**Context**  
Project Name: Trackio  
Team Members:

* Yurchenko Illia –Frontend Developer
* Maslo Veronika – Fullstack Developer, Team Lead

This project is an educational project to build a simplified version of Trello aimed at personal productivity. The app allows individual users to create and manage their tasks visually.

**Goals and non-goals  
✅ Goals**

Our objective is to develop a minimal Trello-like task management system as a single-user application. The core features we plan to implement include:

* **User authentication:** A seamless authentication flow with session management. This will encompass user registration, login, logout, and authentication token refreshing.
* **Task creation:** Users will be able to create tasks with a title, description, tags, status, and deadline.
* **Customizable statuses:** Each user can create their own task statuses and select their colors. Default statuses will be provided upon user registration and cannot be renamed: "To Do," "In Progress," "Done," "Backlog," and "In Review."
* **Customizable tags:** Users can create their own tags in a similar manner. Default tags will also be created at registration - "Bug," "Feature," "Enhancement," "Documentation," and "Urgent." The names of these default tags cannot be changed, but their colors can be customized.
* **Task visualization:** Tasks will be displayed on a Kanban-style board, grouped by status.
* **Drag and drop**: Users will have the ability to drag and drop tasks between statuses directly on the Kanban board.
* **Detailed task view**: Users will be able to view and update task details (such as title, description, status, etc.) on a dedicated task page.

❌ **Non-Goals**

To keep the scope small and focused (as this is an educational project), we are not planning to implement the following features:

* **Collaboration features**: There will be no multi-user boards, team features, or shared tasks.
* **Groups or organizations**: We will not group users into teams or companies.
* **Email verification and password recovery**: We will provide basic authentication only, without "forgot password" or email confirmation features.
* **Google authentication:** We will not integrate Google authentication.
* **Performance optimization or scalability**: We are not focusing on large-scale performance tuning or heavy optimization.

**Subsystems**

A diagram of a software application

AI-generated content may be incorrect.Our application will utilize a modular monolithic architecture, built with Next.js, incorporating both the frontend and backend within the same codebase. The main components (subsystems) are as follows:

1. HTTP Frontend (UI)

2. API Routes (Business Logic + Controller)

4. Data Access Layer (Prisma)

5. Database (MongoDB)

**Subsystem Descriptions**

1. HTTP Frontend (UI)

* Built using React components and rendered via Next.js pages.
* Utilizes the Next.js App Router for page routing located in `/src/app/`.
* Implements custom hooks stored in `/src/hooks/` for reusable logic.
* Employs a global state management system with Zustand in `/src/store/`.
* Sends JSON requests to internal API routes.
* Manages client-side state and supports a drag-and-drop UI.

2. API Routes

* Located under `/app/api/` (App Router).
* They manage authentication through middleware (found in `src/middleware.ts`) and utilize a custom axios instance for automatic token refresh.
* Organized by resource:
  + /api/tasks/ - Task management
  + /api/statuses/ - Status management
  + /api/tags/ - Tag management
  + /api/auth/ - Authentication
* Implements RESTful endpoints for:
  + Task management
  + Status management
  + Tag management
  + User authentication
  + Token refresh

Additionally, a cron job runs daily at 3 AM to clean up expired tokens from the database.

3. Data Access Layer

* Prisma ORM handles querying, creating, and updating records in MongoDB.
* Prisma Client is generated and shared across service / controller modules.
* Models defined in prisma/schema.prisma:
  + User
  + Task
  + Status
  + Tag
  + TaskTag (junction table)
  + RefreshToken

4. Database

* MongoDB as the primary database
* Stores:
  + User accounts with authentication data
  + Tasks with relationships to statuses and tags
  + Status categories with color and default settings
  + Tags with color and default settings
  + Refresh tokens for authentication
  + Task-Tag relationships

**How Subsystems Are Connected**

1. **Data Flow**

* Frontend components make requests to API routes
* API routes are responsible for all business logic and database operations.
* Authentication is managed through middleware, Axios interceptors, and refresh tokens.

1. **State Management**

* Uses a combination of React hooks and global state management (Zustand with persist)
* Implements custom hooks for reusable logic
* Employs providers for global context to maintain state.

1. **Security**

* Uses JWT-based authentication for secure access.
* Incorporates refresh and access tokent for session management. Both are atored as secure HTTP-only cookies.
* Middleware is implemented to protect routes.
* Passwords are hashed to enhance user security.

1. **Testing**

* Jest configuration for testing (jest.config.mjs)
* Test setup in jest.setup.js

1. **Development Tools**

* TypeScript for type safety
* ESLint for code quality

This architecture establishes a robust foundation for a task management application, featuring capabilities such as task organization, status tracking, and tag management, all while ensuring security and scalability.

**Authorization subsystem**

**Responsibilities**

* User registration
* User login
* User logout
* Session token (JWT) creation
* Session token (JWT) verification
* Token refreshing.
* Middleware protection for private routes

**Implementation Details**

1. **User Registration:**
   * Validate user input (e.g., email format, password length).
   * Check if the email is already in use.
   * Hash the password (using bcrypt) and store the user in the database.
   * Generate signed JWT tokens.
   * Set the tokens in an HTTP-only cookies in the response.
2. **User Login:**
   * Validate user input.
   * Look up the user by email.
   * Compare the provided password with the stored hashed password using bcrypt.compare.
   * If valid, generate signed JWT tokens.
   * Set the tokens in an HTTP-only cookies in the response.
3. **User Logout:**
   * Invalidate the current session by clearing the JWT cookies.
   * Set the token cookie with an empty value and a Max-Age=0 to expire it.
   * Delete the refresh token from the database.
   * Redirect the user to the login page.
4. **Token Management:**

**Token Types and Lifetimes**

1. Access Token:

* Short-lived (15 minutes)
* Contains userId and email
* Used for API authorization

1. Refresh Token:

* Long-lived (7 days)
* Contains only userId
* Stored in database with expiration
* Used to obtain new access tokens

**Token Storage**

* Both tokens stored in HTTP-only cookies
* Refresh tokens also stored in database with:
* Token value
* User ID
* Expiration timestamp
* Creation timestamp

1. **Token Refresh Flow**

**Automatic Refresh Process**

* Middleware intercepts all non-public routes
* Checks for valid access token
* If access token is invalid/expired:
  + Attempts to use refresh token
  + Makes request to /api/auth/refresh
  + Gets new access and refresh tokens
  + Updates cookies automatically
  + Continues with original request

**Refresh Token Endpoint (/api/auth/refresh)**

* Validates existing refresh token
* Verifies token exists in database
* Checks token hasn't expired
* Generates new access and refresh tokens
* Deletes old refresh token from database
* Saves new refresh token
* Sets new tokens in cookies

**Security Measures in Refresh Flow**

* Refresh tokens are single-use (rotated after refresh)
* Database validation of refresh tokens
* Automatic cleanup of expired tokens (daily at 3am, cron job)
* HTTP-only cookies prevent XSS attacks
* Token rotation on each refresh

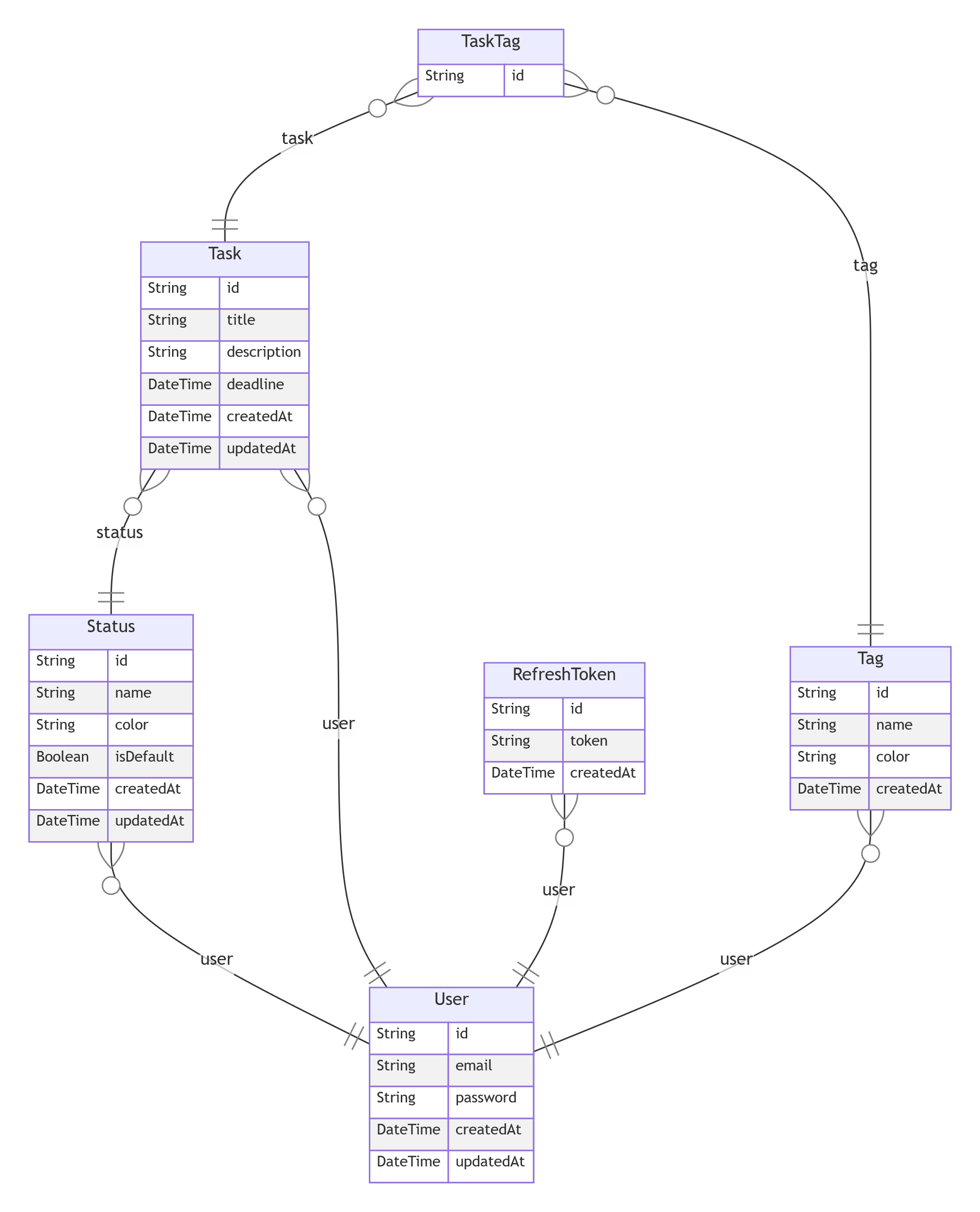
This implementation provides a robust and secure authentication system with automatic token refresh, protecting against common security vulnerabilities while maintaining a good user experience.

**Data storage subsystem**

**Responsibilities**

* Persist user data including authentication information and preferences
* Store tasks with their associated metadata (title, description, deadlines)
* Manage task statuses with customizable colors and default states
* Handle task tagging system for better organization
* Maintain user sessions through refresh tokens
* Support user-defined statuses and tags with unique constraints per user

**Schema Overview (Prisma)**

****

**User Management**

* User model serves as the central entity with relationships to all other models
* Stores essential user information (email, password)
* Maintains creation and update timestamps
* Links to user's tasks, statuses, tags, and refresh tokens

**Task Management**

* Task model represents individual tasks with:
* Basic information (title, description)
* Status tracking through Status relation
* Optional deadline
* Tagging capability through TaskTag junction
* User ownership and timestamps

**Status System**

* Status model enables customizable task states:
* User-defined status names
* Optional color coding
* Default status flag
* Unique status names per user
* Links to associated tasks

**Tagging System**

* Tag model for task categorization:
* User-defined tag names
* Optional color coding
* Default tag flag
* Unique tag names per user
* TaskTag junction model for many-to-many relationships between tasks and tags

**Authentication**

* RefreshToken model for session management:
* Secure token storage
* Expiration tracking
* User association
* Indexed for efficient querying

**Data Relationships**

* **One-to-Many:**
* User → Tasks
* User → Statuses
* User → Tags
* User → RefreshTokens
* Status → Tasks
* **Many-to-Many:**
* Tasks ↔ Tags (through TaskTag)

**On-disk Format**

* MongoDB stores data as JSON-like documents
* Prisma handles data mapping from MongoDB to TypeScript types
* Data is persisted across restarts
* Indexes are maintained for efficient querying:
* User lookups
* Status filtering
* Task retrieval
* Tag associations
* Token expiration checks

**Security Features**

* Password hashing (handled at application level)
* Unique constraints on user emails
* User-scoped data access
* Secure token storage
* Timestamp tracking for auditing

**Performance Considerations**

* Indexed fields for common queries
* Efficient relationship mapping
* Optimized data structure for MongoDB
* Proper use of MongoDB ObjectId for IDs

**Business Logic**

The application's core functionality revolves around task management with a focus on user-specific customization. Users can create and manage tasks, each associated with a status and optional tags. The system enforces user ownership of all data, ensuring users can only access and modify their own tasks, statuses, and tags.

**Task Management**

Tasks are the central entity in the system. Each task requires:

* A title
* A valid status
* User ownership
* Optional description, tags, and deadline

Tasks can be moved between statuses through drag-and-drop operations, with the system validating ownership of both the task and target status. When a task is moved, the system updates its statusId.

**Status System**

Statuses are user-defined columns that represent different stages of task completion. Each status has:

* A unique name per user
* Optional color coding
* Default status flag
* User ownership

The system ensures at least one status is always marked as default, and users can customize their status columns to match their workflow needs.

**Tagging System**

The tagging system provides a flexible way to categorize tasks. Tags are user-defined and can be:

* Applied to multiple tasks
* Colored for visual distinction
* Removed from tasks

**Data Access and Security**

All data operations are scoped to the authenticated user. The system:

* Verifies ownership before any modification
* Prevents unauthorized access
* Maintains data integrity
* Optimizes query performance

**Error Handling**

The system handles various error scenarios:

* Invalid input validation
* Authorization checks
* Business rule enforcement
* Duplicate prevention

**HTTP frontend**

The frontend of our application is built using Next.js, providing a modern, responsive interface for task management. The application follows a clean architecture where the frontend interacts exclusively with the business logic layer through well-defined API endpoints.

**Core Responsibilities**

The frontend layer is responsible for converting HTTP JSON requests into internal function calls and rendering the UI based on API responses. It maintains a clear separation of concerns by interacting only with the business logic layer, never directly with the data storage.

**Main Pages**

**Authentication Pages**

The /login and /register pages handle user authentication. The login page provides a clean interface for existing users to access their workspace, while the registration page allows new users to create an account. Both pages include form validation and error handling to ensure a smooth user experience.

**Main page**

The main page features a flexible view system that allows users to view their tasks in a Kanban-like board. It displays tasks organized by their status columns, supporting drag-and-drop operations for task management.

**Task Management**

The /task/[id] page provides a detailed view of individual tasks. This page allows users to:

* View and edit task details
* Modify task status
* Manage task tags
* Update deadlines
* View task history

**Technical Implementation**

The frontend is built with modern web technologies:

* Next.js for server-side rendering and routing
* React for UI components
* Tailwind CSS for styling
* TypeScript for type safety

The application implements proper error handling, loading states, and optimistic updates to provide a responsive user experience. All API calls are properly typed and handled through custom hooks, ensuring consistent data management across the application.

**Alternatives Considered**

**Alternative 1: Firebase Auth + Firestore**

We considered using Firebase for authentication and data storage, which would have provided a fully managed backend solution. However, this approach was rejected because:

* It would have limited our control over the authentication flow and data structure
* The learning goal was to gain experience with Prisma and MongoDB
* We wanted to maintain full control over our data and security

**Alternative 2: Separate Backend (Express)**

A traditional approach with a separate Express backend was evaluated. This was not chosen because:

* Next.js API routes provide sufficient functionality for our backend needs
* A separate backend would increase deployment complexity
* The current architecture reduces the number of moving parts
* Next.js provides good performance and developer experience

**Alternative 3: PostgreSQL**

We evaluated PostgreSQL as an alternative to MongoDB. While PostgreSQL is a robust relational database, we chose MongoDB because:

* MongoDB's flexible schema better suits our dynamic status system
* The document-based structure aligns well with our task management needs
* The schema-less nature allows for easier modifications to the status system

**Additional Considerations**

We also evaluated other alternatives that were quickly dismissed:

* Using a different frontend framework (like Vue.js) - Next.js was chosen for its React integration and server-side rendering capabilities
* Implementing a microservices architecture - deemed unnecessary for the current scale
* Using a different authentication provider - custom implementation was chosen for learning purposes

The final architecture choice balances learning goals, development efficiency, and future scalability while maintaining a clean and maintainable codebase.