ChatGPT

Next-Gen To-Do Application: Requirements Document

This document outlines the **functional** and **non-functional requirements** for a React/Node.js to-do application, as well as an Agile execution plan (user stories and sprints). We will use React.js (with Bootstrap CSS for responsive UI) on the front end and Node.js on the back end, with a PostgreSQL database via Supabase. Google OAuth will handle user authentication, and Supabase's Realtime feature will sync data across devices [1].

Functional requirements define **what the system should do** [2] (features visible to the user), while non-functional requirements define **how well the system operates** (qualities like performance, security, usability) [3]. Below we enumerate all required features and qualities of the app.

Functional Requirements

- **Task CRUD**: Users can create, read, update, and delete tasks. Each task has a *title*, optional *description*, a *due date*, and a *priority level* (e.g. High/Medium/Low) [4] [5].
- **Task Listing & Sorting**: The app lists a user's tasks in a clear order (by deadline or priority). Tasks due sooner or with higher priority should appear first. Completed tasks should be visibly marked (e.g. strikethrough) and not shown in the active list [6]. Users can filter or search tasks by date, priority, or keyword.
- **Task Completion**: A user can mark a task as *completed*; completed tasks are moved out of the active list and visually indicated (e.g. checkmark or strikethrough) [7].
- **Recurring Tasks**: Users can set tasks to repeat on a schedule (daily, weekly, monthly, custom intervals). Recurring tasks automatically re-create themselves after completion or on schedule, reducing manual entry.
- **Reminders and Notifications**: Users can set reminders for tasks. The system will send notifications (push or email) when a task's due date/time approaches. Users can choose how far in advance to be alerted (e.g. 1 hour before deadline).
- **Task Prioritization**: Users can assign a priority level to each task (e.g. High/Medium/Low). The UI uses priority to help users focus on urgent tasks (for example, by highlighting high-priority tasks or sorting by priority [5]).
- **Responsive UI**: The interface must work on all device types (phones, tablets, desktops). We will use Bootstrap's mobile-first, responsive grid system to ensure layouts adapt seamlessly [8] [9]. On small screens, the layout should remain usable and navigation clear.
- **User Authentication**: Users must sign in via Google OAuth 2.0 (handled by Supabase Auth) [10] [1]. This provides secure single-sign-on without building a custom auth system. Only authenticated users can view or modify their to-dos.
- **Real-Time Sync**: All changes to tasks (add/edit/delete/complete) should propagate in real-time across a user's devices. Using Supabase Realtime (Postgres change listening) [1], if a user edits their list on one device, other open devices will see the update instantly.

- **Admin Panel**: A separate admin interface (restricted to admin users) should allow viewing all users and their data. Admins can search users, see counts of tasks per user, and optionally remove or deactivate accounts.
- **Analytics and KPIs**: The admin dashboard should display key metrics (KPIs) such as: total registered users, daily/weekly active users (DAU/WAU), tasks created per day, tasks completed per day, average tasks per user, and similar. Charts or tables should visualize trends. These help measure app usage and growth.
- **Additional Features (optional)**: While not core, useful extras include tagging or categorizing tasks, sub-tasks, and integration with calendars. (These can be planned in later sprints.)

Each functional requirement is validated by user stories and test cases. For example, an agile user story might be: *"As a logged-in user, I can add a new task with a title, description, due date and priority so that I can track that task* [4] [11]*."* The list above captures all planned features, including the advanced ones (recurring tasks, reminders, prioritization, real-time sync, and admin analytics).

Non-Functional Requirements

- **Performance:** The app should be fast and responsive. Typical actions (like loading the task list, or creating a task) should complete within acceptable time (e.g. <1 second for listing tasks) [12] [13]. Backend queries should be optimized and UI rendering efficient.
- **Scalability:** The architecture must support growth in user base and data volume. Using Supabase (Postgres) ensures we can scale the database; the design should allow for horizontal scaling (e.g. stateless Node.js servers behind a load balancer). The system should handle increasing loads (e.g. thousands of users) without major refactoring [13].
- **Security:** User data must be protected. All API endpoints require authentication. Google OAuth (via Supabase) provides secure login [10]. Use HTTPS for all traffic. Implement Row-Level Security (RLS) in Postgres so each user only accesses their own data [1]. Admin endpoints should require an admin role. Sensitive data (like user tokens) should be stored securely. Non-functional requirements explicitly include "Security – protection against unauthorized access" [13].
- **Usability:** The UI must be intuitive and accessible. Use a consistent design system (colors, fonts, layouts) for coherence. Provide clear feedback on actions (e.g. task creation confirmation). The app should include basic guidance (tooltips or a quick "getting started" help). It should support keyboard navigation and follow accessibility best practices (e.g. proper labels, color contrast) to be easy for all users [13].
- **Responsiveness:** As noted, the layout must adapt to all screen sizes (mobile-first). Using Bootstrap (the world's most popular mobile-first framework [8]) satisfies this requirement by design. Testing on various device viewports is required.
- **Maintainability:** The codebase should follow clean architecture: modular React components and a clear Node.js API. We will write unit and integration tests to make future changes safer. Use of TypeScript (optional) or linting tools can improve code quality. This supports "Maintainability – ease of updates and fixes" [13].
- **Reliability & Availability:** The app should have high uptime; deploy on a reliable hosting with monitoring. Use error handling and logging so the system can recover gracefully (e.g. retry on transient failures). Consider uptime targets (e.g. 99.9%).
- **Compatibility:** The app should work on all modern browsers (Chrome, Firefox, Safari, Edge) and not rely on platform-specific features. It should degrade gracefully if a new browser or device appears.

- **Monitoring and Logging:** The system should log key events (errors, user sign-ins, significant failures) and ideally include basic analytics hooks. Admins should be alerted on critical issues.

In summary, the non-functional requirements ensure the to-do app is **efficient, secure, usable, and scalable** [3] [12]. Meeting these "-ilities" is crucial to a quality user experience.

Technology Stack and Architecture

- **Frontend:** React.js with Bootstrap (HTML/CSS/JS). Bootstrap's mobile-first grid makes responsive design straightforward [8] [9]. React components will handle the interactive UI.
- **Backend:** Node.js (e.g. with Express) exposing a RESTful JSON API. The backend will implement the task business logic and enforce authentication/authorization.
- **Database:** PostgreSQL, using Supabase as a managed backend. Every project has a full Postgres database [1]. Supabase also provides ready-made APIs for CRUD and real-time subscriptions.
- **Authentication:** Supabase Auth with Google as the OAuth provider [10] [1]. This simplifies login flows. Users are identified by Google accounts; no password handling is needed in our code.
- **Real-Time Sync:** Supabase Realtime (listening to Postgres changes) will push updates to clients instantly [1]. This avoids building a custom WebSocket server.
- **Deployment:** The frontend can be deployed on a static host (Netlify/Vercel) and the backend on a Node-friendly host (Heroku, Render, or containerized in Docker). Supabase handles the database and auth infrastructure.
- **Additional Tools:** We will use Git for version control, a CI/CD pipeline for testing and deployment, and monitoring tools (like Postgres analytics or Sentry for errors).

This tech stack is fixed by requirements (React, Node, Bootstrap, Postgres/Supabase, Google OAuth) and leverages popular, well-supported technologies. Supabase's integrated features (DB, Auth, Realtime) mean we rely on proven services for core needs [1] [10].

User Stories (Epics)

We will use agile user stories (formatted as "As a *<role>*, I want *<feature>* so that *<benefit>*" [14]). Example stories include:

- **User Authentication**: *As an unauthenticated user, I want to sign in or register via Google OAuth so that I can securely access my tasks* [10].
- **Add Task**: *As a logged-in user, I want to add a new task (title/description/due date/priority) so that I can track something I need to do* [4] [11].
- **View Tasks**: *As a logged-in user, I want to see the list of my to-dos (sorted by deadline/priority) so that I know what tasks I have upcoming* [6] [11].
- **Edit Task**: *As a user, I want to edit an existing task's title, description, or due date so that I can correct or update it* [7].
- **Delete Task**: *As a user, I want to delete a task so that I can remove tasks I no longer need to do* [15].
- **Complete Task**: *As a user, I want to mark a task as complete so that I can distinguish done items from pending ones* [7].
- **Recurring Task**: *As a user, I want to set a task to repeat on a schedule (e.g. every week) so I don't have to re-enter routine tasks each time*.
- **Reminders**: *As a user, I want to set reminders for a task so that I receive a notification before it's due*.

- **Priority Filter**: *As a user, I want to filter or sort tasks by priority so that I can focus on high-priority items first* [5].
- **Real-Time Sync**: *As a user, I want updates on one device to appear immediately on my other devices so that I always see the latest task list* [1].
- **Admin – User List**: *As an admin, I want to view all registered users so that I can manage accounts and monitor usage*.
- **Admin – Analytics**: *As an admin, I want to see metrics (total users, tasks/day, completion rate) so that I can understand app adoption and engagement*.

Each story will have acceptance criteria (e.g., "Given I am on the task list page, when I click 'Add Task' and fill the form, then the new task appears") and will be entered into a sprint backlog. These stories capture all the requirements above and will be broken into tasks during sprint planning.

Sprint Plan (Execution Schedule)

We plan to develop the app in **two-week Scrum sprints**, building an MVP first and then iterating on advanced features. Below is a high-level breakdown:

- **Sprint 1 (Weeks 1–2): Setup & Core Auth** – Initialize the code repository, CI/CD, and project structure. Configure Supabase project (Postgres & Auth) [1] and set up Google OAuth credentials [10]. Implement basic backend endpoints and React scaffolding. Build the Google Sign-In flow so users can authenticate. Prepare the database schema for users and tasks.

- **Sprint 2 (Weeks 3–4): Basic Task CRUD & UI** – Develop task CRUD APIs in Node.js (using Supabase client or REST API). Create React components for: viewing the task list, adding a task, and displaying task details. Integrate Bootstrap to design a responsive task list UI [8]. Enable marking tasks complete, and editing/deleting tasks. Test end-to-end that tasks persist in the database and reflect in the UI.

- **Sprint 3 (Weeks 5–6): Advanced Task Features** – Add task *priority* support (High/Med/Low) with corresponding UI controls [5]. Implement **recurring tasks** logic: allow users to set repetition (daily/weekly) when creating/editing a task, and auto-schedule future instances. Add **reminders/notifications**: let users set a reminder time (e.g. 1 day before due date) and trigger an email or push notification at that time. Enhance filtering and sorting UI (e.g. drop-down to show only high-priority tasks). Continue UI polishing for mobile.

- **Sprint 4 (Weeks 7–8): Real-Time Sync & Polishing** – Integrate Supabase Realtime subscriptions so the frontend listens to task changes and updates instantly [1]. Test multi-device scenarios (e.g. browser and mobile open) to ensure tasks appear in sync. Optimize performance: index important DB columns (due date, user ID), lazy-load lists if needed. Ensure listing tasks meets our performance target [12]. Conduct security review on APIs (verify users cannot access others' data).

- **Sprint 5 (Weeks 9–10): Admin Panel & Analytics** – Build an admin interface (protected by an admin role flag). This can be a special React route (e.g. `/admin`) only accessible to admins (as defined in Supabase claims or a database flag). Admin panel features: a table of all users (name/email, registration date), and for each user a summary of task counts. Add an **analytics dashboard**: use a chart library (or simple stats) to show KPIs like total active users, tasks created per day/week, tasks

completed per day, average tasks per user, and retention metrics. Store these metrics in the database or compute on the fly from logs/tasks.

- **Sprint 6 (Weeks 11–12): Final Testing & Launch** – Perform comprehensive testing: write unit tests for backend logic, and integration tests for API endpoints. Conduct user testing on different devices/browsers to verify responsiveness [8]. Fix any bugs and refine UI based on feedback. Optimize final performance (minify assets, ensure fast queries). Prepare deployment: containerize Node app (if needed), set up environment variables, and publish to production. Review all requirements and ensure each is met. Finally, create documentation (README, API docs, and a brief user guide).

Each sprint will have a **sprint goal** (listed above) and include tasks such as design, development, testing, and review. We will hold sprint planning (to select user stories for the sprint), daily standups, sprint review (demonstration of features), and sprint retrospective. This Agile approach follows guidance that a project should start with clear user stories and proceed iteratively [14] [11].

All sources for this plan are industry best practices: requirements engineering principles [2] [3], examples from to-do app case studies [16] [5], and Agile planning guides [14].

---

[1] Supabase | The Postgres Development Platform.
https://supabase.com/

[2] [3] [13] Functional and Non Functional Requirements - GeeksforGeeks
https://www.geeksforgeeks.org/software-engineering/functional-vs-non-functional-requirements/

[4] [6] [7] [12] [15] [16] Designing a To-Do List Application | by Chakrin Deesit | Medium
https://medium.com/@ckn.deesit/designing-a-to-do-list-application-ffcbe80f2fdf

[5] 7 best to do list apps of 2025 | Zapier
https://zapier.com/blog/best-todo-list-apps/

[8] Bootstrap · The world's most popular mobile-first and responsive front-end framework.
https://getbootstrap.com/docs/3.3/

[9] Grid system - Bootstrap
https://getbootstrap.com/docs/4.0/layout/grid/

[10] Login with Google | Supabase Docs
https://supabase.com/docs/guides/auth/social-login/auth-google

[11] [14] 10 Steps to Plan Better so you can Write Less Code
https://www.freecodecamp.org/news/10-steps-to-plan-better-so-you-can-write-less-code-ece655e03608/