# Personal Time Tracker Web Application – Detailed Requirements

## 1. Project Overview and Goals

The Personal Time Tracker is a simple web application that helps individuals log and visualize how they spend their time throughout the day. The goal is to promote productivity and mindful time usage by providing a clear record of activities and insightful reports. Users can input activities they engage in (work, sleep, exercise, etc.), and the app will present daily and periodic summaries in an easy-to-understand format. By reflecting on personal habits through these reports, users can identify where their time goes and make adjustments to improve their routines.

**Key Objectives:** - Allow users to easily record activities with time spans (start and end times) and categories. - Provide visual summaries (charts and stats) of time spent on different activities for various time ranges (daily, weekly, monthly, yearly). - Encourage self-improvement with data-driven insights (e.g. seeing proportions of time spent in each category). - Keep the user experience quick, clean, and accessible with modern web technologies and no complex setup (no login required for the MVP).

## 2. Scope of Work

### 2.1 In-Scope (MVP Features)

* **Activity Logging:** Users can add new time entries by specifying an activity name, choosing a category (predefined or custom), and providing start and end times (the app can calculate duration). Entries can be edited or deleted as needed.
* **Dashboard & Statistics:** A landing page (dashboard) shows an overview of the user’s time distribution. It includes:
* A summary of total time spent per category for *today* by default.
* Options to view stats for other ranges: **Today, This Week, Last 7 Days, This Month, This Year** (and possibly all-time). The dashboard should update to reflect the selected range.
* Visual charts (bar charts, pie charts, etc.) illustrating the breakdown of time. For example, a pie chart showing percentage of time spent in each category today, or a bar chart for the past week’s daily totals.
* An inspiring quote or interesting stat (optional nice-to-have) to keep users motivated (e.g. “You spent 8 hours on Productive tasks this week!”).
* **Activity Management:** A dedicated page or section to add activities (the "Add Activity" form) and to view/manage existing entries. This includes:
* Form fields to input **Activity Name**, **Category**, **Start Time**, **End Time**, and **Notes** (notes are optional). The difference between end and start will determine the duration.
* Category field supports both **predefined categories** (e.g. Work, Health, Sleep, Exercise, Leisure, etc.) and **user-defined categories**. Users can select from a list of common categories or type a new category name to create a custom one.
* The user’s local time zone is used for recording and displaying times, ensuring the timeline reflects their actual day. Internally, timestamps may be stored in UTC but will be converted to local time in the UI.
* After adding an entry, the new activity should be saved to the database (or local storage for anonymous usage) and the UI should update (the entry list and dashboard stats should reflect the change in real-time).
* A list or table of existing entries (for the current day or selected range) with options to **edit** or **delete** each entry. Editing an entry opens it in the form for modification; deleting removes it after a confirmation prompt. This ensures users can correct mistakes or remove an activity if needed.
* **Responsive UI:** The interface must be mobile-friendly and desktop-friendly. On all devices, it should maintain clarity and ease of use:
* Use a clean, modern design with soft, neutral colors and highly readable fonts.
* Layout should adjust for smaller screens (e.g., collapsible navigation, charts resizing or stacking vertically on mobile, etc.).
* **Data Persistence:** All activity data is stored persistently using a database (Supabase PostgreSQL). Even without user accounts, the data entered should not be lost on page refresh:
* In MVP, no user login is required. Data can be stored either in local storage (browser) or in the cloud database under an anonymous user context.
* The application will utilize Supabase as the backend-as-a-service to store and retrieve entries via its API, so that data can persist across sessions on the same device.

### 2.2 Out of Scope (Future Enhancements)

The following features are acknowledged as valuable but will **not** be implemented in the MVP: - **User Authentication & Profiles:** No sign-up or login in the MVP. In the future, each user could have an account to sync data across devices and ensure privacy. - **Data Export:** Ability to export the tracked time data to CSV or other formats for external analysis. - **Notifications/Reminders:** Automated reminders to log activities or alerts when spending too long on one activity. - **AI-generated Insights:** Intelligent analysis of habits (e.g., *“You spent 30% of your time on Health last week, which is 10% more than the week before.”*) and personalized advice. - **Collaborative or Team Features:** The MVP focuses on personal tracking. Sharing data or tracking team projects is not included. - **Advanced Calendar Features:** While a basic calendar view is considered (see Phase 2), advanced scheduling (recurring activities, drag-and-drop scheduling, conflict detection for overlapping events) is not in MVP.

These future features should be kept in mind (to avoid architectural dead-ends), but they will not be part of the first release.

## 3. Tech Stack and Platform

The application will be built with a modern web stack, leveraging a serverless and cloud-hosted approach for ease of deployment and maintenance.

* **Frontend:** HTML5, CSS3, and JavaScript, using **React** (preferred library for building the UI). React offers component-based architecture, a rich ecosystem, and efficient state management for dynamic UIs.
* UI styling can be done with plain CSS or a CSS-in-JS solution or utility framework (for example, using Tailwind CSS or styled-components for convenience, though this is up to the implementation).
* For charts and visualizations, a library like **Chart.js**, **Recharts**, or **D3.js** can be used to create bar and pie charts easily within React.
* **Backend:** **Supabase** (which provides a Postgres database, RESTful API endpoints, and client libraries). Supabase will handle data storage and basic CRUD operations via its auto-generated APIs. We will use Supabase’s JavaScript client in the React app to perform database operations. No custom server is needed for the MVP (the app is essentially a static frontend that communicates directly with Supabase).
* Supabase being built on PostgreSQL allows us to define schema, use SQL, and even take advantage of real-time subscriptions or Row-Level Security if needed for future auth.
* **No separate application server** is planned in MVP – logic will reside in the client and in Supabase (via database functions or security rules as needed).
* **Database:** PostgreSQL (hosted by Supabase). The database will store time entries and categories. It offers relational integrity (we can link entries to categories, etc.) and the flexibility to run analytical queries if needed for reports.
* **Hosting & Deployment:** The app code will be hosted on **GitHub** and deployed via **Vercel**.
* Continuous Deployment is set up: any push to the main branch triggers Vercel to automatically build and deploy the latest version of the app. This ensures a smooth CI/CD pipeline with minimal manual steps.
* Vercel provides hosting for the React app (as a static or Next.js app, etc.), and environment variables (like Supabase keys) will be configured there securely.
* **Version Control:** All source code will be managed in a GitHub repository, facilitating collaboration and version tracking. Issues and feature branches can be used to manage future enhancements.
* **AI Assistance (Optional in Development):** OpenAI Codex or GitHub Copilot may be used during development to speed up coding (generating boilerplate, suggesting functions), but the core logic and final code will be reviewed and maintained by the developer.

## 4. Features and Functional Requirements

### 4.1 Dashboard (Landing Page)

When the user opens the app, they are greeted with a **Dashboard** that provides a quick overview of their time usage. This page is crucial for giving instant feedback and insights to the user about their day or other time period.

**Components of the Dashboard:** - **Time Range Selector:** By default, show statistics for **Today** (the current date). Provide easy options (tabs or a dropdown) to switch the view to: - Today – statistics from 00:00 to 23:59 of the current day. - This Week – aggregate stats for the current week (e.g. Monday through Sunday, or Monday up to current day if the week is still in progress; define week start as Monday by default). - Last 7 Days – a rolling window of the past 7 days (e.g. today and the 6 days prior). - This Month – aggregate stats for the calendar month to date. - This Year – aggregate stats for the year to date. - (Potentially All Time or a custom range in the future, but the above are the main presets for MVP.) - **Summary Statistics:** Display total time tracked and breakdown by category for the selected range. For example: - *“Total time logged: 9 hours 30 minutes”* (for the period selected). - A list or grid showing each category and the sum of hours: e.g., **Work: 4h, Health: 2h, Sleep: 6h, Exercise: 1h, Others: 30m** (these are examples). - Possibly highlight the top category (*“Most time spent on: Work”*) or the number of activities logged. - **Charts:** Visual representations of the data: - **Pie Chart** – showing the proportion of time each category took in the selected period. This gives a quick sense of distribution (e.g., a pie where the slice for "Sleep" might be 30% if 30% of the selected time range was spent sleeping). - **Bar Chart or Line Chart** – showing trends over time. For example, a bar chart with each day of the last week on the x-axis and total hours logged that day on the y-axis, or a bar per category displaying hours. We could have: - A bar chart of daily totals (especially for weekly view or last 7 days view). - Alternatively, a stacked bar chart for the week where each bar is divided by category colors. - The exact chart types can be refined during implementation, but the requirement is to have at least one **time-series or comparative chart** and one **distribution chart**. - Charts should be color-coded by category (e.g., Work = blue, Sleep = green, etc., consistent colors across charts). - **Daily Activity List (optional on dashboard):** For the “Today” view, it could be useful to list today’s activities in chronological order (with times and durations). This can help the user quickly verify what has been logged. For other ranges, a condensed list or summary might be shown (e.g., for week view, perhaps nothing or just a note “See calendar for detailed entries.”). This is a nice-to-have; the primary focus is on summary stats and charts. - **Motivational Quote or Stat (optional):** An area on the dashboard could show a rotating motivational quote about time management or a personal stat to encourage the user. Examples: - Quote: “Time is what we want most, but what we use worst.” – William Penn. - Stat: “You have logged 50 hours this month. Great job on staying mindful of your time!” - This is not a core functionality, so it can be implemented if time permits or left for future polish.

**Behavior:** - The dashboard updates in real-time. If the user adds or edits an activity on another page (or modal), and then returns to the dashboard (or if the dashboard is active), the statistics and charts reflect the changes without requiring a page refresh. This can be achieved by re-fetching data after changes or using a real-time subscription from Supabase. - If no data is available for the selected range (e.g., a new user who hasn’t added anything today), the dashboard should handle this gracefully: - Show an empty state message like “No activities logged yet for this period. Start by adding an activity!” and prompt the user toward the add activity form. - Possibly show default placeholder chart or hide the chart section when no data is present. - The dashboard page also likely contains navigation (e.g., a header with an “Add Activity” button or link to go to the activity entry page, and a toggle to switch to Calendar view if implemented). The navigation should be simple since the app is small (for example, a top bar with the app name and two links: *Dashboard* and *Add Activity*, plus maybe *Calendar* if applicable).

### 4.2 Activity Entry & Management Page

This is the form where users input their activities. It’s the core data entry interface of the application, designed to be minimal and user-friendly so that logging time takes only a few seconds.

**Add Activity Form:** - **Activity Name:** A short description of the activity. Example inputs: "Breakfast", "Meeting with client", "Gym workout", "Sleeping". This is a free-text field (max length perhaps 100 characters for sanity). It’s required to fill in. - **Category:** A dropdown or combo-box field where the user categorizes the activity. This field serves to group activities for summary: - The app will provide a set of **predefined categories** by default, such as: **Work, Health, Sleep, Leisure, Exercise, Study, Travel, Personal, Other** (this list can be refined, but these cover common areas). - The user can choose one of these or type a new category name to create a **custom category**. If a new category is entered, the system should add it to the list for future use (persist it for that user). - *Implementation detail:* For MVP, custom categories can be stored either in a dedicated table or simply recognized from existing entries. The recommended approach is a separate **Categories** table, where default categories are pre-seeded and user-added categories are inserted with an association to that user (so others don’t get them, though in an anonymous scenario all users might technically see all if not isolated – see Section 5 on Data for more on handling this). - Category selection is required. If the user doesn’t choose one, the form should prompt (maybe default to "Other" if unspecified). - **Start Time & End Time:** Date-time pickers or input fields for when the activity started and when it ended. - The user should be able to pick a date and time for each. By default, if adding an activity on the same day, the date can default to today. - For convenience, the UI might allow quick selection for common durations (e.g. buttons like +15 min, +30 min when setting end time after picking start) – not required, but a possible enhancement for UX. - Both start and end are required to log a complete entry. (Alternatively, the form could allow entering a start time and a duration instead of end time, but internally both will be recorded. MVP will use explicit start and end times.) - The app should ensure **End Time is after Start Time**. Basic validation: if a user picks an end time that is before the start, show an error. - **Duration Calculation:** The difference between end and start will be the duration. This can be automatically displayed as feedback (e.g., “Duration: 1 hour 30 minutes”) once both times are selected. The duration will also be stored or computed for reporting purposes. - Timezone considerations: The input times correspond to the user’s local timezone (the app should use the browser’s locale or a setting to interpret the times). In the database, times can be stored in UTC or with timezone info; on retrieval, they will be converted back to local for display. This ensures that if the app is used across time zones (in future with accounts), the data remains consistent. - **Notes:** An optional text area for any additional details about the activity. For example, a user might log "Work: Project X" as activity with a note "Focus on feature Y implementation". Notes can be kept short or up to a few sentences (maybe limit to 255 characters or so). This field can be left blank. - **Submit Button:** “Add Activity” (or just "Save") – saves the entry. On submit: - Validate the inputs (all required fields present, times make sense). - Write the new entry to the database (via Supabase API call). If using local storage for MVP data, then save in local storage as well (though ideally Supabase is used for persistence). - Update the UI: perhaps clear the form for a new entry, and give feedback like a toast message “Activity added!”. The new entry should then appear in the list of activities and the dashboard stats should update.

**Viewing & Editing Entries:** - Below or alongside the form, the page should show a **list of activities logged**. This can be filtered by date: - It makes sense to show recent entries or entries for today by default (since one usually adds and might review today’s entries). Alternatively, show all entries with a scroll or allow filtering by date. - Each entry in the list can display: **Activity name, category, start time – end time, duration,** and perhaps an icon or indicator of category (like a color dot). - **Edit:** Next to each entry, an edit icon/button allows the user to modify the entry. Clicking edit could populate the form above with that entry’s data (switching the form into “edit mode”), so the user can change any details and then save. After editing, the changes are written to the database and the list/dashboard refresh accordingly. - **Delete:** Next to each entry, a delete icon/button allows removing the entry. For safety, ask for confirmation (“Are you sure you want to delete this entry?”) before deletion. Once confirmed, the entry is removed from the database and the list is updated (and charts should reflect the removal). - The list should be pageable or lazy-loaded if the user has many entries, but in MVP we can assume a typical user won’t input an extremely large number of activities (and we can optimize later). - Possibly include a **date filter or picker** to view entries from a specific day. If the user wants to see what they did yesterday or any past date, they could select that date and the list updates to show activities from that day. (This is a lighter alternative to a full calendar view, and could be implemented easily with a date picker control and filtering the entries by that date.) - Entries should be sorted by start time (chronologically) when displayed for a given day to show the timeline of the day.

**Overlapping Entries:** The system will not strictly prevent overlapping time entries in MVP (meaning a user could log two activities whose times overlap). However, this is generally not desired for accurate tracking. We may: - Warn the user if they create an entry that overlaps with an existing one on the same day (nice-to-have validation). - Or simply allow it and assume the user might intentionally overlap (perhaps multitasking). This can be revisited in future versions.

### 4.3 Calendar View (Phase 2 / Future Feature)

*Note: The Calendar view is planned as a Phase 2 enhancement, not required for the initial MVP. We describe it here for completeness and future development planning.*

The **Calendar View** will provide an intuitive visual representation of activities placed in a calendar format, similar to how Google Calendar displays events.

* **Monthly Overview:** A month grid where each day is a cell. Each day’s cell might show a summary of total hours tracked or key activities (e.g., the top 1-2 categories or a small list if not too many).
* The user can click on a specific date to drill down into that day’s details.
* **Detailed Day/Week View:** Upon selecting a date (or switching to week view), display a timeline for that day (or week). For example:
* A vertical list of hours (e.g., 0:00 to 23:00) on the side, and each activity rendered as a colored block spanning from its start to end time on a timeline. This is the "time blocks" view reminiscent of Google Calendar’s day view.
* If in week view, multiple columns for each day of that week, so the user can see their entire week’s schedule laid out.
* Activities are color-coded by category (matching the colors used in charts for consistency).
* This view clearly shows how the day was filled, and gaps (free time) or overlaps.
* **Navigation:** The calendar should allow moving to previous/next month or week, and a quick “back to today” button. This lets the user explore past records easily.
* **Integration with Entries:** The calendar is another way to view (and possibly manage) entries:
* Clicking on an activity block in the calendar could open an edit dialog or show the details (name, exact times, notes).
* Users could add new activities from the calendar view as well by clicking/tapping on a time slot (this would open the add activity form pre-filled with that chosen time). This is a nice-to-have interactive feature.
* **Synchronization:** The calendar should reflect the same data as the list and dashboard. Changes made via one interface (e.g., deleting an entry from the list) should update the calendar as well. This will require the data source to be consistent (which it will be, since all come from the same database).
* The calendar view is not required for MVP delivery, but the data model and structure set in MVP (especially having start and end times for activities) is designed to enable this feature easily in the next phase.

### 4.4 Non-Functional Requirements (Usability, Performance, Reliability)

* **Ease of Use:** The application should be intuitive. Even without a tutorial, a first-time user should understand how to add an activity and view the reports. Use clear labels, placeholders (e.g., “e.g. Meeting with Bob” in the Activity Name field), and maybe a one-time tooltip or example to guide new users.
* **Performance:** Given the app’s scope, performance requirements are modest. All operations (adding an entry, loading the dashboard) should feel instantaneous:
* The app should fetch only necessary data (e.g., only today’s entries for today’s dashboard) to reduce load.
* Use indexes in the database for queries on date or user\_id to keep retrieval snappy (more on data model in Section 5).
* The frontend should efficiently update components (leveraging React’s state updates and possibly useMemo/useCallback to avoid unnecessary re-renders of charts).
* **Offline & Resilience:** The app is primarily online (since it uses Supabase). If the network is offline, the app should handle gracefully:
* Possibly cache the last known data (using local storage or browser caching) so the dashboard doesn’t break if offline (this can be an enhancement).
* Show a message like “You are offline. Any new entries will be saved locally and synced when back online.” (Implementing full offline sync might be complex, so at minimum detect and inform user if their action cannot be completed due to no connectivity).
* **Data Privacy:** In MVP with no login, data is technically not private on the backend (unless we implement an anonymous user per device – see Section 5). We should inform users (perhaps in an About/FAQ) that this is a personal tool and not secure for sensitive data. When authentication is added later, each user’s data will be protected.
* **Scalability:** The design should allow adding more features later (like those in future scope) without major refactoring:
* E.g., adding user accounts later might mean adding a user\_id foreign key to entries (we already plan for that).
* The use of Supabase and React is scalable to many users, but as MVP is single-user centric, we mostly ensure internal scalability (clean code structure, etc.).
* **Compatibility:** The app should work on modern browsers (Chrome, Firefox, Safari, Edge) and ideally be functional on both desktop and mobile web. It doesn’t target old browsers like IE11 (not necessary in 2025 environment).
* **Responsive Design:** As mentioned, the UI will adjust to different screen sizes. This involves possibly using flexible grids and media queries:
* On mobile: maybe a single-column layout with larger touch-friendly buttons, charts that can scroll or be viewed one at a time.
* On desktop: multi-column layout (e.g., form on left, list on right, etc., as suitable).

## 5. Data Model and Database Structure

The application’s data will revolve around tracking time entries (activities) and organizing them by category. Below is the proposed database schema using PostgreSQL (Supabase). All tables will use **UUIDs** for primary keys by default (Supabase can generate these automatically) or auto-increment integers if preferred – but UUID is good for unique identification especially if merging with future user accounts.

### 5.1 **Activities Table** (time entries)

This table stores each time log entry the user creates.

* **id** (UUID, Primary Key): Unique identifier for the activity entry.
* **user\_id** (UUID or TEXT): Identifier for the user who owns this entry. In the MVP (no auth), this could be an anonymous user’s ID or a device ID. All entries from the same device/share will carry the same user\_id to group them. (In future, this will link to an authenticated users table).
* **category\_id** (UUID, Foreign Key to Categories table, nullable): Reference to the category of the activity. If using a Categories table, this links to it. If an entry uses a custom category that is not in the categories table, this could be null and we may use the category\_name field instead (see next). In practice, we will ensure category\_id is set by either a predefined or newly created category.
* **category\_name** (TEXT): Redundant storage of the category name. This can be useful for quick queries or if we allow entries without a predefined category. It will store the human-readable category (e.g., "Work", "Sleep"). If category\_id is provided, category\_name should match the name from the Categories table at the time of entry (to avoid an extra join for simple listing). This field also allows quick distinct queries on categories used.
* **activity\_name** (TEXT): The description of the activity (what the user did). Example: "Breakfast", "Project meeting", etc.
* **start\_time** (TIMESTAMP WITH TIME ZONE): When the activity began. Stored in UTC in the database (Supabase by default can handle timezone conversion; the app will send in local time which the DB can store with offset). This, combined with end\_time, indicates the time block of the activity.
* **end\_time** (TIMESTAMP WITH TIME ZONE): When the activity ended. Must be >= start\_time. If an activity is instantaneous or the user only wants to log a point event, end\_time could equal start\_time (or we could allow end to be null and treat it as 0 duration, but that’s edge case).
* **duration\_minutes** (INT): The duration of the activity in minutes. This can be *derived* as end\_time - start\_time, but we may store it for convenience and faster querying (especially for total calculations). Alternatively, we store duration in a human-readable format or as an interval type. Storing as integer minutes or seconds is simple. (If we trust calculation in queries or in frontend, this field is optional. It could also be a generated column in Postgres).
* **notes** (TEXT, nullable): Any additional notes the user entered for this entry.
* **created\_at** (TIMESTAMP): Timestamp when the entry was created (auto-filled by database default to NOW()).
* **updated\_at** (TIMESTAMP): Timestamp when the entry was last modified (auto-updated on edit, e.g., using a trigger or Supabase functionality).

**Indexes & Constraints:** - Primary key on id. - Index on user\_id to quickly retrieve all entries for a given user (or device). This is important as most queries will be scoped to the current user's data. - Index on start\_time (maybe composite index with user\_id, start\_time) to query by date or range for the dashboard (e.g., all entries in a date range). - Foreign key constraint on category\_id referencing Categories table (with ON DELETE SET NULL, perhaps, so if a category is removed the entry isn't deleted). - Constraint that end\_time >= start\_time (can be a CHECK constraint in SQL to ensure no negative durations).

### 5.2 **Categories Table**

This table holds the list of activity categories for classification. It includes predefined categories and any custom categories users add.

* **id** (UUID, Primary Key): Unique identifier for the category.
* **name** (TEXT): The category name (e.g., "Work", "Health", "Sleep", "Chores", etc.). Ideally enforce uniqueness per user (a user shouldn't have two categories with the same name).
* **user\_id** (UUID or TEXT, nullable): Identifies the owner of this category. For predefined global categories, this can be NULL or a special value indicating it's a global category available to all. For user-defined categories, user\_id links to the user (or device) who created it. This way, users can have personal categories that don’t show up for others.
* **created\_at** (TIMESTAMP): When the category was added (could be useful to track, not critical).
* **color\_code** (TEXT, nullable): An optional hex color or label for the category color. This can help ensure the category has a consistent color across charts and calendar. Predefined categories might come with default colors. (This field is optional but can be nice for UI consistency; if not set, colors can be assigned from a default palette in code).

**Indexes & Constraints:** - Unique index on (user\_id, name) to prevent duplicate category names for the same user (and perhaps allow a unique constraint on (name) where user\_id is null for global ones). - Index on user\_id for quick lookup of categories available to a user.

**Preloaded Data:** - Upon initial setup, this table will be populated with a set of common categories (with user\_id = NULL for global). For example: - Work, Health, Sleep, Leisure, Exercise, Study, Travel, Other (each with a distinct id and maybe color). - The app’s category dropdown will load all global categories plus any categories with user\_id matching the current user (if any). - When a user enters a new category via the form, the app will insert a new record into Categories with that user’s ID and the given name (if it doesn’t already exist). Then use that category for the activity entry.

*(If the implementation chooses not to use a categories table for MVP: alternatively, categories can be handled on the fly by simply using distinct category\_name values from the Activities table for that user. However, the above structure is more scalable and makes adding features (like renaming categories or color coding) easier in the future.)*

### 5.3 **Users Table** (Future)

Supabase by default has an auth.users system table for authenticated users. In the MVP with no authentication, we won't manually manage a users table. However, we still need a way to distinguish data of one user (or device) from another: - We will generate an **anonymous user identifier** for each unique client. This could be done by using Supabase’s anonymous sign-in feature (which creates a user behind the scenes without credentials) or by generating a random UUID on the client and storing it in local storage. - Using Supabase Auth (Anonymous Sign-In): The app can call supabase.auth.signInAnonymously() to get an user session without requiring any user input. This provides a user\_id (UUID) for the session that we can use as the user\_id in the entries and categories. This method ensures each device has a unique user token recognized by the database (and can be used with Row Level Security policies if enabled)[[1]](https://supabase.com/docs/guides/auth/auth-anonymous#:~:text=,anon%20key). If the user clears their browser data or signs out, a new anonymous account would be created and they’d lose access to old data (which is a known limitation, acceptable in MVP). - Alternatively, without using Supabase Auth, we can create a random UUID in local storage as device\_id and use that as the user\_id field for all entries. This is simpler but less secure, and we’d have to ensure our Supabase tables are readable/writable with the anon service role and perhaps manually filter by this id in queries. - **Important:** All queries to the database for fetching or modifying entries will include a filter on user\_id to ensure we only handle the current user's (device’s) data. This will prevent mixing data between different users of the app, even without formal authentication. (If using Supabase’s auth with RLS, this can be enforced at the database level as well, but initial implementation might rely on client-side filtering and the fact that the anon API key doesn’t expose data unless queried.)

*(No separate users table schema is defined here, since Supabase will handle it if using auth. If we were to define one later for custom needs, it might include user profile info like email, name, etc.)*

### 5.4 Example Data Scenario

To illustrate, here’s how data might look for a user (with user\_id = U1234-... as a placeholder):

* **Categories table:**
* (NULL, "Work", NULL, [time], "#1f77b4") – a global category Work with some default color.
* (NULL, "Sleep", NULL, [time], "#2ca02c") – global category Sleep.
* (NULL, "Exercise", NULL, [time], "#d62728") – global category Exercise.
* (U1234, "Project X", U1234, [time], "#ff7f0e") – a custom category added by this user for a specific project.
* (U1234, "Chores", U1234, [time], "#9467bd") – another custom category for the user.
* **Activities table (Time entries):**
* id: A1, user\_id: U1234, category\_id: (ref to Work), category\_name: "Work", activity\_name: "Client Meeting", start\_time: 2025-10-10 09:00 (UTC 03:30, which is 09:00 IST local), end\_time: 2025-10-10 10:00, duration\_minutes: 60, notes: "Discussed Q3 targets", created\_at: ..., updated\_at: ...
* id: A2, user\_id: U1234, category\_id: (ref to Health or Exercise), category\_name: "Exercise", activity\_name: "Morning Jog", start\_time: 2025-10-10 06:30, end\_time: 2025-10-10 07:00, duration\_minutes: 30, notes: "", ...
* id: A3, user\_id: U1234, category\_id: NULL (category was custom typed without adding to table for some reason), category\_name: "Meditation", activity\_name: "Meditation", start\_time: 2025-10-10 07:05, end\_time: 2025-10-10 07:20, duration\_minutes: 15, notes: "", ...
* id: A4, user\_id: U1234, category\_id: (ref to Sleep), category\_name: "Sleep", activity\_name: "Sleep", start\_time: 2025-10-09 23:00, end\_time: 2025-10-10 07:00, duration\_minutes: 480, notes: "Overnight sleep", ...

With this data, the dashboard for **October 10, 2025 (Today)** would sum up the durations per category: - Work: 1 hour, - Exercise: 30 min, - Meditation: 15 min, - Sleep: 7 hours (but note part of it started previous day; depending on whether we attribute by start date or split across midnight – for simplicity, if an activity crosses midnight, we might attribute it to the day it started or ended. This is an edge case to clarify in implementation. Possibly treat each entry as belonging to the date of its start\_time for daily grouping.)

The pie chart might show slices for Work, Exercise, Meditation, Sleep. The bar chart (if showing the week) would have a bar for Oct 10 with ~9 hours total, and other days accordingly.

## 6. Operations and API Endpoints

Given that we are using Supabase, we do not need to write our own API server, but we will outline how the front-end will interact with the backend for key operations:

**6.1 Using Supabase Client Library:**  
On the front-end, the plan is to use the Supabase JavaScript SDK. We will initialize it with our Supabase project URL and the public **anon key**. This allows direct calls from the React app to the database in a secure way (depending on RLS policies). Each operation will be something like: supabase.from('<table>').select(...) or .insert(...) etc., which under the hood calls Supabase's REST API.

**6.2 CRUD Operations:**

* **Create (Add Activity):**  
  Endpoint (via Supabase): POST /rest/v1/activities (if using direct REST) or using the client:  
  supabase.from('activities').insert([{ ...fields... }]).  
  The fields will include user\_id, category\_id (or category\_name), activity\_name, start\_time, end\_time, duration\_minutes, notes. Supabase will respond with the inserted object or an error. The app will then update state to include the new entry (or refetch the list).
* **Read (Fetch Activities for dashboard or list):**
* For dashboard stats: a query to sum or group by category might be done. We have two approaches:
  1. **Client-side aggregation:** Fetch all activities for the given range and then sum/group in the React app code.
  2. **Database aggregation:** Use SQL via Supabase RPC (remote procedure call) or a custom view. For MVP, the simpler route is client-side since data volumes are small.
* Example: To get today’s activities for user X:  
  supabase.from('activities').select("\*").eq('user\_id', X).gte('start\_time', start\_of\_day).lte('start\_time', end\_of\_day) (assuming filtering by date via start\_time). Then the client can calculate totals.
* For the list on the Activity page: similarly, select \* from activities where user\_id = X and date = chosen\_date (or a limited range).
* Supabase allows real-time subscriptions: we could subscribe to changes on the activities table for the current user\_id, so that any new entry or update triggers a push update to the UI. This is an optional improvement to make the app feel real-time synchronized.
* **Update (Edit Activity):**  
  Supabase endpoint: PATCH /rest/v1/activities?id=eq.<id> or via client:  
  supabase.from('activities').update({ fields... }).match({ id: ..., user\_id: X }).  
  We include user\_id in the match to be safe (so one device can’t accidentally edit another’s entry if it had the id). This will update the record. On success, we refresh the relevant data in the UI (or optimistic update).
* **Delete (Remove Activity):**  
  Endpoint: DELETE /rest/v1/activities?id=eq.<id> or via client:  
  supabase.from('activities').delete().match({ id: ..., user\_id: X }).  
  This will remove the record from the database. The UI will then filter it out from the list and refresh charts.
* **Fetch Categories:**  
  On loading the form, we need the list of categories. We can fetch from the Categories table: supabase.from('categories').select("\*").or(user\_id.is.null,user\_id.eq.X) – meaning get categories where user\_id is null (global categories) or equals the current user.  
  This list populates the dropdown. (Alternatively, maintain a local list of global categories to reduce calls, but let's assume dynamic fetch to allow future changes.)

**6.3 Security and Access:** - Initially, we might keep the Supabase tables open to read/write with the anon key (no RLS, just make them accessible since there's no sensitive data). However, this means any user of the app technically could access everyone’s data if they knew how to query it. This is acceptable for a quick MVP but not ideal. - A better approach (if time permits) is to enable **Row Level Security (RLS)** on the activities table in Supabase, and write a policy such that: - A logged-in user (including an anonymous auth user) can only select/insert/update their own rows (where user\_id = auth.uid() in the policy). - This requires that we use the Supabase auth session (anonymous or real) so that auth.uid() is available. - If we do implement anonymous sign-in, then from the app’s perspective there will be a call at startup like:  
await supabase.auth.signInAnonymously();  
which obtains a session and user. All subsequent calls include a JWT so RLS knows the user’s ID[[2]](https://supabase.com/docs/guides/auth/auth-anonymous#:~:text=Calling%20,data%2C%20or%20use%20another%20device). The user\_id in our table would correspond to this auth.uid(). This is a more secure design. - For simplicity, if RLS is not configured, we must rely on client-side filtering of user\_id and trust the front-end. This is easier to implement but should be revisited when adding real auth.

**6.4 Database Schema Deployment:** - The database (Supabase) can be set up via their dashboard or using an SQL script. We should prepare the SQL statements for creating the tables as described: - Create table categories (...fields...) and insert default categories. - Create table activities (...fields..., foreign key references). - (If using Supabase Auth, ensure email auth is disabled if not used, and enable anonymous auth in the project settings). - If using RLS: enable RLS on activities and categories, and add policies: - For activities: allow select/insert/update/delete for authenticated users where user\_id = auth.uid() (and maybe allow insert with user\_id = auth.uid() automatically). - For categories: allow select for all where user\_id is null or user\_id = auth.uid(), allow insert for auth users with user\_id = auth.uid() on their rows. - These details can be handled in the Supabase config; they need to be set up once. Codex or the developer should ensure to document or automate this setup.

## 7. User Journey Examples

This section describes typical flows a user will take, tying together the UI and data interactions:

* **First-Time Use (New User, Day 1):**  
  Alice opens the Time Tracker web app on her phone. She sees a welcome message on the dashboard saying “No activities logged yet today.” She taps the "Add Activity" button.
* On the Add Activity form, she enters:
  + Activity Name: "Morning Routine"
  + Category: She sees suggestions like Work, Exercise, etc., but chooses to type a new category "Personal Care".
  + Start Time: 7:00 AM (today’s date preselected by default)
  + End Time: 7:30 AM
  + Notes: "Breakfast and getting ready"
* She hits "Add". The entry is saved. The dashboard now updates: “Total time today: 30 minutes” with a pie chart maybe showing 100% Personal Care (30m).
* She adds a couple more activities: a 2-hour "Work: Project X coding" from 9:00–11:00, and a 1-hour "Exercise: Gym" from 6:00–7:00.
* Now the dashboard shows total 3h30m today, broken into categories Work (2h), Exercise (1h), Personal Care (0.5h). She sees a pie chart with those three slices and a bar chart showing today’s total (3.5h).
* **Continuing Use (Later that day):**  
  Alice realizes she forgot to log her lunch break. She goes to Add Activity, inputs "Lunch" with category "Food" (another new category), 1:00–1:30 PM. Now her charts update accordingly. She decides to edit "Morning Routine" to break it into "Breakfast" and "Getting Ready" as two separate entries for more detail:
* She clicks edit on "Morning Routine" entry, changes the name to "Breakfast", and adjusts time to 7:00–7:15, then saves.
* She then adds another entry "Getting Ready" from 7:15–7:30 in category Personal Care.
* The app now has separate entries and the dashboard updates.
* **Viewing Past Data:**  
  The next day, Alice opens the app. The dashboard defaults to Today (which is empty in morning). She switches to "Yesterday" (if implemented via date picker) or sees "This Week" stats. She can see yesterday’s total and breakdown. If the Calendar view existed, she could click on yesterday’s date to see the timeline of her activities.
* **Edge Case – Clearing Data:**  
  If Alice clears her browser cache or opens the app in a different browser/device during the MVP phase, she will not see her old data because there’s no account tying it (unless we use a shared anonymous token strategy). This is a known limitation and would be communicated. In a future version with login, this would not be an issue.

Throughout these journeys, the focus is on making adding and viewing data as seamless as possible so that users build a habit of logging time. Any friction (like complicated forms or slow loading) could discourage regular use, so the design emphasizes simplicity and speed.

## 8. Deployment and Development Notes

* The application repository will be set up on **GitHub** under an appropriate name (e.g., personal-time-tracker). It will contain the frontend React code (likely bootstrapped with Create React App, Vite, or Next.js – any framework that works on Vercel; Next.js could be beneficial if server-side rendering is desired for fast load, but MVP can be purely client-rendered).
* **Vercel Integration:** The GitHub repo will be connected to Vercel. On each push to the main branch, Vercel will:
* Install dependencies (from package.json, including React, chart library, supabase client, etc.).
* Build the project (e.g., npm run build) into an optimized static bundle.
* Deploy the static site to a global CDN. The app will have an address (and possibly a custom domain) where it’s accessible.
* Environment variables like SUPABASE\_URL and SUPABASE\_ANON\_KEY will be set in Vercel’s settings so that the production build can connect to the correct backend. (We will ensure not to expose secrets in the repo; the anon key is public but still best kept in env).
* **Testing:** Before deployment, the developer should test the application locally:
* Unit tests for any utility functions (if any).
* Manual testing of adding, editing, deleting entries; ensuring charts update; verifying data persists after page refresh (and across days).
* Possibly test on mobile viewport to fine-tune responsiveness.
* **Success Criteria Validation:** We will verify the success criteria as follows:
* **Add/Edit/Delete Entries:** Confirm that a user can perform all three actions and see the changes immediately reflected in the list and dashboard. No data should mysteriously disappear or fail to save during normal use.
* **Data Persistence:** After adding entries, a page reload (or closing and reopening the browser) still shows the data (i.e., Supabase storage works or local storage is intact). If using Supabase, also check that the data actually went to the cloud by looking in the Supabase table.
* **Dashboard Visualization:** Check that the charts and summary make sense (e.g., if 2 hours Work and 1 hour Exercise, the pie chart slices correspond roughly 67%/33%, etc.). Also ensure the time range filter (Today, This Week, etc.) updates the data correctly.
* **Deployment Pipeline:** Push a trivial update to the repo and watch Vercel deploy it automatically; then access the live site to ensure it reflects the update. This confirms CI/CD is working.
* **Documentation & Handover:** Provide a README in the repo detailing how to run the project, how to set up the Supabase (schema and config), and any assumptions made. This will help others (or future me) understand the system. For example, include the SQL for creating tables in the repo or notes that the Supabase migration tool was used.

By covering the above aspects, we ensure that the delivered web application is robust, user-friendly, and maintainable. The MVP focuses on core functionality (tracking and visualizing personal time usage) with a clear path to extend features in subsequent phases.

[[1]](https://supabase.com/docs/guides/auth/auth-anonymous#:~:text=,anon%20key) [[2]](https://supabase.com/docs/guides/auth/auth-anonymous#:~:text=Calling%20,data%2C%20or%20use%20another%20device) Anonymous Sign-Ins | Supabase Docs

<https://supabase.com/docs/guides/auth/auth-anonymous>