**Technical Decision Log for Journal Application**

### **Decision: Choosing the Authentication Strategy (JWT vs Session-Based Authentication)**

#### **Problem:**

We needed to decide how to handle user authentication for the Journal application. The system requires secure login and session management, while also supporting scalability and performance under load.

#### **Options Considered:**

* **JWT (JSON Web Tokens)**: A token-based authentication mechanism where the server generates a token containing user information after successful login. The token is sent with each request, which allows the system to be stateless.
* **Session-Based Authentication**: A traditional method where user sessions are stored on the server side. Each session is assigned to a unique ID, and the server keeps track of active sessions.

#### **Chosen Approach and Rationale:**

We chose **JWT authentication**. JWT allows for stateless authentication, meaning the server does not need to store session data, which simplifies scaling and reduces server-side memory usage. It is more efficient in cloud-based, distributed systems where horizontal scaling is important. JWT also supports mobile applications and third-party integrations easily.

#### **Trade-offs and Consequences:**

* **Trade-off**: Managing JWT tokens requires additional security measures like token expiration and refreshing mechanisms. If a token is compromised, an attacker has access until it expires.
* **Consequences**: JWT provides a more scalable and lightweight solution, but it necessitates careful implementation of token expiration and revocation mechanisms to maintain security.

### **Technical Decision Log: Choosing Next.js for Full-Stack Development**

#### **Problem:**

We needed to choose a backend framework or a full-stack framework to build the Journal application. The decision had to take into account the need for both frontend and backend development, the need to support server-side rendering (SSR), and the ability to scale as the application grows.

#### **Options Considered:**

1. **Next.js** (Full-Stack Framework):  
   * **Overview**: Next.js is a full-stack React framework that supports both frontend and backend functionality. It offers features like server-side rendering (SSR), static site generation (SSG), and API routes. It allows developers to handle both frontend rendering and backend API logic within the same application, which reduces the need for a separate backend.
2. **Express.js with React** (Separate Backend):  
   * **Overview**: Express.js is a minimal and flexible Node.js web application framework that can be used as the backend to manage API routes and serve static assets, while React is used for frontend rendering. In this setup, the frontend and backend would be separate, meaning managing API routes and UI components would require separate deployments and configurations
3. **Django/Flask (Python Frameworks)**:  
   * **Overview**: Django and Flask are both popular Python web frameworks. Django offers more features out of the box (like ORM, admin panel, etc.), whereas Flask is more minimalistic and flexible. Both frameworks would require a separate frontend (React or other frameworks), which increases complexity and deployment overhead.

#### **Chosen Approach and Rationale:**

We chose **Next.js** for the following reasons:

1. **Full-Stack Capabilities**:  
   * Next.js allows us to write both frontend and backend code in the same repository, which reduces complexity. With API routes integrated into the framework, we can define backend logic (like handling authentication, interacting with the database, managing journal entries) directly alongside the frontend code.
2. **Server-Side Rendering (SSR)**:  
   * One of the key features of Next.js is its ability to handle server-side rendering, which improves the performance and SEO of the Journal application. SSR allows pages to load quickly and ensures that search engines can index dynamic content properly, like journal entries and user profiles.
3. **Optimized for React**:  
   * Since we are using React for the frontend, Next.js integrates seamlessly with it. This minimizes the complexity of managing two separate codebases (frontend in React and backend in Node.js) and provides a more streamlined development experience.
4. **Automatic Code Splitting and Optimized Performance**:  
   * Next.js automatically splits the code into smaller chunks and optimizes loading performance. This is important for the Journal application, which may have a large number of journal entries and images, as it ensures users only download the code they need for each page.
5. **Scalability and Developer Experience**:  
   * Next.js comes with built-in tools to optimize performance, such as image optimization and automatic static generation for common pages. It also provides good development experience with features like hot reloading, a powerful routing system, and a solid set of pre-configured tools to work with modern JavaScript/TypeScript.
6. **Simplified Deployment**:  
   * With Next.js, deployment becomes simpler since we only need one application to manage both frontend and backend logic. This is especially advantageous in a cloud environment (such as Vercel or AWS), where deployment processes are often optimized for full-stack applications. Additionally, Next.js is optimized for serverless environments, which makes it easy to scale without needing separate deployments for frontend and backend.

#### **Trade-offs and Consequences:**

* **Trade-off**: One potential downside of using Next.js as a full-stack framework is that it might not be as flexible as choosing a separate backend like Express.js or Django. For example, Next.js is not as feature-rich as Django when it comes to handling complex database relationships, admin panels, or other backend-specific features. However, these features can be integrated with external tools or APIs if necessary.
* **Consequences**: Since Next.js combines frontend and backend, developers need to be proficient in both the frontend and backend parts of the application. This could lead to more complex codebases in some cases, though the overall experience remains simplified with the full-stack nature of Next.js. Additionally, if the app grows very large in terms of features and traffic, certain aspects of Next.js (such as API routes) might need optimization or rethinking.