Please find the python challenges given so far listed below for your reference!!

✅ 15 Days Python Challenge – Tasks

**\*\*\*\*\* Day 1 (Mon, Sep 8):**

Build a small form that takes name and age (slider) and shows a greeting. Use only Streamlit.

# **📄 Technical Documentation: FormApp (Name + Age Greeting Form)**

## **1. Problem Statement & Goals**

**Problem Statement:** Users often need a quick and simple way to input their basic details (name and age) to personalize interactions.

**Goal:** Build a lightweight form application that collects **Name** and **Age**, and displays a personalized greeting message:  
 👉 Example: *“Hello, John Doe! You are 25 years old.”*

**Success Criteria:**

* User can input a valid name.
* User can select an age via slider.
* Greeting message dynamically updates with entered values.

## **2. Personas & User Journeys**

**Primary Persona:**

* **User Type:** General web user
* **Goal:** Quickly enter details and see a personalized greeting.
* **Technical Skill:** Basic computer skills.

**User Journey:**

1. User opens the app.
2. User enters their full name in the input field.
3. User selects their age using a slider.
4. User clicks **Continue** after each input.
5. User sees a greeting message with their name and age.

## **3. Epics & User Stories (with Acceptance Criteria)**

### **Epic 1: User Input - Name**

**User Story:** As a user, I want to enter my name so that the system can address me personally.

**Acceptance Criteria:**

* Input field accepts alphabetic characters and spaces.
* Input field cannot be empty.
* On clicking **Continue**, name is stored and available for greeting.

### **Epic 2: User Input - Age**

**User Story:** As a user, I want to select my age using a slider so that the greeting reflects my age.

**Acceptance Criteria:**

* Slider allows selection of values between **1 and 100**.
* Slider displays the selected age dynamically.
* On clicking **Continue**, age is stored and available for greeting.

### **Epic 3: Display Greeting**

**User Story:** As a user, I want to see a greeting with my name and age so that I feel the app is personalized.

**Acceptance Criteria:**

* Greeting message should follow the format:  
   Hello, <Name>! You are <Age> years old.
* Greeting updates only after both fields are completed.
* Greeting should handle edge cases (e.g., missing name → prompt user to enter again).

## **4. System Architecture / Workflows**

**Architecture:**

* **Frontend only** (for MVP)
* No backend persistence needed (data lives in browser memory).

**Workflow:**

1. User enters name → App validates → stores in local state.
2. User selects age → App validates → stores in local state.
3. Once both are captured, greeting message is rendered.

## **5. Technical Specifications**

### **Frontend**

* **Framework:** React (or Vanilla JS for MVP).
* **UI Components:**
  + Input field (Name)
  + Slider (Age)
  + Button (Continue)
  + Greeting display box

### **Data Model**

{

"name": "string",

"age": "number"

}

### **APIs**

* No backend API required for MVP.
* Future: API endpoint (POST /greeting) to send data to server for persistence/logging.

### **Validation Rules**

* Name: must not be empty, min length = 2 characters.
* Age: integer between 1–100.

## **6. Non-Functional Requirements**

* **Performance:** Greeting should render instantly (<100ms).
* **Security:**
  + Input sanitization to prevent script injection.
* **Compliance:**
  + GDPR not required (no data storage).

## **7. Open Questions / Assumptions**

* Should the form allow **real-time greeting update** (as user types/changes age) or only after clicking **Continue**?
* Should the greeting persist if the page is refreshed (requires localStorage)?
* Should age have a max cap (100) or allow higher values?
* Future scope: Should we log these inputs in a backend for analytics?

👉 This is the **MVP technical documentation**.

**\*\*\*\*\* Day 2 (Tue, Sep 9):**

**Friends go out for dinner/trip and want to split expenses fairly.**

• Input: total amount + number of people

• Optionally: each person’s name & contribution

• Output: how much each person should pay or get back

**📄 Technical Documentation: Expense Splitter App**

## **1. Problem Statement & Goals**

**Problem Statement:** When friends go out for dinner or on a trip, managing shared expenses fairly is often challenging. People either forget contributions or struggle to calculate who owes whom.

**Goal:** Build a simple app where users can input the **total amount** and **number of people**, optionally include **individual names and contributions**, and get an output showing how much each person should **pay or get back** to make the split fair.

**Success Criteria:**

* Users can calculate equal split instantly.
* Users can handle uneven contributions and see the balance for each person.
* Output is clear: who needs to **pay extra** or who should **receive back** money.

## **2. Personas & User Journeys**

**Primary Persona:**

* **User Type:** Friends, colleagues, or family splitting group expenses.
* **Goal:** Quickly know how much everyone owes or is owed.
* **Technical Skill:** Basic mobile/web app usage.

**User Journeys:**

1. User enters **total bill amount** and **number of people**.
2. System calculates equal split → shows amount per person.
3. (Optional) User enters each person’s **name + contribution amount**.
4. System compares contribution vs expected split → shows balances.
5. Each person sees how much they **owe** or should **get back**.

## **3. Epics & User Stories (with Acceptance Criteria)**

### **Epic 1: Equal Split Calculation**

**User Story:** As a user, I want to enter the total amount and number of people so that the system can calculate equal split.

**Acceptance Criteria:**

* Input fields: total\_amount, number\_of\_people.
* System divides total evenly.
* Output: “Each person should pay X.”
* Validation: total > 0, number\_of\_people ≥ 1.

### **Epic 2: Unequal Contributions**

**User Story:** As a user, I want to enter each person’s name and contribution so that the system can calculate who owes or gets back money.

**Acceptance Criteria:**

* Input fields: name, contribution.
* System calculates expected split = total\_amount / number\_of\_people.
* For each person: balance = contribution - expected\_split.  
  + If balance > 0 → Person should get back money.
  + If balance < 0 → Person owes money.
* Output: Table with name, contribution, and balance.

### **Epic 3: Result Presentation**

**User Story:** As a user, I want to see a clear summary so that I immediately know who owes or gets money.

**Acceptance Criteria:**

* Show per-person breakdown.
* Totals should balance to 0.
* Format: “Alice pays 200, expected 150 → gets back 50.”

## **4. System Architecture / Workflows**

**Architecture:**

* **Frontend:** Input form for amount, people, and contributions.
* **Business logic layer:** Splitting algorithm.
* **Backend (optional for MVP):** Not required unless persistence/sharing needed.

**Workflow:**

1. Input total + number of people.
2. System computes equal split.
3. If contributions entered → validate & store.
4. Compute balances: contribution - equal\_share.
5. Render results in structured table.

## **5. Technical Specifications**

### **Data Model**

{

"total\_amount": 1200,

"number\_of\_people": 4,

"people": [

{ "name": "Alice", "contribution": 200 },

{ "name": "Bob", "contribution": 400 },

{ "name": "Charlie", "contribution": 300 },

{ "name": "Diana", "contribution": 300 }

]

}

### **Calculation Logic**

* **Equal share:** total\_amount / number\_of\_people
* **Balance per person:** contribution - equal\_share

### **Example**

* Total = 1200, People = 4 → Equal share = 300
* Contributions:  
  + Alice = 200 → Owes 100
  + Bob = 400 → Gets back 100
  + Charlie = 300 → Settled
  + Diana = 300 → Settled

**Output:**

* Alice owes 100
* Bob should get 100
* Charlie settled
* Diana settled

### **APIs (future)**

* POST /calculate-split → Accepts JSON input, returns split breakdown.
* GET /history → (Optional) Retrieve past calculations.

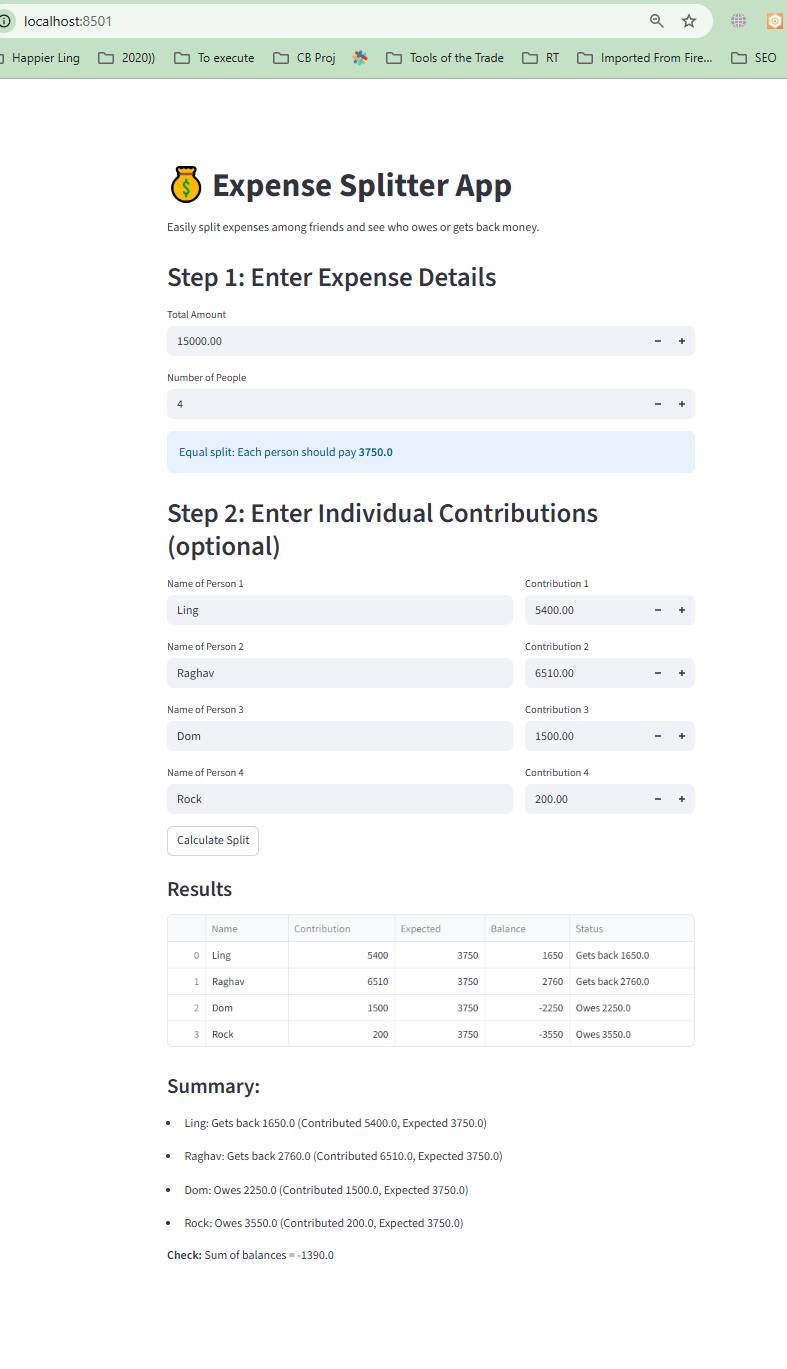
## **6. Non-Functional Requirements**

* **Performance:** Results calculated instantly (<200ms).
* **Scalability:** Should support up to 50 people per group.
* **Security:** Input sanitization, no sensitive data stored.
* **Compliance:** No PII persistence unless explicit consent.

## **7. Open Questions / Assumptions**

* Should we support **multiple bills** (e.g., trip with multiple expenses) or just a single bill per session?
* Should we allow fractional amounts (e.g., 3 people splitting 100 → 33.33 each)?
* Should balances be rounded (to 2 decimals)?
* Do we need export/share options (PDF, WhatsApp, email)?
* Should user accounts/logins exist for saving history?

👉 This covers the **technical documentation for MVP**.

Web app screenshot:  
  


**\*\*\*\*\* Day 3 (Wed, Sep 10):**

**Simple Calculator ➕➖✖️➗**

• Inputs: two numbers + operation

• Output: result  
  
Got it 🚀 Let’s prepare the **full technical documentation + PRD** for the **Simple Calculator App** (basic arithmetic: add, subtract, multiply, divide).

# **📄 Technical Documentation: Simple Calculator App**

## **1. Problem Statement & Goals**

**Problem Statement:** Users often need a quick and reliable way to perform basic arithmetic (addition, subtraction, multiplication, division) without unnecessary complexity.

**Goal:** Build a **Simple Calculator** that accepts two numbers and an operation, then outputs the result.

**Success Criteria:**

* App supports **+ , – , × , ÷** operations.
* Handles invalid inputs gracefully (e.g., division by zero).
* Provides instant results.

## **2. Personas & User Journeys**

**Persona 1: Student**

* Needs: Quick math help while studying.

**Persona 2: Professional**

* Needs: Quick calculation during work without opening a full spreadsheet.

**User Journey:**

1. User opens the app.
2. User enters **two numbers**.
3. User selects an **operation (+, –, ×, ÷)**.
4. System calculates and displays result instantly.

## **3. Epics & User Stories (with Acceptance Criteria)**

### **Epic 1: Number Input**

**User Story:** As a user, I want to enter two numbers so that I can perform a calculation.

**Acceptance Criteria:**

* Input fields accept integers and decimals.
* Inputs must not be empty.
* Validation prevents invalid characters.

### **Epic 2: Operation Selection**

**User Story:** As a user, I want to select an operation (+, –, ×, ÷) so that I can perform the calculation I need.

**Acceptance Criteria:**

* Four operations available: +, –, ×, ÷.
* Operation must be chosen before result is calculated.

### **Epic 3: Calculation & Output**

**User Story:** As a user, I want to see the result of my calculation instantly so that I don’t need to calculate manually.

**Acceptance Criteria:**

* Result displays immediately after pressing "Calculate".
* Supports decimals.
* Handles division by zero with a user-friendly message (“Error: Cannot divide by zero”).

## **4. System Architecture / Workflows**

**Architecture:**

* **Frontend only (MVP):** Inputs, operation buttons, result display.
* **Logic:** Basic arithmetic functions implemented in JS/Python/etc.
* **Backend (optional):** Not required unless history/logging is needed.

**Workflow:**

1. User enters numbers.
2. User selects operation.
3. App validates inputs.
4. App executes calculation function.
5. Result displayed instantly.

## **5. Technical Specifications**

### **Data Model**

{

"number1": 12,

"number2": 6,

"operation": "divide",

"result": 2

}

### **Calculation Logic**

* Addition: result = num1 + num2
* Subtraction: result = num1 - num2
* Multiplication: result = num1 \* num2
* Division:  
  + If num2 == 0 → Error message.
  + Else → result = num1 / num2.

### **APIs (future)**

* POST /calculate → Accepts JSON with num1, num2, operation, returns result.
* GET /history → Returns past calculations.

## **6. Non-Functional Requirements**

* **Performance:** <50ms response time.
* **Usability:** Simple, minimal UI, mobile-friendly.
* **Scalability:** Not critical for MVP (small app).
* **Security:** Input sanitization to prevent script injection.

## **7. Open Questions / Assumptions**

* Should results support **very large numbers** (e.g., scientific notation)?
* Should we store **calculation history** locally?
* Should we allow **multiple chained operations** (like 2 + 3 × 4), or just two inputs per operation?
* Should we include a **clear/reset** button for inputs and results?

# **📑 Product Requirements Document (PRD)**

## **Product: Simple Calculator App**

### **1. Overview**

A minimal calculator app to perform basic arithmetic operations between two numbers.

### **2. Goals & Non-Goals**

**Goals:**

* Support 4 basic arithmetic operations.
* Provide instant results.
* Handle invalid inputs gracefully.

**Non-Goals (MVP):**

* Advanced math functions (square root, powers, percentages).
* Scientific calculator features.
* Persistent history across sessions.

### **3. User Stories (Prioritized)**

**MVP Stories:**

1. Input two numbers.
2. Select operation.
3. See calculation result.
4. Handle division by zero error.

**Future Stories:**

* Save calculation history.
* Support chained operations.
* Add advanced operations (%, √, ^, etc.).
* Export/share results.

### **4. Functional Requirements**

* **Inputs:** Two numeric fields.
* **Operations:** Buttons or dropdown for +, –, ×, ÷.
* **Output:** Result displayed in text box/label.
* **Validation:** Block empty/invalid inputs.

### **5. User Flows**

**Flow 1: Basic Calculation**

1. Enter first number.
2. Enter second number.
3. Select operation.
4. Click “Calculate.”
5. See result.

**Flow 2: Error Handling**

* If dividing by zero → Show error message.
* If inputs invalid → Show warning.

### **6. Success Metrics**

* ✅ Result displayed in <50ms.
* ✅ Error handling works correctly 100% of time.
* ✅ User satisfaction: app rated as “easy to use” in testing.

### **7. Release Plan**

**MVP (Phase 1):**

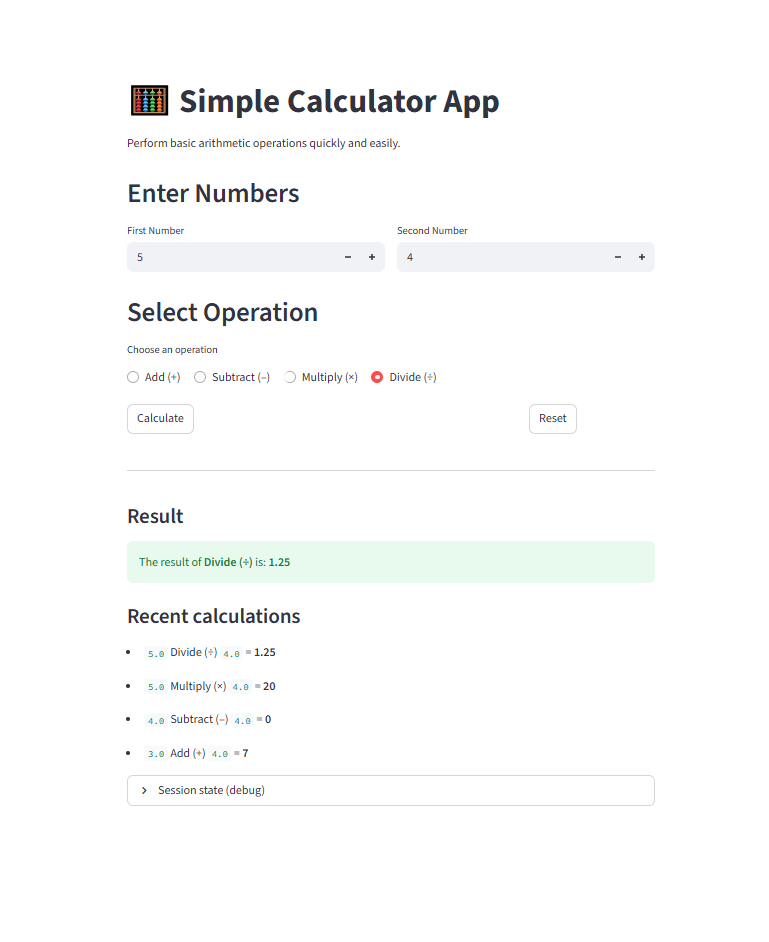
* Support +, –, ×, ÷ with two numbers.
* Handle division by zero.

**Phase 2 (Nice-to-Have):**

* History of calculations.
* Clear/reset button.

**Phase 3 (Advanced):**

* Advanced math functions.
* Multi-step expressions.
* User accounts + cloud history sync.

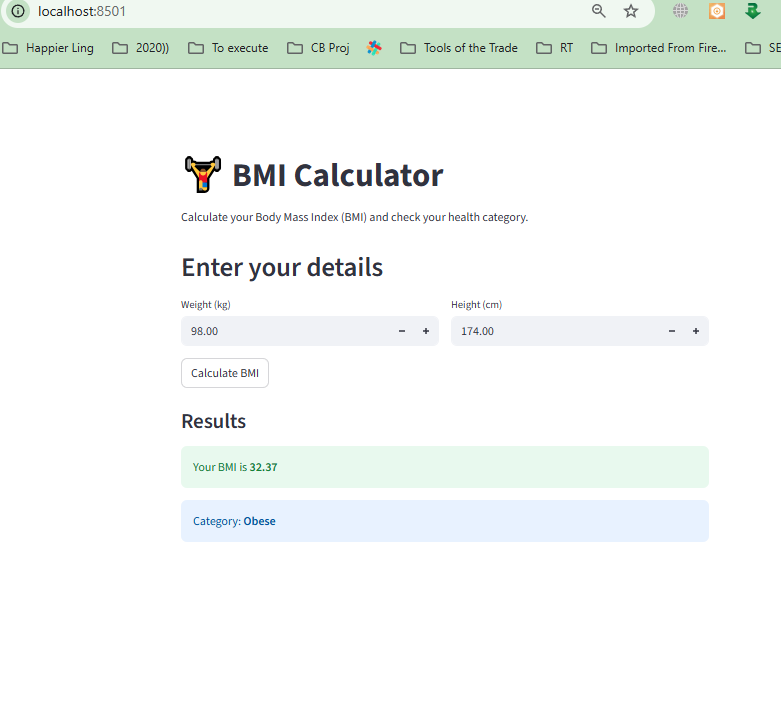


**\*\*\*\*\* Day 4 (Thu, Sep 11):**

**BMI Calculator 🏋️**

• Inputs: height & weight

• Output: BMI value + health category (Underweight / Normal / Overweight)



**\*\*\*\*\* Day 5 (Fri, Sep 12):**

**Unit Converter 🔄**

• Convert: currency, temperature, length, weight

• Show results instantly  
  
**📄 Technical Documentation: Unit Converter App**

## **1. Problem Statement & Goals**

**Problem Statement:** People frequently need to convert between units (currency, temperature, length, weight). Existing tools can feel cluttered or slow.

**Goal:** Build a **Unit Converter App** that:

* Converts **currency, temperature, length, and weight**.
* Shows results **instantly** as the user types/selects units.

**Success Criteria:**

* Supports common units per category.
* Provides accurate conversions (currency via API, others via formula).
* Instant result updates (<100ms).

## **2. Personas & User Journeys**

**Persona 1: Traveler**

* Needs: Currency conversion while abroad.

**Persona 2: Student**

* Needs: Quick length/weight/temperature conversions while studying.

**Persona 3: Professional (Engineer/Scientist)**

* Needs: Reliable and accurate conversion for calculations.

**User Journey:**

1. User selects conversion type (currency, temperature, length, weight).
2. User enters input value.
3. User selects source and target units.
4. Result updates instantly.

## **3. Epics & User Stories (with Acceptance Criteria)**

### **Epic 1: Currency Conversion**

**User Story:** As a user, I want to convert between currencies so that I know the equivalent value.

**Acceptance Criteria:**

* Input: amount, source currency, target currency.
* Conversion rate fetched via API.
* Result shown instantly.
* Handle API errors gracefully (fallback: “Rates unavailable”).

### **Epic 2: Temperature Conversion**

**User Story:** As a user, I want to convert between Celsius, Fahrenheit, and Kelvin so that I can understand values in my preferred unit.

**Acceptance Criteria:**

* Supports C ↔ F ↔ K.
* Conversion formulas:  
  + F = (C × 9/5) + 32
  + C = (F - 32) × 5/9
  + K = C + 273.15

### **Epic 3: Length Conversion**

**User Story:** As a user, I want to convert between common length units so that I can use appropriate measurements.

**Acceptance Criteria:**

* Units: meters, kilometers, miles, feet, inches, centimeters.
* Accurate conversions using fixed conversion factors.

### **Epic 4: Weight Conversion**

**User Story:** As a user, I want to convert between common weight units so that I can measure appropriately.

**Acceptance Criteria:**

* Units: grams, kilograms, pounds, ounces, tons.
* Accurate conversions using fixed conversion factors.

### **Epic 5: Instant Results**

**User Story:** As a user, I want results to update instantly as I type so that I save time.

**Acceptance Criteria:**

* No “submit” button needed.
* Input value change triggers recalculation immediately.

## **4. System Architecture / Workflows**

**Architecture:**

* **Frontend:** Input box, dropdowns (source & target units), output display.
* **Logic:** Conversion functions.
* **Backend:** Only required for currency (fetch live rates from an API).

**Workflow:**

1. User selects category (currency, temp, length, weight).
2. User enters value + selects units.
3. If currency → fetch latest rate.
4. Perform calculation → display result instantly.

## **5. Technical Specifications**

### **Data Model**

{

"category": "currency",

"input\_value": 100,

"from\_unit": "USD",

"to\_unit": "INR",

"result": 8325

}

### **APIs**

* **Currency:** Use external API (e.g., ExchangeRate-API, Open Exchange Rates).  
  + Endpoint: GET /latest?base=USD
  + Response: { "rates": { "INR": 83.25, "EUR": 0.92 } }

### **Conversion Logic**

* **Temperature:** Formula-based.
* **Length/Weight:** Fixed conversion constants.
* **Currency:** Real-time API rates.

## **6. Non-Functional Requirements**

* **Performance:**
  + Instant results (<100ms) for formula-based conversions.
  + Currency API call <1s latency.
* **Usability:** Mobile-friendly, clean UI.
* **Scalability:** Support adding new unit categories in future.
* **Reliability:** Graceful API fallback for currency.

## **7. Open Questions / Assumptions**

* Should currency rates be **live** (every API call) or cached for the session?
* Should we support **offline mode** (for non-currency units)?
* Should results allow **decimal precision control** (e.g., 2 vs 5 decimal places)?
* Should we add **favorites/history** for frequent conversions?

# **📑 Product Requirements Document (PRD)**

## **Product: Unit Converter App**

### **1. Overview**

A fast, lightweight converter that supports **currency, temperature, length, and weight conversions** with instant results.

### **2. Goals & Non-Goals**

**Goals:**

* Perform fast, accurate conversions.
* Provide simple UI with instant results.
* Support core categories: currency, temperature, length, weight.

**Non-Goals (MVP):**

* Advanced categories (volume, speed, area, time).
* Scientific constants.
* Persistent user accounts.

### **3. User Stories (Prioritized)**

**MVP Stories:**

1. Convert between two currencies with live rates.
2. Convert between temperature units (C, F, K).
3. Convert between length units (m, km, ft, in, miles, cm).
4. Convert between weight units (g, kg, lbs, oz, tons).
5. Show results instantly.

**Future Stories:**

* Save favorite conversions.
* Show history of conversions.
* Support offline conversions.
* Add more categories (volume, speed, area).

### **4. Functional Requirements**

* Input: numeric field.
* Dropdowns: select source unit + target unit.
* Conversion result: displayed instantly.
* Validation: input must be a number.
* Currency API integration.

### **5. User Flows**

**Flow 1: Currency Conversion**

1. Select “Currency.”
2. Enter amount.
3. Select source currency → target currency.
4. API fetches rate → Result displayed instantly.

**Flow 2: Temperature/Length/Weight Conversion**

1. Select category.
2. Enter amount.
3. Select source + target unit.
4. Formula applied → Result displayed instantly.

### **6. Success Metrics**

* ✅ Formula conversions <100ms.
* ✅ Currency conversions <1s.
* ✅ 95%+ user satisfaction in usability testing.

### **7. Release Plan**

**MVP (Phase 1):**

* Currency, temperature, length, weight conversions.
* Instant results.

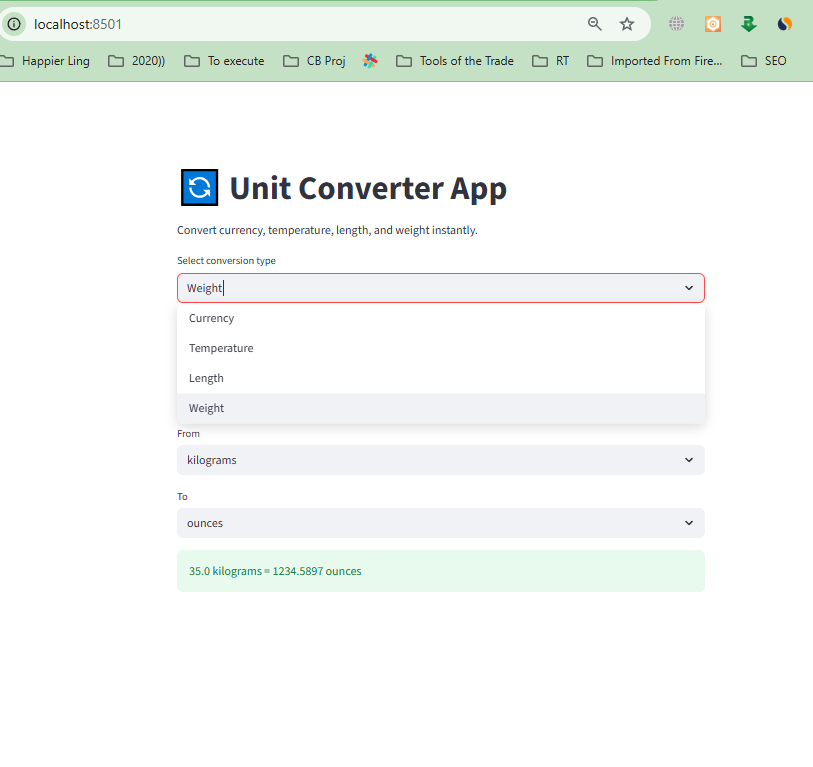
**Phase 2 (Nice-to-Have):**

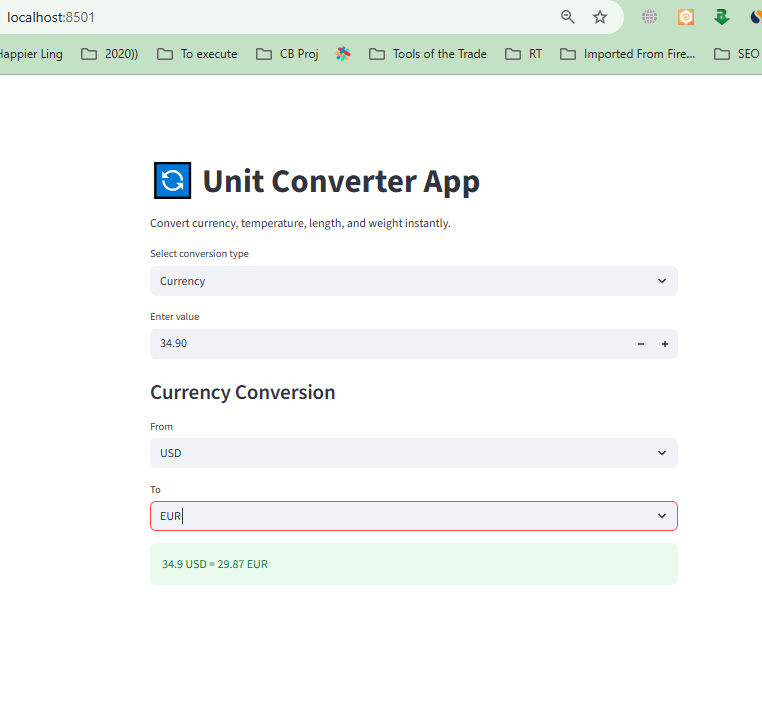
* Favorites/history.
* Offline support (non-currency).

**Phase 3 (Advanced):**

* More unit categories (volume, speed, area, time).
* User accounts with saved preferences.
* Multi-language support.

✅ This PRD + spec provides a clear roadmap for MVP → future expansions.





**\*\*\*\*\* Day 6 (Sat, Sep 13):**

**Water Intake Tracker 💧**

• Input: daily water intake

• Show: progress toward a goal (like 3L/day)

• Plot: weekly hydration chart

**full technical documentation + PRD** for the **Water Intake Tracker App**.

# **📄 Technical Documentation: Water Intake Tracker 💧**

## **1. Problem Statement & Goals**

**Problem Statement:** Many people struggle to drink enough water daily. Without tracking, it’s hard to know progress toward hydration goals.

**Goal:** Build a **Water Intake Tracker App** that:

* Lets users input their daily water intake.
* Tracks progress toward a daily goal (default: 3L/day).
* Displays a **weekly hydration chart** to visualize trends.

**Success Criteria:**

* User can input water intake multiple times per day.
* App aggregates total per day and compares to goal.
* Weekly chart shows daily totals vs goal.

## **2. Personas & User Journeys**

**Persona 1: Health Enthusiast**

* Wants to stay hydrated to support fitness.

**Persona 2: Busy Professional**

* Needs reminders and tracking to build a hydration habit.

**User Journey:**

1. User opens app.
2. User logs water intake (e.g., 250ml).
3. App updates daily total.
4. Progress bar shows % of daily goal reached.
5. At end of week → user sees hydration chart (7 days).

## **3. Epics & User Stories (with Acceptance Criteria)**

### **Epic 1: Input Water Intake**

**User Story:** As a user, I want to log water intake so that I can track hydration throughout the day.

**Acceptance Criteria:**

* Input field for volume (ml or L).
* Multiple entries per day allowed.
* Entries aggregate into daily total.

### **Epic 2: Track Progress Toward Goal**

**User Story:** As a user, I want to see my progress toward a daily goal so that I know how much more to drink.

**Acceptance Criteria:**

* Default goal = 3L/day (customizable in future).
* Progress bar updates after each entry.
* Show remaining amount (e.g., “You need 750ml more to reach your goal”).

### **Epic 3: Weekly Hydration Chart**

**User Story:** As a user, I want to see a weekly hydration chart so that I can review my habits.

**Acceptance Criteria:**

* X-axis = days of the week.
* Y-axis = water intake (liters).
* Bars show intake vs goal (e.g., shaded bar for 3L).

## **4. System Architecture / Workflows**

**Architecture:**

* **Frontend:** Input form, progress bar, weekly chart.
* **Storage:** Local storage for MVP (or lightweight DB).
* **Logic:** Aggregation by date + visualization.

**Workflow:**

1. User logs intake → store entry with timestamp.
2. App aggregates daily total.
3. Update progress toward daily goal.
4. End of week → display 7-day chart.

## **5. Technical Specifications**

### **Data Model**

{

"daily\_goal": 3000,

"entries": [

{ "date": "2025-09-25", "amount": 250 },

{ "date": "2025-09-25", "amount": 500 }

],

"totals": {

"2025-09-25": 750,

"2025-09-24": 2800

}

}

### **Components**

* **Input Component:** text/number field (ml).
* **Progress Component:** progress bar showing daily % goal.
* **Chart Component:** weekly hydration bar chart.

### **APIs (future)**

* POST /log-water → Log intake.
* GET /progress → Fetch daily total & goal.
* GET /weekly → Fetch past 7 days intake.

## **6. Non-Functional Requirements**

* **Performance:** Update progress instantly.
* **Persistence:** Data stored locally (MVP), expandable to cloud.
* **Usability:** Mobile-first design, minimal steps to log water.
* **Reliability:** Handle timezones/dates correctly.

## **7. Open Questions / Assumptions**

* Should users be able to **customize their daily goal** (e.g., 2.5L instead of 3L)?
* Should the app **send reminders** to drink water (push notifications)?
* Should chart show **average hydration** across weeks?
* Should units support both **ml and oz**?

# **📑 Product Requirements Document (PRD)**

## **Product: Water Intake Tracker 💧**

### **1. Overview**

A lightweight app that helps users stay hydrated by tracking daily water intake, showing progress toward a daily goal, and visualizing weekly hydration patterns.

### **2. Goals & Non-Goals**

**Goals:**

* Easy water intake logging.
* Daily progress tracking with goal visualization.
* Weekly chart for hydration trends.

**Non-Goals (MVP):**

* Multi-user support.
* Cloud sync or account system.
* Advanced health analytics.

### **3. User Stories (Prioritized)**

**MVP Stories:**

1. Log water intake (multiple entries per day).
2. Track daily progress toward 3L goal.
3. Display weekly hydration chart.

**Future Stories:**

* Customizable daily goal.
* Notifications/reminders.
* Unit conversion (ml ↔ oz).
* Export/share hydration data.

### **4. Functional Requirements**

* Input field for intake amount.
* Progress bar showing % of goal.
* Weekly chart with bars per day.
* Persistent data storage.

### **5. User Flows**

**Flow 1: Log Intake**

1. Enter amount (ml).
2. Tap “Add.”
3. Progress bar updates.

**Flow 2: View Progress**

* Progress bar shows % completed toward goal.

**Flow 3: Weekly Chart**

* Chart shows intake for last 7 days vs 3L goal.

### **6. Success Metrics**

* ✅ Logging takes <5 seconds per entry.
* ✅ Progress updates instantly.
* ✅ Weekly chart loads in <500ms.
* ✅ 80%+ daily active users consistently log intake.

### **7. Release Plan**

**MVP (Phase 1):**

* Intake logging.
* Daily progress bar.
* Weekly chart.

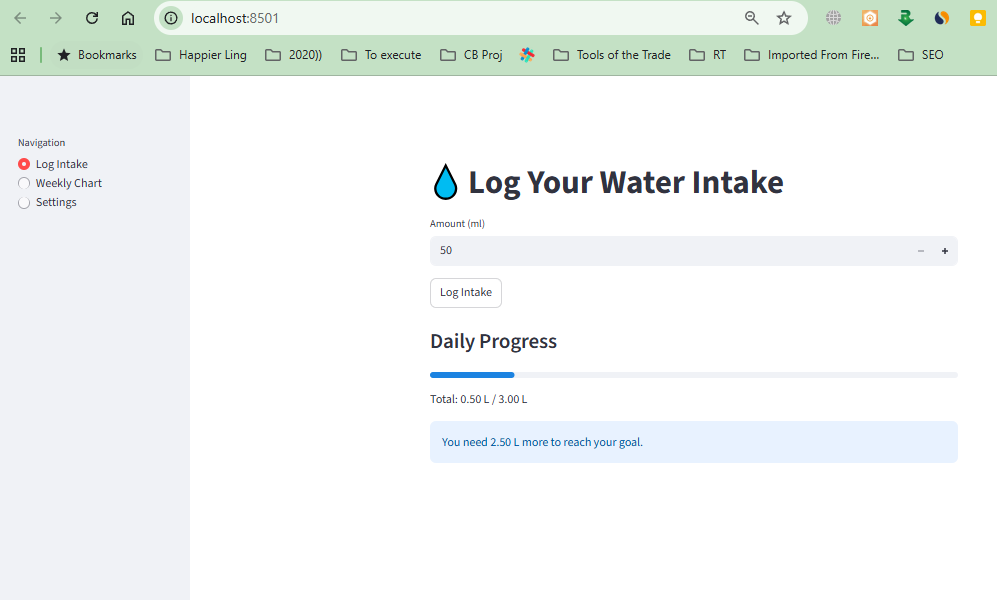
**Phase 2 (Nice-to-Have):**

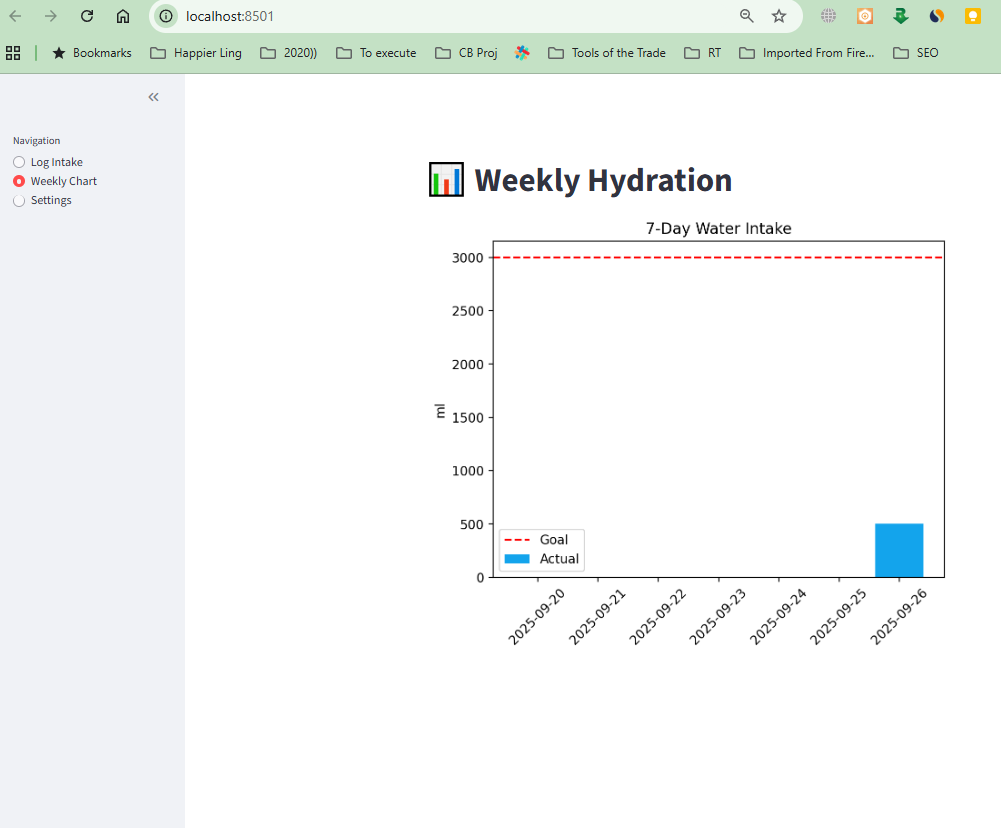
* Customizable daily goal.
* Unit options (ml/oz).
* Local notifications.

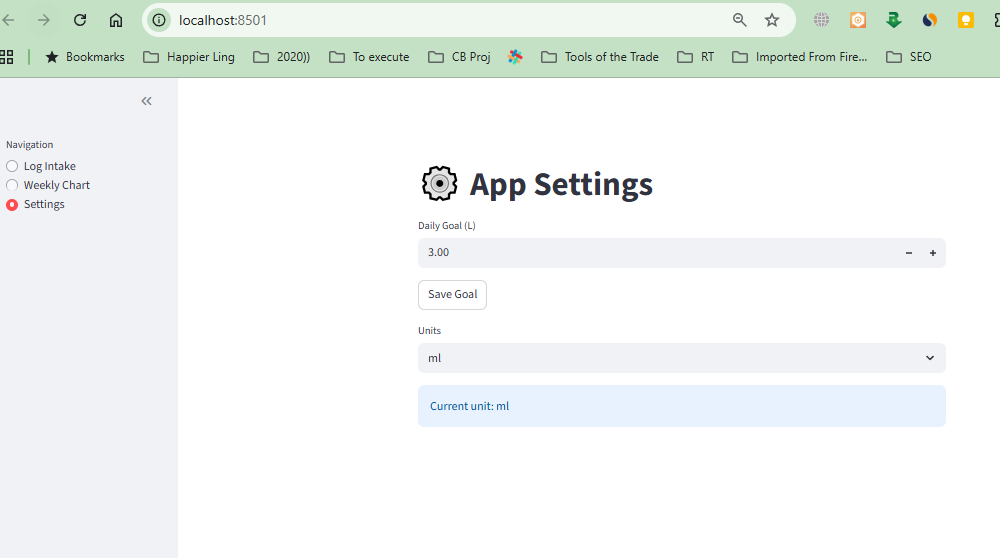
**Phase 3 (Advanced):**

* Cloud sync + accounts.
* Long-term hydration analytics.
* Integration with health apps (Apple Health, Google Fit).

✅ With this PRD + spec, devs can start building the **Water Intake Tracker MVP** right away.







**\*\*\*\*\* Day 7 (Mon, Sep 15):**

**Gym Workout Logger 🏋️**

• Log exercises (sets, reps, weight)

• Store history in a table

• Show weekly progress graph

Let’s build the **end-to-end technical documentation + PRD** for the **Gym Workout Logger App**.

# **📄 Technical Documentation: Gym Workout Logger 🏋️**

## **1. Problem Statement & Goals**

**Problem Statement:** Fitness enthusiasts often struggle to track workouts consistently. Without logging exercises, sets, reps, and weights, it’s difficult to measure progress and stay motivated.

**Goal:** Build a **Gym Workout Logger App** that:

* Lets users log exercises with sets, reps, and weights.
* Stores workout history in a table for review.
* Visualizes weekly progress with a graph.

**Success Criteria:**

* Users can log multiple exercises per session.
* History is persistently stored.
* Weekly progress graph shows clear trends (e.g., total volume lifted per week).

## **2. Personas & User Journeys**

**Persona 1: Beginner Gym-Goer**

* Needs: Simple logging tool to track workouts.

**Persona 2: Fitness Enthusiast**

* Needs: Detailed history to measure performance improvements.

**Persona 3: Trainer/Coach**

* Needs: Quick overview of client progress.

**User Journey:**

1. User opens the app and logs exercises (name, sets, reps, weight).
2. System stores the entry in history.
3. User views workout history in a table.
4. User checks weekly progress graph for performance trends.

## **3. Epics & User Stories (with Acceptance Criteria)**

### **Epic 1: Log Exercises**

**User Story:** As a user, I want to log my exercises with sets, reps, and weight so that I can track my workouts.

**Acceptance Criteria:**

* Input fields: exercise name, sets, reps, weight (kg/lbs).
* Multiple entries allowed per session.
* Entries timestamped by date.

### **Epic 2: Store Workout History**

**User Story:** As a user, I want to see a history of my logged exercises so that I can review past workouts.

**Acceptance Criteria:**

* History displayed in table format.
* Columns: Date, Exercise, Sets, Reps, Weight.
* Data stored persistently (local storage for MVP).

### **Epic 3: Weekly Progress Graph**

**User Story:** As a user, I want to see my weekly progress so that I can measure improvements over time.

**Acceptance Criteria:**

* Graph shows weekly trend (e.g., total weight lifted per week).
* X-axis = weeks, Y-axis = total volume.
* Volume formula = sets × reps × weight.

## **4. System Architecture / Workflows**

**Architecture:**

* **Frontend:** Input form, history table, progress chart.
* **Storage:** Local storage (MVP); DB backend for future.
* **Visualization:** Chart library (e.g., Chart.js / Recharts).

**Workflow:**

1. User logs exercise → stored in local database.
2. History table updates instantly.
3. Weekly progress calculation → aggregated from stored data.
4. Graph rendered with weekly totals.

## **5. Technical Specifications**

### **Data Model**

{

"exercises": [

{

"date": "2025-09-25",

"exercise": "Bench Press",

"sets": 3,

"reps": 10,

"weight": 60

},

{

"date": "2025-09-25",

"exercise": "Squats",

"sets": 4,

"reps": 8,

"weight": 80

}

],

"weekly\_totals": {

"2025-W39": 3240,

"2025-W38": 2800

}

}

### **Calculations**

* **Total Volume (per entry):** sets × reps × weight
* **Weekly Total:** Sum of all volumes for a week.

### **APIs (future)**

* POST /log-exercise → Log new exercise.
* GET /history → Retrieve all logs.
* GET /weekly-progress → Return aggregated data.

## **6. Non-Functional Requirements**

* **Performance:** Logging and graph updates <200ms.
* **Persistence:** Data stored locally for MVP.
* **Usability:** Simple UI for quick entry between sets.
* **Scalability:** Should handle thousands of entries smoothly.

## **7. Open Questions / Assumptions**

* Should users be able to **edit/delete past entries**?
* Should weights support both **kg and lbs** with toggle?
* Should graph show **per-exercise progress** (e.g., bench press strength over time) or only total volume?
* Should we allow **exporting history** (CSV, PDF)?

# **📑 Product Requirements Document (PRD)**

## **Product: Gym Workout Logger 🏋️**

### **1. Overview**

A workout tracker that lets users log exercises with sets, reps, and weights, view history, and track progress through weekly graphs.

### **2. Goals & Non-Goals**

**Goals:**

* Enable fast and simple logging of workouts.
* Store workout history.
* Visualize weekly progress trends.

**Non-Goals (MVP):**

* Social/community features.
* Automated workout recommendations.
* Integration with wearables or fitness trackers.

### **3. User Stories (Prioritized)**

**MVP Stories:**

1. Log exercises (sets, reps, weight).
2. View workout history in a table.
3. See weekly progress graph (total volume lifted).

**Future Stories:**

* Edit/delete past entries.
* Track per-exercise strength progress.
* Unit toggle (kg ↔ lbs).
* Export/share history.
* Cloud sync + multi-device access.

### **4. Functional Requirements**

* **Input Form:** Exercise name, sets, reps, weight.
* **History Table:** Displays logs with timestamp.
* **Weekly Graph:** Shows total volume over last 7 days or by week.
* **Validation:** Inputs must be positive numbers.

### **5. User Flows**

**Flow 1: Log Workout**

1. Enter exercise name, sets, reps, weight.
2. Tap “Log.”
3. Entry stored & table updates.

**Flow 2: View History**

* Table lists all past entries (sortable by date/exercise).

**Flow 3: Weekly Progress**

* Graph shows weekly total volume vs previous weeks.

### **6. Success Metrics**

* ✅ Logging takes <5 seconds per exercise.
* ✅ Graph updates instantly.
* ✅ Users log workouts at least 3x per week (engagement).

### **7. Release Plan**

**MVP (Phase 1):**

* Exercise logging.
* History table.
* Weekly progress graph.

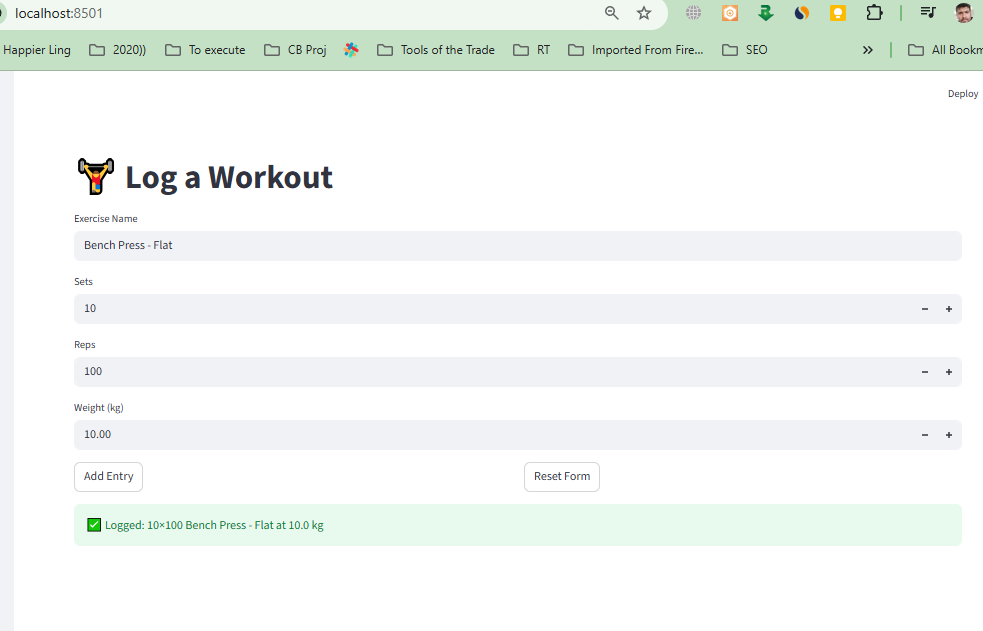
**Phase 2 (Nice-to-Have):**

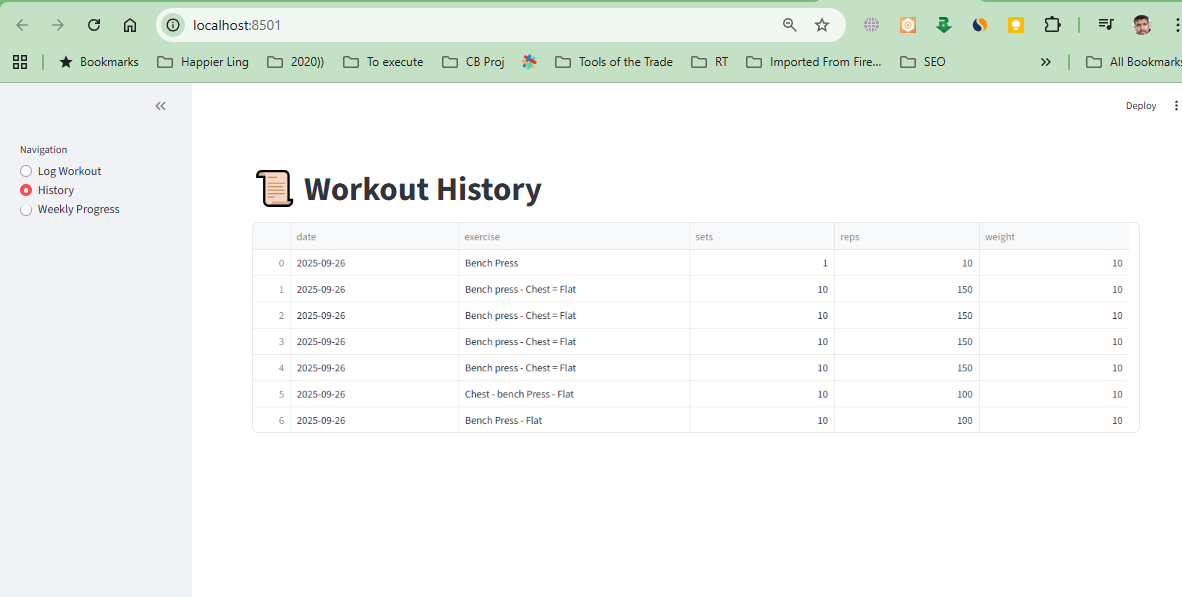
* Edit/delete entries.
* Per-exercise progress visualization.
* Unit toggle (kg/lbs).

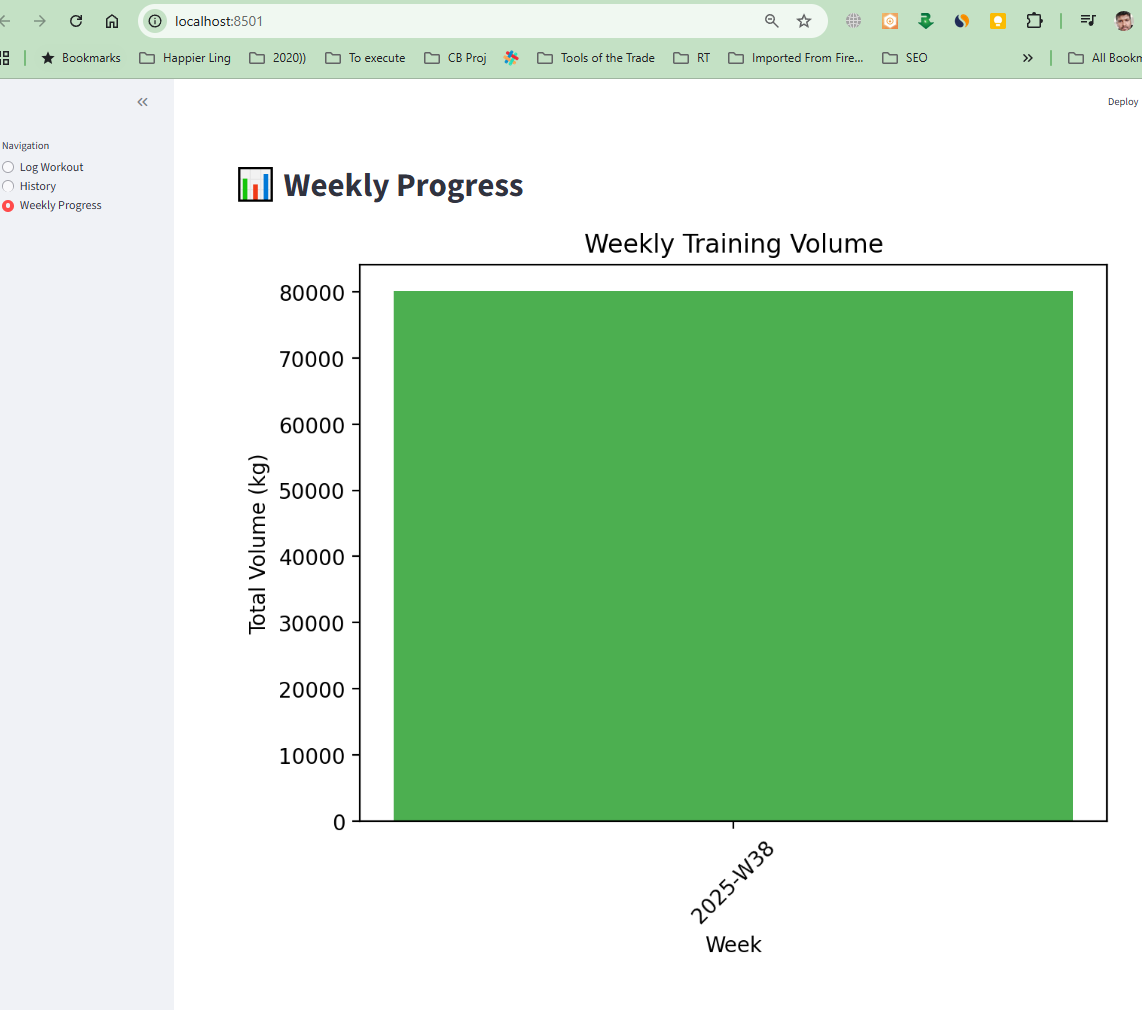
**Phase 3 (Advanced):**

* Cloud sync + accounts.
* Integration with wearables (Apple Watch, Fitbit).
* Smart insights & workout recommendations.

✅ With this PRD + spec, the **Gym Workout Logger MVP** is fully defined and can be built immediately.







**\*\*\*\*\* Day 8 (Tue, Sep 16):**

**Currency Converter 💱**

• Convert between INR, USD, EUR, etc. (static rates)

• Simple dropdown + number input

**full technical documentation + PRD** for the **Currency Converter App (with static rates)**.

# **📄 Technical Documentation: Currency Converter 💱**

## **1. Problem Statement & Goals**

**Problem Statement:** Users frequently need to quickly convert amounts between currencies (INR, USD, EUR, etc.). Live rates are not always required, and static conversion is sufficient for a simple utility app.

**Goal:** Build a **Currency Converter App** with:

* Input: Amount + dropdowns for source and target currencies.
* Conversion: Based on predefined static rates.
* Output: Converted amount shown instantly.

**Success Criteria:**

* User can select any two currencies from the supported list.
* Conversion result updates instantly after input/change.
* Static rates are accurate and easy to update in code.

## **2. Personas & User Journeys**

**Persona 1: Student**

* Needs: Quick reference conversions for study/travel planning.

**Persona 2: Traveler**

* Needs: Simple offline tool for estimating costs.

**User Journey:**

1. User selects source currency (e.g., INR).
2. User enters an amount.
3. User selects target currency (e.g., USD).
4. App instantly shows converted amount.

## **3. Epics & User Stories (with Acceptance Criteria)**

### **Epic 1: Input & Currency Selection**

**User Story:** As a user, I want to enter an amount and select currencies so that I can perform a conversion.

**Acceptance Criteria:**

* Numeric input field for amount.
* Dropdown for source currency.
* Dropdown for target currency.
* Validation: amount must be > 0.

### **Epic 2: Conversion Calculation**

**User Story:** As a user, I want to see the converted amount instantly so that I can know the equivalent value.

**Acceptance Criteria:**

* Static conversion rates stored in the app.
* Result updates instantly on input or selection change.
* Rounds to 2 decimal places.

### **Epic 3: Output Display**

**User Story:** As a user, I want to see the converted amount clearly so that I can read it quickly.

**Acceptance Criteria:**

* Result displayed in text format.
* Format: <amount> <source\_currency> = <result> <target\_currency>.

## **4. System Architecture / Workflows**

**Architecture:**

* **Frontend:** Input box + dropdowns + result display.
* **Logic:** Conversion using static rate matrix.
* **Backend:** Not required (all static).

**Workflow:**

1. User inputs amount + selects source & target.
2. System fetches static rate from a conversion matrix.
3. Formula: converted\_amount = input × (target\_rate / source\_rate)
4. Result displayed instantly.

## **5. Technical Specifications**

### **Static Rates Example**

(Base currency = USD for simplicity)

{

"USD": 1,

"INR": 83.25,

"EUR": 0.92,

"GBP": 0.79,

"JPY": 148.5

}

### **Conversion Formula**

converted\_amount = amount × (rate[to\_currency] / rate[from\_currency])

### **Example**

* Input: 1000 INR → USD
* Formula: 1000 × (1 / 83.25) ≈ 12.01 USD

### **Data Model**

{

"amount": 1000,

"from\_currency": "INR",

"to\_currency": "USD",

"result": 12.01

}

## **6. Non-Functional Requirements**

* **Performance:** Conversion <50ms.
* **Usability:** Mobile-first, minimal UI.
* **Offline:** Works without internet (since static rates).
* **Scalability:** Easy to add new currencies by updating rate table.

## **7. Open Questions / Assumptions**

* Should we allow **swap button** (switch source ↔ target)?
* Should users be able to **update static rates** manually?
* Should the result allow **custom decimal precision** (2 vs 4 decimals)?
* Should we support **more than 10 currencies** or keep it minimal?

# **📑 Product Requirements Document (PRD)**

## **Product: Currency Converter 💱**

### **1. Overview**

A lightweight currency converter app using **static exchange rates**, allowing quick conversions between INR, USD, EUR, and other major currencies.

### **2. Goals & Non-Goals**

**Goals:**

* Quick conversions between currencies.
* Simple dropdown + input UI.
* Offline functionality with static rates.

**Non-Goals (MVP):**

* Real-time/live exchange rates.
* Complex analytics/history.
* Integration with financial services.

### **3. User Stories (Prioritized)**

**MVP Stories:**

1. Input amount + select source & target currencies.
2. Display converted amount instantly.
3. Provide static conversion rates for INR, USD, EUR, GBP, JPY.

**Future Stories:**

* Swap button to switch currencies.
* Customizable static rates.
* Add more currencies.
* Conversion history (recent lookups).

### **4. Functional Requirements**

* Input field: amount (numeric).
* Dropdowns: from\_currency, to\_currency.
* Output field: converted result.
* Conversion formula applied using static matrix.

### **5. User Flows**

**Flow 1: Conversion**

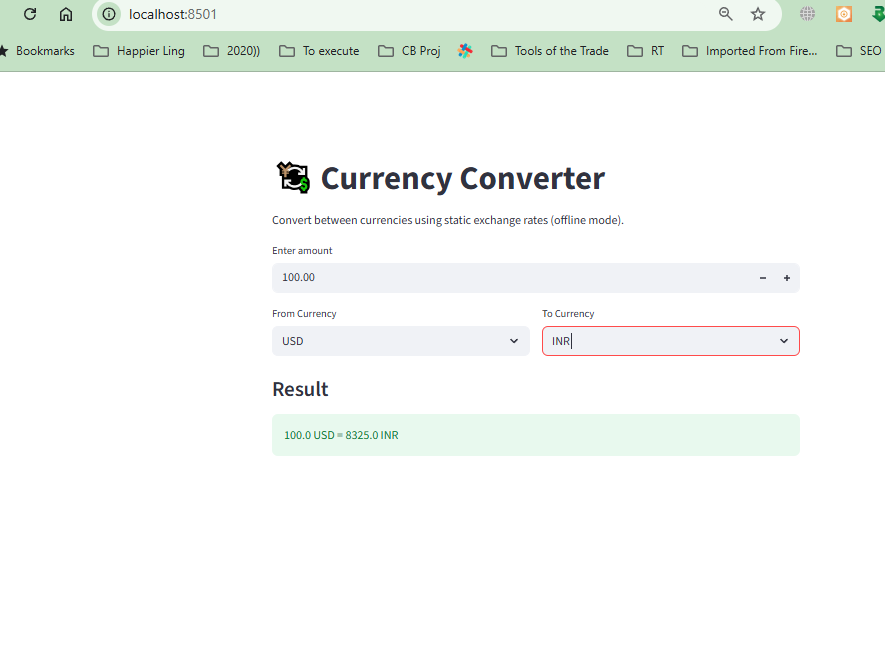
1. Enter amount.
2. Select source currency.
3. Select target currency.
4. Result instantly displayed.

**Flow 2: Error Handling**

* If invalid amount → show error (“Enter valid number”).
* If same currency selected → show result = input.

### **6. Success Metrics**

* ✅ Conversion result <50ms.
* ✅ Error handling always works correctly.
* ✅ At least 90% of users find app easy to use (usability testing).



**\*\*\*\*\*\* Day 09 Restaurant Order & Billing App 🍔**

Display menu items with prices.

User selects items + quantity.

Generate a bill summary (subtotal + tax + total).

Option to download invoice as CSV/PDF.  
  
the **end-to-end technical documentation + PRD** for the **Restaurant Order & Billing App**.

# **📄 Technical Documentation: Restaurant Order & Billing App 🍔**

## **1. Problem Statement & Goals**

**Problem Statement:** Restaurants and food stalls often need a quick digital tool to display menus, take orders, calculate bills, and generate invoices. Existing POS systems can be too complex or expensive for small setups.

**Goal:** Build a **lightweight Restaurant Order & Billing App** that:

* Displays menu items with prices.
* Lets users select items + quantity.
* Generates a bill (subtotal, tax, total).
* Provides an option to **download invoice as CSV/PDF**.

**Success Criteria:**

* Menu is clearly displayed.
* Order selection is simple (add/remove items, adjust quantity).
* Bill summary is accurate and formatted.
* Invoice export works in both CSV and PDF formats.

## **2. Personas & User Journeys**

**Persona 1: Customer (Self-Ordering Kiosk)**

* Wants to view menu and place an order.

**Persona 2: Restaurant Cashier/Waiter**

* Needs a fast billing tool to handle orders and generate invoices.

**User Journey:**

1. User views the menu with items + prices.
2. User selects items and quantities.
3. App generates bill summary (subtotal, tax, total).
4. User/downloads invoice (CSV or PDF).

## **3. Epics & User Stories (with Acceptance Criteria)**

### **Epic 1: Display Menu**

**User Story:** As a user, I want to view menu items with prices so that I can select what to order.

**Acceptance Criteria:**

* Menu displayed in grid/list format.
* Each item has a name, price, and “Add” button.

### **Epic 2: Select Items + Quantity**

**User Story:** As a user, I want to select items and adjust quantities so that I can create my order.

**Acceptance Criteria:**

* Quantity selector (+/– buttons).
* Multiple items can be added.
* Item subtotal updates with quantity changes.

### **Epic 3: Generate Bill Summary**

**User Story:** As a user, I want to see the bill summary so that I know the total cost.

**Acceptance Criteria:**

* Show subtotal, tax (configurable %, e.g., 5%), and total.
* Bill updates dynamically as items are added/removed.

### **Epic 4: Download Invoice**

**User Story:** As a user, I want to download the invoice in CSV or PDF so that I can keep a record of my order.

**Acceptance Criteria:**

* CSV file with columns: Item, Quantity, Price, Subtotal.
* PDF invoice with restaurant name, items, subtotal, tax, total.

## **4. System Architecture / Workflows**

**Architecture:**

* **Frontend:** Menu UI, order cart, bill summary, export option.
* **Logic:** Billing & tax calculation, invoice generation.
* **Storage:** Local storage for session; DB (future) for multiple orders.
* **Libraries:** PDF generator (jsPDF / Puppeteer), CSV export (built-in JS).

**Workflow:**

1. User selects menu items + quantities.
2. App calculates subtotal, applies tax, computes total.
3. Bill summary displayed dynamically.
4. User downloads invoice as CSV/PDF.

## **5. Technical Specifications**

### **Data Model**

{

"menu": [

{ "id": 1, "name": "Burger", "price": 120 },

{ "id": 2, "name": "Fries", "price": 80 },

{ "id": 3, "name": "Coke", "price": 50 }

],

"order": [

{ "item\_id": 1, "name": "Burger", "quantity": 2, "price": 120, "subtotal": 240 },

{ "item\_id": 2, "name": "Fries", "quantity": 1, "price": 80, "subtotal": 80 }

],

"bill": {

"subtotal": 320,

"tax": 16,

"total": 336

}

}

### **Billing Logic**

* **Subtotal:** sum of (quantity × price)
* **Tax:** subtotal × tax\_rate
* **Total:** subtotal + tax

### **APIs (future)**

* GET /menu → Fetch menu items.
* POST /order → Save order.
* GET /invoice/:id → Fetch invoice details.

## **6. Non-Functional Requirements**

* **Performance:** Bill updates <100ms.
* **Usability:** Mobile-first, clean design.
* **Scalability:** Support 100+ menu items.
* **Reliability:** Invoice download works consistently.

## **7. Open Questions / Assumptions**

* Should tax be **fixed (e.g., 5%)** or configurable by restaurant?
* Should invoices include **restaurant details (logo, address)**?
* Should app support **multiple orders per session** or just one?
* Should users be able to **save past orders**?

# **📑 Product Requirements Document (PRD)**

## **Product: Restaurant Order & Billing App 🍔**

### **1. Overview**

A lightweight app for restaurants to display menu items, take orders, calculate bills, and generate invoices in CSV/PDF format.

### **2. Goals & Non-Goals**

**Goals:**

* Provide a simple ordering and billing solution.
* Support invoice export in CSV and PDF.
* Make system usable offline (no API dependency for MVP).

**Non-Goals (MVP):**

* Payment integration.
* Table/waiter management.
* Inventory tracking.

### **3. User Stories (Prioritized)**

**MVP Stories:**

1. Display menu items with prices.
2. Select items + quantities.
3. Generate bill summary (subtotal + tax + total).
4. Download invoice in CSV and PDF.

**Future Stories:**

* Add discounts/coupons.
* Multiple order tracking (table numbers, takeaways).
* Cloud sync + analytics dashboard.
* Payment integration (UPI, PayPal, Stripe).

### **4. Functional Requirements**

* Menu display with item names and prices.
* Order cart with adjustable quantities.
* Dynamic bill summary with tax.
* Invoice download (CSV & PDF).

### **5. User Flows**

**Flow 1: Place Order**

1. Browse menu.
2. Add items + adjust quantities.
3. View bill summary.

**Flow 2: Generate Invoice**

1. Click “Download Invoice.”
2. Choose CSV or PDF.
3. File downloaded with all order details.

### **6. Success Metrics**

* ✅ Bill summary updates instantly.
* ✅ CSV/PDF invoice generation <2s.
* ✅ 95%+ error-free calculations.

### **7. Release Plan**

**MVP (Phase 1):**

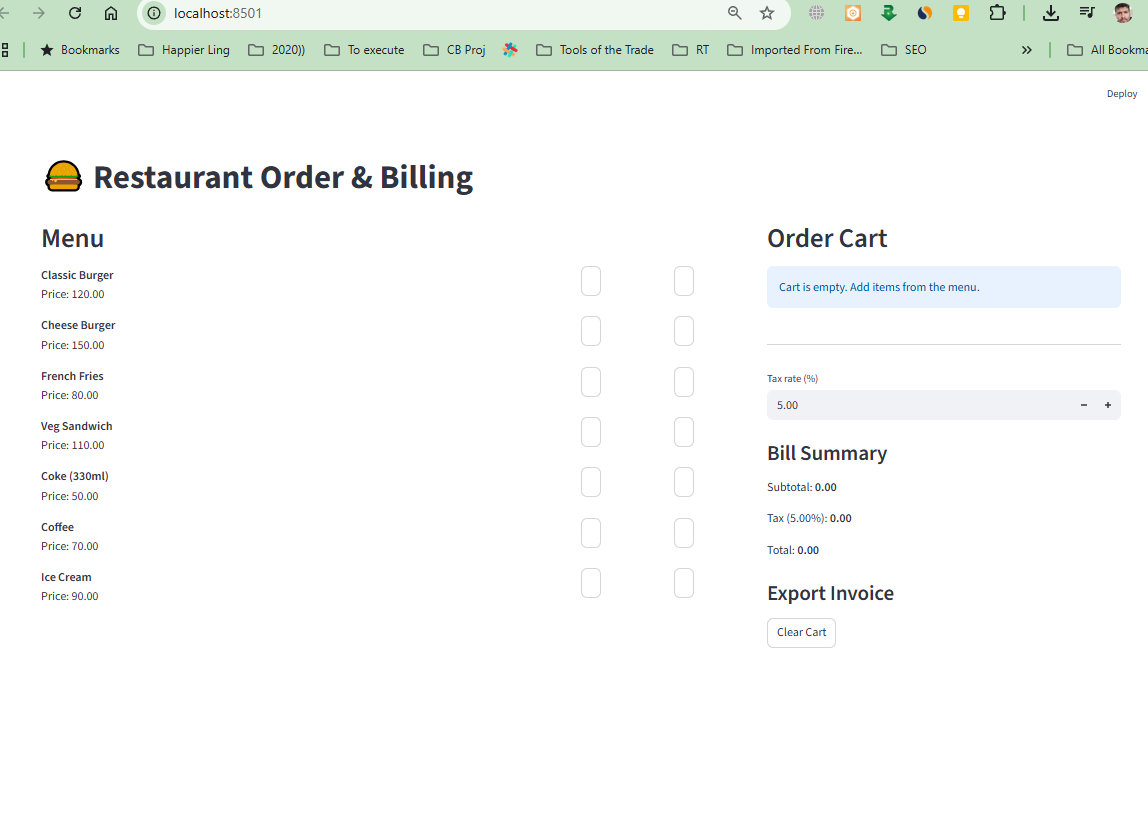
* Menu, order cart, bill summary.
* CSV/PDF invoice export.

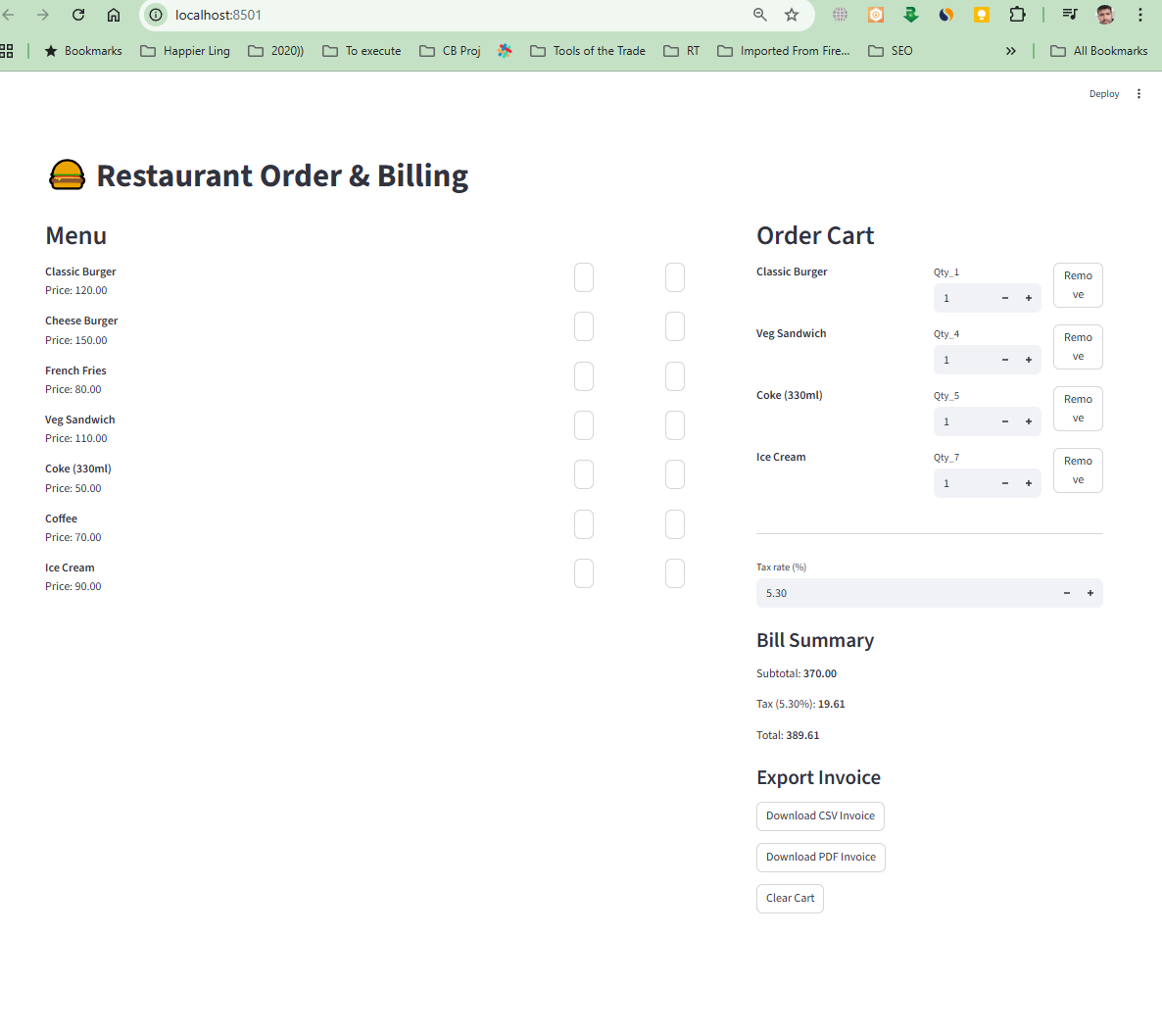
**Phase 2 (Nice-to-Have):**

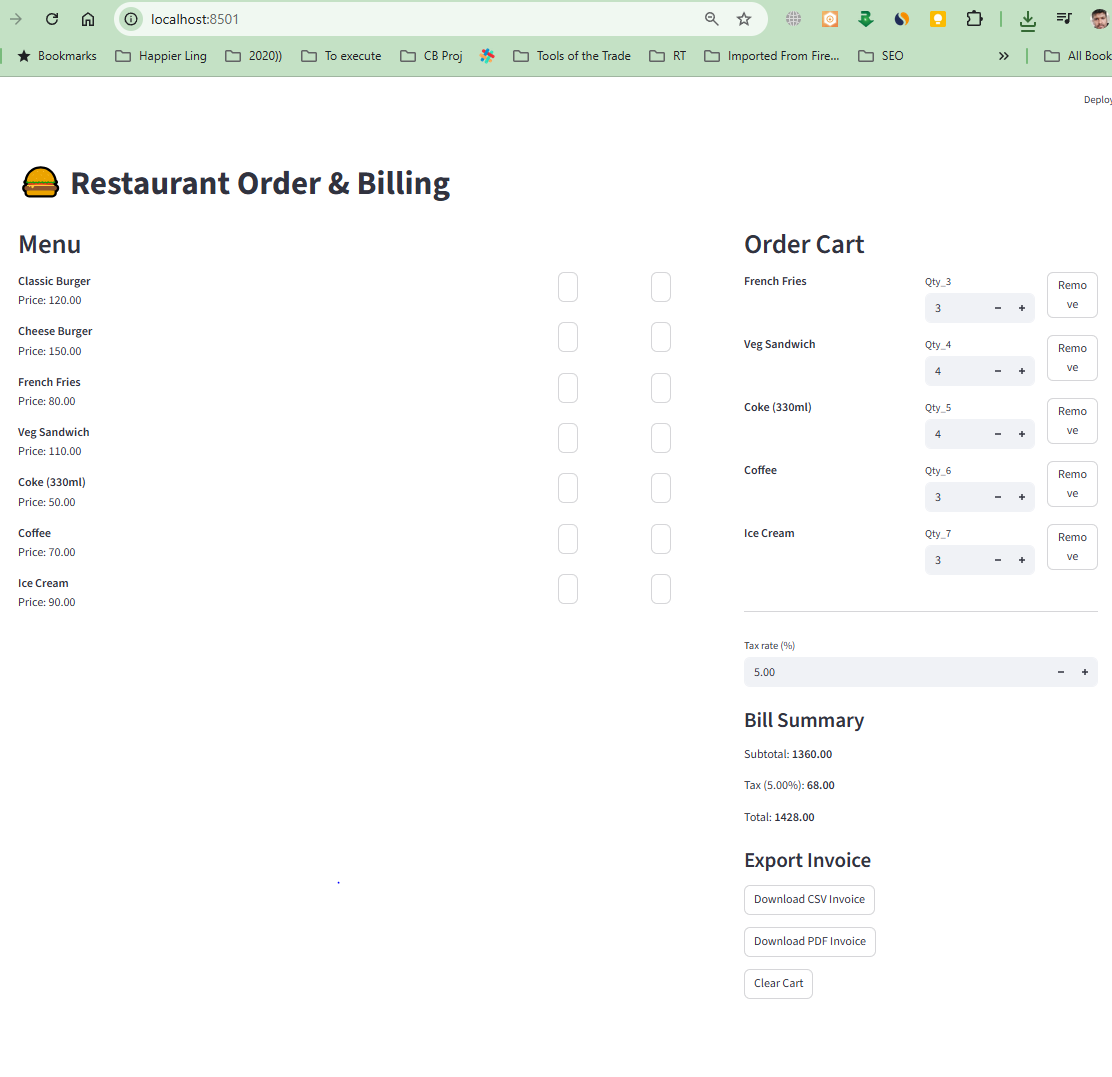
* Discounts & coupons.
* Configurable tax.
* Multiple order management.

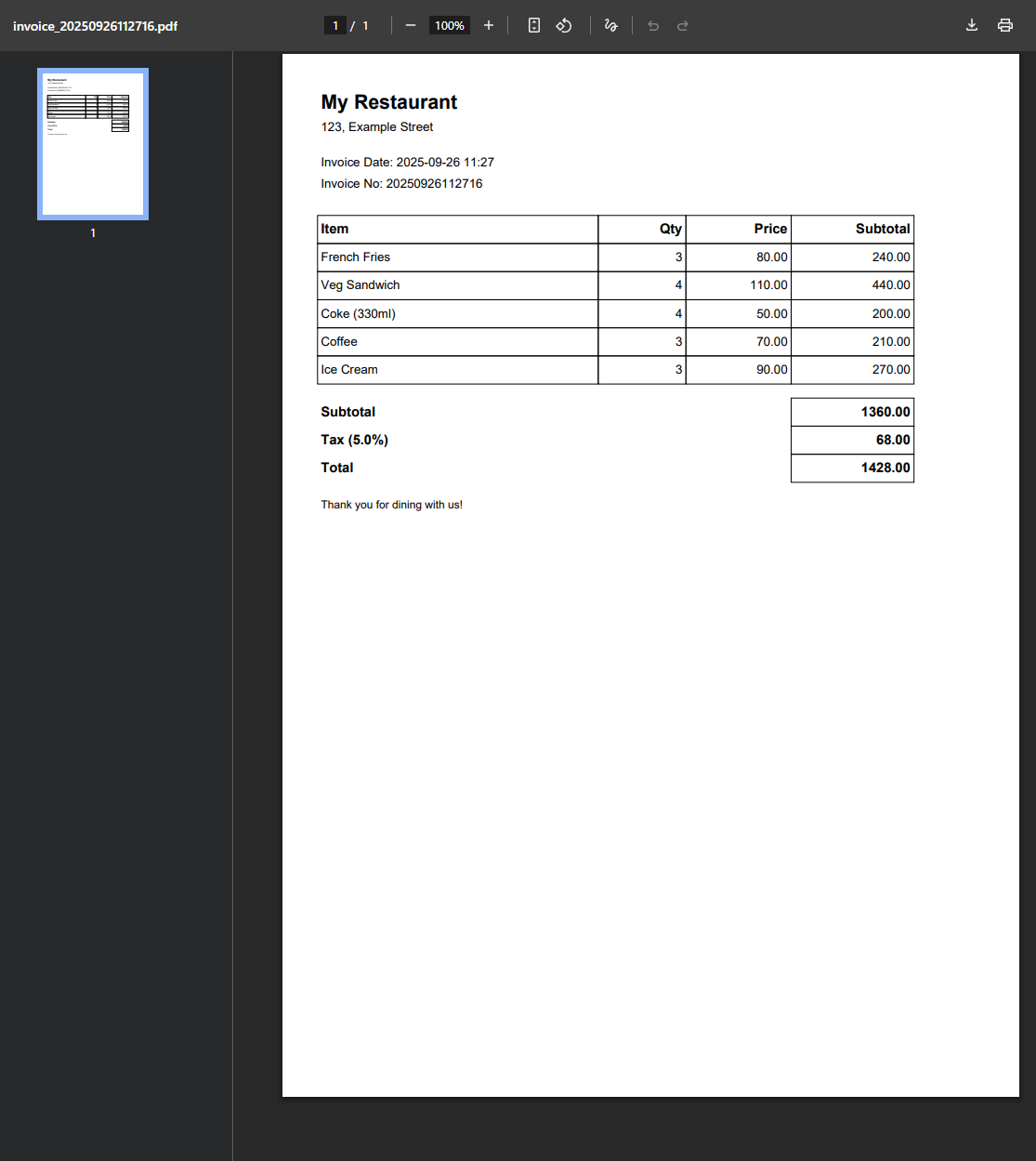
**Phase 3 (Advanced):**

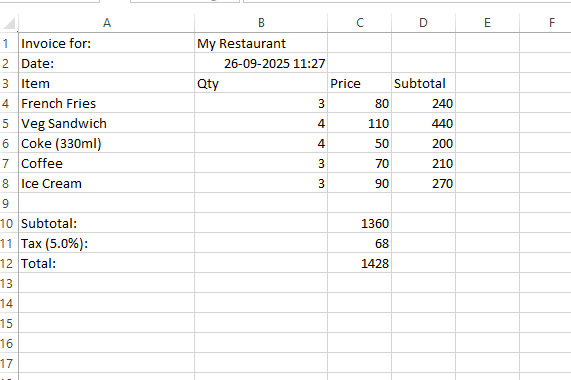
* Cloud storage + analytics.
* Payment integration.
* Multi-branch restaurant support.

✅ This PRD + spec is ready for devs to start building the **Restaurant Order & Billing MVP**.  
  










**\*\*\*\*\* Day 10 challenge**

Event Registration System 🎉

Registration form: Name, Email, Event Choice.

Save all responses in st. session state (or CSV).

Show live count of total registrations.

Allow CSV export for organizers.

the **end-to-end technical documentation + PRD** for the **Event Registration System**.

# **📄 Technical Documentation: Event Registration System 🎉**

## **1. Problem Statement & Goals**

**Problem Statement:** Event organizers need a simple way to collect participant registrations, track sign-ups in real time, and export data for management. Existing solutions can be overly complex for small-scale events.

**Goal:** Build a lightweight **Event Registration System** that:

* Provides a **registration form** (Name, Email, Event Choice).
* Saves responses in **session state (MVP)** or CSV file.
* Shows a **live count of total registrations**.
* Allows organizers to **export registrations as CSV**.

**Success Criteria:**

* Form is easy to fill.
* Data is stored persistently.
* Organizers can see live total count.
* CSV export works reliably.

## **2. Personas & User Journeys**

**Persona 1: Attendee**

* Wants to register quickly without complexity.

**Persona 2: Organizer**

* Wants to track number of registrations in real time.
* Needs CSV export for managing participant lists.

**User Journey (Attendee):**

1. User opens registration form.
2. User enters name, email, selects event.
3. Clicks “Register.”
4. Registration stored + confirmation shown.

**User Journey (Organizer):**

1. Opens dashboard or view.
2. Sees total registrations count.
3. Clicks “Export CSV” → downloads list of participants.

## **3. Epics & User Stories (with Acceptance Criteria)**

### **Epic 1: Registration Form**

**User Story:** As an attendee, I want to register with my name, email, and event choice so that I can attend the event.

**Acceptance Criteria:**

* Fields: Name (text), Email (valid format), Event Choice (dropdown).
* Validation: No empty fields, valid email.
* Confirmation shown after submission.

### **Epic 2: Save Registrations**

**User Story:** As the system, I want to store registrations so that data is available for organizers.

**Acceptance Criteria:**

* Store in **Streamlit session state** for MVP.
* Optionally persist to CSV file on disk.
* Each registration is appended to dataset.

### **Epic 3: Live Count**

**User Story:** As an organizer, I want to see the total number of registrations so that I can monitor attendance in real time.

**Acceptance Criteria:**

* Counter updates instantly when a new registration is submitted.
* Displayed prominently on organizer’s view.

### **Epic 4: CSV Export**

**User Story:** As an organizer, I want to export registrations in CSV format so that I can manage attendees offline.

**Acceptance Criteria:**

* Button: “Export as CSV.”
* Generates file with headers: Name, Email, Event Choice, Timestamp.

## **4. System Architecture / Workflows**

**Architecture:**

* **Frontend:** Streamlit-based form.
* **Storage:** Session state (in-memory) + optional CSV file.
* **Organizer tools:** Live counter + CSV export.

**Workflow:**

1. User submits form → validate inputs.
2. Registration saved in session state (and CSV if enabled).
3. Live counter increments.
4. Organizer clicks “Export” → downloads CSV.

## **5. Technical Specifications**

### **Data Model**

{

"registrations": [

{

"name": "Alice Johnson",

"email": "alice@example.com",

"event": "Workshop A",

"timestamp": "2025-09-25T12:30:00Z"

},

{

"name": "Bob Smith",

"email": "bob@example.com",

"event": "Workshop B",

"timestamp": "2025-09-25T12:32:00Z"

}

],

"count": 2

}

### **Storage**

* **MVP:** In-memory (st.session\_state).
* **Export:** CSV file with headers.

### **CSV Example**

Name,Email,Event,Timestamp

Alice Johnson,alice@example.com,Workshop A,2025-09-25 12:30

Bob Smith,bob@example.com,Workshop B,2025-09-25 12:32

## **6. Non-Functional Requirements**

* **Performance:** Form submit <200ms.
* **Usability:** Mobile-friendly form.
* **Reliability:** CSV export should always include all registrations.
* **Scalability (future):** Move to database when events exceed 1000+ registrations.

## **7. Open Questions / Assumptions**

* Should **duplicate emails** be allowed (e.g., multiple events per user)?
* Should CSV export include **only current session** or **all past events**?
* Should organizers have a **separate dashboard view** or just use the same interface?
* Should attendees receive a **confirmation email** (future feature)?

# **📑 Product Requirements Document (PRD)**

## **Product: Event Registration System 🎉**

### **1. Overview**

A simple event registration system where attendees fill a form, organizers see live registration counts, and export all responses as CSV.

### **2. Goals & Non-Goals**

**Goals:**

* Collect attendee data (Name, Email, Event Choice).
* Store data in session state and allow CSV export.
* Provide live registration count.

**Non-Goals (MVP):**

* Payment processing.
* Email confirmations.
* Multi-event scheduling.

### **3. User Stories (Prioritized)**

**MVP Stories:**

1. Attendee registers with name, email, event choice.
2. System saves registrations in session state.
3. Organizer sees live count of registrations.
4. Organizer exports CSV of all registrations.

**Future Stories:**

* Confirmation email for attendees.
* Admin dashboard with filtering by event.
* Export in multiple formats (Excel, PDF).
* Database backend for large-scale events.

### **4. Functional Requirements**

* Form with fields: Name, Email, Event Choice.
* Validation: required fields, valid email.
* Store data in memory (session state).
* Show live counter of registrations.
* Button: “Export as CSV.”

### **5. User Flows**

**Flow 1: Attendee Registration**

1. Fill form (Name, Email, Event Choice).
2. Submit.
3. Success message.
4. Data stored in session state.

**Flow 2: Organizer View**

1. See total registrations counter.
2. Click “Export CSV.”
3. Download file with all data.

### **6. Success Metrics**

* ✅ Registration submission <200ms.
* ✅ Live counter updates immediately.
* ✅ CSV export always contains all registered entries.

### **7. Release Plan**

**MVP (Phase 1):**

* Registration form.
* Session state storage.
* Live count display.
* CSV export.

**Phase 2 (Nice-to-Have):**

* Confirmation email.
* Filter by event.
* Excel/PDF export.

**Phase 3 (Advanced):**

* Database backend.
* Multi-event management.
* Organizer login & dashboard.

✅ With this PRD + spec, the **Event Registration System MVP** is fully ready to build.

Let’s extend the **Event Registration System 🎉** with the **CSV schema + export function** so devs can implement it directly.

# **📄 CSV Schema for Event Registrations**

The exported CSV should contain **one row per registration**.

### **Headers (columns)**

| **Column** | **Type** | **Description** |
| --- | --- | --- |
| Name | String | Full name of attendee |
| Email | String | Valid email address |
| Event | String | Selected event name |
| Timestamp | String | ISO datetime or formatted (YYYY-MM-DD HH:mm) |

### **Example**

Name,Email,Event,Timestamp

Alice Johnson,alice@example.com,Workshop A,2025-09-25 12:30

Bob Smith,bob@example.com,Workshop B,2025-09-25 12:32

Charlie Lee,charlie@example.com,Workshop A,2025-09-25 12:35

# **📄 Export Function**

Below is a **Python implementation** for Streamlit (since you mentioned st.session\_state).

import streamlit as st

import pandas as pd

from datetime import datetime

# Initialize session state for registrations

if "registrations" not in st.session\_state:

st.session\_state.registrations = []

# Registration form

st.title("🎉 Event Registration")

with st.form("registration\_form"):

name = st.text\_input("Name")

email = st.text\_input("Email")

event\_choice = st.selectbox("Select Event", ["Workshop A", "Workshop B", "Workshop C"])

submitted = st.form\_submit\_button("Register")

if submitted:

if name and email and event\_choice:

st.session\_state.registrations.append({

"Name": name,

"Email": email,

"Event": event\_choice,

"Timestamp": datetime.now().strftime("%Y-%m-%d %H:%M")

})

st.success(f"✅ Thanks {name}, you are registered for {event\_choice}!")

else:

st.error("⚠️ Please fill all fields")

# Live count

st.subheader("📊 Total Registrations")

st.write(len(st.session\_state.registrations))

# Show registrations (organizer view)

if st.session\_state.registrations:

df = pd.DataFrame(st.session\_state.registrations)

st.dataframe(df)

# CSV export

csv = df.to\_csv(index=False).encode("utf-8")

st.download\_button(

label="⬇️ Download CSV",

data=csv,

file\_name="registrations.csv",

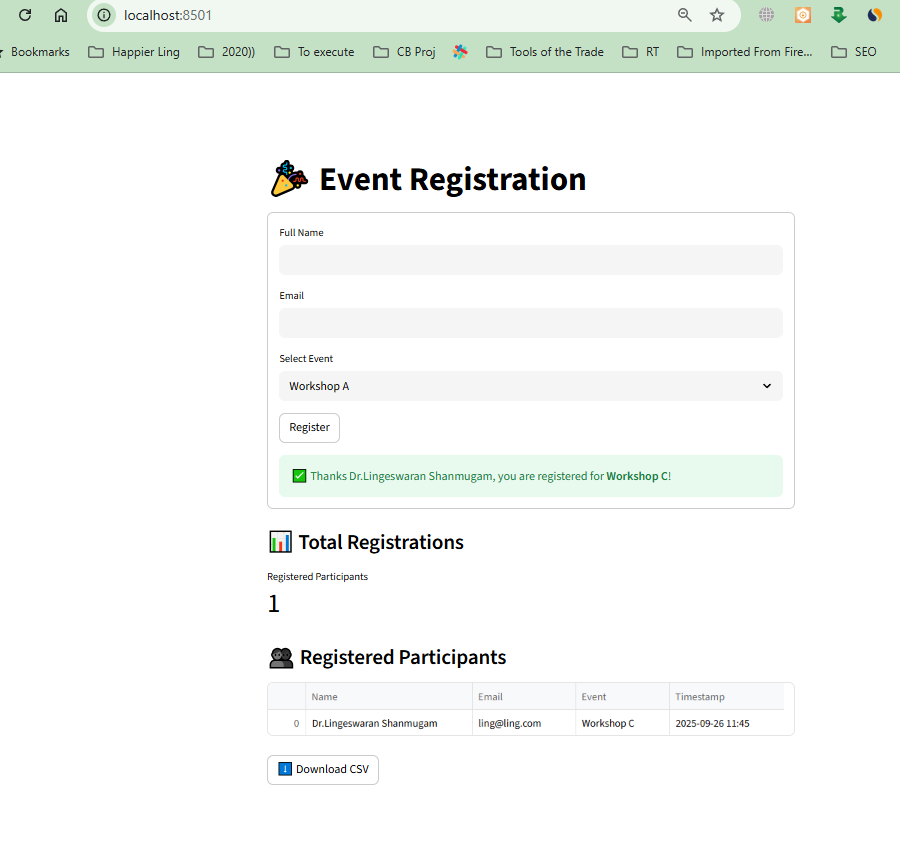
mime="text/csv"

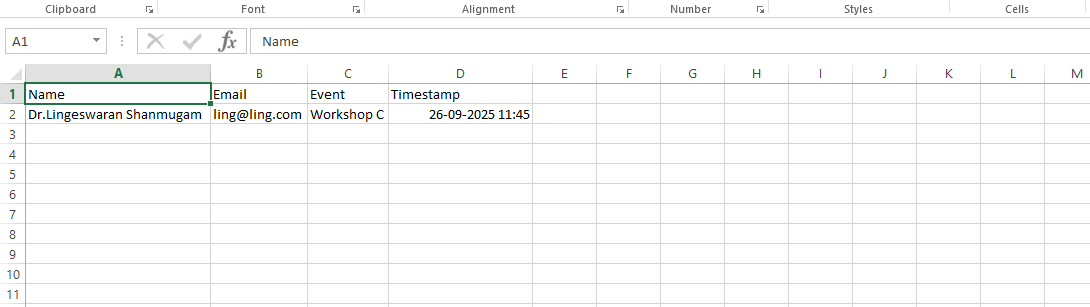
)

### **🔑 How it Works**

1. **Form Submission:** Attendee fills in name, email, event choice.
2. **Session State Storage:** Registration appended to st.session\_state.registrations.
3. **Live Counter:** Uses len(st.session\_state.registrations).
4. **Organizer View:** Displays registrations in a table (st.dataframe).
5. **CSV Export:** Uses Pandas to generate CSV → st.download\_button triggers download.

👉 This version is **MVP-ready**: organizers can directly download all collected registrations as CSV.





**\*\*\*\*\* day 11 Rock, Paper, Scissors – User vs computer → keep score.**

the **Rock, Paper, Scissors Game** (User vs Computer, with score tracking). I’ll prepare **technical documentation + PRD** so devs can build it smoothly.

# **📄 Technical Documentation: Rock, Paper, Scissors Game ✊📄✂️**

## **1. Problem Statement & Goals**

**Problem Statement:** Users enjoy simple casual games like *Rock, Paper, Scissors* for fun. A digital version with score tracking makes it interactive and competitive.

**Goal:** Build a **Rock, Paper, Scissors Game** where:

* User selects Rock, Paper, or Scissors.
* Computer randomly selects one.
* Winner is determined by standard rules.
* Score is tracked across rounds.

**Success Criteria:**

* Game runs smoothly and instantly.
* User vs Computer outcomes are clear.
* Score updates after every round.

## **2. Personas & User Journeys**

**Persona 1: Casual Player**

* Wants quick, fun gameplay.

**Persona 2: Child / Student**

* Uses the game for entertainment and simple logic learning.

**User Journey:**

1. User opens the game.
2. User selects Rock/Paper/Scissors.
3. Computer makes random choice.
4. System decides winner & updates score.
5. Score displayed (User vs Computer).
6. User can play multiple rounds.

## **3. Epics & User Stories (with Acceptance Criteria)**

### **Epic 1: User Input**

**User Story:** As a user, I want to choose Rock, Paper, or Scissors so that I can play against the computer.

**Acceptance Criteria:**

* Buttons for Rock, Paper, Scissors.
* User must select exactly one choice per round.

### **Epic 2: Computer Choice**

**User Story:** As the system, I want to randomly select Rock, Paper, or Scissors so that the game is fair.

**Acceptance Criteria:**

* Computer picks randomly with equal probability.

### **Epic 3: Determine Winner**

**User Story:** As a user, I want to know who won the round so that I can track the game.

**Acceptance Criteria:**

* Rules applied correctly:  
  + Rock beats Scissors
  + Scissors beats Paper
  + Paper beats Rock
* Tie handled correctly.
* Result message displayed.

### **Epic 4: Keep Score**

**User Story:** As a user, I want to see the score so that I know how I’m performing against the computer.

**Acceptance Criteria:**

* Scoreboard: User Wins | Computer Wins | Ties.
* Updates after every round.

## **4. System Architecture / Workflows**

**Architecture:**

* **Frontend:** Game UI (buttons + results + scoreboard).
* **Logic:** Random choice generator + winner calculation.
* **Storage:** Local session (score resets when page refreshed).

**Workflow:**

1. User clicks a choice (Rock/Paper/Scissors).
2. Computer randomly picks one.
3. Compare choices → decide winner.
4. Update scoreboard.
5. Display round result + updated scores.

## **5. Technical Specifications**

### **Data Model**

{

"user\_choice": "Rock",

"computer\_choice": "Scissors",

"round\_result": "User Wins",

"score": {

"user": 3,

"computer": 2,

"ties": 1

}

}

### **Logic**

import random

choices = ["Rock", "Paper", "Scissors"]

def play\_round(user\_choice):

computer\_choice = random.choice(choices)

if user\_choice == computer\_choice:

return "Tie"

elif (

(user\_choice == "Rock" and computer\_choice == "Scissors") or

(user\_choice == "Paper" and computer\_choice == "Rock") or

(user\_choice == "Scissors" and computer\_choice == "Paper")

):

return "User Wins"

else:

return "Computer Wins"

## **6. Non-Functional Requirements**

* **Performance:** Result displayed <100ms.
* **Usability:** Simple UI with clear buttons.
* **Fairness:** Computer choice must be random.
* **Scalability:** Not relevant (single-player).

## **7. Open Questions / Assumptions**

* Should the game **end after fixed rounds** (e.g., best of 5) or be infinite until user stops?
* Should there be a **reset score** button?
* Should the UI show **previous round history** or just current round?

# **📑 Product Requirements Document (PRD)**

## **Product: Rock, Paper, Scissors Game ✊📄✂️**

### **1. Overview**

A lightweight fun game where a user plays Rock, Paper, Scissors against the computer, with live score tracking.

### **2. Goals & Non-Goals**

**Goals:**

* Simple and quick gameplay.
* Clear win/loss/tie logic.
* Score tracking across rounds.

**Non-Goals (MVP):**

* Multiplayer mode.
* Persistent score history.
* Animations/sounds (can be future enhancements).

### **3. User Stories (Prioritized)**

**MVP Stories:**

1. User selects Rock, Paper, or Scissors.
2. Computer randomly selects one.
3. System declares winner.
4. Scoreboard updates (User, Computer, Ties).

**Future Stories:**

* Reset score button.
* Game modes: Best of 5/10.
* Visual animations for choices.
* Persistent score history (local storage).

### **4. Functional Requirements**

* Buttons for Rock, Paper, Scissors.
* Random generator for computer choice.
* Result logic and scoreboard.
* Display of round result + score.

### **5. User Flows**

**Flow 1: Play Round**

1. User clicks Rock/Paper/Scissors.
2. Computer randomly picks.
3. Winner calculated.
4. Score updated & displayed.

**Flow 2: Reset Game (future)**

1. User clicks Reset.
2. Scoreboard resets to zero.

### **6. Success Metrics**

* ✅ Round result displayed instantly.
* ✅ Scoreboard updates correctly.
* ✅ Users play multiple rounds without confusion.

### **7. Release Plan**

**MVP (Phase 1):**

* User vs computer game.
* Score tracking.

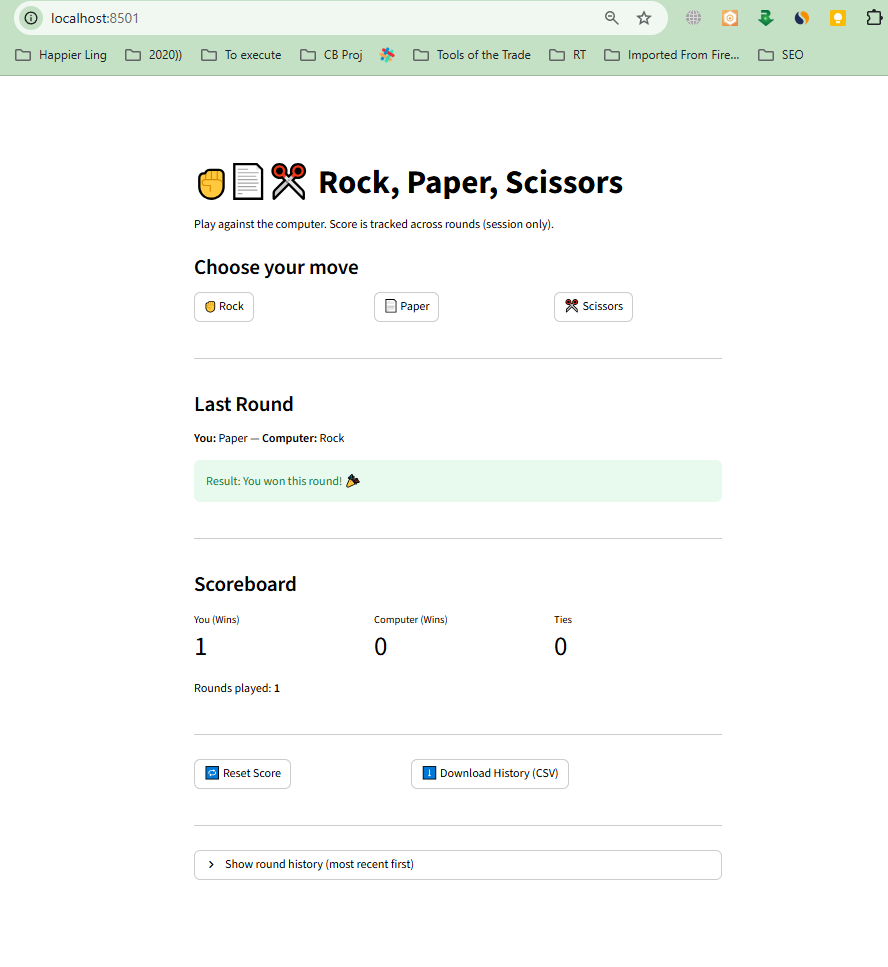
**Phase 2 (Nice-to-Have):**

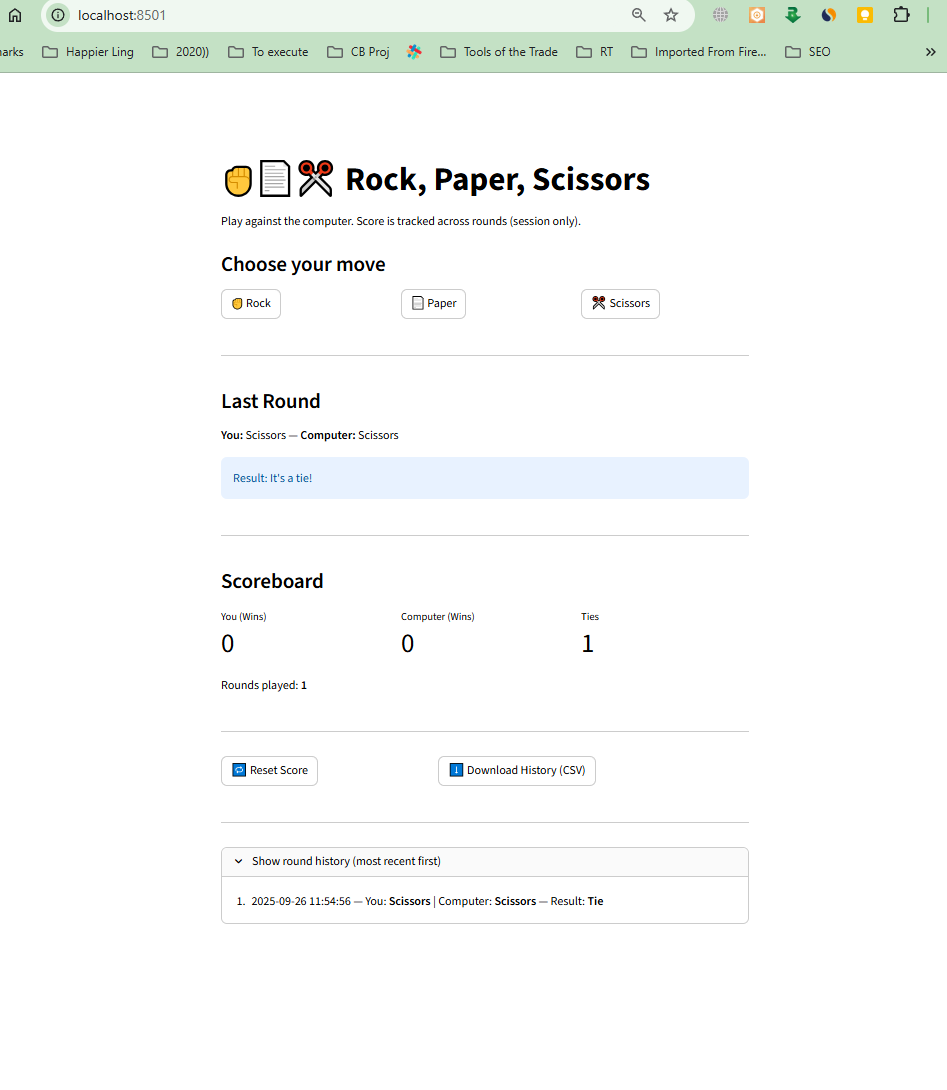
* Reset score button.
* Best of N rounds.

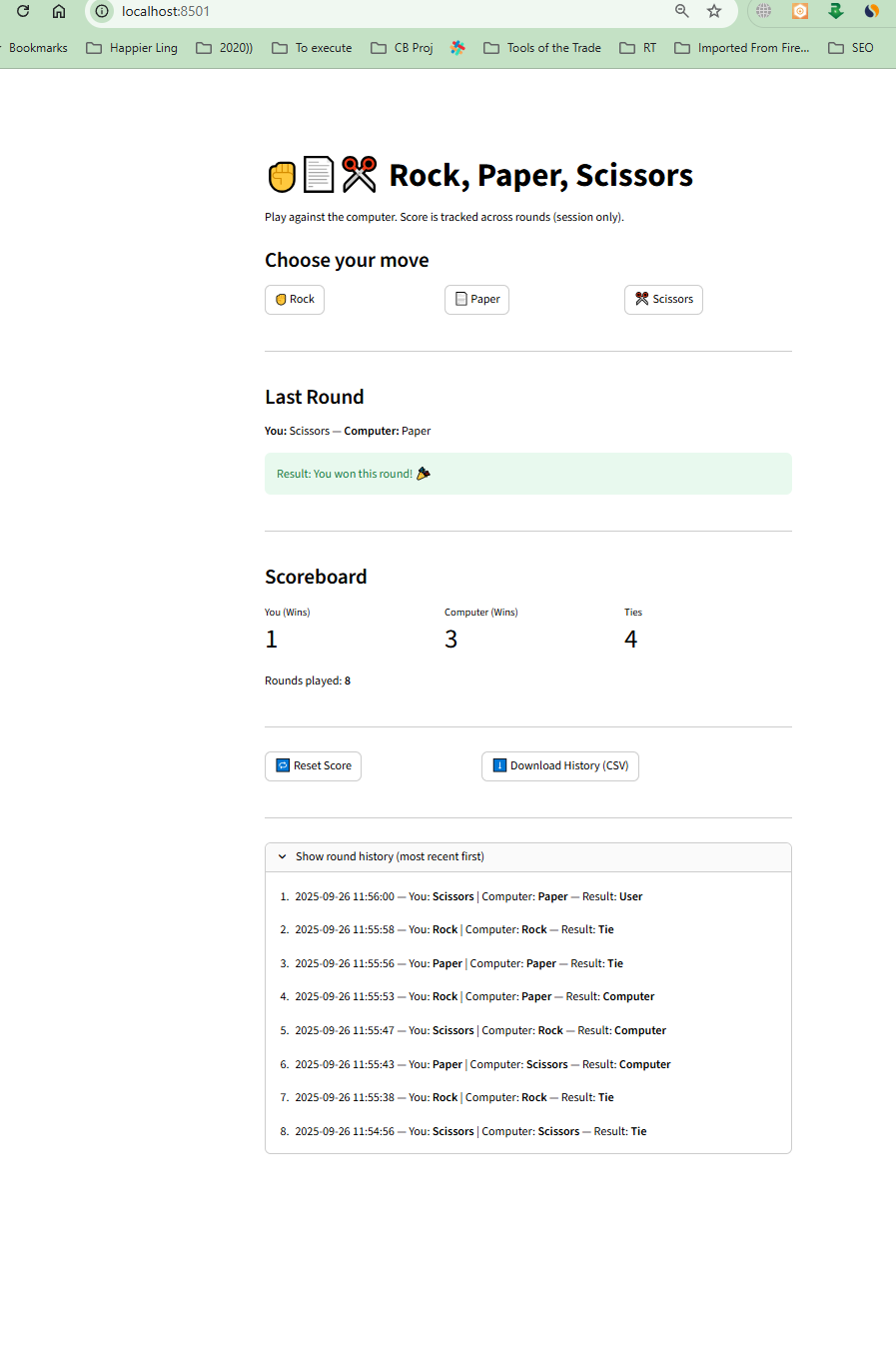
**Phase 3 (Advanced):**

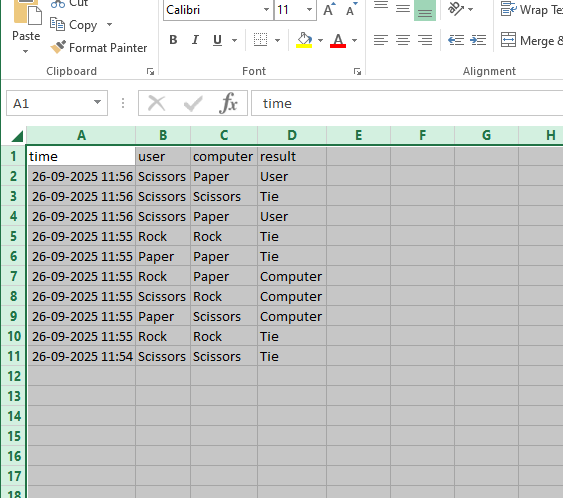
* Animations & sounds.
* Multiplayer mode.
* Persistent score history.

✅ With this PRD + spec, devs can directly implement the **Rock, Paper, Scissors MVP**









**\*\*\*\*\* day 12 Stopwatch – Start/stop/reset timer  
  
the Stopwatch app spec. Quick, focused, and ready for implementation. ⏱️**

# **📄 Technical Documentation: Stopwatch — Start / Stop / Reset Timer**

## **1. Problem statement & goals**

**Problem:  
 Users need a simple, reliable stopwatch for timing activities (workouts, cooking, tests, productivity sprints). Many devices have cluttered timers; a lightweight stopwatch focused on accuracy and usability is useful.**

**Goal:  
 Create a minimal stopwatch that supports Start, Stop, and Reset, displays elapsed time with good precision, and optionally supports laps.**

**Success criteria:**

* **Start/stop actions are instantaneous.**
* **Reset clears elapsed time to 0.**
* **Display updates smoothly (recommended 100ms refresh).**
* **Timekeeping remains accurate across typical browser/device conditions.**

## **2. Personas & user journeys**

**Personas**

* **Athlete / Gym-goer: times sets/rests.**
* **Student / Professional: times tests or focused work sessions.**
* **Home cook / DIYer: tracks short tasks.**

**User journeys**

1. **Open app → press Start → elapsed time begins increasing → press Stop to pause → press Start to resume → press Reset to clear.**
2. **(Optional) Press Lap during running to record lap times; view list of laps.**

## **3. Epics & user stories (with acceptance criteria)**

### **Epic 1 — Core Controls**

**User story: As a user, I want to Start, Stop, and Reset the stopwatch so I can time tasks.  
 Acceptance criteria:**

* **Start button begins timing from current elapsed (0 if reset).**
* **Stop pauses the timer; elapsed time remains visible.**
* **Reset sets elapsed time to 00:00:00.000 and clears laps (if enabled).**
* **Start after Stop resumes from paused elapsed time.**

### **Epic 2 — Time Display**

**User story: As a user, I want to see elapsed time with readable precision so I can measure accurately.  
 Acceptance criteria:**

* **Display format: HH:MM:SS.mmm (hours optional if 0; show at least MM:SS.mmm).**
* **UI updates at least every 100 ms (10 updates/sec) for smoothness; internal time measured with high-resolution timer.**

### **Epic 3 — (Optional) Laps**

**User story: As a user, I want to record lap times while stopwatch is running so I can capture split intervals.  
 Acceptance criteria:**

* **Lap button captures current elapsed time into a list.**
* **Each lap shows lap number, lap time (since previous lap), and cumulative time.**
* **Reset clears lap list.**

### **Epic 4 — Persist / Accuracy (Optional)**

**User story: As a user, I want the stopwatch to remain accurate if the browser tab is backgrounded or device sleeps.  
 Acceptance criteria:**

* **Use system clock deltas (timestamps) rather than relying only on setInterval for accumulated time.**
* **On resume/regain focus, displayed elapsed equals actual real-world elapsed since start minus paused durations.**

## **4. System architecture / workflows**

**Architecture (MVP):**

* **Single-page frontend (React / vanilla JS) — no backend needed.**
* **UI components: Timer display, Start/Stop button, Reset button, (optional) Lap button + Lap list.**

**Workflow:**

1. **When user presses Start:**
   * **record startTimestamp = performance.now() or Date.now() minus any accumulatedPausedTime (if resuming).**
   * **start an animation/update loop (requestAnimationFrame or setInterval) to refresh display.**
2. **When user presses Stop:**
   * **compute elapsed = now - startTimestamp and add to accumulatedElapsed.**
   * **stop update loop.**
3. **When user presses Reset:**
   * **set accumulatedElapsed = 0, startTimestamp = null, clear laps, update display to zero.**
4. **(Lap) Capture nowTotalElapsed and compute lap delta from last lap.**

**Timing approach:**

* **Use performance.now() for high-resolution timing.**
* **Display using computed elapsed = accumulatedElapsed + (running ? performance.now() - startTimestamp : 0).**

## **5. Technical specifications (APIs, data models, functions)**

### **Data model (in-memory)**

**{**

**running: false,**

**startTimestamp: null, // performance.now() when started**

**accumulatedElapsed: 0, // milliseconds**

**laps: [ // optional**

**{ lapNumber: 1, time: 12345, lapTime: 12345 } // time in ms**

**]**

**}**

### **Core functions (pseudocode)**

**function start() {**

**if (!state.running) {**

**state.startTimestamp = performance.now();**

**state.running = true;**

**startUpdateLoop(); // requestAnimationFrame or setInterval**

**}**

**}**

**function stop() {**

**if (state.running) {**

**state.accumulatedElapsed += performance.now() - state.startTimestamp;**

**state.startTimestamp = null;**

**state.running = false;**

**stopUpdateLoop();**

**}**

**}**

**function reset() {**

**state.running = false;**

**state.startTimestamp = null;**

**state.accumulatedElapsed = 0;**

**state.laps = [];**

**updateDisplay(0);**

**}**

**function getElapsed() {**

**return state.accumulatedElapsed + (state.running ? (performance.now() - state.startTimestamp) : 0);**

**}**

**function recordLap() {**

**const elapsed = getElapsed();**

**const lastLapTime = state.laps.length ? state.laps[state.laps.length -1].time : 0;**

**state.laps.push({**

**lapNumber: state.laps.length + 1,**

**time: elapsed,**

**lapTime: elapsed - lastLapTime**

**});**

**}**

### **Display formatting**

* **Convert ms → HH:MM:SS.mmm**
  + **hours = floor(ms / 3\_600\_000)**
  + **minutes = floor((ms % 3\_600\_000) / 60\_000)**
  + **seconds = floor((ms % 60\_000) / 1\_000)**
  + **millis = ms % 1\_000 (pad to 3 digits)**

### **UI behavior**

* **Start button toggles to Pause/Stop (label changes) depending on running.**
* **Disable Lap when not running.**
* **Buttons accessible via keyboard (Enter/Space) and aria labels for accessibility.**

## **6. Non-functional requirements**

* **Accuracy: Keep accurate to within a few milliseconds over normal use; use performance.now() to avoid timer drift.**
* **Responsiveness: UI update ≤ 100 ms interval for smooth readout.**
* **Resource usage: Prefer requestAnimationFrame when animating in foreground; use setInterval fallback or throttling when backgrounded to save battery.**
* **Accessibility: High-contrast display, large digits, keyboard operability, ARIA roles.**
* **Cross-platform: Works in modern browsers and mobile web views.**

## **7. Open questions / assumptions**

* **Assume single-session, in-memory stopwatch (no persistence across page reloads for MVP).**
* **Should we include Lap functionality? (Optional; recommended.)**
* **Should stopwatch support best-of-N rounds or alarms? (Out of scope for MVP.)**
* **Target platforms: web (desktop + mobile). Native apps would require platform-specific timers but same logic.**

# **📑 Product Requirements Document (PRD)**

## **Product: Stopwatch — Start / Stop / Reset Timer ⏱️**

### **1. Overview**

**A minimal, highly-accurate stopwatch for timing tasks with Start, Stop, and Reset controls, plus optional lap recording.**

### **2. Goals & non-goals**

**Goals**

* **Provide accurate elapsed-time display.**
* **Start/Stop/Reset controls with instant feedback.**
* **Clean, mobile-first UI and accessibility.**

**Non-goals (MVP)**

* **Persistent history across sessions.**
* **Complex scheduling/alarms or countdown timers.**
* **Integrations with external services.**

### **3. User stories (prioritized)**

**MVP**

1. **As a user, I can Start the stopwatch so timing begins.**
2. **As a user, I can Stop/Pause the stopwatch so timing pauses.**
3. **As a user, I can Reset the stopwatch to zero.**
4. **As a user, I can read elapsed time with millisecond precision.**

**Nice-to-have**

* **As a user, I can record Lap times.**
* **As a user, I can export lap list as CSV.**
* **As a user, I can set a target time and get notification when reached.**

### **4. Functional requirements**

* **Start button: initiate timing.**
* **Stop button: pause timing and preserve elapsed time.**
* **Reset button: set elapsed to zero and clear laps.**
* **Time display: HH:MM:SS.mmm formatting.**
* **Lap button (optional): record splits.**
* **Keyboard accessibility and screen-reader friendly.**

### **5. User flows**

**Play**

* **User taps Start → timer runs → tap Stop to pause → tap Start to resume → tap Reset to clear.**

**Lap**

* **While running, user taps Lap → new lap appended with lap time and cumulative time.**

### **6. Success metrics**

* **Timer responsiveness: button action reflected within 100ms.**
* **Time display accuracy: no more than minimal drift over 1 hour (use high-res timing).**
* **User satisfaction in usability tests: >90% rate easy to use.**

### **7. Release plan**

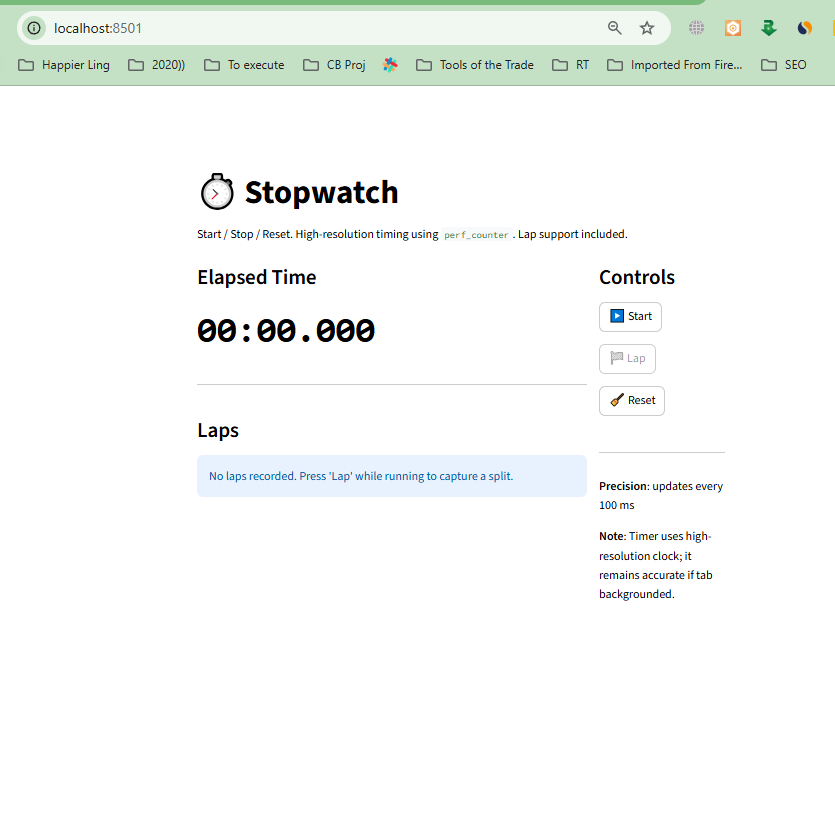
**MVP (Phase 1)**

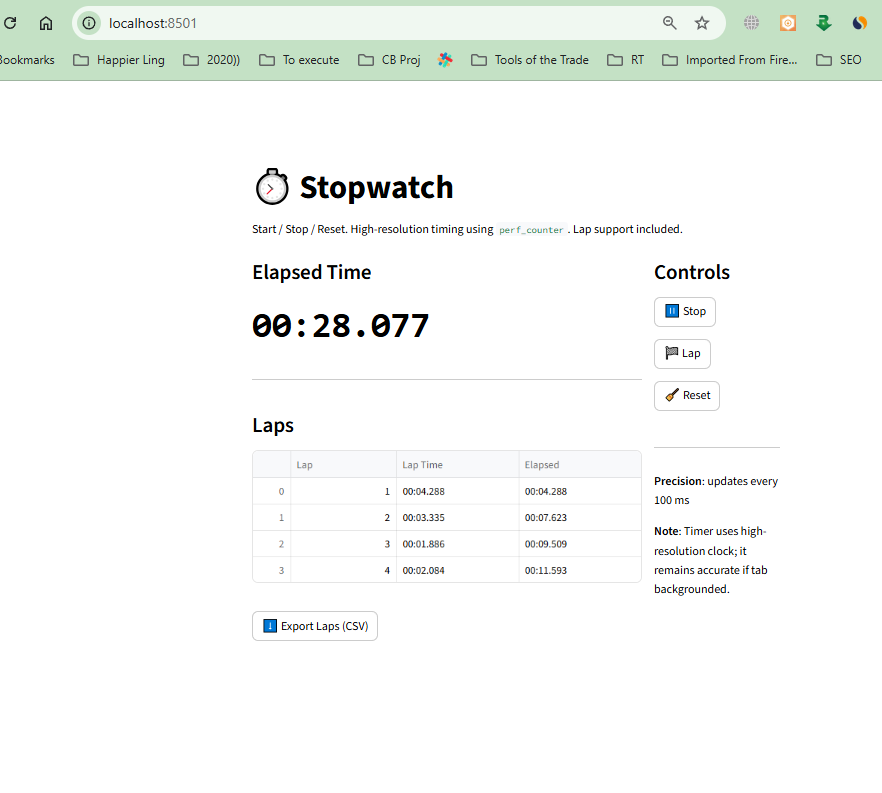
* **Core Start/Stop/Reset, accurate display, mobile-friendly UI.**

**Phase 2**

* **Add Lap recording + lap list UI, CSV export of laps.**

**Phase 3**

* **Add persistence (localStorage) to retain state across reloads, add alarms/alerts and native app ports.**

****

**\*\*\*\*\*\* Day 13 Quiz Game App ❓**

**Features:**

Multiple-choice questions (hardcoded in list/dict).

User selects answers via radio buttons.

Keep score using st.session\_state.

Show final score at the end.

# **📄 Technical Documentation: Quiz Game App ❓**

## **1. Problem statement & goals**

**Problem:** Users want a simple interactive quiz to test knowledge with immediate feedback and a final score. Many quick prototypes need an easy-to-build pattern for questions, UI, and score tracking.

**Goal:** Build a lightweight **Quiz Game App** that:

* Uses a hardcoded list/dict of multiple-choice questions.
* Presents each question with radio-button choices.
* Keeps score in st.session\_state.
* Shows final score at the end and optionally a per-question summary.

**Success criteria:**

* Questions render correctly with choices.
* User selects one answer per question.
* Score is tracked and persisted during the session.
* Final score and optional review are shown at the end.

## **2. Personas & user journeys**

**Personas**

* **Student:** practices quick drills and immediate feedback.
* **Teacher/Trainer:** wants a simple demo quiz for learners.
* **Casual user:** plays a quick quiz for fun.

**User journey**

1. User opens the quiz.
2. User reads question 1 and selects an answer via radio buttons.
3. User clicks **Next** (or **Submit** for single-question flow).
4. Score updates in st.session\_state.
5. Repeat through all questions.
6. App shows final score and per-question feedback.

## **3. Epics & user stories (with acceptance criteria)**

### **Epic 1 — Question Rendering**

**User story:** As a user, I want to see multiple-choice questions so I can choose correct answers.  
 **Acceptance criteria:** Questions display text + 3–5 radio options; only one option selectable.

### **Epic 2 — Answer Submission & Navigation**

**User story:** As a user, I want to submit answers and move to the next question.  
 **Acceptance criteria:** Submit/Next button validates selection, stores response, navigates to next question.

### **Epic 3 — Score Tracking**

**User story:** As a user, I want my score tracked so I can see performance.  
 **Acceptance criteria:** Correct answers increment st.session\_state['score']. Score persists for navigation until final screen or reset.

### **Epic 4 — Final Results**

**User story:** As a user, I want to see my final score and review answers.  
 **Acceptance criteria:** App shows total correct / total questions and optionally a table of Q / selected / correct / correct? boolean.

## **4. System architecture / workflows**

**Architecture (MVP):**

* Frontend: Streamlit UI (radio buttons, buttons, status text).
* State: st.session\_state stores current\_q\_index, score, answers list, and quiz\_finished flag.
* Questions: hardcoded Python list/dict in app code.

**Workflow:**

1. On app start, initialize st.session\_state (index=0, score=0, answers=[]).
2. Render question at current\_q\_index with radio options.
3. User selects option → clicks Next/Submit.
4. App evaluates answer, updates score and answers, increments current\_q\_index.
5. If current\_q\_index >= len(questions), set quiz\_finished = True and render final score + review.

## **5. Technical specifications (data model, functions, sample code)**

### **Data model (hardcoded questions)**

questions = [

{

"id": 1,

"question": "What is the capital of France?",

"options": ["Paris", "Berlin", "Rome", "Madrid"],

"answer": "Paris"

},

{

"id": 2,

"question": "2 + 2 = ?",

"options": ["3", "4", "5"],

"answer": "4"

},

# ...

]

### **Session state schema**

st.session\_state = {

"current\_q": 0,

"score": 0,

"answers": [ # list of dicts {q\_id, selected, correct\_answer, is\_correct}

# ...

],

"quiz\_finished": False

}

### **Core functions (conceptual)**

* init\_state() — ensure st.session\_state keys exist.
* render\_question(index) — show question text and radio options, return selected value.
* submit\_answer(selected) — compare with questions[index]['answer'], update score & answers, advance index or finish.
* render\_results() — show final score and optional review table.

### **Streamlit MVP Implementation (drop-in)**

import streamlit as st

# Hardcoded questions

QUESTIONS = [

{"id": 1, "question": "Capital of France?", "options": ["Paris", "Berlin", "Rome"], "answer": "Paris"},

{"id": 2, "question": "2 + 2 = ?", "options": ["3", "4", "5"], "answer": "4"},

{"id": 3, "question": "Largest planet?", "options": ["Earth", "Jupiter", "Mars"], "answer": "Jupiter"},

]

def init\_state():

if "current\_q" not in st.session\_state:

st.session\_state.current\_q = 0

if "score" not in st.session\_state:

st.session\_state.score = 0

if "answers" not in st.session\_state:

st.session\_state.answers = []

if "finished" not in st.session\_state:

st.session\_state.finished = False

def submit(selected\_option):

idx = st.session\_state.current\_q

q = QUESTIONS[idx]

is\_correct = (selected\_option == q["answer"])

if is\_correct:

st.session\_state.score += 1

st.session\_state.answers.append({

"id": q["id"],

"question": q["question"],

"selected": selected\_option,

"correct": q["answer"],

"is\_correct": is\_correct

})

st.session\_state.current\_q += 1

if st.session\_state.current\_q >= len(QUESTIONS):

st.session\_state.finished = True

def reset\_quiz():

for k in ["current\_q", "score", "answers", "finished"]:

if k in st.session\_state:

del st.session\_state[k]

init\_state()

# App

st.title("❓ Quiz Game")

init\_state()

if st.session\_state.finished:

st.subheader("🎉 Quiz Finished!")

st.write(f"Your score: \*\*{st.session\_state.score} / {len(QUESTIONS)}\*\*")

st.write("### Review")

import pandas as pd

df = pd.DataFrame(st.session\_state.answers)

df = df[["question", "selected", "correct", "is\_correct"]]

st.table(df)

st.button("Restart", on\_click=reset\_quiz)

else:

q\_idx = st.session\_state.current\_q

q = QUESTIONS[q\_idx]

st.write(f"\*\*Question {q\_idx + 1} of {len(QUESTIONS)}\*\*")

st.write(q["question"])

choice = st.radio("Choose an answer:", q["options"], key=f"choice\_{q\_idx}")

col1, col2 = st.columns([1,1])

with col1:

if st.button("Submit"):

if choice is None or choice == "":

st.warning("Please select an option before submitting.")

else:

submit(choice)

st.experimental\_rerun()

with col2:

if st.button("Reset Quiz"):

reset\_quiz()

st.experimental\_rerun()

**Notes on the implementation**

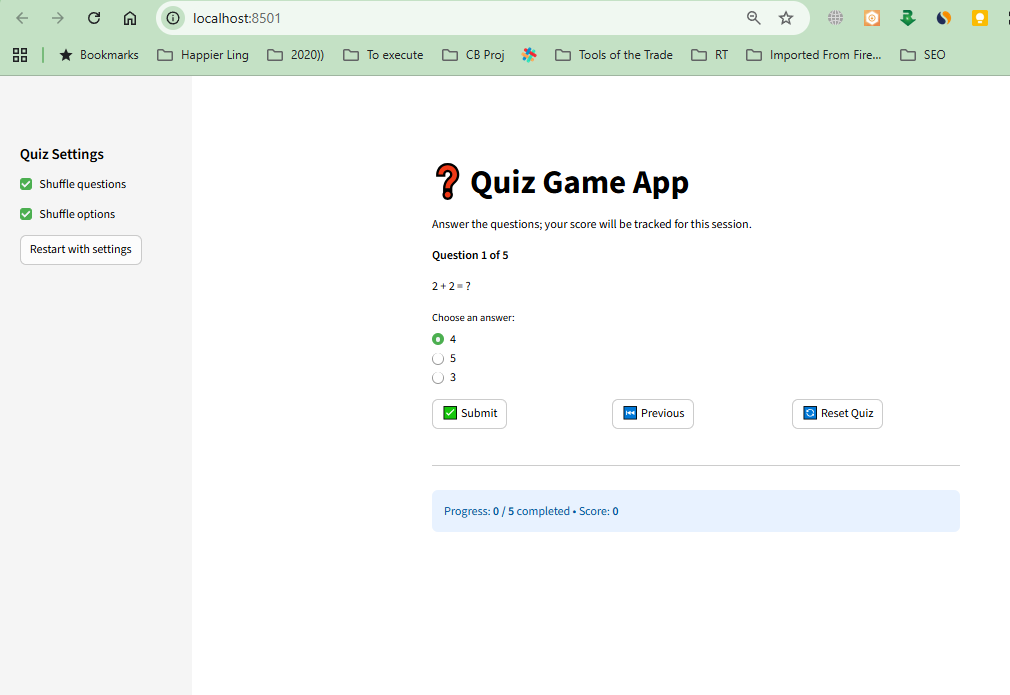
* Uses st.session\_state to persist current\_q, score, answers, finished.
* After submit, st.experimental\_rerun() ensures the app re-renders with updated state.
* Final results show a table with question, selected answer, correct answer, and correctness flag.

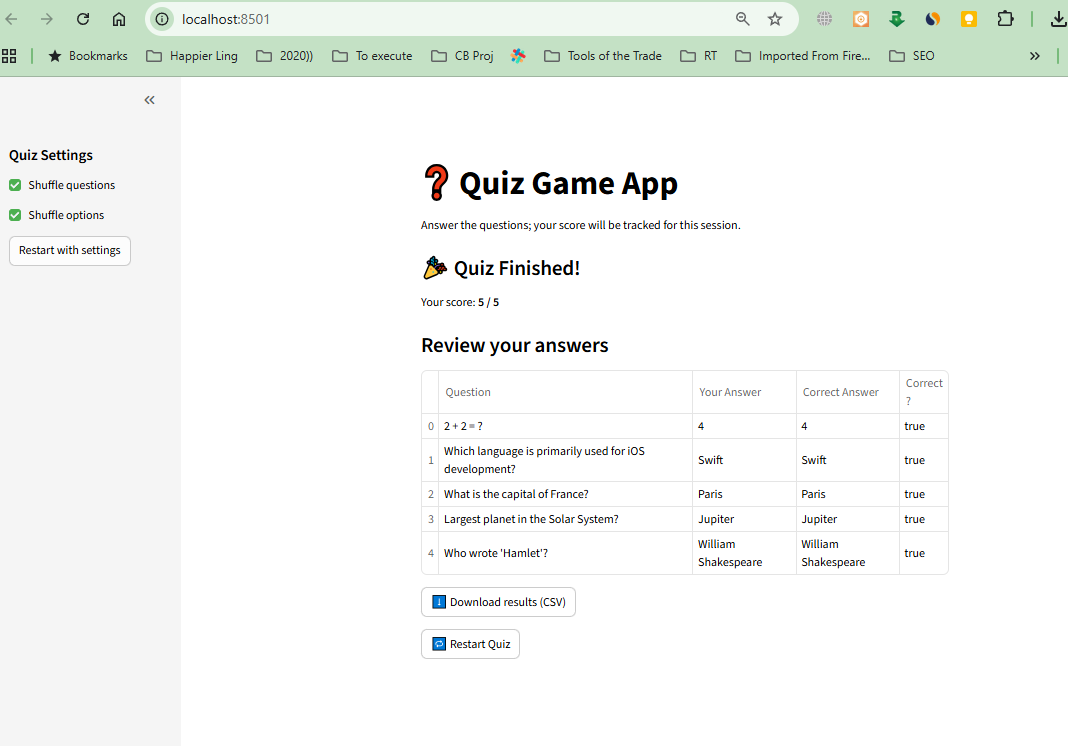
## **6. Non-functional requirements**

* **Performance:** Rendering and state updates should be instant for small question sets (<200ms).
* **Usability:** Clear question numbering, visible progress (e.g., 2/10).
* **Accessibility:** Radio buttons accessible via keyboard; readable fonts and contrast.
* **Scalability:** Hardcoded list OK for small quizzes; migrate to JSON or DB for large sets.
* **Security / Privacy:** No sensitive user data; session-only storage by default.

## **7. Open questions / assumptions**

* Should questions be randomized (shuffle order & options)? — easy to add.
* Should immediate feedback be shown after each question (correct/incorrect) or only final summary?
* Will questions include images or media? If yes, UI must support embedding assets.
* Should time limits per question be enforced? (future feature)
* Do we need persistent user accounts or leaderboards? (out of scope for MVP)





**\*\*\*\*\* Day 14 Tic-Tac-Toe ❌⭕**

3x3 grid using buttons.

Two-player mode (or vs computer with random moves).

Highlight winning line.

Option to reset the board  
  
the **end-to-end technical documentation + PRD** for the **Tic-Tac-Toe Game**.

# **📄 Technical Documentation: Tic-Tac-Toe Game ❌⭕**

## **1. Problem Statement & Goals**

**Problem:** Tic-Tac-Toe is a classic strategy game often used for learning and entertainment. A simple digital version should allow quick play between two players (or vs computer) with clear win/tie outcomes.

**Goal:** Build a **Tic-Tac-Toe App** that:

* Displays a **3x3 grid** of clickable buttons.
* Supports **two-player mode** (player X and O).
* Optionally allows **vs computer** (random moves).
* Highlights the **winning line** when someone wins.
* Provides a **Reset** option.

**Success Criteria:**

* Players alternate turns correctly.
* Game detects win, tie, or ongoing state.
* Winning line visually highlighted.
* Reset clears the board and score.

## **2. Personas & User Journeys**

**Persona 1: Student**

* Wants a quick casual game.

**Persona 2: Casual Gamer**

* Wants simple entertainment with friends.

**User Journey:**

1. User starts a new game.
2. Player X clicks a square → symbol placed.
3. Player O (or computer) makes move.
4. Game checks for winner/tie after each turn.
5. If win → highlight winning row/column/diagonal.
6. User can click **Reset** to start again.

## **3. Epics & User Stories (with Acceptance Criteria)**

### **Epic 1: Board Setup**

**User Story:** As a user, I want to see a 3x3 grid so I can play Tic-Tac-Toe.

**Acceptance Criteria:**

* 9 clickable buttons, initially blank.
* Grid layout displayed clearly.

### **Epic 2: Player Turns**

**User Story:** As a user, I want alternating turns between X and O so the game is fair.

**Acceptance Criteria:**

* Player X always starts.
* After each valid move, turn switches.
* Cannot click an already occupied square.

### **Epic 3: Win & Tie Detection**

**User Story:** As a user, I want the system to detect when the game is won or tied.

**Acceptance Criteria:**

* Check after every move.
* Winning line (row, column, diagonal) highlighted.
* Tie declared if board is full and no winner.

### **Epic 4: Computer Mode (Optional)**

**User Story:** As a user, I want to play against the computer so I can practice.

**Acceptance Criteria:**

* Computer picks random empty square.
* Computer plays as O.

### **Epic 5: Reset Game**

**User Story:** As a user, I want to reset the board so I can start a new match.

**Acceptance Criteria:**

* Reset clears board, current turn resets to X.
* Scores can be cleared (optional).

## **4. System Architecture / Workflows**

**Architecture:**

* **Frontend:** Grid UI with buttons, status display, reset button.
* **Logic:** Game state management (turn, win check, random AI).
* **Storage:** In-memory (session state).

**Workflow:**

1. Game initialized → empty 3x3 grid.
2. User clicks a square → symbol placed (X or O).
3. Game checks win/tie → update status if needed.
4. If vs computer → system makes move after user.
5. Winning state highlights line → disables further moves.
6. Reset clears everything → new game starts.

## **5. Technical Specifications**

### **Data Model**

{

"board": ["", "", "", "", "", "", "", "", ""],

"current\_player": "X",

"winner": null,

"winning\_line": [],

"game\_over": false

}

### **Win Conditions**

Index combinations for win check:

WIN\_COMBOS = [

[0,1,2], [3,4,5], [6,7,8], # rows

[0,3,6], [1,4,7], [2,5,8], # columns

[0,4,8], [2,4,6] # diagonals

]

### **Computer Move (Random AI)**

import random

def computer\_move(board):

empty = [i for i, v in enumerate(board) if v == ""]

return random.choice(empty) if empty else None

### **Example (Streamlit Implementation)**

import streamlit as st

import random

# Initialize state

if "board" not in st.session\_state:

st.session\_state.board = [""] \* 9

st.session\_state.current = "X"

st.session\_state.winner = None

st.session\_state.game\_over = False

WIN\_COMBOS = [

[0,1,2],[3,4,5],[6,7,8],

[0,3,6],[1,4,7],[2,5,8],

[0,4,8],[2,4,6]

]

def check\_winner():

for combo in WIN\_COMBOS:

a,b,c = combo

if st.session\_state.board[a] == st.session\_state.board[b] == st.session\_state.board[c] != "":

st.session\_state.winner = st.session\_state.board[a]

st.session\_state.game\_over = True

return combo

if "" not in st.session\_state.board:

st.session\_state.winner = "Tie"

st.session\_state.game\_over = True

return None

def reset\_game():

st.session\_state.board = [""] \* 9

st.session\_state.current = "X"

st.session\_state.winner = None

st.session\_state.game\_over = False

st.title("❌⭕ Tic-Tac-Toe")

cols = st.columns(3)

for i in range(9):

if st.session\_state.board[i] == "":

if cols[i % 3].button(" ", key=f"btn{i}"):

if not st.session\_state.game\_over:

st.session\_state.board[i] = st.session\_state.current

combo = check\_winner()

if not st.session\_state.game\_over:

st.session\_state.current = "O" if st.session\_state.current == "X" else "X"

else:

cols[i % 3].button(st.session\_state.board[i], disabled=True, key=f"btn{i}")

if st.session\_state.winner:

if st.session\_state.winner == "Tie":

st.success("It's a Tie! 🤝")

else:

st.success(f"🎉 Winner: {st.session\_state.winner}")

if st.button("Reset Game"):

reset\_game()

## **6. Non-Functional Requirements**

* **Performance:** Button clicks → instant updates (<50ms).
* **Usability:** Mobile-friendly grid, clear status messages.
* **Fairness:** Computer picks randomly (upgradeable to smart AI later).
* **Accessibility:** Buttons labeled with X/O for screen readers.

## **7. Open Questions / Assumptions**

* Should we support **score tracking across games**?
* Should computer AI be **random only** or use a smarter algorithm (minimax)?
* Should game auto-switch between **two-player vs vs-computer** modes?

# **📑 Product Requirements Document (PRD)**

## **Product: Tic-Tac-Toe Game ❌⭕**

### **1. Overview**

A digital Tic-Tac-Toe game with 3x3 grid, two-player mode, optional vs computer mode, win detection, winning line highlight, and reset option.

### **2. Goals & Non-Goals**

**Goals:**

* Provide simple two-player and vs computer gameplay.
* Detect win/tie outcomes.
* Highlight winning line.
* Reset game easily.

**Non-Goals (MVP):**

* Advanced AI (minimax).
* Multiplayer over network.
* Persistent scores.

### **3. User Stories (Prioritized)**

**MVP Stories:**

1. Display 3x3 grid of buttons.
2. Support two-player mode with alternating turns.
3. Detect win/tie → highlight winning line.
4. Reset game.

**Future Stories:**

* Play vs computer (random moves).
* Scoreboard tracking.
* Smarter AI difficulty levels.

### **4. Functional Requirements**

* 3x3 board of buttons.
* Alternating turns (X starts).
* Win/tie detection after each move.
* Highlight winning line.
* Reset button clears board.

### **5. User Flows**

**Flow 1: Play Game**

1. Player X clicks → X placed.
2. Player O clicks → O placed.
3. System checks win/tie.
4. Status displayed → continue or end.

**Flow 2: Reset**

1. User clicks Reset → board cleared.
2. New game starts with X’s turn.

### **6. Success Metrics**

* ✅ Game runs smoothly with no delays.
* ✅ Win/tie detection works 100% correctly.
* ✅ Reset always restores board.

### **7. Release Plan**

**MVP (Phase 1):**

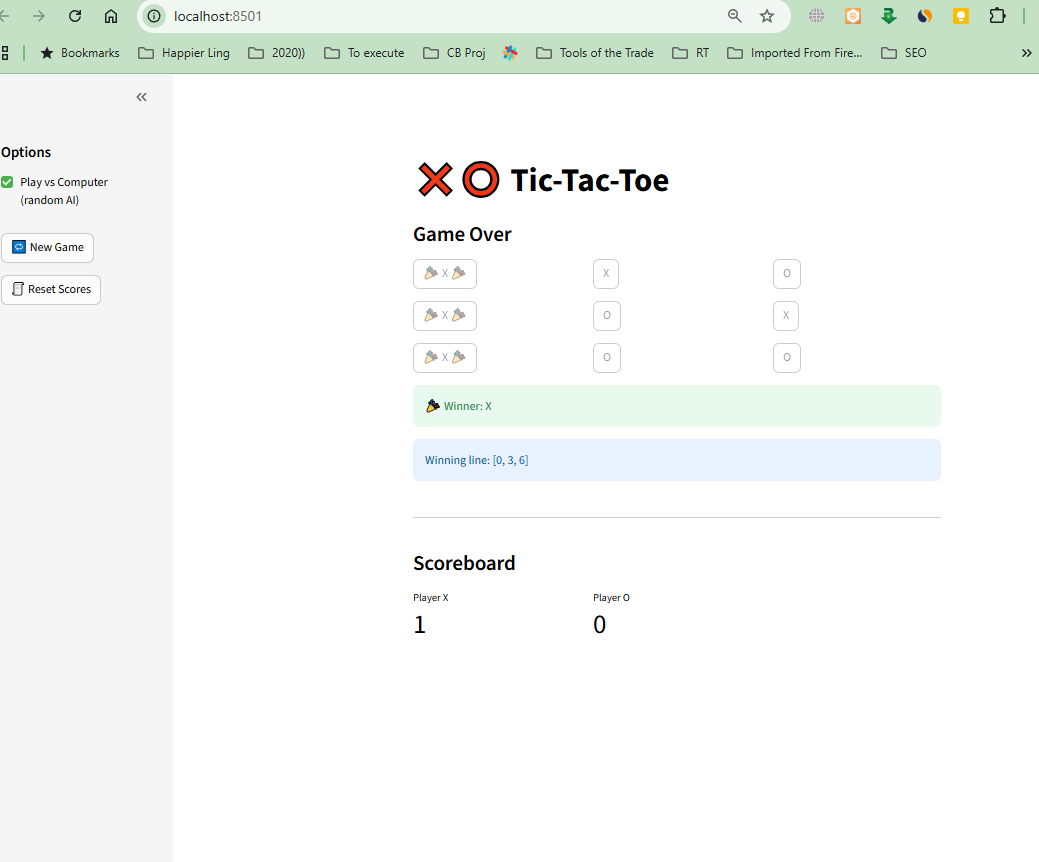
* Two-player mode.
* Win/tie detection.
* Reset option.

**Phase 2 (Nice-to-Have):**

* Play vs computer (random AI).
* Winning line highlight.
* Scoreboard across games.

**Phase 3 (Advanced):**

* Smarter AI (minimax).
* Online multiplayer.
* Custom board sizes (4x4, 5x5).



**\*\*\*\*\*\* Day 15 Problem Statement: Snake Game in Streamlit**

Build a classic Snake Game where the player controls a snake to eat food and grow in size, while avoiding collisions with the walls or itself.

Requirements:

1. Create a grid-based game board.

2. The snake moves automatically in the chosen direction (Up/Down/Left/Right).

3. Player can change direction using buttons or keyboard input.

4. Food appears randomly on the grid.

5. Each time the snake eats food, its length increases and the score updates.

6. The game ends if the snake collides with the wall or itself.

7. Display the current score and a "Restart" option.

Goal:

Practice real-time updates, session state management, and dynamic rendering in Streamlit.

the **Snake Game** is a classic real-time game, excellent for practicing state management, animations, and interactive UI. Let’s prepare the **technical documentation + PRD** for building it in **Streamlit** (grid-based with session state).

# **📄 Technical Documentation: Snake Game 🐍**

## **1. Problem Statement & Goals**

**Problem:** The Snake Game is a great way to practice **real-time rendering, state persistence, and user input handling**. A grid-based implementation with Streamlit can simulate animations using repeated re-renders.

**Goal:** Build a classic **Snake Game** where:

* Player controls a snake that moves automatically.
* Snake eats food to grow and score points.
* Colliding with wall or itself ends the game.
* Player can restart anytime.

**Success Criteria:**

* Snake moves smoothly at timed intervals.
* Player input changes direction instantly.
* Food placement is random and valid (not inside the snake).
* Score increments correctly with each food eaten.
* Game ends correctly on collision.

## **2. Personas & User Journeys**

**Persona 1: Gamer / Student**

* Wants a fun, interactive game.

**Persona 2: Developer / Learner**

* Wants to understand how to manage state and updates in Streamlit.

**User Journey:**

1. User starts game → Snake appears on grid with food.
2. Snake moves automatically.
3. User presses direction (button or arrow key).
4. Snake grows when eating food → score updates.
5. If snake hits wall/self → game over message.
6. User clicks **Restart** → grid resets with new snake.

## **3. Epics & User Stories (with Acceptance Criteria)**

### **Epic 1: Game Board**

**User Story:** As a user, I want to see a grid board so that I can play the game.

**Acceptance Criteria:**

* Grid displayed as 2D matrix (e.g., 20x20).
* Snake and food rendered in grid cells.

### **Epic 2: Snake Movement**

**User Story:** As a user, I want the snake to move automatically so the game feels real.

**Acceptance Criteria:**

* Snake moves one cell per tick.
* Default direction = right.
* Timer-based updates (e.g., every 200ms).

### **Epic 3: Player Input**

**User Story:** As a user, I want to control the snake using buttons/keyboard so I can play interactively.

**Acceptance Criteria:**

* Buttons for Up/Down/Left/Right.
* Cannot reverse direction directly (e.g., right → left).
* Input updates direction instantly.

### **Epic 4: Food Generation**

**User Story:** As a user, I want food to appear randomly so the game is engaging.

**Acceptance Criteria:**

* Food spawns on empty cell.
* When snake eats food:  
  + Snake length increases by 1.
  + Score increments by +1.
  + New food spawns.

### **Epic 5: Game Over Condition**

**User Story:** As a user, I want the game to end if I crash so that the rules are fair.

**Acceptance Criteria:**

* Collision with wall ends game.
* Collision with self ends game.
* “Game Over” message + Restart button shown.

### **Epic 6: Score & Restart**

**User Story:** As a user, I want to see my score and restart the game.

**Acceptance Criteria:**

* Score displayed at top.
* Restart button resets game state.

## **4. System Architecture / Workflows**

**Architecture:**

* **Frontend:** Grid display using Streamlit (st.markdown / st.table / st.pyplot).
* **Logic:** Game loop (timer-based refresh).
* **State Management:** st.session\_state stores snake body, direction, food, score, and game\_over flag.

**Workflow:**

1. Initialize game (snake length = 1, random food, score = 0).
2. Every tick → move snake → check collision → render.
3. On input → update direction.
4. If snake eats food → grow, increment score.
5. If collision → end game.

## **5. Technical Specifications**

### **Data Model (Session State)**

{

"board\_size": 20,

"snake": [(10,10), (10,9), (10,8)], # head-first list

"direction": "RIGHT", # "UP", "DOWN", "LEFT", "RIGHT"

"food": (5,15),

"score": 0,

"game\_over": False

}

### **Functions (Core)**

def init\_game():

st.session\_state.snake = [(10,10)]

st.session\_state.direction = "RIGHT"

st.session\_state.food = random\_food()

st.session\_state.score = 0

st.session\_state.game\_over = False

def move\_snake():

head = st.session\_state.snake[0]

dx, dy = direction\_delta(st.session\_state.direction)

new\_head = (head[0] + dx, head[1] + dy)

# Check collisions

if (out\_of\_bounds(new\_head) or new\_head in st.session\_state.snake):

st.session\_state.game\_over = True

return

st.session\_state.snake.insert(0, new\_head)

if new\_head == st.session\_state.food:

st.session\_state.score += 1

st.session\_state.food = random\_food()

else:

st.session\_state.snake.pop()

def random\_food():

import random

while True:

x, y = random.randint(0, st.session\_state.board\_size-1), random.randint(0, st.session\_state.board\_size-1)

if (x,y) not in st.session\_state.snake:

return (x,y)

### **Rendering**

* Grid can be displayed using:  
  + st.markdown with emojis (🟩 for snake, 🍎 for food, ⬛ for empty).
  + Or matplotlib heatmap with colored cells.

## **6. Non-Functional Requirements**

* **Performance:** Game tick refresh ≤ 200ms.
* **Usability:** Responsive controls (buttons/keys).
* **Persistence:** State stored in session, resets on restart.
* **Fairness:** Food never spawns inside snake.

## **7. Open Questions / Assumptions**

* Should board size be **fixed (20x20)** or configurable?
* Should speed increase as snake grows?
* Should we add **high score tracking** across sessions?
* Should we allow **keyboard controls** (↑ ↓ ← →) or buttons only?

# **📑 Product Requirements Document (PRD)**

## **Product: Snake Game 🐍**

### **1. Overview**

A grid-based Snake Game built in Streamlit with automatic movement, player input, random food, scoring, collision detection, and restart option.

### **2. Goals & Non-Goals**

**Goals:**

* Provide a classic Snake Game experience.
* Implement in Streamlit using session state.
* Practice real-time updates and rendering.

**Non-Goals (MVP):**

* Multiplayer or online play.
* Custom skins/themes.
* Persistent high scores (can be Phase 2).

### **3. User Stories (Prioritized)**

**MVP Stories:**

1. Display grid game board.
2. Snake moves automatically.
3. Player changes direction via input.
4. Food spawns randomly.
5. Eating food → grow snake + increase score.
6. Collision detection → game over.
7. Show score + restart option.

**Future Stories:**

* Keyboard arrow controls.
* Increasing speed with score.
* High score persistence.
* Multiple food types (bonus items).

### **4. Functional Requirements**

* Grid board visualization.
* Snake moves every tick.
* Direction input (buttons/keys).
* Random food spawn.
* Collision detection logic.
* Score display.
* Restart button.

### **5. User Flows**

**Flow 1: Play Game**

1. Game starts with snake + food.
2. Snake moves automatically.
3. Player changes direction.
4. If snake eats food → grows, score +1.
5. If snake hits wall/self → game ends.

**Flow 2: Restart**

1. User clicks Restart.
2. Board resets, score cleared, new game starts.

### **6. Success Metrics**

* ✅ Game runs smoothly (200ms refresh).
* ✅ Snake growth and scoring work correctly.
* ✅ Collision detection always accurate.
* ✅ Restart resets board cleanly.