ChatGPT

NIST CSF Trust Scoring Platform Design Specification

Overview and Context

The **NIST Cybersecurity Framework (CSF)** is a well-known set of guidelines and best practices for improving cybersecurity and risk management at an organizational level [1]. It defines a **Framework Core** consisting of five high-level **Functions** – **Identify, Protect, Detect, Respond,** and **Recover** – which cover the lifecycle of cybersecurity risk management [2]. (The latest CSF 2.0 adds a sixth **Govern** function, but our current implementation focuses on the original five.) Each Function is further broken down into Categories and Subcategories (108 subcategories in CSF 1.1) that represent specific security outcomes [3].

Our **Trust Scoring System** leverages this framework to assess an organization's security posture. We have developed a series of **React.js form components** aligned with the NIST CSF Functions. Users input information via these forms (essentially a structured questionnaire for each function), and the system computes a "trust score" indicating cybersecurity maturity or risk level. This score can be viewed as analogous to an implementation tier or maturity level; for example, responses might be evaluated in a binary or tiered manner to calculate scores [4]. Ultimately, the goal is to provide a quantitative measure of how well the organization meets the CSF criteria, helping identify gaps and prioritize improvements.

System Architecture

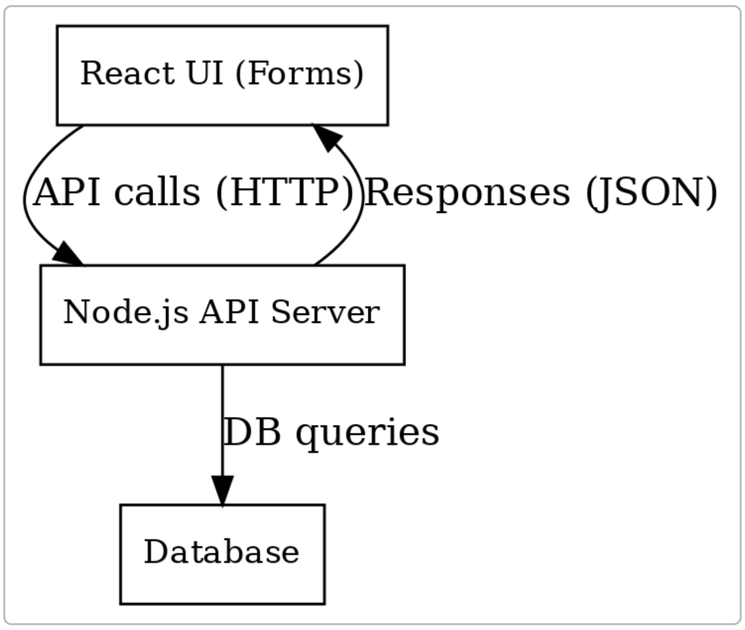

Figure 1: High-Level System Architecture. The platform follows a typical web application architecture. The front-end is a **React** single-page application (SPA) running in the browser, and the back-end will be a **Node.js** (Express) API server with a database. The React client collects user inputs through the CSF forms and, when the user submits their assessment, it communicates with the Node.js back-end via

HTTP API calls. The Node server will process and store the data in a persistent **database**, and can also compute the trust score on the server side if needed. The server then returns results (e.g. the computed score or saved profile data) as JSON responses back to the client for display. This separation of concerns ensures a robust, scalable platform: the **front-end** handles user interaction and input validation, while the **back-end** handles data persistence, business logic, and integration with other systems (if any).

This design follows modern web best practices. The React app (client) sends requests to the Node.js API for any heavy processing or for saving/retrieving data. As NIST recommends, the front-end and back-end communicate over a secure HTTP(S) interface (e.g. RESTful API). Node.js can handle concurrent requests efficiently using its event-driven, non-blocking I/O architecture [5]. In our architecture, when a user completes the forms and submits, the React app will send an aggregate payload (or individual form data) to the Node API (e.g. an endpoint like `/api/trust-score`). The Node API will queue and handle these requests, perform any server-side calculations (such as aggregating scores or looking up reference data), and then store the results. We anticipate using a database (SQL or NoSQL) to persist the assessment results. For example, there may be a **table** for assessment submissions, capturing the answers to each CSF Category/Subcategory and the resulting trust score. Designing this schema is part of the back-end development – each form's data corresponds to columns or documents in the database, so that we can easily query an organization's posture in each area.

Implementation Note: Until the back-end is fully built, the React front-end can mock this process by computing the score on its own and perhaps storing data in memory or local storage. However, the architecture is prepared to connect to a real database. In the MERN stack (MongoDB, Express, React, Node) style, for instance, the Node server would interface with MongoDB to save the form inputs and retrieve prior results [6]. The design is flexible to use any database (SQL relational tables or a document DB). The key point is that all data flows through defined APIs, maintaining a clean separation between UI and data layer.

UI Structure and Component Relationships

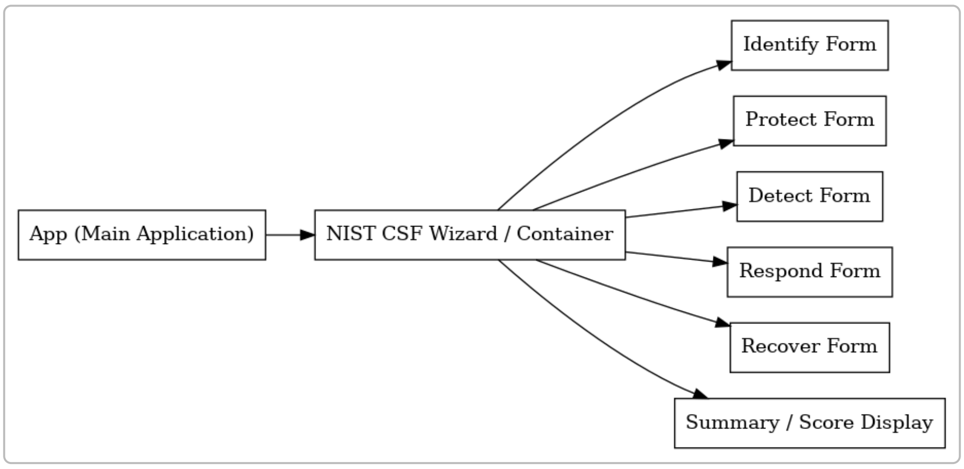

Figure 2: React Component Hierarchy for the CSF Forms. The front-end is built with a component-based architecture in React. At a high level, our main application (`App`) contains a **CSF Assessment Wizard** component, which in turn manages five form sub-components corresponding to the NIST CSF Functions: an **Identify Form**, **Protect Form**, **Detect Form**, **Respond Form**, and **Recover Form**. Each of these form components is responsible for collecting user input about that particular function area. For example, the **Identify Form** gathers information on asset management, business environment, risk assessment processes, etc., aligning with what the Identify function covers [7]. The **Protect Form** covers areas like access control, training, data security measures (reflecting the Protect function's scope)

[8], and so on for Detect, Respond, and Recover (each form's content is based on the relevant CSF categories to ensure comprehensive coverage of that function's domain).

This parent-child component structure promotes modularity and clarity. React encourages breaking the UI into reusable pieces [9], so each form is a self-contained unit that can be developed and tested independently. The **Wizard/Container component** coordinates these forms – it handles which form is currently visible and aggregates the data from all forms. In our implementation, the Wizard maintains the overall state of the multi-step form (the answers for all steps), passing down relevant data or handlers to each child form as needed. This avoids each form being isolated; instead, they are connected through the wizard's state. When a user navigates to the next form, the wizard component can store the previous form's data and load the next form component.

We also implemented some **shared UI elements** to ensure consistency across forms. For instance, each form uses common sub-components like **input controls** (text fields, checkboxes, radio buttons, dropdowns, etc.) and possibly a **QuestionItem** component for rendering each question/prompt in a consistent style. These could live in a shared `/components` directory (as per our project structure) and be imported into each form. This approach follows React's UI component principles, making the UI easier to maintain and keeping design consistent (e.g., all forms might use the same custom styled button for "Next" or "Submit").

Layout: The forms are presented in a logical sequence (more on navigation in the next section). The UI has a straightforward layout – typically a header or title indicating the current section (e.g., "Identify – Assessment"), the form fields/questions in the main content area, and navigation controls (Back/Next buttons) at the bottom. We ensure the layout remains consistent from Identify through Recover forms, so users have a smooth experience. The design is also responsive so that the forms can be used on various screen sizes. Each form page is kept to a reasonable length (by focusing on one Function at a time) which improves usability compared to one gigantic form. Short paragraphs or grouped questions with instructions can guide the user through each section. The overall look-and-feel is that of a step-by-step survey or wizard, which is a familiar pattern for users and keeps them from being overwhelmed.

Under the hood, **state management** is crucial for these components to work together. The wizard holds the composite state of all form inputs. We utilize React's state and context APIs to manage this. For example, when the user enters data in the Protect Form and clicks "Next", the wizard component's state is updated with those answers before moving to the Detect Form. We can use React Context or a state management library (like Redux or Zustand) to pass data between steps without prop-drilling through many layers [10]. This centralized state approach ensures that the data is consistent and available when we need to calculate the final score. It also means the user can navigate backward (e.g., from Detect back to Protect) and see their previously entered answers. The design emphasizes a single source of truth for the form data (the wizard's state or a global store) so that all components reflect the latest inputs [11]. Validation logic can also live at this level – for instance, the wizard can prevent moving to the next step until required fields in the current form are filled, ensuring data completeness.

Form Flow and Relationships Between Forms

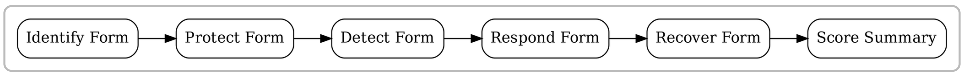

Figure 3: Multi-Step Form Flow Through CSF Functions. The five forms have a natural **sequence** based on the CSF function order: *Identify* → *Protect* → *Detect* → *Respond* → *Recover*. The user progresses through these steps in order, as illustrated in Figure 3. This sequential **wizard flow** enforces a structured approach: you first **Identify** assets and risks, then define how you **Protect** them, ensure you

can **Detect** incidents, plan how to **Respond**, and finally **Recover** from incidents. This mirrors the logical flow of the NIST CSF and helps the user think through their security posture in a comprehensive way.

The **relationship between the forms** is primarily navigational and hierarchical (not so much data dependency). Each step must be completed before moving on to the next, and all steps collectively contribute to the final trust score. We designed the wizard so that the **“Next” button** on each form saves that form’s data and then renders the next form component. There is also a **“Back” button** (except on the first step) allowing users to revisit and edit previous answers. Changes propagate through the centralized state, so the trust score will always consider the latest answers from all forms.

Notably, some answers in earlier forms could influence later ones in terms of context. For example, if in the Identify Form the user indicates they have no critical third-party vendors, perhaps certain questions in Detect or Respond related to third-party risk might be skipped or marked as not applicable. Our design allows for such conditional logic if needed. The forms can be aware of the global state, so they can conditionally render or adjust questions based on prior inputs. This is an area for future enhancement – currently, the forms are mostly independent, but the structure will accommodate inter-form logic if the scoring model requires it.

At the end of the sequence is a **Summary/Score Display** (as shown in Figure 2 as the “Summary / Score” component). After the user completes the Recover Form, the wizard can calculate the overall **Trust Score** using all the collected data. The Summary page then presents this score, possibly along with a breakdown by function (so the user can see, for example, a sub-score for Identify, Protect, etc., and identify weak spots). The scoring algorithm might assign weights to certain questions or use a simple percentage of “yes” answers – these specifics are configurable. The key is that the Summary component has access to *all* the answers (through the wizard state or via an API call to the backend that computes the score). For transparency, we may also list which areas are lacking (e.g., “You answered NO to having an incident response plan – this lowers the Respond score”).

Data flow: When the user hits “Submit” on the final step, the collected data is packaged (typically as a JSON object) and sent to the back-end API for storage. The front-end may also send the calculated score, or the back-end can recompute it to double-check. The back-end, after saving, might respond with a confirmation and perhaps a copy of the results. In the UI, the Summary page can show a message like “Assessment submitted successfully” along with the score. If the platform supports saving partial progress, the wizard could also allow users to save and resume later (this would involve sending what’s done so far to the database).

In summary, the forms work together as a cohesive workflow. Each form is a chapter of a story: defining assets (Identify), protecting them, monitoring, responding, and recovering – covering the full breadth of NIST CSF topics. The relationships are managed by the wizard container which orchestrates the journey and ensures that when the journey is complete, we have a full set of data ready to be translated into a trust score.

Design Considerations and Future Integration

Validation & User Guidance: Each form includes validation to ensure that required information is provided. We follow standard UX practices for forms – for example, highlighting missing required fields or providing help text where a question might be unclear. Since the CSF can be complex, the UI may include tooltips or descriptions for certain questions to guide non-expert users. The design document notes that maintaining usability is crucial: forms should not overwhelm the user with jargon. Where possible, we map technical controls to layman descriptions (with an option to view the official CSF reference for power users). This keeps the trust scoring exercise accessible to a broad audience.

Component Reusability: The structured approach allows reusing components for efficiency. For instance, if many questions are yes/no checkboxes, a single `YesNoQuestion` component can be used across all forms. This reduces code duplication and ensures uniform behavior (one change fixes the component everywhere). The style (CSS) is likewise reused – a global stylesheet or styled-components ensure each form adheres to the same design language (colors, typography, spacing). This document emphasizes that a consistent UI not only looks professional but also reinforces the structured nature of the assessment.

Scalability: As we plan to integrate a database and possibly support multiple users or organizations, the architecture is built to scale. The front-end can easily be extended with routing if we later have a dashboard or multiple pages (for example, a home page listing multiple assessment entries, etc.). The back-end API is stateless, making it scalable behind a load balancer if needed. Each form's data is clearly defined, which will help in creating database schemas and API payload structures. We also separate concerns cleanly: the front-end doesn't hard-code any database details; it just talks to APIs. This abstraction means we could even swap out the back-end technology without affecting the front-end, as long as the API contract remains the same.

Security: Given this is a trust scoring/cybersecurity tool, we are mindful of security in the design. All communication between front-end and back-end will be over HTTPS. We will implement proper authentication (if the tool requires login) to protect sensitive assessment data. For now, if it's a standalone assessment tool, we at least ensure that no personal data is exposed and that the architecture can accommodate adding an auth layer (e.g., using JWT tokens or session cookies with the Node.js API) in the future.

Database Design (Future): When building the back-end, we will create tables or collections to store assessment results. One likely design is a table with columns corresponding to each CSF subcategory's response (yes/no or maturity level), plus metadata like the user/org, date, and computed scores. Alternatively, a more normalized approach could have an **Assessments** table and a related **Responses** table listing question ID and answer, which is flexible for changes in questions. The document here notes that this design will be done in alignment with the front-end's data structure – since we have the forms divided by function, we might group responses by function in the data model as well. The **trust score** itself can be stored for historical comparisons. This will allow generating reports like "Your score improved from 60% to 75% in the Protect function over time."

In conclusion, the **NIST CSF Trust Scoring Platform** is designed with a clear modular structure and an emphasis on aligning to the NIST framework's logic. The use of a React multi-step form interface provides an intuitive user experience, guiding the user through the five core areas of cybersecurity readiness. Each form component is part of a larger workflow, and together they capture a comprehensive security profile. The architecture diagrams (Figure 1, 2, 3) illustrate how data flows from the user interface through to the back-end storage. This design not only makes the current implementation robust and maintainable but also sets the stage for future enhancements such as database integration, user management, and analytics. By adhering to proven architectural practices and the guidance of NIST CSF, the platform will be a powerful tool for organizations to **assess and improve their cybersecurity posture** in a structured, measurable way.

Sources:

- NIST Cybersecurity Framework – Overview and Functions [1] [2]
- NIST CSF Categories and Subcategories (structure of framework) [3]
- React Component Architecture and State Management [12] [11]
- React & Node Integration (MERN stack best practices) [5] [6]
- Cyber Trust Index research (example of questionnaire-based scoring) [4]
- Vanta Security – NIST CSF function definitions (for context on each function) [7] [13]

---

[1] NIST Cybersecurity Framework: A Cheat Sheet for Professionals
https://www.techrepublic.com/article/nist-cybersecurity-framework-the-smart-persons-guide/

[2] [3] NIST CSF: The NIST CSF components | Infosec
https://www.infosecinstitute.com/resources/nist-csf/nist-csf-the-nist-csf-components/

[4] Cyber Trust Index: A Framework for Rating and Improving Cybersecurity Performance
https://www.mdpi.com/2076-3417/12/21/11174

[5] [6] Using NodeJS with React.js | Perfect Stack for Modern App Development
https://www.itarch.info/2023/07/using-nodejs-with-reactjs-perfect-stack.html

[7] [8] [13] What is NIST CSF and why is it important? | Vanta
https://www.vanta.com/resources/what-is-nist-csf-and-why-is-it-important

[9] [10] [11] [12] React JS Architecture: Components & Implementation Steps
https://www.upgrad.com/blog/react-js-architecture/