ChatGPT

Password Manager System Design

Technology Stack (Recommended)

For the **web dashboard front end**, use a modern JavaScript framework (e.g. **React** or **Vue**) with TypeScript for safety and maintainability. Frameworks like React have become standard for rich interactive UIs [1]. You may use Next.js (React) or Nuxt (Vue) for server-side rendering if needed for SEO, but a single-page app (SPA) with a robust UI library (e.g. Material-UI or TailwindCSS) is common. For **state management**, Redux or Context API (React) or Vuex (Vue) can be used.

On the **backend**, choose a scalable, well-supported platform. Popular choices include **Node.js** (Express or NestJS) or **Python** (Django/DRF or FastAPI) or **.NET Core (C#)**. As of 2025, Node.js remains popular for event-driven APIs and has an enormous ecosystem [2]. Python/Django or Flask are also mature, and .NET Core (C#) is used by some password managers (e.g. Bitwarden's server is built on .NET). All have libraries for web APIs and crypto. Using TypeScript on Node.js (e.g. NestJS) can help maintain type safety.

The **browser extension** should be implemented as a WebExtension (compatible with Chrome and Firefox). It typically involves a **background script**, **content scripts**, and a **popup/options UI** [3]. The extension UI can even be built with React/TypeScript and then bundled into the extension, as React is commonly used in extensions [4]. The extension will communicate with the backend API over HTTPS (using fetch or XMLHttpRequest) to sync data.

For the **database**, a relational store like **PostgreSQL** or **MySQL** is suitable (for ACID compliance and complex queries), although a document store (e.g. MongoDB) could also work. If you need flexible schema for vault items, a JSONB column in PostgreSQL or a NoSQL DB is fine. The database should support encryption-at-rest (e.g. AWS KMS integration or built-in transparent encryption). Session data or caching can use **Redis**.

Finally, for **mobile apps** (future): develop cross-platform apps with **React Native** or **Flutter** (both allow one codebase for iOS/Android) or native iOS (Swift) and Android (Kotlin). These should use the same backend API and security model, carrying over the encryption and sync logic.

Key Features

- **Core Features (Individual Users):** Securely store credentials (logins, credit cards, notes) in an encrypted vault. Automatic password capture (when logging into a site) and **autofill** of forms via the browser extension. Password generator for strong passwords. Search and organize entries with folders/tags. Import/export from/to CSV or other managers. Sync vault data across devices (via server).
- **User Profile & Settings:** Master password management, PIN/biometric unlock (on mobile), personal profile. Vault health report (identifying weak or reused passwords). Password change reminders.
- **Advanced Features:** Built-in **TOTP (Time-based One-Time Password)** generator (store 2FA codes). Secure sharing of items (e.g. share a login with another user or between your accounts, encrypting with the recipient's key). Breach monitoring (alert if stored credentials appear in

known breaches). “Emergency access” feature to grant a trusted contact access. Secure notes and file attachments (encrypted). Support for multiple accounts/profiles. Optional two-factor methods (push notifications or hardware keys).

Security Mechanisms

- **Zero-Knowledge End-to-End Encryption:** All sensitive data (passwords, notes, etc.) must be encrypted on the client side *before* sending to the server [5]. For example, Bitwarden encrypts vault data locally with AES-256 immediately upon entry [5]. The encryption key is derived from the user’s master password (and email) using a strong KDF. This ensures the server stores *only encrypted blobs* and never has the decryption key, so it has “zero knowledge” of plaintext data [6] [7]. Store only a salted hash of the master password on the server for authentication.
- **Cryptography:** Use modern algorithms and sufficient parameters. For vault encryption, use **AES-256-GCM** (authenticated encryption). Derive encryption keys from the master password using Argon2id (current recommended KDF) with high memory/work factors [8]. (OWASP recommends Argon2id with $\geq$19 MB memory, iter=2 as a minimum [8].) This thwarts brute-force attacks on the master password. Each vault item or vault itself should use a unique IV/nonce. Consider adding an HMAC (or using AEAD mode) for integrity. Never store master passwords or raw keys on the server.
- **Authentication & 2FA:** Use a secure authentication flow. After login (verifying the hashed password), issue session tokens (JWT or server-side sessions) over HTTPS. Require **multi-factor authentication** (2FA) on login: support TOTP (RFC6238) apps, email/SMS OTP, and especially hardware/WebAuthn (FIDO2) like YubiKeys for phishing-resistant login. Optionally allow login via third-party SSO (OAuth/OIDC) for enterprise. Protect the login API with rate-limiting and device fingerprinting.
- **Secure Development:** Follow OWASP best practices. Enforce HTTPS/TLS everywhere; use secure cookies and proper CORS. Implement Content Security Policy (CSP) to prevent XSS. Sanitize all inputs, use prepared statements to prevent injection. Apply strict CSRF protections on the web app. Use helmet or similar middleware. Keep dependencies updated and perform regular security audits (e.g. third-party code review, penetration tests). Maintain a bug bounty program.
- **Client Security:** On the client (browser and mobile), use secure coding practices. The extension should use the browser’s extension APIs and least-privilege permissions (e.g. only access specific pages, minimal host permissions). Consider storing a local cache of the vault encrypted in the extension (IndexedDB) or mobile app (secure storage/keystore) so users can use passwords offline; decrypt it in-memory only. The extension should lock after inactivity. On mobile, use device biometrics/PIN and secure enclave/keychain to store an encryption key.
- **Zero Trust Mindset:** Minimize trust even in infrastructure: use IAM roles with least privilege, encrypt backups, segregate admin interfaces, rotate keys, and enforce network segmentation internally.

System Architecture

The system follows a **client-server architecture** [9]. On the client side are:
- **Web App (Dashboard):** Runs in the browser as an SPA. It communicates with backend APIs for user management (auth, profile), vault operations (fetch/update passwords), settings, etc. Data displayed is decrypted in the browser.
- **Browser Extension:** Installed in Chrome/Firefox, it consists of background scripts (for handling sync and messaging), content scripts (injected into web pages to perform autofill and capture credentials), and a popup/options UI. It connects to the same backend APIs to sync vault data, and interacts with the web page DOM to fill login forms.

- **Mobile/Desktop Apps:** (Future) Native or cross-platform apps that also use the APIs; they keep a local encrypted vault and sync with the server.

On the server side are stateless **API services** behind a load balancer:
- **Authentication Service:** Verifies login credentials (compares hashed passwords), issues JWTs or session tokens, and handles 2FA verification. It uses HTTPS and can integrate SSO/identity providers.
- **Vault Service:** Handles CRUD operations on vault items. It stores only encrypted payloads in the database. When a client saves or updates data, it sends the encrypted blob to this service to store under the user's account ID.
- **Other Microservices (Optional):** For advanced features, there may be separate services for sharing, auditing, breach checking, etc. For example, an audit-log service could record metadata of operations (timestamps, IPs) in a write-once log. A sharing service could manage public keys for sharing credentials securely between users.

APIs: Use a RESTful JSON API (or GraphQL) over HTTPS. Design endpoints like `/api/auth/login`, `/api/vault/items`, etc. Ensure every request is authenticated (via token) and authorized (user only accesses own data). Follow REST security guidelines (e.g. OWASP REST security cheat sheet [10]).

Data Flow: On login, the client derives the vault encryption key (using the master password). When saving a credential, the client encrypts it and sends it to `/vault/save`. The server writes the encrypted data to the DB. When fetching data, the client retrieves the encrypted items and decrypts them locally. This ensures that secret data never travels or resides in plaintext on the server.

Database Schema: A typical design has tables like `Users` (id, email, password_hash, settings) and `VaultItems` (id, user_id, type, encrypted_data, timestamp). The `encrypted_data` column is a blob or JSON that contains the ciphertext of the password/notes (and any per-item metadata). You may encrypt individual fields (e.g. URL, username, password) separately. Ensure the DB is encrypted at rest.

Browser Extension Architecture: The extension will use the WebExtension model [3]. For example, a background script maintains the current decrypted vault in memory (or refetches it) and listens for browser events. Content scripts run on login pages: when a user visits a login form, the extension matches the domain to a stored credential and autofills by injecting into the form. When the user logs in to a new site, the content script can capture the site, username, and password fields and send them (securely) to the background script, which prompts the user to save them. The popup UI (action) can be a React component letting the user search/view the vault.

Overall, the system is multi-tiered: the presentation (web/extension/mobile) layer, the API/microservices layer, and the data/storage layer, all communicating over secure channels. The **client-server split** allows access from multiple devices and central sync [9]. One trade-off noted is that developing both client and server adds complexity [9], but it is necessary for cross-device sync.

Hosting & Deployment

- **Cloud Infrastructure:** Use a major cloud provider (AWS, Azure, or GCP) for reliability and scalability. For example, AWS with EKS (Kubernetes) or ECS (Docker containers) for the application services, and RDS (PostgreSQL) for the database.
- **Containers & Orchestration:** Containerize services with Docker. Deploy on Kubernetes (EKS/GKE/AKS) or serverless containers (AWS Fargate) for auto-scaling. This simplifies updates and horizontal scaling.

- **CI/CD Pipeline:** Implement continuous integration and deployment (e.g. GitHub Actions or Jenkins). Run automated tests (security linters, unit/integration tests) on each commit. Use IaC (Infrastructure as Code) like Terraform or CloudFormation to provision resources reproducibly.
- **Network & Security:** Place servers in private subnets, use load balancers (ALB/NLB) for the API with TLS termination (HTTPS). Obtain TLS certificates via a managed service (AWS Certificate Manager or Let's Encrypt). Use a Web Application Firewall (WAF) and DDoS protection (e.g. AWS Shield) at the edge.
- **Monitoring & Logging:** Centralize logs (e.g. Elasticsearch/Logstash/Kibana or CloudWatch). Monitor performance (Prometheus, Grafana) and security alerts. Implement audit logging for sensitive actions.
- **High Availability:** Use multi-AZ database deployments and deploy services in multiple availability zones. Regularly backup the database (encrypt backups). For global scale, consider multi-region deployments with geo-redirects.
- **Compliance:** Plan for GDPR (encrypt personal data), SOC2/ISO27001 (audit trails, policies) especially if expanding to enterprise.
- **Deployment of Extensions:** For the browser extension, package and submit to Chrome Web Store and Mozilla Add-ons with an automated release process.

Scalability and Future Expansion

- **Horizontal Scaling:** Design servers stateless (no local user data) so you can add more instances behind a load balancer as load increases. Use auto-scaling groups based on CPU or request latency.
- **Database Scaling:** For heavy read loads, use read replicas. For very large user bases, consider sharding or multi-tenancy in the database. Use connection pooling to handle many concurrent users.
- **Caching:** While vault contents are encrypted (and typically small), you can cache user metadata (not sensitive) in Redis or CDN. Cache static assets (web app) on a CDN.
- **Microservices:** As features grow, split into microservices so each can scale independently (e.g. a separate service for handling breach checks or analytics).
- **Rate Limiting & Throttling:** Implement rate limits on login and API endpoints to prevent brute force and abuse, scaling limits per IP or user.
- **Mobile and Desktop Clients:** Future mobile apps should reuse the API layer. Ensure APIs are versioned to allow backward compatibility.
- **Enterprise Features:** When adding enterprise/team features, support Single Sign-On (SAML or OIDC), Active Directory/LDAP integration, SCIM user provisioning, admin dashboards, permission roles, and extensive audit logs. Architect data to support workspaces or organizations (separate vaults per org). You may also offer an on-premises/self-hosted option for enterprises with even stricter controls.
- **Continuous Improvement:** Regularly revisit cryptographic parameters (e.g. increase Argon2 memory cost as hardware improves). Keep libraries up to date.

By combining these modern technologies and security practices, the password manager will be robust, user-friendly, and able to grow from individual users up to enterprise scale. The zero-knowledge, end-to-end encryption model ensures user data remains safe even if the server is compromised [5] [6]. Using well-supported stacks (e.g. React/TypeScript, Node.js, PostgreSQL) ensures maintainability and community support [1] [2].

Sources: We参考ed best practices from industry leaders (e.g. Bitwarden's security whitepaper [5]) and OWASP guidelines [8], and architectural case studies [9] [3] to inform this design.

---

1 2 How to Choose the Best Tech Stack for Web App Development in 2025
https://5ly.co/blog/best-web-app-tech-stack/

3 4 Creating a Chrome extension with React and TypeScript - LogRocket Blog
https://blog.logrocket.com/creating-chrome-extension-react-typescript/

5 6 7 How End-to-End Encryption Paves the Way for Zero Knowledge - White Paper | Bitwarden
https://bitwarden.com/resources/zero-knowledge-encryption-white-paper/

8 Password Storage - OWASP Cheat Sheet Series
https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

9 10 High-Level Design
http://www.cs.ucf.edu/courses/cop4331/fall2014/cop4331-8/hldesign/