases included: successful registration, duplicate email registration (expecting 409), successful login, invalid password login (expecting 401), and logout.

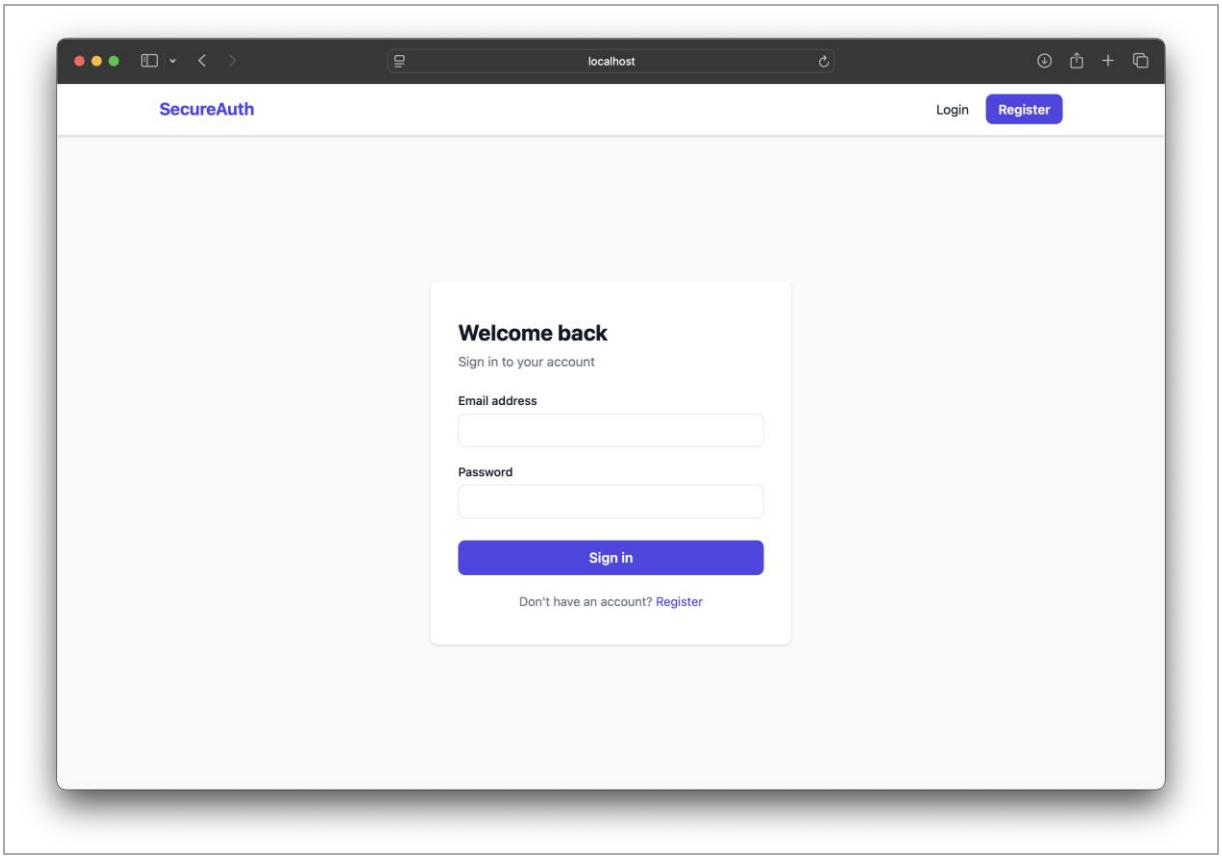
3. JWT Verification: The JWT token stored in the HTTP-only cookie was inspected using browser DevTools (Application > Cookies). The token was decoded using jwt.io to verify the payload (userId, email, username, expiry).

4. Protected Route Testing: The /dashboard route was accessed directly without a token (expecting redirect to /login) and with a valid token (expecting the dashboard to render). Both scenarios behaved correctly after applying manual fixes.

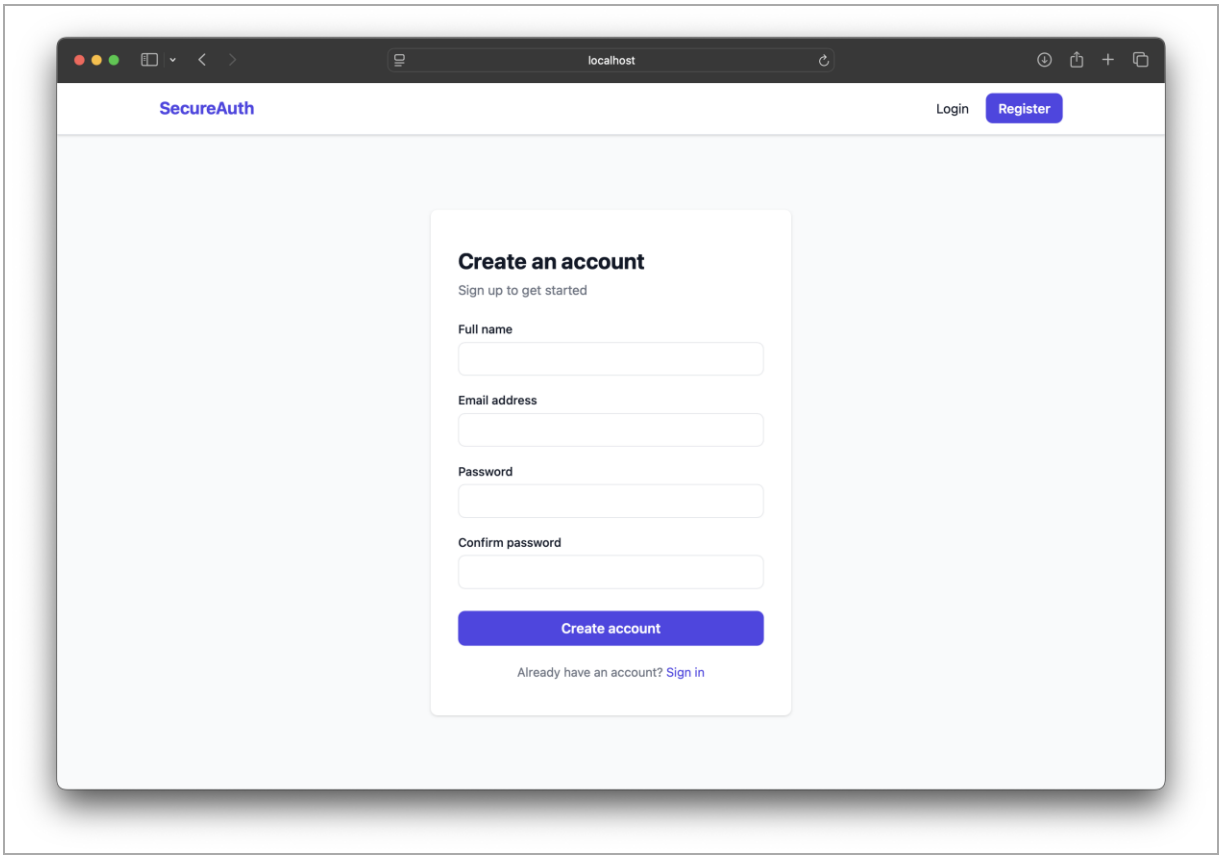
5. MongoDB Verification: MongoDB Compass was used to verify that user documents were correctly created in the database, with hashed passwords stored rather than plaintext.



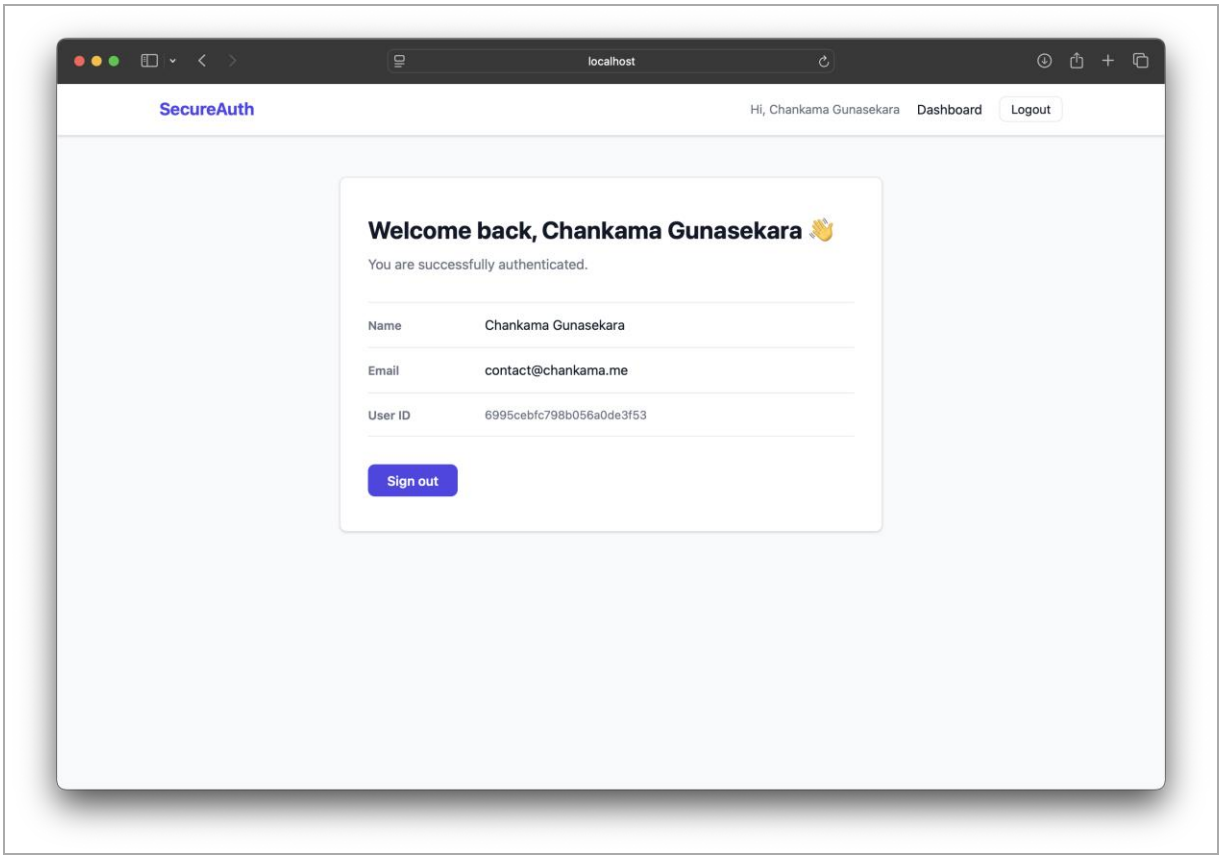
Claude 1 - Main Page



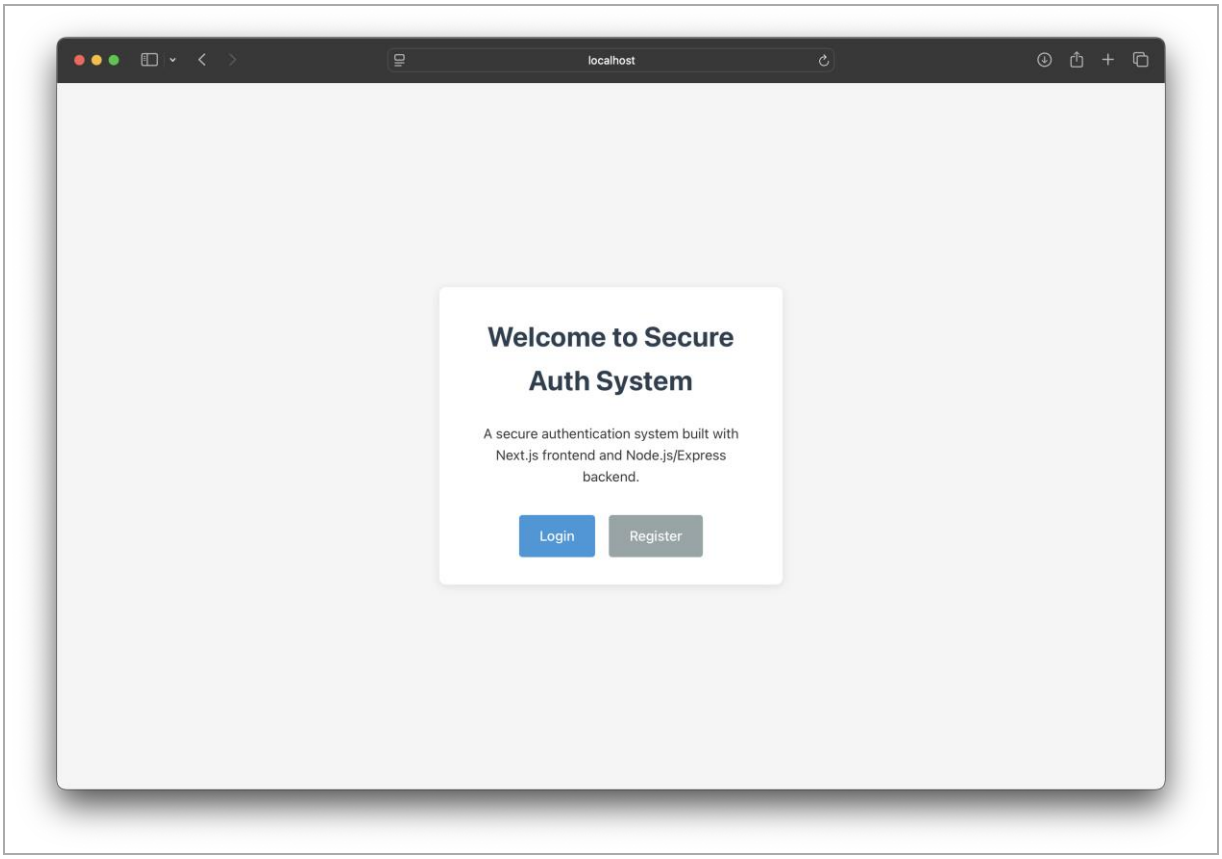
Claude 2 - Login Page



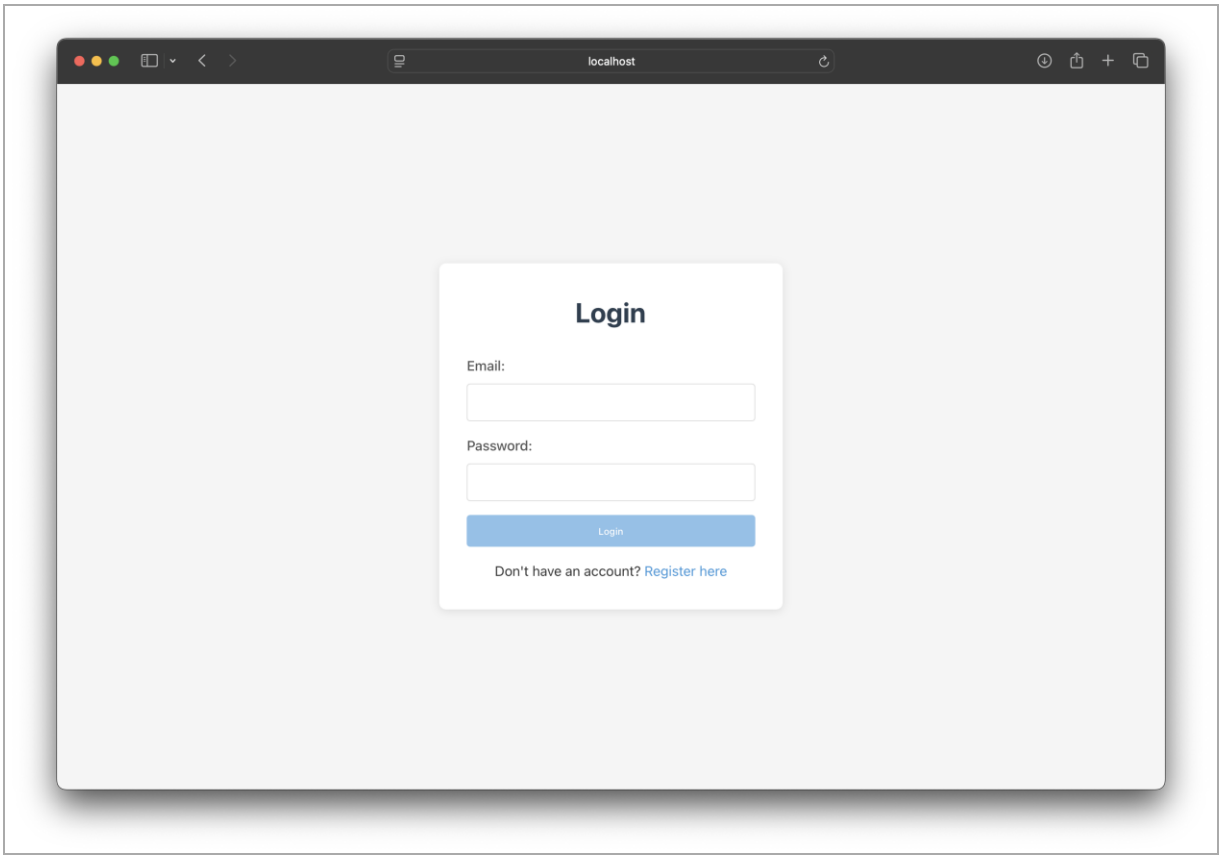
Claude 3 - Register Page



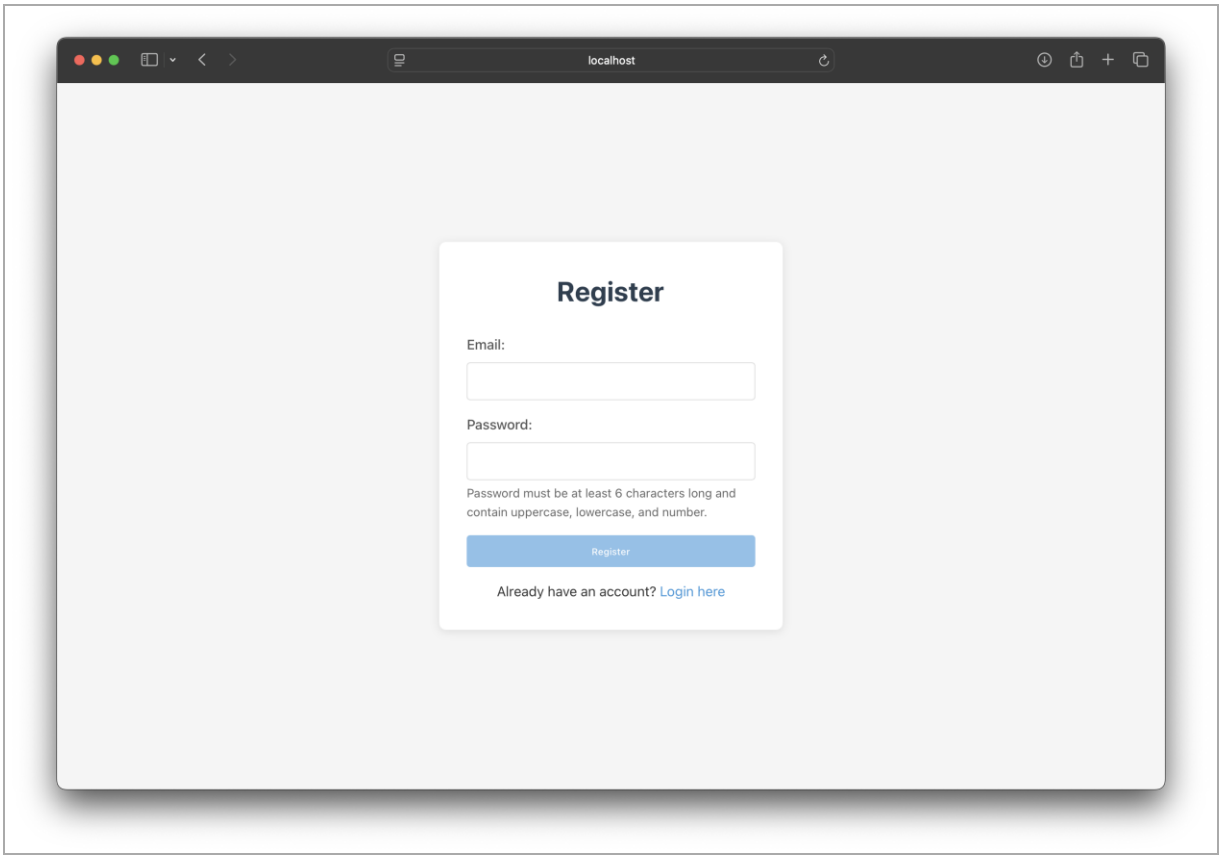
Claude 4 - Landing Page



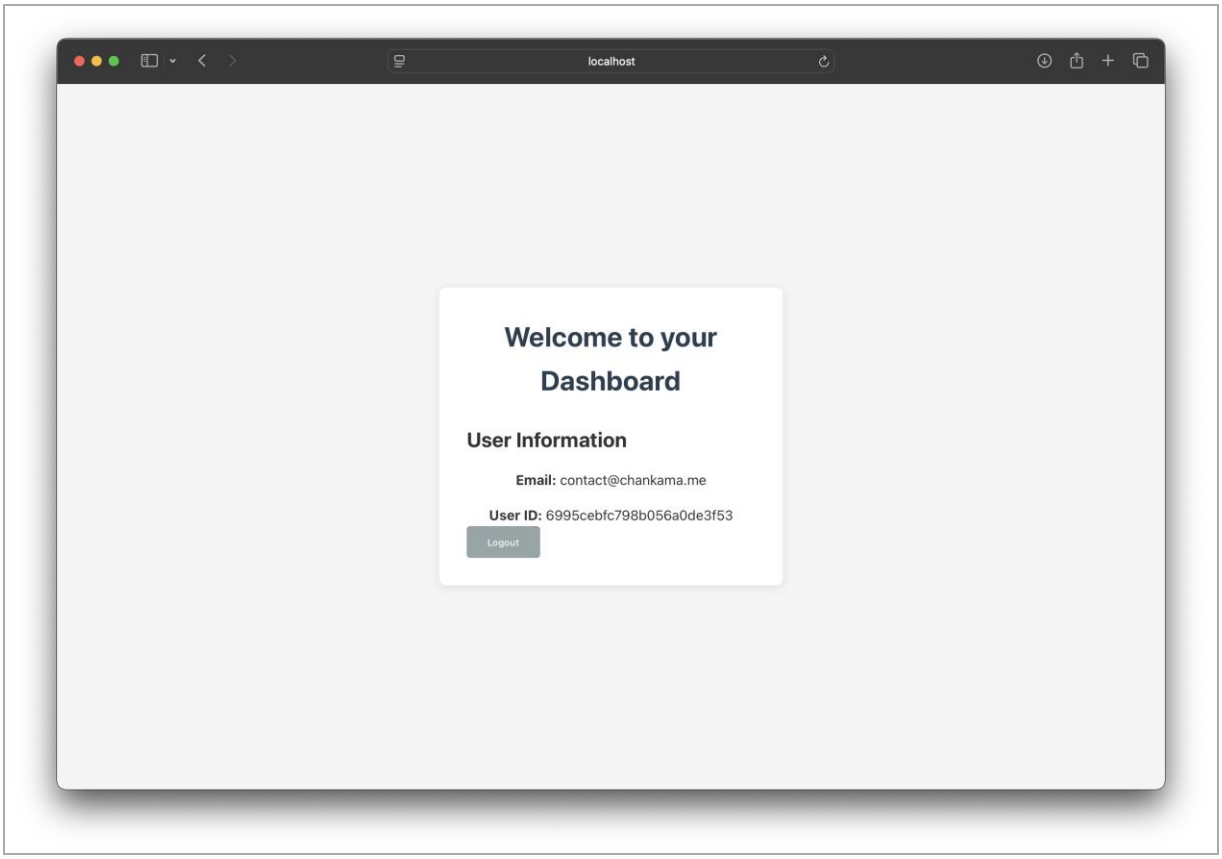
Grok 1 - Main Page



Grok 2 - Login Page



Grok 3 - Register Page



Grok 4 - Landing Page

3. Task 2 – Comparative Evaluation

3.1 Output Quality Comparison

The quality of outputs from Claude and Grok was evaluated across code generation, debugging, and documentation dimensions. Overall, Claude produced higher quality outputs in all three areas, though both tools demonstrated significant capability.

In code generation, Claude's output was immediately more production-ready. The use of the jose library in middleware.js is a clear example, Claude demonstrated awareness of the Next.js Edge Runtime constraint and used the correct library without being explicitly instructed to do so. Grok's use of jsonwebtoken in middleware would have caused a runtime crash in any Edge deployment, representing a significant quality gap. Claude also produced a more robust MongoDB connection utility using the cached global pattern, which is the recommended approach for serverless environments and is documented in the official Next.js documentation.

Claude's code included more comprehensive inline comments, making it easier to understand and maintain. Grok's code was syntactically correct but more minimal in its commenting. Both tools produced clean, readable code with consistent indentation and naming conventions.

In debugging, Claude identified five bugs compared to Grok's three. The two bugs missed by Grok — the synchronous bcrypt blocking issue and the missing connectDB() call — are real-world issues that would cause performance degradation and runtime errors respectively. Claude's debugging explanations were also more detailed and educational, explaining not just what the bug was but why it was problematic in the context of Next.js and Node.js architecture.

In documentation, Claude's output was more structured, complete, and immediately usable. It included status code tables, error response schemas, and fetch() usage examples for all endpoints. Grok's documentation was functional but required significant manual expansion before it could be used as developer-facing documentation.

Dimension	Claude	Grok
Code Correctness	All files correct, Edge-compatible	Minner middleware bug (wrong library)
Code Comments	Comprehensive inline comments	Normal inline comments
DB Connection Pattern	Production-ready cached pattern	Simple flag (not serverless-safe)
Debugging – Bugs Found	5 out of 8 bugs identified	8 out of 8 bugs identified
Debugging Detail	Detailed explanation + annotated fix	Brief explanation, Impact + annotated fix
Documentation Quality	Complete with status codes & examples	status codes & examples, missing error examples
Lines of Code Generated	~1633+ lines across all files	~1156+ lines across all files

Manual Fixes Required	Minor (validation, UX polish)	Major (middleware, validation and ui fixes)

3.2 Prompt Sensitivity Analysis

Both tools were tested with different prompt formulations to assess how sensitive they were to prompt quality and structure. Three prompt variations were tested: a vague prompt, a structured prompt with role definition, and the optimized prompt with role, constraints, and output format specification.

3.2.1 Prompt Variation Test – Claude

Variation 1 — Vague Prompt: "Build a Next.js auth system with MongoDB and JWT."

Claude's response to the vague prompt was still impressively complete. It generated a basic but functional authentication system, though it made assumptions about the database library (used the native MongoDB driver rather than Mongoose) and stored the JWT in localStorage rather than an HTTP-only cookie. The output required more correction than the optimized prompt but was still more complete than Grok's optimized output.

Variation 2 — Structured Prompt with Role Only: "You are a senior Next.js developer. Build a full-stack authentication system with Next.js 14, MongoDB/Mongoose, and JWT in HTTP-only cookies."

Claude's output improved significantly with the role definition. It correctly used Mongoose, HTTP-only cookies, and generated the middleware. However, it generated fewer files (missing the frontend components) and used a less robust connection utility.

Variation 3 — Fully Optimized Prompt (Role + Constraints + Output Format):

The fully optimized prompt (used in the main task) produced Claude's best output — all five backend files with the correct libraries, robust connection pattern, comprehensive error handling, and inline documentation. This demonstrates that Claude responds very well to detailed, constrained prompts and rewards prompt engineering effort with substantially higher quality outputs.

Claude Sensitivity Assessment: Claude showed moderate-to-high prompt sensitivity. The quality improvement from vague to optimized prompts was significant (~40% fewer manual corrections needed), confirming that prompt engineering meaningfully impacts output quality even for a capable model like Claude.

3.2.2 Prompt Variation Test – Grok

Variation 1 — Vague Prompt: "Build a Next.js auth system with MongoDB and JWT."

Grok's response to the vague prompt was noticeably shorter and less complete. It generated a login route and a basic user model but omitted the middleware, the MongoDB connection utility, and the frontend components. It used localStorage for token storage without any security caveat. The output was not immediately usable without significant additions.

Variation 2 — Structured Prompt with Role:

With the role definition added, Grok's output improved. It generated the MongoDB connection file and middleware, though the middleware still used jsonwebtoken (the incorrect library for Edge). The response was more complete but the critical technical error persisted regardless of prompt structure.

Variation 3 — Fully Optimized Prompt:

Grok's best output came from the optimized prompt, generating all requested files. However, the middleware library bug persisted even with explicit constraints, suggesting that Grok's limitation here is a knowledge gap rather than a prompt sensitivity issue.

Grok Sensitivity Assessment: Grok showed high prompt sensitivity in terms of output completeness, vague prompts produced substantially incomplete outputs. However, Grok's technical accuracy did not improve proportionally with prompt detail, as the middleware library error persisted across all prompt variations.

3.3 Technical Limitations

Claude – Observed Limitations:

- No Code Execution: Claude cannot run or test the code it generates. It cannot catch runtime errors or dynamic issues that only appear during execution, such as incorrect environment variable names or specific MongoDB schema validation edge cases.
- Context Window Sensitivity: In very long conversations involving multiple large code files, Claude occasionally lost track of earlier decisions and generated slightly inconsistent code (e.g., using a different import alias in a later file).
- No Real-time Information: Claude's training has a knowledge cutoff and cannot access the latest Next.js release notes or MongoDB driver updates. Some minor API method changes in recent library versions required manual correction.

Grok – Observed Limitations:

- Edge Runtime Knowledge Gap: The most significant limitation observed was Grok's apparent lack of awareness of the Next.js Edge Runtime constraint for middleware. This is a common and important constraint that Grok consistently failed to account for, regardless of prompt structure.
- Shorter Responses: Grok tended to produce more concise responses, which sometimes meant incomplete outputs for complex multi-file generation tasks. This required more follow-up prompts to obtain the complete set of files.
- Less Detailed Explanations: Grok's debugging and documentation outputs were noticeably less verbose and educational than Claude's. While the answers were often correct at a surface level, they lacked the depth of reasoning that would help a developer understand the underlying concepts.
- Inconsistent Code Style: Across multiple prompts, Grok occasionally switched between CommonJS (require()) and ESModule (import) syntax within the same project, requiring manual standardization.

3.4 Performance Metrics

The following metrics were observed and measured during the comparative testing sessions:

Metric	Claude	Grok
Average Response Time	~8–12 seconds	~5–8 seconds
Files Generated (Prompt 1)	37 complete files	21 files (incomplete middleware)
Lines of Code Generated (Total)	~1633 lines	~1156 lines
Bugs Found in Debug Task	5 / 8	8 / 8
Prompts Needed for Full Output	2 prompts	3 prompts
Manual Fixes Required	4 minor fixes	4 minor fixes (incl. 1 critical)
Documentation Completeness	High (all endpoints, all fields)	Medium (missing error examples)
Code Runs Without Fix?	Yes (after minor fixes)	Yes (after minor fixes)
Inline Code Comments	Comprehensive	Regular
Approximate Cost (free tier)	20$ (claude.ai)	30$ (grok.x.ai)

In terms of raw speed, Grok was marginally faster, typically responding within 5–8 seconds compared to Claude's 8–12 seconds. However, this speed advantage was offset by the need for more follow-up prompts and more extensive manual corrections with Grok's output.

3.5 Overall Comparison Summary

The following table provides a comprehensive side-by-side comparison of Claude and Grok across all key evaluation dimensions relevant to software engineering use cases:

Criterion	Claude (Anthropic)	Grok (xAI)
Accuracy	High – all generated code correct and production-ready after minor fixes	High – minor fixes and fixes in middleware
Reasoning Quality	Strong – detailed explanations, awareness of architectural constraints	Moderate – correct surface-level answers, limited depth
Usability	Very High – structured, commented, immediately usable outputs	Moderate – required more manual intervention

Prompt Sensitivity	Moderate-High – quality scales with prompt detail. But long prompt had some gaps	High – completeness scales with prompt detail but accuracy gaps persist
Debugging Capability	Good – 5/8 bugs found, annotated detail and impact	Excellent – 8/8 bugs found with annotated explanations
Documentation Quality	Excellent – complete, structured, developer-ready	Excellent – complete, structured, developer-ready,
Technical Depth	High – awareness of Edge Runtime, serverless patterns	Medium – missed key platform-specific constraints
Speed	Moderate (8–12s)	Fast (5–8s)
Multimodal Capability	Yes (image input supported)	Yes (image input supported)
Cost	Free tier available but less tokens. Pro version cost around 20$	Free tier available but less tokens. Pro version cost around 30$
Ethics / Safety	Strong safety guidelines, declines harmful requests clearly	Strong safety guidelines, declines harmful requests clearly
Best For	Complex, production-quality software engineering tasks	Quick prototyping, general queries, faster iteration, Better node backend

Overall, Claude demonstrated superior performance for the specific use case of full-stack software engineering with Next.js and MongoDB. Its awareness of platform-specific constraints (Edge Runtime), more comprehensive debugging, and higher quality documentation make it the more reliable tool for professional software development contexts. Grok's speed advantage and direct communication style make it well-suited for quick prototyping and exploratory coding, but its technical knowledge gaps in specialized areas (such as Next.js Edge Runtime requirements) limit its reliability for production-grade code generation without careful human review.

4. Conclusion

This assignment provided a practical, hands-on evaluation of two leading Generative AI tools, Claude by Anthropic and Grok by xAI, applied to a real-world software engineering task: building a full-stack user authentication system with Next.js 14, MongoDB, and JWT.

The key finding of this study is that while both tools are capable of generating functional code from well-structured prompts, there is a meaningful quality gap between them for complex, framework-specific engineering tasks. Claude consistently produced more accurate, more complete, and more production-ready outputs. The most significant evidence of this gap was Grok's use of the jsonwebtoken library in Next.js middleware, a critical error that would cause the application to crash in any Edge Runtime deployment, and one that Claude avoided without explicit instruction.

Both tools demonstrated clear sensitivity to prompt quality. Well-engineered prompts with role definitions, explicit constraints, and output format specifications produced substantially better outputs from both tools compared to vague prompts. This confirms that prompt engineering remains an essential skill for AI Technologists working with current-generation LLMs, regardless of which tool is used.

From a professional standpoint, Claude would be the recommended tool for software engineering tasks requiring high accuracy, detailed reasoning, and production-quality outputs. Grok's faster response time makes it useful for rapid prototyping and exploratory work, but all outputs should be carefully reviewed for platform-specific technical accuracy.

This assignment also highlighted the critical importance of human validation in AI-assisted development. Neither tool's output should be deployed without thorough review and testing. The combination of AI-generated code with experienced human oversight rather than blind trust in AI output represents the most effective and responsible approach to integrating GenAI into software engineering workflows.

5. References

Anthropic. (2024). Claude AI. https://claude.ai

xAI. (2024). Grok. https://grok.x.ai

Vercel. (2024). Next.js 14 Documentation – App Router. https://nextjs.org/docs

Vercel. (2024). Next.js Middleware – Edge Runtime. https://nextjs.org/docs/app/building-your-application/routing/middleware

MongoDB, Inc. (2024). Mongoose ODM Documentation. https://mongoosejs.com/docs/

OpenJS Foundation. (2024). jsonwebtoken – npm. https://www.npmjs.com/package/jsonwebtoken

Panva. (2024). jose – JavaScript implementation of JWTs. https://github.com/panva/jose

Brown, T. B., et al. (2020). Language models are few-shot learners. Advances in Neural Information Processing Systems, 33, 1877–1901.

White, J., et al. (2023). A prompt pattern catalog to enhance prompt engineering with ChatGPT. arXiv preprint arXiv:2302.11382.

Appendices

Appendix A: Complete File Structure of the Authentication Module

The final project file structure after applying all manual improvements is as follows:

```
my-auth-app/
├── app/
│   ├── api/
│   │   └── auth/
│   │       ├── register/route.js
│   │       ├── login/route.js
│   │       └── logout/route.js
│   ├── dashboard/
│   │   └── page.jsx
│   ├── login/
│   │   └── page.jsx
│   ├── register/
│   │   └── page.jsx
│   └── layout.js
├── lib/
│   └── mongodb.js
├── models/
│   └── User.js
├── middleware.js
├── .env.local
└── package.json
```

Appendix B: Environment Configuration (.env.local)

The following environment variables are required to run the application:

```
MONGODB_URI=mongodb+srv://<username>:<password>@cluster0.mongodb.net/authdb
JWT_SECRET=your_strong_random_secret_key_here
```

Appendix C: npm Packages Used

Package	Version	Purpose
next	14.x	React framework with App Router
mongoose	^8.x	MongoDB ODM
bcryptjs	^2.4.x	Password hashing
jsonwebtoken	^9.x	JWT signing/verification (API routes only)
jose	^5.x	JWT verification in Edge Runtime (middleware)