📚 COMPLETE CODE DOCUMENTATION - Line by Line Explanation

Bug Tracker MERN Project - Detailed File Analysis
Date: January 21, 2026
Status: Day 2 Complete - Authentication System

📑 TABLE OF CONTENTS

BACKEND FILES

---

1. 📄 **backend/models/User.js** (User Model)

Purpose:

Defines the MongoDB schema for user data and handles password security.

Line-by-Line Explanation:

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

Line 1: Import Mongoose library

- Mongoose is an ODM (Object Data Modeling) library for MongoDB
- Provides schema-based solution to model application data
- Handles validation, casting, and business logic

```
const bcrypt = require('bcryptjs');
```

Line 2: Import bcryptjs library

- Used for password hashing (one-way encryption)
- More secure than plain bcrypt (pure JavaScript, no C++ dependencies)
- Industry standard for password security

```
const userSchema = new mongoose.Schema({
```

Line 4: Create new Mongoose schema

- Schema defines the structure of documents in MongoDB collection
- Acts as a blueprint for user documents

- Enforces data types and validation rules

```
  name: {
    type: String,
```

Lines 5-6: Define name field

- `type: String` - Only accepts string values
- JavaScript strings will be stored as text in MongoDB

```
    required: [true, 'Please add a name'],
```

Line 7: Make name required with custom error message

- First value (true) - Field is mandatory
- Second value (string) - Error message if missing
- Validation happens before saving to database

```
    trim: true,
```

Line 8: Auto-trim whitespace

- Removes leading/trailing spaces from input
- Example: " John " becomes "John"
- Prevents accidental whitespace issues

```
    maxlength: [50, 'Name cannot be more than 50 characters']
```

Line 9: Set maximum length with error message

- Limits name to 50 characters
- Custom error message for user feedback
- Prevents database overflow and maintains data consistency

```
  },
  email: {
    type: String,
    required: [true, 'Please add an email'],
    unique: true,
```

Lines 11-14: Define email field

- `type: String` - Email stored as text
- `required` - Cannot be empty
- `unique: true` - No two users can have same email (database index)
- Creates unique index in MongoDB for fast lookups

```
    lowercase: true,
```

Line 15: Convert email to lowercase

- "John@Example.COM" becomes "john@example.com"
- Prevents duplicate accounts with different cases
- Makes email lookups case-insensitive

```
    trim: true,
```

Line 16: Remove whitespace from email

- Prevents "user@example.com " vs "user@example.com" issues

```
    match: [
      /^\w+([\.-]?\w+)*@\w+([\.-]?\w+)*(\.\w{2,3})+$/,
      'Please add a valid email'
    ]
```

Lines 17-20: Email format validation with regex

- Uses regular expression to validate email format
- Pattern breakdown:
  - `^\w+` - Start with word characters (a-z, A-Z, 0-9, _)
  - `[\.-]?\w+` - Optional dot or dash followed by word chars
  - `*` - Repeat previous pattern (for subdomains)
  - `@` - Required @ symbol
  - `\w+` - Domain name
  - `\.\w{2,3}` - Extension (.com, .net, .org, etc.)
  - `+$` - End of string
- Example valid: user@example.com, john.doe@mail.co.uk
- Example invalid: user@, @example.com, user.example.com

```
  },
  password: {
    type: String,
    required: [true, 'Please add a password'],
    minlength: [6, 'Password must be at least 6 characters'],
```

Lines 22-25: Define password field

- Stored as string (will be hashed before saving)
- Minimum 6 characters for basic security
- Required field with validation

```
    select: false // Don't return password in queries by default
```

Line 26: Hide password from query results

- **CRITICAL SECURITY FEATURE**
- When querying users, password is NOT included by default
- Must explicitly request with `.select('+password')` to get it
- Prevents accidental password exposure in API responses
- Example: `User.find()` will NOT include password
- Example: `User.findOne().select('+password')` will include it

```
  },
  createdAt: {
    type: Date,
    default: Date.now
```

Lines 28-31: Define createdAt timestamp

- `type: Date` - Stores as MongoDB Date object
- `default: Date.now` - Auto-sets current date/time when document created
- `Date.now` is a function reference (no parentheses)
- Mongoose calls it when creating new document
- Useful for tracking when user registered

```
});
```

Line 33: Close schema definition

```
// Hash password before saving
userSchema.pre('save', async function(next) {
```

Lines 35-36: Mongoose middleware (hook) before save

- `pre('save')` - Runs BEFORE document is saved to database
- `async` - Allows use of await for bcrypt operations
- `function(next)` - Traditional function (not arrow) to access `this`
- `this` refers to the document being saved
- `next` - Callback to continue to next middleware/save operation

```
  if (!this.isModified('password')) {
    next();
  }
```

Lines 37-39: Check if password was modified

- `this.isModified('password')` - Mongoose method to check if field changed
- If password NOT modified (e.g., updating name only), skip hashing
- Prevents re-hashing already hashed password
- `next()` - Move to next middleware or save
- **Why needed:** Updating user without changing password shouldn't re-hash

```
  const salt = await bcrypt.genSalt(10);
```

Line 41: Generate salt for hashing

- Salt = random data added to password before hashing
- `10` = cost factor (2^10 = 1024 hashing rounds)
- Higher number = more secure but slower
- 10 is recommended balance
- Each user gets unique salt
- Prevents rainbow table attacks

```
  this.password = await bcrypt.hash(this.password, salt);
```

Line 42: Hash the password with salt

- `this.password` - Plain text password from user input
- `bcrypt.hash()` - Creates one-way hash
- Result: "$2a$10$abcd...xyz" (60 character string)
- Original password cannot be recovered from hash
- Same password with different salt produces different hash
- Example: "password123" → "$2a$10$N9qo8uLOickgx2ZMRZoMye..."

```
});
```

Line 43: Close pre-save middleware

```
// Compare password method
userSchema.methods.comparePassword = async function(enteredPassword) {
```

Lines 45-46: Create custom instance method

- `methods` - Adds methods to all User documents
- `comparePassword` - Custom method name
- `async` - Returns promise (uses await internally)
- `enteredPassword` - Plain text password from login attempt
- Called on user instance: `user.comparePassword('password123')`

```
  return await bcrypt.compare(enteredPassword, this.password);
```

Line 47: Compare passwords

- `bcrypt.compare()` - Safely compares plain text with hash
- First param: Plain text password entered by user
- Second param: Hashed password from database (this.password)
- Returns: Boolean (true if match, false if not)
- Internally: Bcrypt extracts salt from hash and hashes entered password
- Then compares both hashes byte-by-byte
- Time-constant comparison prevents timing attacks

```
};
```

Line 48: Close comparePassword method

```
module.exports = mongoose.model('User', userSchema);
```

Line 50: Export User model

- `mongoose.model()` - Creates model from schema
- First param: 'User' - Model name (singular)

- MongoDB collection will be named 'users' (plural, lowercase)
- Second param: userSchema - Schema definition
- Export allows importing in other files: `require('./models/User')`
- Creates Model constructor with methods: find, findOne, create, etc.

---

2. 📄 **backend/controllers/authController.js** (Authentication Logic)

Purpose:

Contains business logic for user authentication (register, login, get profile).

Line-by-Line Explanation:

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

Line 1: Import User model

- Loads User model from models folder
- Allows creating, finding, updating users
- Provides access to all Mongoose methods

```
const jwt = require('jsonwebtoken');
```

Line 2: Import JSON Web Token library

- JWT = Compact, URL-safe token for authentication
- Stateless authentication (no server-side session storage)
- Token contains encoded user information

```
// Generate JWT Token
const generateToken = (id) => {
```

Lines 4-5: Helper function to generate JWT

- Takes user ID as parameter
- Returns signed JWT token
- Reusable for register and login

```
  return jwt.sign({ id }, process.env.JWT_SECRET, {
```

Line 6: Create and sign JWT

- `jwt.sign()` - Creates token
- First param: `{ id }` - Payload (data stored in token)
  - Shorthand for `{ id: id }`
  - Contains user's MongoDB _id
- Second param: `process.env.JWT_SECRET` - Secret key from .env
  - Used to sign token (proves token is legitimate)
  - Must match when verifying token
  - Should be long, random string

```
    expiresIn: '30d'
```

Line 7: Set token expiration

- Token valid for 30 days
- After 30 days, token becomes invalid
- User must login again
- Can use: '1h' (1 hour), '7d' (7 days), '90d', etc.
- Balances security (shorter) vs convenience (longer)

```
  });
};
```

Lines 8-9: Close generateToken function

- Returns token string: "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."

```
// @desc    Register new user
// @route   POST /api/auth/register
// @access  Public
```

Lines 11-13: JSDoc-style comments

- Documents what function does
- `@desc` - Description of functionality
- `@route` - HTTP method and endpoint path
- `@access` - Who can access (Public = no auth required)

```
exports.register = async (req, res) => {
```

Line 14: Export register controller function

- `exports.register` - Makes function available for import
- `async` - Can use await for database operations
- `req` - Request object (contains body, params, headers)
- `res` - Response object (send data back to client)

```
  try {
```

Line 15: Try-catch block for error handling

- Catches any errors during registration process
- Prevents server crash
- Allows sending error response to client

```
    const { name, email, password } = req.body;
```

Line 16: Destructure request body

- Extracts name, email, password from POST request body
- `req.body` - Data sent from frontend
- Example: `{ name: "John", email: "john@example.com", password: "pass123" }`
- Requires `express.json()` middleware to parse JSON

```
    // Validation
    if (!name || !email || !password) {
      return res.status(400).json({ message: 'Please provide all fields' });
    }
```

Lines 18-20: Validate required fields

- Checks if any field is missing or empty
- `!name` - Falsy check (null, undefined, empty string, 0, false)
- `return` - Stops function execution
- `res.status(400)` - HTTP 400 Bad Request
- `.json()` - Sends JSON response
- Early return pattern prevents further execution

```
    // Check if user exists
    const userExists = await User.findOne({ email });
```

Lines 22-23: Check for duplicate email

- `User.findOne()` - Mongoose method to find one document
- `{ email }` - Shorthand for `{ email: email }`
- Searches for user with this email
- `await` - Waits for database query to complete
- Returns user object if found, null if not found

```
    if (userExists) {
      return res.status(400).json({ message: 'User already exists' });
    }
```

Lines 24-26: Prevent duplicate registration

- If user found with same email, reject registration
- `status(400)` - Bad Request (client error)
- Provides user-friendly error message
- Prevents unique constraint violation error

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

Lines 28-32: Create new user in database

- `User.create()` - Mongoose method to create and save document
- Shorthand for `new User({...}).save()`
- `await` - Waits for database operation
- Triggers pre-save middleware (password hashing)
- Returns created user document
- Password is plain text here, hashed by pre-save hook

```
    if (user) {
```

Line 34: Check if user created successfully

- Should always be true unless database error
- Extra safety check

```
      res.status(201).json({
```

Line 35: Send success response

- `status(201)` - HTTP 201 Created (successful creation)
- `.json()` - Send JSON response

```
        _id: user._id,
        name: user.name,
        email: user.email,
        token: generateToken(user._id)
```

Lines 36-39: Response payload

- `_id` - MongoDB document ID
- `name`, `email` - User information
- `token` - JWT token for authentication
- Password NOT included (security)
- Frontend will store token and use for authenticated requests

```
      });
    } else {
      res.status(400).json({ message: 'Invalid user data' });
    }
```

Lines 40-42: Handle creation failure

- Fallback if User.create() fails
- Rare case (usually caught by try-catch)

```
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
};
```

Lines 43-45: Error handling

- `catch` - Catches any errors in try block
- `status(500)` - HTTP 500 Internal Server Error
- `error.message` - Error description
- Handles: Database connection errors, validation errors, etc.

```
// @desc    Login user
// @route   POST /api/auth/login
// @access  Public
exports.login = async (req, res) => {
  try {
    const { email, password } = req.body;
```

Lines 47-52: Login function setup

- Similar structure to register
- Only needs email and password (no name)

```
    // Validation
    if (!email || !password) {
      return res.status(400).json({ message: 'Please provide email and password' });
    }
```

Lines 54-56: Validate login inputs

- Ensures both email and password provided
- Early return if missing

```
    // Check for user and include password
    const user = await User.findOne({ email }).select('+password');
```

Lines 58-59: Find user and get password

- `findOne({ email })` - Search by email
- `.select('+password')` - **IMPORTANT:** Override default behavior
- Remember: User model has `select: false` on password
- `+password` explicitly includes password field in result
- Without this, user.password would be undefined
- Needed to compare passwords

```
    if (!user) {
      return res.status(401).json({ message: 'Invalid credentials' });
    }
```

Lines 61-63: Handle user not found

- `status(401)` - HTTP 401 Unauthorized
- Generic "Invalid credentials" message
- **Security:** Don't reveal if email exists or password wrong
- Prevents email enumeration attacks

```
    // Check password
    const isMatch = await user.comparePassword(password);
```

Lines 65-66: Verify password

- Calls custom method from User model
- `password` - Plain text password from request
- `user.comparePassword()` - Uses bcrypt to compare
- Returns boolean: true if match, false if not

```
    if (!isMatch) {
      return res.status(401).json({ message: 'Invalid credentials' });
    }
```

Lines 68-70: Handle wrong password

- Same response as wrong email
- **Security:** Attacker can't tell which is wrong

```
    res.json({
      _id: user._id,
      name: user.name,
      email: user.email,
      token: generateToken(user._id)
    });
```

Lines 72-77: Send login success response

- `status(200)` is default, can omit
- Returns user info + token
- Frontend stores token for authenticated requests

```
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
};
```

Lines 78-80: Error handling for login

```
// @desc    Get current user
// @route   GET /api/auth/me
// @access  Private
exports.getMe = async (req, res) => {
```

Lines 82-85: Get logged-in user profile

- `Private` access - Requires authentication
- Auth middleware must run before this function

```
  try {
    const user = await User.findById(req.user.id);
```

Lines 86-87: Find user by ID

- `req.user.id` - Set by auth middleware
- Middleware decodes JWT and adds user.id to request
- `findById()` - Mongoose method to find by MongoDB _id
- Password NOT included (select: false by default)

```
    if (!user) {
      return res.status(404).json({ message: 'User not found' });
    }
```

Lines 89-91: Handle user not found

- `status(404)` - HTTP 404 Not Found
- Rare: Token valid but user deleted from database

```
    res.json({
      _id: user._id,
      name: user.name,
      email: user.email
    });
```

Lines 93-97: Return user profile

- Returns user information
- No token needed (already authenticated)
- No password (security)

```
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
};
```

Lines 98-100: Error handling for getMe

---

3. 📄 **backend/routes/auth.js** (API Routes)

Purpose:

Defines authentication endpoints and links them to controller functions.

Line-by-Line Explanation:

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

Line 1: Import Express

- Express is the web framework
- Provides routing, middleware, request/response handling

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

Line 2: Create Express router

- `Router()` - Creates modular route handler
- Can be mounted as middleware in main app
- Groups related routes together
- Makes routes modular and reusable

```
const { register, login, getMe } = require('../controllers/authController');
```

Line 3: Import controller functions

- Destructuring import of three functions
- From authController.js file
- These contain actual business logic

```
const auth = require('../middleware/auth');
```

Line 4: Import auth middleware

- Middleware to verify JWT tokens
- Protects routes from unauthorized access

```
// Public routes
router.post('/register', register);
```

Lines 6-7: Register route

- `router.post()` - Define POST route
- `/register` - Route path (will be /api/auth/register)
- `register` - Controller function to handle request
- Anyone can access (no auth middleware)
- When request comes: Express calls register() function

```
router.post('/login', login);
```

Line 8: Login route

- POST /api/auth/login
- Calls login controller
- Public access

```
// Protected routes
router.get('/me', auth, getMe);
```

Lines 10-11: Get current user route

- GET /api/auth/me
- `auth` - Middleware runs FIRST
- `getMe` - Controller runs AFTER middleware
- Execution order: Request → auth middleware → getMe controller → Response
- `auth` middleware validates token and adds user to req.user

```
module.exports = router;
```

Line 13: Export router

- Makes router available for import in server.js
- Will be mounted at `/api/auth` in main app

---

4. 📄 **backend/middleware/auth.js** (JWT Verification)

Purpose:

Middleware to verify JWT tokens and protect routes.

Line-by-Line Explanation:

```
const jwt = require('jsonwebtoken');
```

Line 1: Import JWT library

- Same library used to create tokens
- Now used to verify them

```
const auth = async (req, res, next) => {
```

Line 3: Define middleware function

- `async` - Can use await
- `req` - Request object
- `res` - Response object
- `next` - Function to call next middleware/route handler
- Middleware signature: (req, res, next)

```
  try {
```

Line 4: Try-catch for error handling

- Catches invalid tokens, expired tokens, etc.

```
    // Get token from header
    const token = req.header('Authorization')?.replace('Bearer ', '');
```

Lines 5-6: Extract token from request header

- `req.header('Authorization')` - Gets Authorization header
- Example header: "Authorization: Bearer eyJhbGciOiJIUzI1NiIs..."
- `?.` - Optional chaining (safe if header doesn't exist)
- `.replace('Bearer ', '')` - Remove "Bearer " prefix
- Result: Just the token string
- Standard format for JWT in HTTP headers

```
    if (!token) {
      return res.status(401).json({ message: 'No token, authorization denied' });
    }
```

Lines 8-10: Check if token exists

- If no token provided, reject request
- `status(401)` - HTTP 401 Unauthorized
- `return` - Stop execution, don't call next()

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

Lines 12-13: Verify and decode token

- `jwt.verify()` - Validates token signature
- First param: Token to verify
- Second param: Secret key (must match creation secret)
- Returns: Decoded payload `{ id: userId, iat: ..., exp: ... }`
- `iat` - Issued at (timestamp)
- `exp` - Expiration (timestamp)
- Throws error if:
  - Token tampered with
  - Token expired
  - Invalid signature
  - Malformed token

```
    req.user = decoded;
```

Line 14: Attach user data to request

- `decoded` contains `{ id: userId }`
- Makes user ID available to next middleware/controller
- Controller can access via `req.user.id`
- Common pattern in Express authentication

```
    next();
```

Line 15: Call next middleware/controller

- `next()` - Passes control to next function in chain
- If this is before getMe controller, now getMe runs
- Without next(), request hangs

```
  } catch (error) {
    res.status(401).json({ message: 'Token is not valid' });
  }
};
```

Lines 16-18: Error handling

- Catches: Expired token, invalid token, tamperedtoken
- Returns 401 Unauthorized
- Generic message for security

```
module.exports = auth;
```

Line 21: Export middleware

- Can be used in routes: `router.get('/me', auth, getMe)`

---

5. 📄 **backend/config/db.js** (Database Connection)

Line-by-Line Explanation:

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

Line 1: Import Mongoose

- ORM for MongoDB
- Handles connection, queries, schemas

```
const connectDB = async () => {
```

Line 3: Define async connection function

- `async` - Can use await for connection
- Exported function called in server.js

```
  try {
    const conn = await mongoose.connect(process.env.MONGODB_URI);
```

Lines 4-5: Connect to MongoDB

- `mongoose.connect()` - Establishes connection
- `process.env.MONGODB_URI` - Connection string from .env
- `await` - Waits for connection to establish
- Returns connection object
- Example URI: "mongodb+srv://user:pass@cluster.mongodb.net/bugtracker"

```
    console.log(`✅ MongoDB Connected: ${conn.connection.host}`);
```

Line 6: Log success message

- `conn.connection.host` - MongoDB server hostname
- Confirms connection established
- Shows which cluster connected to

```
  } catch (error) {
    console.error(`❌ Error: ${error.message}`);
```

Lines 7-8: Error handling

- Catches connection errors
- Logs error message
- Examples: Wrong password, network error, wrong URI

```
    process.exit(1);
```

Line 9: Exit application on error

- `process.exit(1)` - Terminates Node.js process
- `1` - Exit code indicating error
- Prevents app from running without database
- Server won't start if can't connect to database

```
  }
};
```

Lines 10-11: Close try-catch and function

```
module.exports = connectDB;
```

Line 13: Export function

- Used in server.js: `connectDB()`

---

6. 📄 backend/server.js (Main Server File)

Line-by-Line Explanation:

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

Line 1: Import Express framework

- Core web framework for Node.js
- Provides routing, middleware, HTTP methods

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

Line 2: Import CORS middleware

- CORS = Cross-Origin Resource Sharing
- Allows frontend (port 3000) to make requests to backend (port 5000)
- Prevents browser security errors

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

Line 3: Import Helmet middleware

- Security middleware
- Sets various HTTP headers for security
- Prevents common vulnerabilities (XSS, clickjacking, etc.)

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

Line 4: Import dotenv

- Loads environment variables from .env file
- Makes variables available via process.env

```
const connectDB = require('./config/db');
```

Line 5: Import database connection function

- Custom function to connect to MongoDB

```
// Load environment variables
dotenv.config();
```

Lines 7-8: Load .env file

- `dotenv.config()` - Reads .env file
- Parses key=value pairs
- Adds to process.env object
- Must run before using process.env variables

```
// Connect to MongoDB
connectDB();
```

Lines 10-11: Establish database connection

- Calls connectDB function
- Runs asynchronously (doesn't block server startup)
- Server can start while connecting to database

```
const app = express();
```

Line 13: Create Express application

- `app` - Main application object
- Used to define routes, middleware, configuration

```
// Middleware
app.use(helmet());
```

Lines 15-16: Add Helmet security middleware

- `app.use()` - Registers middleware
- Runs on every request
- Sets security-related HTTP headers

```
app.use(cors());
```

Line 17: Add CORS middleware

- Allows all origins by default
- Frontend can make requests to API
- In production, specify allowed origins

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

Line 18: Parse JSON request bodies

- Parses `Content-Type: application/json`
- Makes JSON data available in req.body
- Required for POST/PUT requests with JSON

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

Line 19: Parse URL-encoded request bodies

- Parses form data
- `extended: true` - Parse complex objects
- Makes form data available in req.body

```
// Test route
app.get('/', (req, res) => {
  res.json({ message: 'Bug Tracker API is running!' });
});
```

Lines 21-24: Root route

- GET / endpoint
- Simple health check
- Returns JSON message
- Confirms server is running

```
// Routes
app.use('/api/auth', require('./routes/auth'));
```

Lines 26-27: Mount auth routes

- `app.use()` - Registers routes
- `/api/auth` - Base path for auth routes
- `require('./routes/auth')` - Auth router
- All routes in auth.js prefixed with /api/auth
- Example: register route becomes /api/auth/register

```
// Error handling middleware
app.use((err, req, res, next) => {
```

Lines 31-32: Global error handler

- `(err, req, res, next)` - Error middleware signature
- 4 parameters distinguish it from regular middleware
- Catches errors from all routes

```
  console.error(err.stack);
```

Line 33: Log error

- `err.stack` - Full error stack trace
- Helps debugging
- Logs to console/file

```
  res.status(500).json({
    message: 'Something went wrong!',
    error: process.env.NODE_ENV === 'development' ? err.message : {}
  });
```

Lines 34-37: Send error response

- `status(500)` - Internal Server Error
- Generic message
- `error` - Only show details in development
- Production hides error details (security)

```
});
```

Line 38: Close error handler

```
const PORT = process.env.PORT || 5000;
```

Line 40: Define port

- Reads PORT from .env
- Falls back to 5000 if not set
- `||` - Logical OR operator

```
app.listen(PORT, () => {
  console.log(`🚀 Server is running on port ${PORT}`);
});
```

Lines 42-44: Start server

- `app.listen()` - Starts HTTP server
- Listens on specified port
- Callback runs when server starts
- Server accepts incoming requests

---

7. 📄 **backend/package.json** (Dependencies)

Key Sections Explained:

```
"dependencies": {
  "express": "^4.18.2",
```

Express - Web framework

- Routing, middleware, HTTP methods
- Core of backend API

```
  "mongoose": "^8.0.0",
```

Mongoose - MongoDB ODM

- Schema definition
- Query building
- Validation

```
  "dotenv": "^16.3.1",
```

Dotenv - Environment variables

- Loads .env file
- Security (keeps secrets out of code)

```
  "bcryptjs": "^2.4.3",
```

Bcryptjs - Password hashing

- Secure password storage
- One-way encryption

```
  "jsonwebtoken": "^9.0.2",
```

Jsonwebtoken - JWT tokens

- Create and verify tokens
- Stateless authentication

```
  "cors": "^2.8.5",
```

CORS - Cross-origin requests

- Allows frontend to call API
- Browser security

```
  "helmet": "^7.1.0",
```

Helmet - Security headers

- Protects against common attacks
- XSS, clickjacking, etc.

```
  "express-validator": "^7.0.1"
```

Express-validator - Input validation

- Validates request data
- Sanitizes input

```
"devDependencies": {
  "nodemon": "^3.0.1"
}
```

Nodemon - Auto-restart server

- Development tool
- Restarts on file changes
- Not needed in production

```
"scripts": {
  "start": "node server.js",
  "dev": "nodemon server.js"
}
```

Scripts

- `npm start` - Production (manual restart)
- `npm run dev` - Development (auto-restart)

---

8. 📄 backend/.env (Environment Variables)

Line-by-Line Explanation:

```
PORT=5000
```

PORT - Server port number

- Server listens on this port
- Default: 5000
- Can change if port busy

```
MONGODB_URI=mongodb+srv://chaturvedipuneet200_db_user:i4kJxfNTaQSQZb7F@gigflow-cluster.czk3cti.mongodb.net/bugtracker?
retryWrites=true&w=majority&appName=gigflow-cluster
```

MONGODB_URI - Database connection string

- **Format:** `mongodb+srv://username:password@cluster/database`
- `chaturvedipuneet200_db_user` - Database username
- `i4kJxfNTaQSQZb7F` - Database password
- `gigflow-cluster.czk3cti.mongodb.net` - Cluster hostname
- `bugtracker` - Database name
- `retryWrites=true` - Retry failed writes
- `w=majority` - Write concern (wait for majority)
- `appName` - Application identifier

```
JWT_SECRET=bug_tracker_jwt_secret_key_2026_change_in_production_12345
```

JWT_SECRET - Secret key for JWT signing

- Used to sign and verify tokens
- Must be long and random
- Keep secret (never commit to Git)
- Change in production to strong random string

```
NODE_ENV=development
```

NODE_ENV - Environment mode

- `development` - Shows detailed errors
- `production` - Hides errors, optimizations
- Used in conditional logic

FRONTEND FILES

---

9. 📄 **frontend/src/context/AuthContext.jsx** (Global Auth State)

Purpose:

Manages authentication state globally across the application.

Line-by-Line Explanation:

```
import { createContext, useContext, useState, useEffect } from 'react';
```

Line 1: Import React hooks

- `createContext` - Create context object
- `useContext` - Access context value
- `useState` - Manage component state
- `useEffect` - Run side effects

```
import { useNavigate } from 'react-router-dom';
```

Line 2: Import navigation hook

- `useNavigate` - Programmatic navigation
- Redirect after login/logout
- From react-router-dom library

```
import API from '../utils/api';
```

Line 3: Import Axios instance

- Custom Axios configuration
- Base URL and interceptors
- Makes API calls

```
import { toast } from 'react-toastify';
```

Line 4: Import toast notifications

- `toast` - Show toast messages
- Success, error, info notifications
- User feedback

```
const AuthContext = createContext();
```

Line 6: Create context object

- `createContext()` - Creates context
- AuthContext - Container for auth data
- Shared across components

```
export const useAuth = () => {
```

Line 8: Custom hook to use context

- `useAuth` - Easier way to access context
- Returns context value
- Throws error if used outside provider

```
  const context = useContext(AuthContext);
```

Line 9: Get context value

- `useContext(AuthContext)` - Access context
- Returns value from nearest Provider

```
  if (!context) {
    throw new Error('useAuth must be used within AuthProvider');
  }
```

Lines 10-12: Error handling

- Ensures hook used correctly
- Must be inside AuthProvider
- Helps catch mistakes early

```
  return context;
};
```

Lines 13-14: Return context value

- Makes {user, login, logout, etc.} available

```
export const AuthProvider = ({ children }) => {
```

Line 16: AuthProvider component

- Wraps app to provide auth state
- `children` - Components inside provider
- Props destructuring

```
  const [user, setUser] = useState(null);
```

Line 17: User state

- `user` - Current logged-in user object
- `setUser` - Function to update user
- Initial: null (not logged in)
- After login: { _id, name, email, token }

```
  const [loading, setLoading] = useState(true);
```

Line 18: Loading state

- `loading` - Boolean for auth check
- Initial: true (checking if logged in)
- Shows spinner while checking
- After check: false

```
  const navigate = useNavigate();
```

Line 19: Navigation function

- `navigate('/path')` - Go to route
- Used after login/logout

```
  // Check if user is logged in on mount
  useEffect(() => {
```

Lines 21-22: Auto-login effect

- Runs once when component mounts
- Checks if user already logged in
- Empty dependency array [] means run once

```
    const checkAuth = async () => {
```

Line 23: Define async check function

- Can't use async directly in useEffect
- Define function inside, call below

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

Line 24: Get token from storage

- `localStorage.getItem()` - Retrieve stored token
- Returns token string or null
- Token saved during login

```
      if (token) {
```

Line 25: Check if token exists

- If token found, validate it

```
        try {
          const { data } = await API.get('/auth/me');
```

Lines 26-27: Validate token with API

- GET /api/auth/me
- Axios interceptor adds token to header
- Backend verifies token
- Returns user data if valid

```
          setUser(data);
```

Line 28: Set user if token valid

- Updates user state with data
- User now logged in

```
        } catch (error) {
          localStorage.removeItem('token');
```

Lines 29-30: Handle invalid token

- Catch: Expired token, invalid token
- Remove bad token from storage

```
        }
      }
```

Lines 31-32: Close try-catch and if

```
      setLoading(false);
```

Line 33: Stop loading

- Auth check complete
- Show app content

```
    };
    checkAuth();
  }, []);
```

Lines 34-36: Call check function and close effect

- `checkAuth()` - Run the check
- `[]` - Run once on mount

```
  // Register user
  const register = async (name, email, password) => {
```

Lines 38-39: Register function

- `async` - Can await API call
- Takes registration data

```
    try {
      const { data } = await API.post('/auth/register', { name, email, password });
```

Lines 40-41: Call register API

- POST /api/auth/register
- Send name, email, password
- Axios sends as JSON

```
      localStorage.setItem('token', data.token);
```

Line 42: Store token

- `setItem()` - Save to localStorage
- Key: 'token', Value: JWT string
- Persists across page refreshes

```
      setUser(data);
```

Line 43: Update user state

- User now logged in
- Triggers re-render

```
      toast.success('Registration successful!');
```

Line 44: Show success message

- Green toast notification
- User feedback

```
      navigate('/dashboard');
```

Line 45: Redirect to dashboard

- Programmatic navigation
- Go to protected route

```
      return { success: true };
```

Line 46: Return success status

- Component can handle result
- Optional

```
    } catch (error) {
      const message = error.response?.data?.message || 'Registration failed';
```

Lines 47-48: Error handling

- Catch API errors
- Extract error message from response
- `?.` - Safe navigation
- Fallback message if none provided

```
      toast.error(message);
```

Line 49: Show error message

- Red toast notification
- Shows backend error

```
      return { success: false, message };
```

Line 50: Return failure status

```
    }
  };
```

Lines 51-52: Close register function

```
  // Login user
  const login = async (email, password) => {
```

Lines 54-55: Login function

- Similar to register
- Only email and password

```
    try {
      const { data } = await API.post('/auth/login', { email, password });
      localStorage.setItem('token', data.token);
      setUser(data);
      toast.success('Login successful!');
      navigate('/dashboard');
      return { success: true };
```

Lines 56-62: Login process

- Call login API
- Store token
- Update user state
- Show success toast
- Redirect to dashboard

```
    } catch (error) {
      const message = error.response?.data?.message || 'Login failed';
      toast.error(message);
      return { success: false, message };
    }
  };
```

Lines 63-67: Login error handling

```
  // Logout user
  const logout = () => {
```

Lines 69-70: Logout function

- Not async (no API call)
- Just clears local state

```
    localStorage.removeItem('token');
```

Line 71: Remove token

- Delete from localStorage

- User can't make authenticated requests

```
    setUser(null);
```

Line 72: Clear user state

- User object null
- Triggers re-render

```
    toast.info('Logged out successfully');
```

Line 73: Show logout message

- Blue info toast

```
    navigate('/login');
```

Line 74: Redirect to login

- Can't access protected routes

```
  };
```

Line 75: Close logout function

```
  const value = {
    user,
    loading,
    register,
    login,
    logout
  };
```

Lines 77-83: Context value object

- All data/functions available to consumers
- Components can access these
- Via `const { user, login } = useAuth()`

```
  return <AuthContext.Provider value={value}>{children}</AuthContext.Provider>;
};
```

Lines 85-86: Provide context

- `Provider` - Makes value available
- `children` - Wrapped components can access
- All descendant components can useAuth()

---

10. 📄 **frontend/src/components/ProtectedRoute.jsx** (Route Guard)

Line-by-Line Explanation:

```
import { Navigate } from 'react-router-dom';
```

Line 1: Import Navigate component

- `Navigate` - Redirect component
- From react-router-dom
- Used to redirect unauthorized users

```
import { useAuth } from '../context/AuthContext';
```

Line 2: Import auth hook

- Access user and loading state
- From AuthContext

```
const ProtectedRoute = ({ children }) => {
```

Line 4: Define component

- `children` - The page to protect (Dashboard, etc.)
- Props destructuring

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

Line 5: Get auth state

- `user` - Current user (null if not logged in)
- `loading` - True while checking auth

```
  if (loading) {
    return (
      <div className="flex items-center justify-center h-screen">
        <div className="animate-spin rounded-full h-12 w-12 border-b-2 border-primary-600"></div>
      </div>
    );
  }
```

Lines 7-13: Show loading spinner

- While checking if user logged in
- Prevents flash of login page
- Tailwind classes:
  - `flex items-center justify-center` - Center content
  - `h-screen` - Full viewport height
  - `animate-spin` - Rotate animation
  - `rounded-full` - Circle
  - `h-12 w-12` - Size
  - `border-b-2 border-primary-600` - Colored bottom border

```
  if (!user) {
    return <Navigate to="/login" replace />;
  }
```

Lines 15-17: Redirect if not logged in

- If no user, redirect to login
- `Navigate` - Declarative redirect
- `to="/login"` - Destination
- `replace` - Replace history (can't go back)
- Prevents accessing protected pages

```
  return children;
};
```

Lines 19-20: Render protected page

- If user exists, show the page
- `children` - Dashboard or other protected component

```
export default ProtectedRoute;
```

Line 22: Export component

- Can be imported and used in routes

Usage Example:

```
<Route path="/dashboard" element={
  <ProtectedRoute>
    <Dashboard />
  </ProtectedRoute>
} />
```

11. 📄 **frontend/src/pages/Register.jsx** (Registration Page)

Key Sections Explained:

```
const [formData, setFormData] = useState({
  name: '',
  email: '',
  password: '',
  confirmPassword: ''
});
```

Form state

- Object with all form fields
- All start empty
- Updated as user types

```
const [errors, setErrors] = useState({});
```

Error state

- Object to store validation errors
- Keys are field names
- Values are error messages

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

Get auth functions

- `register` - Function to register user
- `user` - Current user (for redirect check)

```
if (user) {
  return <Navigate to="/dashboard" replace />;
}
```

Prevent double login

- If already logged in, go to dashboard
- Can't register if logged in

```
const handleChange = (e) => {
  setFormData({ ...formData, [e.target.name]: e.target.value });
  if (errors[e.target.name]) {
    setErrors({ ...errors, [e.target.name]: '' });
  }
};
```

Handle input changes

- `e.target.name` - Input field name
- `e.target.value` - Input field value
- Spread operator `...` - Keep other fields
- `[e.target.name]` - Computed property name
- Clear error when user starts typing

```
const validate = () => {
  const newErrors = {};

  if (!formData.name.trim()) {
    newErrors.name = 'Name is required';
  }

  if (!formData.email.trim()) {
    newErrors.email = 'Email is required';
  } else if (!/\S+@\S+\.\S+/.test(formData.email)) {
    newErrors.email = 'Email is invalid';
  }

  if (!formData.password) {
    newErrors.password = 'Password is required';
  } else if (formData.password.length < 6) {
    newErrors.password = 'Password must be at least 6 characters';
  }

  if (formData.password !== formData.confirmPassword) {
    newErrors.confirmPassword = 'Passwords do not match';
  }

  return newErrors;
};
```

Validation function

- Checks all fields
- `.trim()` - Remove whitespace
- Regex test for email format
- Password length check
- Password match check
- Returns object of errors

```
const handleSubmit = async (e) => {
  e.preventDefault();

  const newErrors = validate();
  if (Object.keys(newErrors).length > 0) {
    setErrors(newErrors);
    return;
  }

  await register(formData.name, formData.email, formData.password);
};
```

Form submission

- `e.preventDefault()` - Stop page reload
- Run validation first
- If errors exist, show them and stop
- `Object.keys().length` - Count errors
- If no errors, call register function

```
<input
  type="text"
  id="name"
  name="name"
  value={formData.name}
  onChange={handleChange}
  className={`w-full px-4 py-2 border rounded-lg focus:ring-2 focus:ring-primary-500 focus:border-transparent outline-none
transition ${
    errors.name ? 'border-red-500' : 'border-gray-300'
  }`}
  placeholder="John Doe"
/>
```

Input field

- Controlled component (value from state)
- `name` matches state key
- `onChange` updates state
- Dynamic className based on error
- `errors.name ?` - Conditional style
- Red border if error, gray if not

```
{errors.name && <p className="text-red-500 text-sm mt-1">{errors.name}</p>}
```

Error message

- Conditional rendering
- Only show if error exists
- Red text, small font

---

12. 📄 **frontend/src/pages/Login.jsx** (Login Page)

Similar to Register, Key Differences:

```
const [formData, setFormData] = useState({
  email: '',
  password: ''
});
```

Simpler form state

- Only email and password
- No name or confirm password

```
const validate = () => {
  const newErrors = {};

  if (!formData.email.trim()) {
    newErrors.email = 'Email is required';
  } else if (!/\S+@\S+\.\S+/.test(formData.email)) {
    newErrors.email = 'Email is invalid';
  }

  if (!formData.password) {
    newErrors.password = 'Password is required';
  }

  return newErrors;
};
```

Simpler validation

- Only email and password checks
- No length or match validation

```
await login(formData.email, formData.password);
```

Call login instead

- Uses login function from context
- Not register

```
<div className="flex items-center">
  <input
    id="remember"
    type="checkbox"
    className="h-4 w-4 text-primary-600 focus:ring-primary-500 border-gray-300 rounded"
  />
  <label htmlFor="remember" className="ml-2 block text-sm text-gray-700">
    Remember me
  </label>
</div>
```

Remember me checkbox

- Currently UI only
- Not connected to functionality
- Can implement persistent login later

```
<a href="#" className="text-primary-600 hover:text-primary-700 font-medium">
  Forgot password?
</a>
```

Forgot password link

- Placeholder for now
- Can implement password reset later

---

13. 📄 **frontend/src/pages/Dashboard.jsx** (User Dashboard)

Key Sections:

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

Get auth data

- `user` - Display user info
- `logout` - Logout button functionality

```
<header className="bg-white shadow">
  <div className="max-w-7xl mx-auto px-4 py-4 sm:px-6 lg:px-8 flex justify-between items-center">
```

Header section

- White background with shadow
- Max width container
- Responsive padding
- Flex layout (space between)

```
<p className="font-medium text-gray-900">{user?.name}</p>
```

Display user name

- `user?.name` - Safe access
- Won't error if user null

```
<button
  onClick={logout}
  className="bg-red-600 text-white px-4 py-2 rounded-lg hover:bg-red-700 transition"
>
  Logout
</button>
```

Logout button

- Calls logout function
- Red button (destructive action)
- Hover effect

```
<div className="grid grid-cols-1 md:grid-cols-3 gap-4 mb-6">
```

Stats grid

- 1 column on mobile
- 3 columns on medium+ screens
- Gap between cards

```
<div className="bg-gray-50 rounded-lg p-4 border border-gray-200">
  <div className="text-3xl font-bold text-primary-600 mb-1">0</div>
  <div className="text-sm text-gray-600">Projects</div>
  <div className="text-xs text-gray-500 mt-1">Coming in Day 3</div>
</div>
```

Stat card

- Shows count (placeholder 0)
- Label
- Status message
- Will be dynamic in Day 3

---

14. 📄 **frontend/src/App.jsx** (Main App Component)

Line-by-Line Explanation:

```
import { BrowserRouter as Router, Routes, Route, Navigate } from 'react-router-dom';
```

Line 1: Import routing components

- `BrowserRouter` as `Router` - Router wrapper
- `Routes` - Container for routes
- `Route` - Individual route definition
- `Navigate` - Programmatic redirect

```
import { ToastContainer } from 'react-toastify';
import 'react-toastify/dist/ReactToastify.css';
```

Lines 2-3: Import toast notifications

- `ToastContainer` - Container for toasts
- CSS import for toast styles

```
import { AuthProvider } from './context/AuthContext';
```

Line 4: Import auth context provider

- Wraps app to provide auth state

```
import ProtectedRoute from './components/ProtectedRoute';
```

Line 5: Import route guard

- Protects routes from unauthorized access

```
import Login from './pages/Login';
import Register from './pages/Register';
import Dashboard from './pages/Dashboard';
```

Lines 7-9: Import pages

- All page components

```
function App() {
  return (
    <Router>
```

Lines 11-13: App component with Router

- `Router` - Enables routing
- Must wrap entire app

```
      <AuthProvider>
```

Line 14: Wrap with AuthProvider

- Provides auth state to all components
- Must be inside Router (uses useNavigate)

```
        <div className="min-h-screen bg-gray-50">
```

Line 15: Main container

- Minimum full screen height
- Light gray background

```
          <Routes>
```

Line 16: Routes container

- Holds all route definitions

```
              <Route path="/" element={<Navigate to="/login" replace />} />
```

Line 18: Root route redirect

- `/` redirects to `/login`
- `replace` - Replace history
- User lands on login page

```
              <Route path="/login" element={<Login />} />
              <Route path="/register" element={<Register />} />
```

Lines 19-20: Public routes

- Anyone can access
- No authentication required

```
              <Route
                path="/dashboard"
                element={
                  <ProtectedRoute>
                    <Dashboard />
                  </ProtectedRoute>
                }
              />
```

Lines 22-28: Protected route

- Wrapped in ProtectedRoute component
- Checks authentication
- Redirects to login if not authenticated

```
            </Routes>
            <ToastContainer position="top-right" autoClose={3000} />
```

Lines 29-30: Toast container

- Must be in component tree
- Top-right position
- Auto-close after 3 seconds

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

Lines 31-35: Close tags

```
export default App;
```

Line 38: Export App component

---

15. 📄 **frontend/src/utils/api.js** (Axios Configuration)

Line-by-Line Explanation:

```
import axios from 'axios';
```

Line 1: Import Axios

- HTTP client library
- Makes API requests

```
const API = axios.create({
  baseURL: 'http://localhost:5000/api',
});
```

Lines 3-5: Create Axios instance

- `axios.create()` - Custom instance
- `baseURL` - Prepended to all requests
- Example: `API.get('/auth/me')` → `http://localhost:5000/api/auth/me`

```
// Add token to requests
API.interceptors.request.use((config) => {
```

Lines 7-8: Request interceptor

- Runs before every request
- Modifies request configuration
- `config` - Request configuration object

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

Line 9: Get token from storage

- Retrieve stored JWT token

```
  if (token) {
    config.headers.Authorization = `Bearer ${token}`;
  }
```

Lines 10-12: Add token to header

- If token exists, add to Authorization header
- Format: "Bearer eyJhbGciOiJIUzI1..."
- Backend expects this format

```
  return config;
});
```

Lines 13-14: Return modified config

- Request continues with token added

```
export default API;
```

Line 16: Export Axios instance

- Used throughout app
- Example: `import API from '../utils/api'`

Usage:

```
const response = await API.get('/auth/me');
// Automatically adds: Authorization: Bearer <token>
```

---

16. 📄 **frontend/src/main.jsx** (React Entry Point)

Line-by-Line Explanation:

```
import React from 'react';
```

Line 1: Import React library

- Core React library
- Required for JSX

```
import ReactDOM from 'react-dom/client';
```

Line 2: Import ReactDOM

- React 18 client rendering API
- Creates root for rendering

```
import App from './App';
```

Line 3: Import main App component

- Root component of application

```
import './index.css';
```

Line 4: Import global styles

- CSS with Tailwind directives
- Applied to entire app

```
ReactDOM.createRoot(document.getElementById('root')).render(
```

Line 6: Create React root

- `createRoot()` - React 18 API
- `document.getElementById('root')` - Get root div from HTML
- `.render()` - Start rendering

```
  <React.StrictMode>
```

Line 7: Enable Strict Mode

- Development mode checks
- Highlights potential problems
- Double-renders components (development only)

```
    <App />
```

Line 8: Render App component

- Root component
- Renders entire application

```
  </React.StrictMode>
);
```

Lines 9-10: Close tags

---

17. **frontend/src/index.css** (Global Styles)

Line-by-Line Explanation:

```
@tailwind base;
```

Line 1: Tailwind base styles

- CSS reset
- Normalize browser defaults
- Base element styles

```
@tailwind components;
```

Line 2: Tailwind component classes

- Pre-built component utilities

- Can add custom components

```
@tailwind utilities;
```

Line 3: Tailwind utility classes

- All utility classes (flex, grid, text-, bg-, etc.)
- Core of Tailwind

```
* {
  margin: 0;
  padding: 0;
  box-sizing: border-box;
}
```

Lines 5-9: Global reset

- Remove default margins/padding
- `box-sizing: border-box` - Include padding in width
- Applied to all elements

```
body {
  font-family: 'Inter', -apple-system, BlinkMacSystemFont, 'Segoe UI', 'Roboto',  'Oxygen',
    'Ubuntu', 'Cantarell', 'Fira Sans', 'Droid Sans', 'Helvetica Neue',
    sans-serif;
```

Lines 11-14: Font family

- Inter font (if available)
- System font fallbacks
- Native look on each platform

```
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
```

Lines 15-16: Font smoothing

- Improves font rendering
- Smoother text on Mac/iOS

```
}
```

Line 17: Close body styles

```
code {
  font-family: source-code-pro, Menlo, Monaco, Consolas, 'Courier New',
    monospace;
}
```

Lines 19-22: Code font

- Monospace fonts for code blocks
- Different from body text

---

18. 📄 **frontend/vite.config.js** (Vite Configuration)

Line-by-Line Explanation:

```
import { defineConfig } from 'vite';
```

Line 1: Import defineConfig

- Type-safe config helper
- Provides TypeScript types

```
import react from '@vitejs/plugin-react';
```

Line 2: Import React plugin

- Enables React Fast Refresh
- JSX transformation

```
export default defineConfig({
```

Line 4: Export configuration object

- `defineConfig()` wraps config

```
  plugins: [react()],
```

Line 5: Add React plugin

- Enables React features
- Fast refresh, JSX, etc.

```
  server: {
    port: 3000,
```

Lines 6-7: Dev server config

- Port 3000 (instead of default 5173)
- Easier to remember

```
    proxy: {
      '/api': {
        target: 'http://localhost:5000',
        changeOrigin: true,
      },
    },
```

Lines 8-13: Proxy configuration

- Routes `/api/*` requests to backend
- Example: `fetch('/api/auth/login')` → `http://localhost:5000/api/auth/login`
- `changeOrigin: true` - Change host header
- Avoids CORS issues in development

```
  },
});
```

Lines 14-15: Close config

---

19. 📄 **frontend/tailwind.config.js** (Tailwind Configuration)

Line-by-Line Explanation:

```
/** @type {import('tailwindcss').Config} */
```

Line 1: TypeScript type hint

- Provides autocomplete in VS Code
- Not executed code (comment)

```
export default {
```

Line 2: Export config object

- ES6 module syntax

```
  content: [
    "./index.html",
    "./src/**/*.{js,ts,jsx,tsx}",
  ],
```

Lines 3-6: Content paths

- Files to scan for Tailwind classes
- `./index.html` - HTML file
- `./src/**/*` - All files in src
- `{js,ts,jsx,tsx}` - File extensions
- Tailwind removes unused classes (tree-shaking)

```
  theme: {
    extend: {
```

Lines 7-8: Theme customization

- `extend` - Add to default theme
- Don't replace, add more

```
      colors: {
        primary: {
          50: '#eff6ff',
          100: '#dbeafe',
          // ... more shades
          900: '#1e3a8a',
        },
      },
```

Lines 9-21: Custom color palette

- `primary` - Custom color name
- 50-900 - Shades (50 lightest, 900 darkest)
- Use like: `bg-primary-500`, `text-primary-600`
- Blue theme for bug tracker

```
    },
  },
  plugins: [],
}
```

Lines 22-25: Close config

- `plugins: []` - No plugins yet
- Can add: forms, typography, etc.

---

20. 📄 **frontend/package.json** (Frontend Dependencies)

Key Sections:

```
"dependencies": {
  "react": "^18.2.0",
  "react-dom": "^18.2.0",
```

React - UI library

- Component-based
- Virtual DOM
- Declarative

```
  "react-router-dom": "^6.20.0",
```

React Router - Routing library

- Client-side routing
- No page reloads

```
  "axios": "^1.6.2",
```

Axios - HTTP client

- Makes API requests
- Better than fetch

```
  "react-beautiful-dnd": "^13.1.1",
```

React Beautiful DnD - Drag and drop

- For Kanban board (Day 8)
- Accessible drag-drop

```
  "react-toastify": "^9.1.3"
```

React Toastify - Toast notifications

- User feedback
- Success/error messages

```
"devDependencies": {
  "@vitejs/plugin-react": "^4.2.0",
  "vite": "^5.0.0",
```

Vite - Build tool

- Fast dev server
- Hot module replacement
- Production builds

```
  "tailwindcss": "^3.3.6",
  "postcss": "^8.4.32",
  "autoprefixer": "^10.4.16"
```

Tailwind + PostCSS - CSS framework

- Utility-first CSS
- PostCSS processes CSS
- Autoprefixer adds vendor prefixes

```
"scripts": {
  "dev": "vite",
  "build": "vite build",
  "preview": "vite preview"
}
```

Scripts

- `npm run dev` - Start dev server
- `npm run build` - Build for production
- `npm run preview` - Preview build

---

21. 📄 **frontend/index.html** (HTML Template)

Line-by-Line Explanation:

```
<!DOCTYPE html>
```

Line 1: Document type

- HTML5 document

```
<html lang="en">
```

Line 2: HTML root element

- `lang="en"` - English language

```
  <head>
    <meta charset="UTF-8" />
```

Lines 3-4: Character encoding

- UTF-8 supports all characters
- Emojis, international characters

```
    <link rel="icon" type="image/svg+xml" href="/bug-icon.svg" />
```

Line 5: Favicon link

- Browser tab icon
- SVG format
- File should be in public folder

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

Line 6: Viewport meta tag

- Responsive design
- Scale to device width
- Required for mobile

```
    <title>Bug Tracker - Project Management</title>
```

Line 7: Page title

- Shows in browser tab
- Search engine title

```
  </head>
  <body>
    <div id="root"></div>
```

Lines 8-10: Body and root div

- `id="root"` - React mounts here
- Empty initially
- Filled by React

```
    <script type="module" src="/src/main.jsx"></script>
```

Line 11: Load React app

- `type="module"` - ES6 modules
- Entry point: main.jsx
- Vite processes this

```
  </body>
</html>
```

Lines 12-13: Close tags

🔐 SECURITY CONCEPTS EXPLAINED

Password Hashing with Bcrypt

What Happens:

1. **User Registration:**

   ```
   User enters: "mypassword123"
   ↓
   Generate salt: "randomsalt12345"
   ↓
   ```

```
Hash: bcrypt("mypassword123" + "randomsalt12345")
↓
Result: "$2a$10$abcdefghij..."  (60 characters)
↓
Store in database
```

2. **User Login:**

```
User enters: "mypassword123"
↓
Get hash from database: "$2a$10$abcdefghij..."
↓
Extract salt from hash
↓
Hash entered password with same salt
↓
Compare: New hash === Stored hash?
↓
If match: Login successful
```

Why It's Secure:

- **One-way**: Can't reverse hash to get password
- **Unique salt**: Same password → Different hashes
- **Slow**: Takes time to hash (prevents brute force)
- **Industry standard**: Trusted and tested

JWT Authentication Flow

Token Creation:

```
// Backend creates token
const token = jwt.sign(
  { id: user._id },           // Payload (user info)
  "secret_key",               // Secret (only backend knows)
  { expiresIn: '30d' }        // Options
);
```

Token Structure:

```
Token:
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjEyMzQ1Njc4OTAiLCJpYXQiOjE1MTYyMzkwMjJ9.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c"

Parts (separated by dots):
1. Header:    eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
              Base64({ "alg": "HS256", "typ": "JWT" })

2. Payload:   eyJpZCI6IjEyMzQ1Njc4OTAiLCJpYXQiOjE1MTYyMzkwMjJ9
              Base64({ "id": "1234567890", "iat": 1516239022 })

3. Signature: SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
              HMACSHA256(header + "." + payload, secret_key)
```

Authentication Flow:

```
1. User logs in
   ↓
2. Backend validates credentials
   ↓
3. Backend creates JWT token
   ↓
4. Frontend stores token in localStorage
   ↓
5. User navigates to protected route
   ↓
6. Frontend adds token to request header
   Authorization: Bearer <token>
   ↓
7. Backend auth middleware extracts token
```

```
   ↓
8. Backend verifies signature with secret
   ↓
9. If valid: Decode payload, get user ID
   ↓
10. Backend processes request
```

Why It's Secure:

- **Signature**: Can't modify without secret
- **Stateless**: No server-side storage needed
- **Expiration**: Tokens expire automatically
- **Portable**: Can be used across services

📊 DATA FLOW DIAGRAMS

Complete Authentication Flow

```
REGISTRATION:
User → Frontend Form → Validation → API.post('/auth/register')
                                          ↓
Backend receives → Check email exists → Hash password
                                          ↓
Create user in MongoDB → Generate JWT → Send response
                                          ↓
Frontend receives → Store token → Update user state → Navigate to dashboard

LOGIN:
User → Frontend Form → Validation → API.post('/auth/login')
                                          ↓
Backend receives → Find user → Compare passwords
                                          ↓
If match: Generate JWT → Send response
                                          ↓
Frontend receives → Store token → Update user state → Navigate to dashboard

PROTECTED ROUTE ACCESS:
User visits /dashboard → ProtectedRoute checks user
                                          ↓
If no user: Redirect to /login
                                          ↓
If user exists: Render Dashboard → Dashboard loads
                                          ↓
Frontend makes API call → Axios interceptor adds token
                                          ↓
Backend auth middleware verifies token → Process request → Send response
```

🎯 SUMMARY

Files Created (Day 1 + Day 2):

Backend (8 files):

1. ✅ server.js - Main server
2. ✅ config/db.js - Database connection
3. ✅ middleware/auth.js - JWT verification
4. ✅ models/User.js - User schema
5. ✅ controllers/authController.js - Auth logic
6. ✅ routes/auth.js - Auth endpoints
7. ✅ package.json - Dependencies
8. ✅ .env - Environment variables

Frontend (13 files):

1. ✅ index.html - HTML template
2. ✅ src/main.jsx - React entry
3. ✅ src/App.jsx - Main component
4. ✅ src/index.css - Global styles

5. ✅ src/utils/api.js - Axios config
6. ✅ src/context/AuthContext.jsx - Auth state
7. ✅ src/components/ProtectedRoute.jsx - Route guard
8. ✅ src/pages/Login.jsx - Login page
9. ✅ src/pages/Register.jsx - Register page
10. ✅ src/pages/Dashboard.jsx - Dashboard
11. ✅ vite.config.js - Vite config
12. ✅ tailwind.config.js - Tailwind config
13. ✅ package.json - Dependencies

Total: 21 code files + documentation

End of Documentation
Project Status: Day 2 Complete ✅
Next: Day 3 - Project Management