INTRODUCTION ABOUT THE PROJECT

In the vibrant and complex landscape of modern Indian nuptials, the need for efficient planning and reliable service discovery has become paramount. While traditional methods of wedding organization—such as word-of-mouth recommendations and physical directories—remain common, they often lack the transparency and breadth required by modern couples. The booming wedding industry has transformed how services are sourced, yet many generic listing sites fail to offer a focused, user-centric environment free from clutter and unverified information. This is where **SubhVivah** comes in.

SubhVivah bridges the gap between couples, families, and wedding service providers by offering a comprehensive, feature-rich marketplace. Designed with a focus on trust, visual appeal, and ease of navigation, the platform provides a streamlined approach to wedding management. Users can explore various vendor "Categories" such as Venues, Photographers, Makeup Artists, and Groom Wear, with features like city-based filtering, price-range sorting, and the ability to shortlist favorites. Users can view detailed portfolios, access verified contact information, and send direct inquiries through a robust interaction system.

The goal of SubhVivah is to create a one-stop solution that fosters a stress-free planning experience while assisting vendors in reaching their target audience efficiently. By leveraging modern web technologies and best practices, the platform not only provides a seamless user experience but also ensures that high-quality, relevant vendors are easily accessible. This democratized approach ensures that the best service providers—regardless of their scale—are discoverable by couples planning their dream day.

As trends shift towards digital-first planning and destination weddings, SubhVivah is built to meet these evolving needs. Its responsive design, dynamic search capabilities, and interactive features—such as real-time shortlisting and dynamic vendor filtering—make it a powerful tool for users seeking to expand their options beyond local boundaries.

From a technical perspective, the platform's development is rooted in modern full-stack web development principles, utilizing the **MERN Stack** (MongoDB, Express.js, React.js, and Node.js) to create a scalable, single-page application (SPA). The project follows a structured development approach, ensuring features like secure JWT authentication, vendor management, and user dashboards are implemented efficiently. The back-end architecture is built using a modular Controller-Service pattern with RESTful APIs, ensuring security, maintainability, and ease of future enhancements.

By combining ease of use, visual elegance, and a personalized planning experience, SubhVivah seeks to stand out as a premier digital hub for wedding coordination. Whether an individual is looking for a luxury venue, budget-friendly catering, or simply wishes to browse latest trends, SubhVivah offers the tools and resources needed to succeed in orchestrating a perfect wedding celebration.

OBJECTIVE AND SCOPE OF PROJECT

The SubhVivah Wedding Planner has the following specific objectives:

- **To foster a seamless planning ecosystem** where couples can discover, compare, and connect with top-tier wedding service providers across dedicated "Categories" (e.g., Venues, Photographers, Makeup Artists).
- **To democratize vendor discovery** by implementing transparent listing systems, ensuring that high-quality vendors—regardless of their size—are discoverable based on merit, ratings, and relevance to the user's needs.
- **To enhance user engagement** through interactive planning tools such as personalized wedding checklists, budget tracking, and a dynamic "Shortlist" feature for saving favorite vendors.
- **To provide a personalized user experience** by allowing couples to manage their wedding profile, track their inquiry status, and customize their dashboard with partner details and wedding dates.
- **To facilitate easy information retrieval** through advanced search functionality and multi-parameter filtering options, allowing users to find specific vendors by City, Category, and Price Range.
- **To ensure accessibility and ease of use** via a responsive, mobile-friendly interface built with Tailwind CSS that works seamlessly across desktops, tablets, and smartphones.

The scope for developing SubhVivah includes:

- **User & Vendor Management:** Implementing secure authentication (Login/Register) using JWT strategies and comprehensive profile management for both couples and vendors.

- **Vendor Marketplace System:** Developing a robust system for administering vendor portfolios, including image galleries, pricing details, and contact information.
- **Interactive Planning Features:** Integrating a user dashboard with a smart checklist system (todo/done status) and a "Favorites" manager to streamline the planning process.
- **Advanced Search & Filtering:** Creating algorithms to sort vendors by "Top Rated," "Price: Low to High," and specific location-based filtering to match couples with local providers.
- **Responsive User Interface:** Utilizing the **React.js** library and **Tailwind CSS** framework to ensure the platform adapts dynamically to different screen sizes and provides a modern aesthetic.
- **Data Security:** Implementing secure backend architecture using **Node.js** and **MongoDB** to protect user contact details and ensure safe interactions between couples and vendors.

INTRODUCTION OF HTML AND CSS

HTML (Hypertext Markup Language) and **CSS (Cascading Style Sheets)** are the foundational technologies of web development. In the context of **Chetan College Forum**, these technologies are utilized within the React.js framework (via JSX) to build a dynamic and visually engaging Single Page Application (SPA).

HTML (Hypertext Markup Language): HTML serves as the structural backbone of the application. In this project, HTML is implemented using **JSX (JavaScript XML)**, a syntax extension for React. This allows us to write HTML structures directly within JavaScript code, defining the layout of components such as the Header, Sidebar, Topic Cards, and Polls. It categorizes elements like input forms, buttons, text blocks, and navigation links, providing the semantic structure necessary for accessibility and SEO.

CSS (Cascading Style Sheets) & Bootstrap: CSS is responsible for the visual presentation of the platform. It controls the layout, colors, fonts, and spacing, ensuring a pleasing aesthetic.

- **Custom CSS:** The project utilizes custom stylesheets (`App.css`, `poll.css`, `Responsive.css`) to define specific visual identities, such as the card layouts for topics and the styling of voting buttons.
- **Bootstrap Framework:** To accelerate development and ensure responsiveness, the project integrates **Bootstrap** (via `react-bootstrap`). This framework provides pre-styled components and a grid system that automatically adjusts the forum's layout based on the user's device, ensuring a consistent experience whether on a laptop or a mobile phone.

Together, HTML (via JSX) and CSS (enhanced by Bootstrap) enable the rapid development of a user-friendly interface that separates content structure from visual design, adhering to modern web standards.

INTRODUCTION OF JAVASCRIPT (FULL STACK)

JavaScript is a versatile and powerful programming language that serves as the core technology for the entire **Chetan College Forum** stack. Unlike traditional web development where JavaScript was limited to the browser, this project utilizes **Full Stack JavaScript**, employing it on both the client-side (React.js) and the server-side (Node.js).

Key Features of JavaScript in this Project:

Unified Language (Full Stack Development): One of the most significant advantages of this project is the use of JavaScript across the entire technology stack (MERN). This allows for seamless data flow (JSON) between the frontend and backend, reducing context switching and simplifying the development process.

Client-Side Scripting (React.js): On the frontend, JavaScript is used to build interactive user interfaces. It handles:

- **DOM Manipulation:** Updating the view dynamically without reloading the page (e.g., when a user votes on a poll or posts a comment).
- **State Management:** Using tools like **Redux Toolkit** to manage application state (user authentication, fetched topics, UI loading states).
- **Event Handling:** Responding to user actions such as button clicks, form submissions, and keyboard shortcuts (e.g., Ctrl+Enter to submit).

Server-Side Scripting (Node.js): On the backend, JavaScript runs on the server via **Node.js**. It handles:

- **API Logic:** Processing requests from the frontend (e.g., fetching topics, registering users).

- **Database Interaction:** Communicating with MongoDB via **Mongoose** to store and retrieve data.
- **Authentication:** Verifying user identities and managing secure sessions (JWT).

Asynchronous Programming: The project heavily relies on JavaScript's asynchronous features (`async/await`, `Promises`). This ensures non-blocking operations, such as waiting for database queries or API responses without freezing the user interface, resulting in a highly performant application.

Object-Oriented & Functional Paradigms: JavaScript supports both object-oriented and functional programming styles. This project utilizes modern **ES6+ features** (Arrow Functions, Destructuring, Modules), allowing for clean, modular, and maintainable code structure across both the client and server environments.

INTRODUCTION OF NODE.JS

Node.js is an open-source, cross-platform JavaScript runtime environment that executes JavaScript code outside of a web browser. Initially created by Ryan Dahl in 2009, Node.js was developed to push JavaScript beyond the client-side, allowing developers to use it for server-side scripting. Unlike PHP, which is a blocking, synchronous language, Node.js is built on the V8 JavaScript engine—the same high-performance engine that powers Google Chrome—converting JavaScript directly into native machine code.

Node.js operates on an event-driven, non-blocking I/O model, which makes it lightweight and efficient. Instead of creating a new thread for every user request (as traditional web servers do), Node.js uses a single-threaded event loop to handle thousands of concurrent connections. This architecture makes it uniquely suited for data-intensive, real-time applications like **Chetan College Forum**, where features such as live polls and instant updates are crucial.

While Node.js is primarily associated with backend web development, it is a versatile environment capable of powering command-line tools, desktop applications, and Internet of Things (IoT) devices. In the context of the MERN stack, Node.js serves as the backbone, hosting the Express.js framework and managing interactions between the React frontend and the MongoDB database.

The Node.js ecosystem is supported by the Node Package Manager (NPM), which is the largest software registry in the world. This vast library of reusable packages allows developers to integrate complex functionalities—such as authentication, image processing, and database connectivity—without reinventing the wheel.

As of 2024, Node.js is the technology of choice for high-traffic platforms like Netflix, Uber, and LinkedIn, proving its reliability and scalability. By allowing the use of a single language (JavaScript)

on both the client and server sides, Node.js unifies web application development, streamlining the coding process and reducing the learning curve for developers.

Features of Node.js:

Asynchronous and Event-Driven: All APIs of the Node.js library are asynchronous, meaning they are non-blocking. A Node.js-based server never waits for an API to return data. The server moves to the next API after calling it, and a notification mechanism helps the server receive a response from the previous API call. This ensures high throughput and speed.

High Performance: Built on Google Chrome's V8 JavaScript Engine, Node.js compiles JavaScript code directly into native machine code. This results in incredibly fast execution speeds, making it superior for tasks requiring high computation or frequent database operations.

Single-Threaded but Scalable: Node.js uses a single-threaded model with event looping. This mechanism helps the server respond in a non-blocking way and makes the server highly scalable as opposed to traditional servers which create limited threads to handle requests. This allows **Chetan College Forum** to handle a large number of students simultaneously without crashing.

Unified Tech Stack (Full Stack JavaScript): One of the distinct advantages of Node.js is that it allows developers to write JavaScript for both the client-side (Frontend) and server-side (Backend). This eliminates the need to context-switch between languages (like PHP and JS), leading to cleaner, more consistent code and faster development cycles.

Cross-Platform: Like PHP, Node.js is completely cross-platform. It can run seamlessly on Windows, Linux, macOS, and Unix systems. This ensures that the application can be deployed on virtually any server environment without modification.

No Buffering: Node.js applications output the data in chunks rather than buffering the entire data set. This is particularly useful for features like uploading large images or streaming video content, providing a smoother user experience.

Active Community and NPM: Node.js boasts a massive, active community of developers. Through NPM (Node Package Manager), developers have access to millions of open-source libraries and modules. This community support ensures that security patches, updates, and new features are continuously available, keeping the platform secure and modern.

JSON Support: Since Node.js is JavaScript-based, it handles JSON (JavaScript Object Notation) natively. This is a significant advantage over other backend languages, as JSON is the standard format for API communication (REST APIs) and is also the native storage format of MongoDB, ensuring seamless data flow throughout the application..

INTRODUCTION OF MONGODB

What is a Database? A database is a specialized application designed to store, manage, and organize data in a structured manner. It provides distinct APIs that allow users to create, access, manage, search, and replicate the data it contains. While traditional systems used tabular data structures, modern web applications like **Chetan College Forum** often require more flexible data storage solutions to handle complex, unstructured data efficiently.

NoSQL Database Management System Unlike Relational Database Management Systems (RDBMS) which organize data into rigid tables, MongoDB is a **NoSQL (Not Only SQL)** database. It is classified as a **Document-Oriented Database**. Instead of rows and columns, it stores data in flexible, JSON-like documents. This structure allows for:

- **Dynamic Schemas:** Data structures can change over time without modifying the entire database.
- **High Scalability:** Designed to handle massive amounts of data across many servers (horizontal scaling).
- **Hierarchical Data Storage:** Complex data structures (like arrays and nested objects) can be stored within a single document, reducing the need for complex joins.

MongoDB Terminology Before exploring the specifics of MongoDB, it is important to understand the key terminology used in Document-Oriented databases and how they differ from traditional RDBMS:

- **Database:** A physical container for collections. Each database gets its own set of files on the file system.
- **Collection:** A collection is a grouping of MongoDB documents. It is the equivalent of an RDBMS **Table**. A collection exists within a single database.

- **Document:** A document is a set of key-value pairs. It is the equivalent of an RDBMS **Row** (or record). Documents have dynamic schemas, meaning documents in the same collection do not need to have the same set of fields.
- **Field:** A field is a key-value pair in a document. It is the equivalent of an RDBMS **Column**.
- **_id:** The primary key for every document. MongoDB automatically assigns a unique `_id` field to every document to ensure it can be uniquely identified.
- **BSON:** Binary JSON. This is the format MongoDB uses to store documents. It extends the JSON model to provide additional data types and to be efficient for encoding and decoding.
- **Embedding:** The practice of nesting data (such as comments inside a topic) directly within a document. This improves read performance by keeping related data together.
- **Sharding:** The process of storing data records across multiple machines to support big data deployments.

What is BSON and MQL? While SQL is used for relational databases, MongoDB uses **MQL (MongoDB Query Language)** and stores data in **BSON (Binary JSON)**.

BSON (Binary JSON): BSON is a binary-encoded serialization of JSON-like documents. It supports embedding of documents and arrays within other documents and records. BSON also contains extensions that allow representation of data types that are not part of the JSON spec, such as `Date` and `BinData`.

CRUD Operations (The Foundation of MQL): Similar to the DML (Data Manipulation Language) in SQL, MongoDB provides powerful operators to interact with data:

1. **Create:** `insertOne()`, `insertMany()` – Used to add new documents to a collection.

2. **Read:** `find()`, `findOne()` – Used to query data. MongoDB supports rich queries, including searching by field, range queries, and regular expression searches.
3. **Update:** `updateOne()`, `updateMany()` – Used to modify existing documents. It supports atomic operators like `$set`, `$inc` (increment), and `$push` (add to array).
4. **Delete:** `deleteOne()`, `deleteMany()` – Used to remove documents from a collection.

MongoDB MongoDB is a widely used, open-source, non-relational database management system developed by MongoDB Inc. It is designed for modern application developers and the cloud era. As a document database, MongoDB makes it easy for developers to store structured or unstructured data.

MongoDB is a core component of the **MERN Stack** (MongoDB, Express, React, Node.js) used in this project. It connects seamlessly with Node.js and React because data is passed as JSON objects throughout the entire application lifecycle.

Advantages of MongoDB:

- **Schema-less:** MongoDB is a schema-less database, which implies that one collection can hold different documents. This flexibility allows for rapid iteration during the development of features like Polls and Spaces in the Chetan College Forum.
- **Performance:** MongoDB stores data in internal memory (RAM) for faster access. It is optimized for high read/write throughput, making it ideal for real-time applications.
- **Scalability:** MongoDB supports horizontal scaling through **Sharding**, allowing the database to handle growing amounts of data by distributing it across multiple servers.
- **Deep Querying:** MongoDB supports dynamic queries on documents using a document-based query language that is nearly as powerful as SQL.

- **High Availability:** MongoDB's replication feature (Replica Sets) provides automatic failover and data redundancy, ensuring the application remains online even if a server fails.
- **JSON Compatibility:** Since the entire project is built on JavaScript (React frontend, Node backend), MongoDB's native JSON-like storage format eliminates the need for complex data translation layers, simplifying the codebase.

DEFINITION OF PROBLEM FOR SHUBHVIVAH

The current landscape of wedding planning and vendor discovery faces several challenges related to transparency, organization, and stress management. While couples utilize various offline methods and generic directories, they often lack a centralized, structured platform dedicated to the specific nuances of Indian weddings. These problems impact the efficiency of budget management and the overall celebration experience.

Below are the primary problems SubhVivah addresses:

1. Fragmented Vendor Discovery:

- **Issue:** Reliable vendor information is currently scattered across word-of-mouth recommendations, ephemeral social media posts, and unverified local directories.
- **Impact:** Couples spend excessive time vetting vendors, leading to decision fatigue and difficulty in comparing prices or quality objectively.

2. Lack of Structured Service Organization:

- **Issue:** Without specific categorization, distinct services (e.g., Candid Photography vs. Traditional Videography, or Banquet Halls vs. Resorts) get mixed together in generic listings.
- **Impact:** Users struggle to find niche services that match their specific vision, leading to a cluttered search experience.

3. Pricing Opacity and Validation:

- **Issue:** In the traditional market, pricing is often hidden or arbitrary ("Call for Price"), making it difficult to distinguish between budget-friendly and luxury options.

- **Impact:** Couples face budget overruns due to hidden costs. The platform addresses this by displaying clear "Starting Prices" and categorizations.

4. Lack of Personalized Planning Tools:

- **Issue:** Static directories offer no way to track progress. Couples often rely on pen-and-paper or disjointed spreadsheets to manage tasks.
- **Impact:** Critical tasks are forgotten. The platform needs interactive features like **Checklists** and **Dashboards** to keep the planning on track.

5. Database Scalability and Media Performance:

- **Issue:** Wedding planning relies heavily on visuals. As the number of vendor portfolios and high-resolution images grows, retrieving data efficiently becomes challenging.
- **Impact:** Slow loading times can deter users. The system needs optimized database queries and efficient image handling to manage visual-heavy content.

6. Mobile Accessibility:

- **Issue:** Modern couples plan their weddings on the go, primarily accessing the internet via smartphones.
- **Impact:** A desktop-centric interface would alienate a vast majority of users. The platform requires a fully responsive design to ensure seamless access on mobile devices.

7. Data Security and User Privacy:

- **Issue:** Handling sensitive personal data (phone numbers, email addresses, wedding dates) requires strict security measures.

- **Impact:** Potential vulnerability to spam or unauthorized access. The system must implement robust authentication (**JWT**) and password encryption (**Bcrypt**) to safeguard user identities.

SYSTEM ANALYSIS

System analysis for **Chetan College Forum** involves understanding the requirements of students and the academic community. The system must provide a seamless user experience while ensuring data security and efficient management of discussion topics. The analysis phase includes identifying key functionalities that will make the forum a vibrant hub for learning.

1. Problem Identification:

- Current methods of campus communication are disorganized and lack permanence.
- Students struggle to find archived answers to common queries.
- There is no unified platform that combines social interaction with structured academic categorization (Spaces).

2. User Requirements:

- **Students (Seekers):** Need a platform to ask questions, search for existing answers, filter topics by interest (Spaces), and receive notifications for replies.
- **Contributors (Experts):** Require features to post detailed answers, share rich content (code snippets, images), create polls, and build a reputation via upvotes and follower counts.

3. System Requirements:

- **Functional:**
    - User Authentication (Login/Register/Email Verify).
    - Topic Management (Create, Read, Update, Delete) with Rich Text.
    - Poll Creation and Voting mechanism.
    - Space and Tag management for categorization.

- Interactive features: Upvoting, Downvoting, and Commenting.

- **Non-Functional:**
  - **Scalability:** The ability to handle growing numbers of users and topics via MongoDB's document model.
  - **Performance:** Fast page loads using React's Virtual DOM and Single Page Application (SPA) architecture.
  - **Security:** Protection against common web vulnerabilities (XSS, NoSQL Injection) and secure session management.

4. System Architecture:

- The system is based on the **MERN architecture**, separating the frontend (Client) from the backend (Server).
- **Frontend (View):** Developed using **React.js** and **Redux Toolkit**, providing a dynamic, responsive user interface that communicates via APIs.
- **Backend (Controller/Model):** Developed using **Node.js** and **Express.js**, managing business logic and API routes.
- **Database:** Uses **MongoDB**, a NoSQL database, to store unstructured data like discussion threads and user profiles efficiently.

5. Data Flow:

- The system manages user input (e.g., creating a poll) through React forms.
- Data is sent via **JSON** payloads over HTTP requests to the Node.js server.
- The server validates the token (JWT), processes the logic, and stores the data in MongoDB collections.

- The response is sent back to the client to update the UI instantly without a page reload (via Redux state updates).

6. User Interface and Experience:

- The system emphasizes specific "Spaces" (e.g., Technology, Music) accessible via a sidebar.
- **React-Bootstrap** is used to ensure the layout is consistent and responsive across all devices.

SYSTEM REQUIREMENTS

Minimum Hardware Requirements for Development

- **CPU:** Intel Core i5 or equivalent (Node.js and React dev servers require moderate processing power).
- **Memory (RAM):** 8 GB (4 GB is absolute minimum, but 8 GB is recommended for running MongoDB, Backend, and Frontend servers simultaneously).
- **Hard Disk:** 256 GB SSD (Solid State Drive is preferred for faster compilation and database reads).
- **System Type:** 64-bit Operating System.

Minimum Software Requirements for Development

- **Operating System:** Windows 10/11, macOS, or Linux (Ubuntu 20.04+).
- **Runtime Environment:** Node.js (v14.0.0 or later).
- **Database:** MongoDB (Local installation or MongoDB Atlas Cloud).
- **Package Manager:** NPM (Node Package Manager) or Yarn.
- **Code Editor:** Visual Studio Code (VS Code) with extensions for ES7+ React/Redux and Prettier.
- **API Testing:** Postman or Thunder Client.
- **Version Control:** Git and GitHub.
- **Browser:** Google Chrome (with React Developer Tools installed).

Minimum Hardware and Software Requirements for Running the Website (Client-Side)

On PCs (Desktops and Laptops)

- **CPU:** Intel Core i3 or equivalent.
- **Memory (RAM):** 4 GB or higher.
- **Internet Connection:** Broadband connection for loading dynamic content and real-time updates.
- **Web Browser:** Modern browsers supporting ES6 JavaScript features (Google Chrome, Mozilla Firefox, Microsoft Edge, Safari).

On Mobile Devices (Phones and Tablets)

- **CPU:** Quad-core processor (Snapdragon 600 series or equivalent) for smooth UI rendering.
- **Memory (RAM):** 3 GB or higher.
- **Operating System:** Android 10+ or iOS 14+.
- **Browser:** Chrome for Android, Safari, or Firefox Mobile.
- **Screen Resolution:** 720p HD or higher (The responsive design adjusts automatically to fit these screens).
- ---

SYSTEM DESIGN AND CODING

INTRODUCTION

Software design is a critical phase in the development of **SubhVivah**, laying the architectural foundation for the entire wedding planning platform. It transcends basic coding by establishing a structured plan for how the MERN stack components (MongoDB, Express, React, Node.js) interact. Design is initiated after the system requirements (such as the need for Vendor Categories, User Checklists, and Vendor Shortlisting) have been specified and analyzed. This step forms the core of the development phase and precedes the actual implementation of features. The primary goal of this phase is to create a scalable marketplace model that ensures seamless data flow between the client and server.

The central focus of the design for SubhVivah is **User Experience (UX)** and **Data Integrity**. Through the design process, the quality of the final platform is ensured by translating user requirements—like the need for city-based vendor filtering and budget-friendly sorting—into a structured framework of React Components and API Endpoints. A robust design minimizes the risk of bugs, ensures secure data handling (e.g., password encryption via Bcrypt), and creates a system that is easy to maintain and expand in the future.

In this phase, detailed representations of data structures (Mongoose Schemas for Vendors and Users), program structures (Redux State Slices for User Authentication), and procedural steps (JWT Authentication Middleware) are defined and implemented. From both a technical and project management perspective, this system design plays a pivotal role in guiding the project toward its successful deployment.

INPUT DESIGN

Input design is the process of converting user-oriented input into a computer-readable format. In **SubhVivah**, input design focuses on the various forms and interactive elements couples use to plan their weddings. As a key component of the overall system design, this was executed with meticulous attention to detail to ensure that users can easily search for vendors, manage checklists, and customize their profiles without technical friction.

The primary objectives of input design in this project were:

1. **Efficiency:** To create input processes (like One-Click Shortlisting or Dynamic City Filtering) that are fast and minimize user effort.
2. **Accuracy:** To ensure high data integrity by implementing client-side validation (React/Formik) and server-side validation (Node.js/Mongoose) to prevent invalid inquiries or duplicate registrations.
3. **User-Friendliness:** To ensure that complex actions, such as filtering vendors by multiple criteria (Category + City + Price), are intuitive and straightforward.

Input Data Considerations in SubhVivah: When designing the input interfaces, the goal was to create a process that is logical and error-free.

- **Data Recording:** Capturing raw data from users via React Forms. This includes text inputs for Search Queries, extensive details for Vendor Inquiries, and checkbox inputs for Checklist management.
- **Data Conversion:** Converting user actions into JSON format. For example, when a user selects "Mumbai" from a dropdown, it is converted into a query string parameter sent to the backend API.

- **Data Verification:** Checking inputs before submission. For instance, the **Registration** form prevents submission if the passwords do not match or if the email format is invalid.
- **Interactive Input:** The system features real-time interactive inputs, such as the **"Heart" (Shortlist)** buttons which capture user preference immediately without a page reload.

Specific Input Interfaces Designed:

- **Authentication Forms:** Secure inputs for Name, Email, and Password with validation checks during registration and login.
- **Vendor Filter Bar:** A comprehensive input module allowing users to select a "City" (e.g., Udaipur, Mumbai) and "Category" (e.g., Photographer, Makeup) to dynamically refine results.
- **Checklist Manager:** A dynamic input module where users can add new tasks, mark items as complete, or delete tasks from their wedding to-do list.
- **Profile Settings:** Inputs for updating personal details, partner's name, and wedding dates, which are then processed to personalize the dashboard experience.

OUTPUT DESIGN

Output design refers to the process of determining how the results of processing will be communicated to the users. In **SubhVivah**, output design plays a crucial role in displaying vendor portfolios, dashboard statistics, and inquiry statuses in a way that is readable and engaging.

The outputs of the system are categorized as follows:

1. **Interactive Outputs:** The most common output type in this Single Page Application (SPA). This includes the **Vendor Listing Grid**, where data fetched from the database is dynamically

rendered as a list of interactive cards. When a user filters by "Venue," the output updates instantly via React state changes.

2. **Visual Outputs:** The **Vendor Details Modal** is a key visual output. Instead of just showing text, the system renders high-resolution images, star ratings, and price points in a visually rich layout, making the data easy to interpret at a glance.
3. **Operational Outputs:** System feedback such as "Added to Shortlist" or "Message Sent Successfully" (Toast Notifications) helps users understand the status of their actions immediately.
4. **Turnaround Outputs:** Data that allows for further action, such as the **Dashboard Overview**, which displays a summary of "Total Vendors Shortlisted" and "Pending Checklist Items," prompting the user to take further planning steps.

Key Output Design Principles Applied:

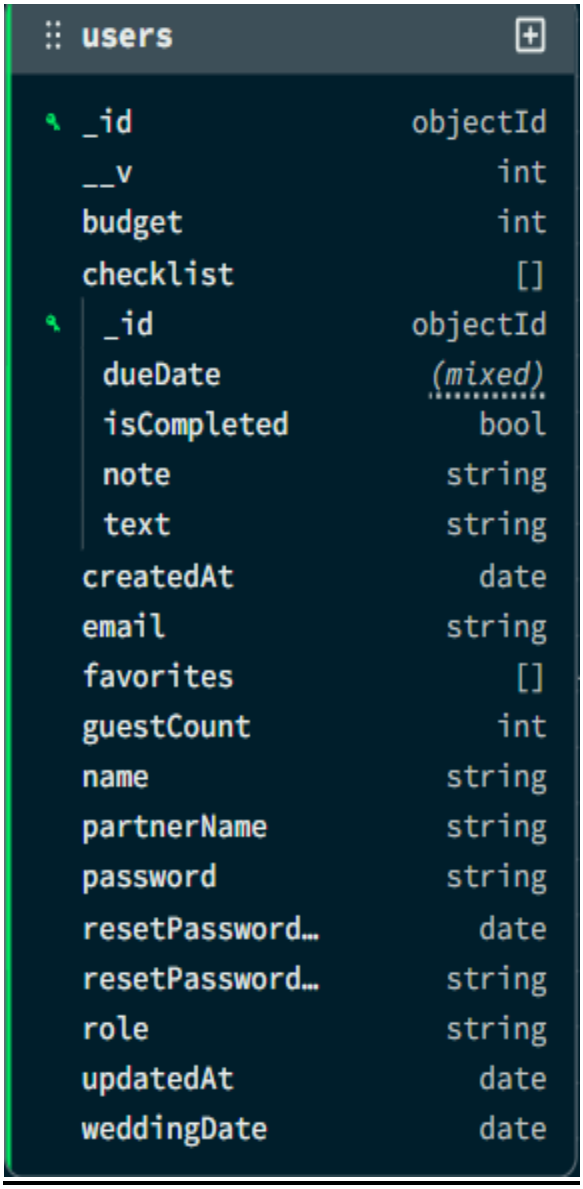
- **Responsiveness:** Using the **Tailwind CSS** framework, outputs are designed to adapt to any screen size. A vendor grid that displays as three columns on desktop transforms into a vertical stack on mobile devices.
- **Clarity:** Information is structured hierarchically. The Vendor Name is prominent, followed by key metadata (City, Category, Rating), and finally the Price, ensuring users can quickly scan for relevant information.
- **Navigation:** The layout includes a dynamic **Sidebar** in the Dashboard (Output) that lists active tabs like Overview, Vendors, and Settings, allowing users to navigate through the system effectively based on the data presented.

CONCLUSION

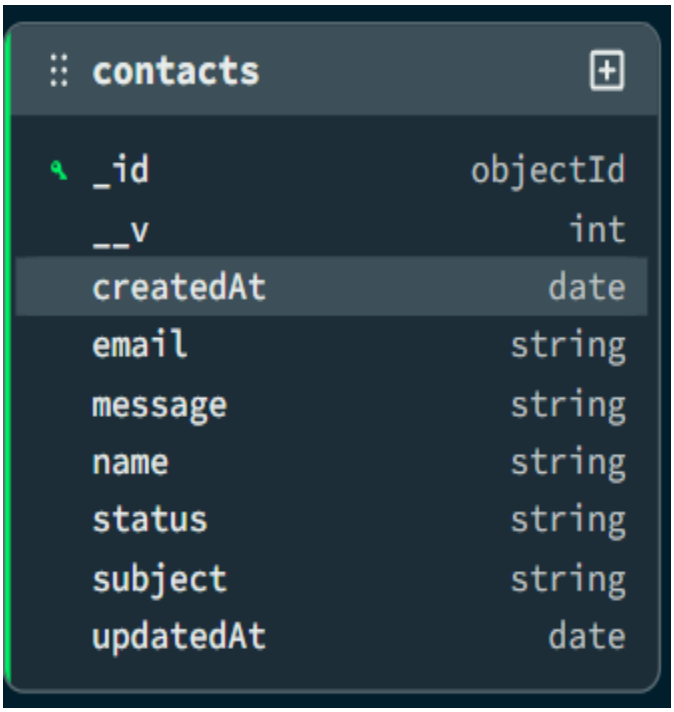
In the software engineering process of **SubhVivah**, system design was foundational for building a quality, reliable, and user-friendly wedding planning platform. The rigorous **Input Design** ensures that vendor searches and user preferences are captured accurately and validated before storage. Meanwhile, the thoughtful **Output Design** guarantees that complex vendor data is presented in a well-structured, visually appealing manner that encourages engagement. By focusing on these design principles using the **MERN stack**, the system successfully meets the needs of modern couples, operating efficiently and remaining scalable for future enhancements.

DATABASE DESIGN

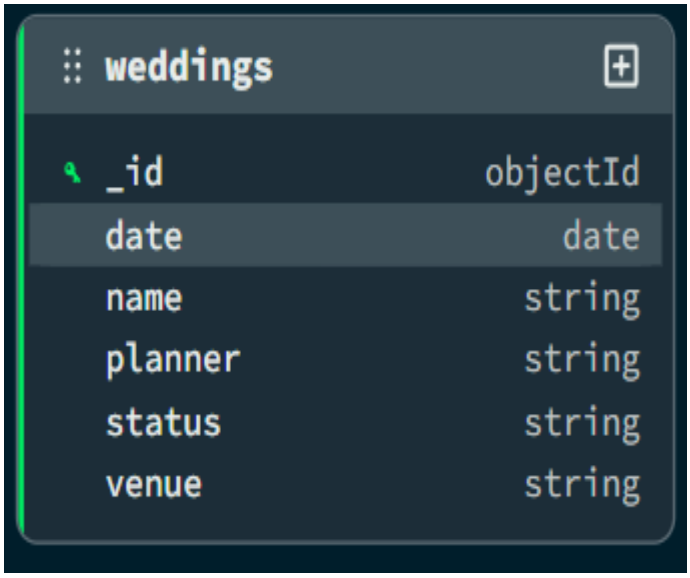
1. Users:



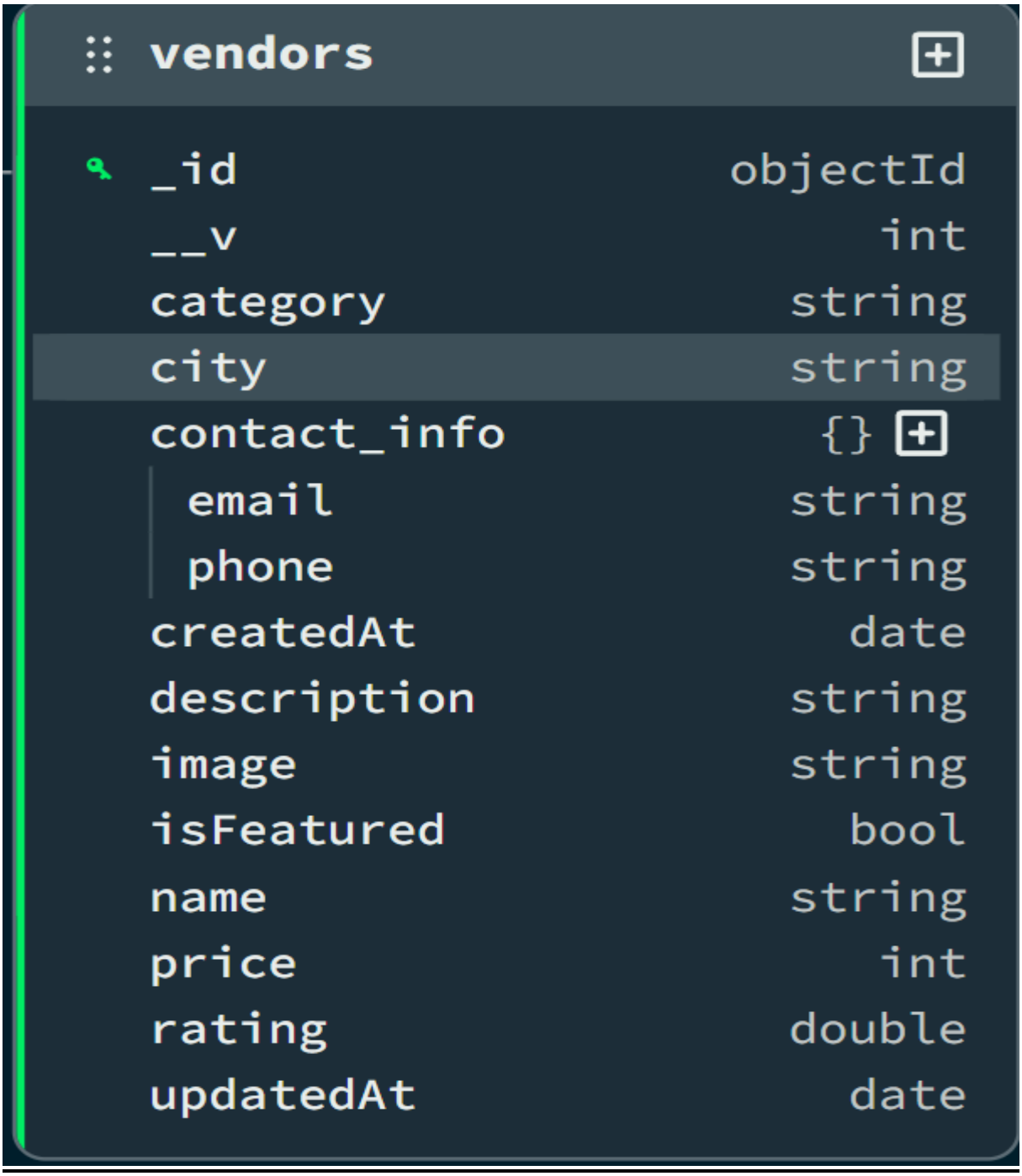
2. **Contacts:**



3. **Weddings:**



4. **Vendors**


vendors
_id objectId
__v int
category string
city string
contact_info {}
email string
phone string
createdAt date
description string
image string
isFeatured bool
name string
price int
rating double
updatedAt date

DATA FLOW DIAGRAM (DFD)

A Data Flow Diagram (DFD) is a graphical tool used to describe and analyze the flow of data through a system. It focuses on the data flowing into the system, the processes that transform the data, and the data flowing out to storage or external entities.

DFDs are of two types:

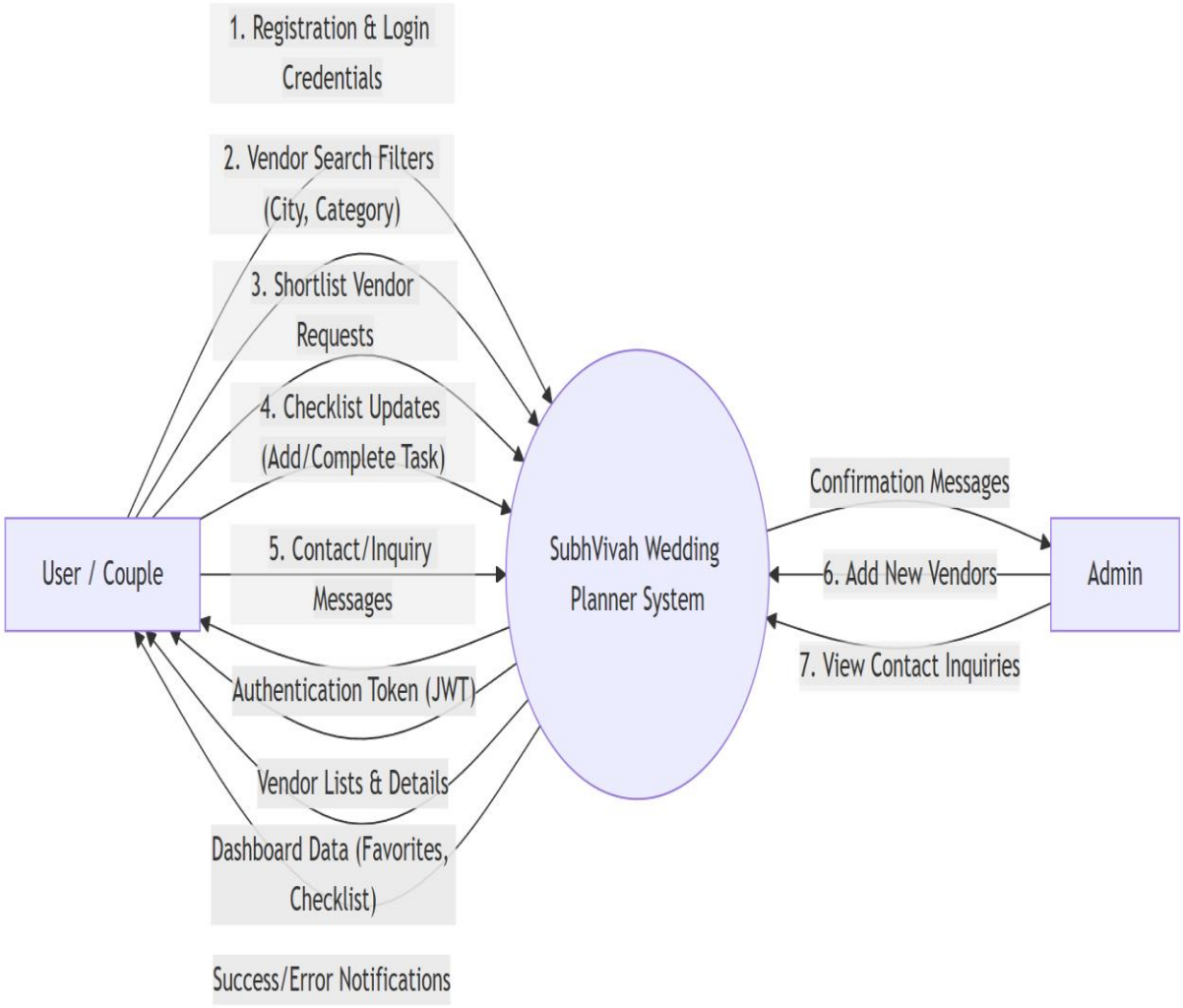
1. **Physical DFD:** The physical DFD is a model of the current system and is used to ensure that the current system has been clearly understood.
2. **Logical DFD:** The logical DFD is a model of the proposed system. It clearly shows the requirements on which the new system should be built, focusing on *what* happens, not *how* it happens.

Notation Used In DFD: There are four simple notations used to complete DFDs:

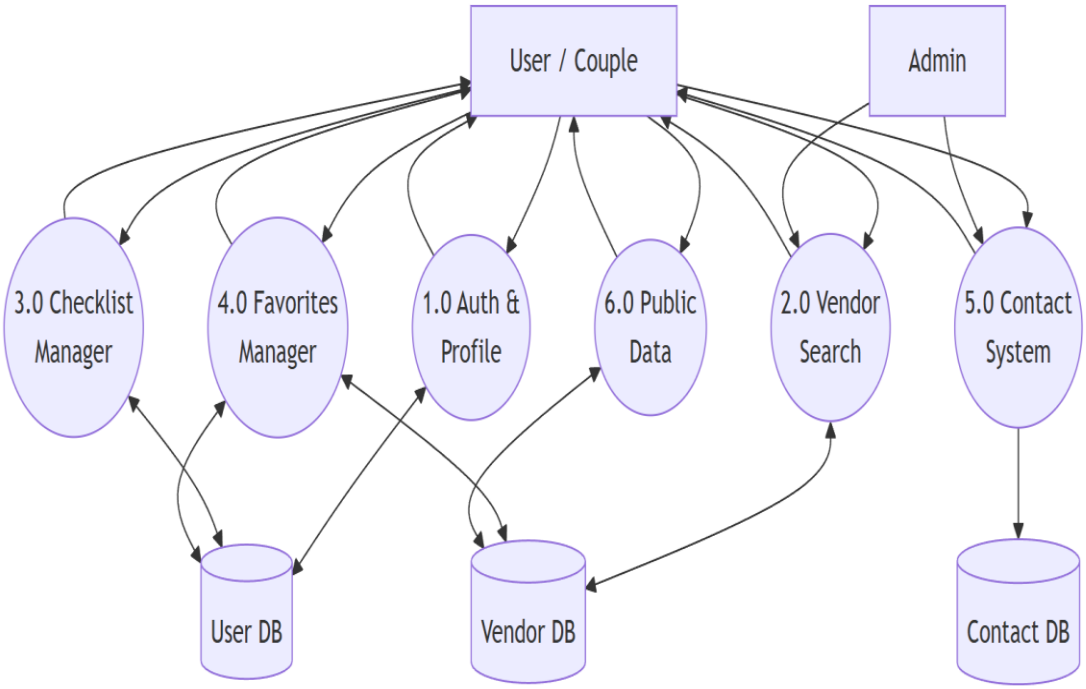
- **Data Flow (Arrow):** Represents the movement of data packets from one part of the system to another.
- **External Entity (Rectangle):** Represents sources (where data comes from) or destinations (where data goes) outside the system boundaries (e.g., Users, Admin).
- **Process (Circle/Rounded Rectangle):** Represents a function or logic that transforms incoming data into outgoing data (e.g., "Verify Login").
- **Data Store (Open Rectangle/Parallel Lines):** Represents repositories where data is stored for later use (e.g., Database Collections like Users, Topics).
-

DFD FOR THE SYSTEM

1. Level 0:



2. Level 1:


User / Couple
Admin
3.0 Checklist Manager
4.0 Favorites Manager
1.0 Auth & Profile
6.0 Public Data
2.0 Vendor Search
5.0 Contact System
User DB
Vendor DB
Contact DB

ENTITY RELATIONSHIP DIAGRAM

An Entity Relationship Diagram (ERD) expresses the overall logical structure of the database graphically. It illustrates the interrelationships between the entities in the **SubhVivah Wedding Planner** system, providing a clear blueprint of how data is organized and connected within the MongoDB database.

The components of the E-R Diagram are:

1. **Entity:** An entity is a real-world object or concept that is distinguishable from all other objects. In this project, examples include **User**, **Vendor**, and **Contact**.
2. **Relationship:** It is an association among several entities. For instance, a "User" shortlists a "Vendor," representing a relationship between the two.
3. **Weak Entity:** A weak entity is an entity that cannot be uniquely identified by its own attributes alone and depends on a strong entity. In this system, a **Checklist Item** can be viewed conceptually as a weak entity because it is embedded within and dependent on the existence of a specific User.
4. **Attributes:** A property or characteristic of an entity. For example, a Vendor entity has attributes like Name, Category, and Price.

ENTITIES IN SUBHVIVAH

Based on the Mongoose models designed for the system, the key entities and their attributes are:

1. User: Represents the couples or individuals registered on the platform to plan their wedding.

- **Attributes:** `_id`, `name`, `email`, `password`, `role` (default: 'user'), `partnerName`, `weddingDate`, `budget`, `guestCount`, `favorites` (Array of Vendor IDs), `checklist` (Array of tasks), `resetPasswordToken`, `resetPasswordExpire`.

2. Vendor: Represents the service providers (venues, photographers, etc.) listed on the marketplace.

- **Attributes:** `_id`, `name`, `category` (Enum: Venue, Photographer, Makeup, etc.), `city`, `price`, `rating`, `image`, `description`, `isFeatured`, `contact_info` (phone, email).

3. Contact: Represents the inquiries or feedback messages sent by users to the platform administrators.

- **Attributes:** `_id`, `name`, `email`, `subject`, `message`, `status` (Enum: New, Read, Replied), `createdAt`.

RELATIONSHIPS

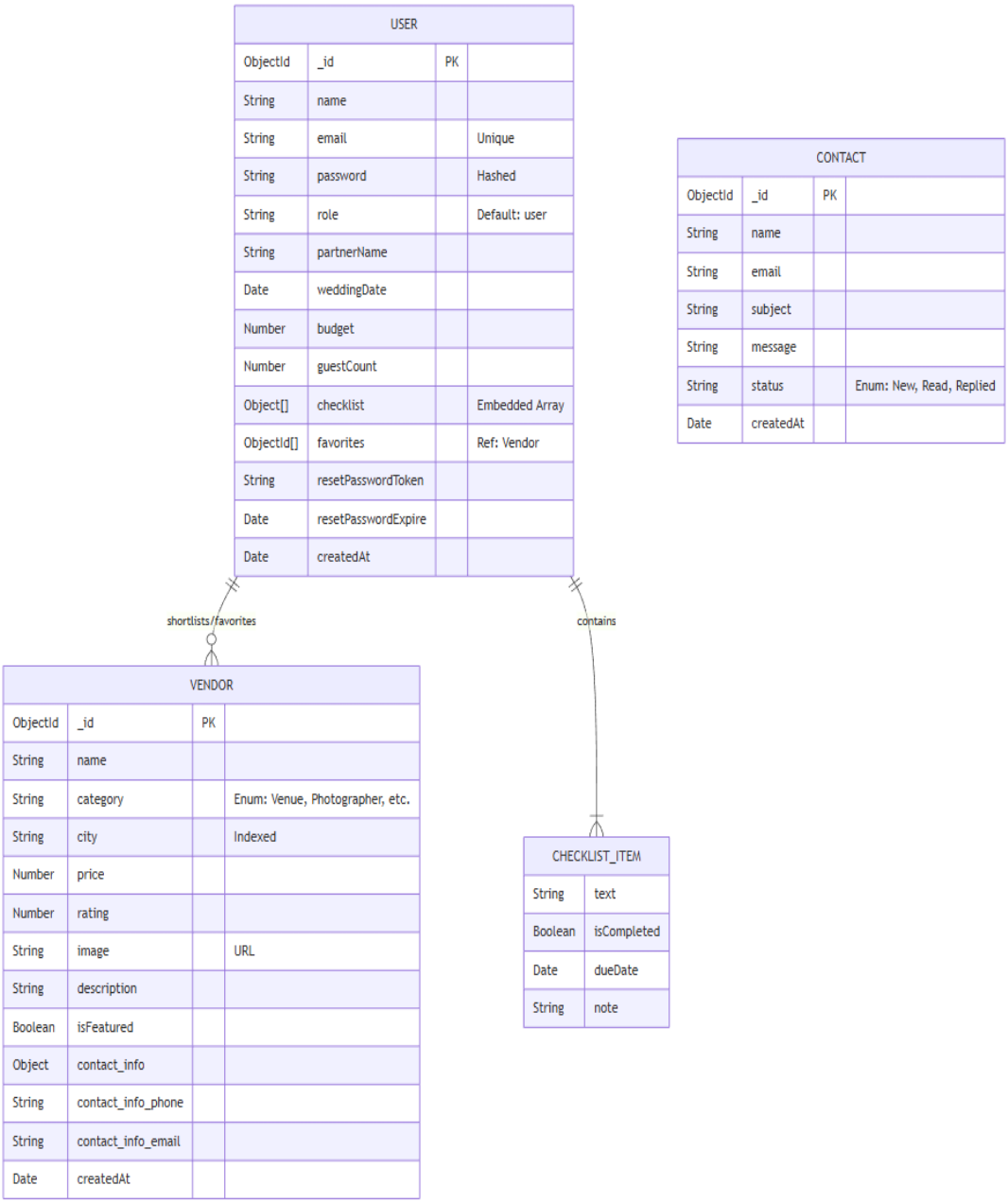
The system enforces the following relationships to ensure data integrity and connectivity:

1. **User — Vendor (M:N):**
    - **Relationship:** "Shortlists" or "Favorites".
    - **Description:** A single User can shortlist multiple Vendors to their dashboard. Conversely, a single Vendor can be shortlisted by multiple Users. This Many-to-Many relationship is implemented via the `favorites` array in the User model which stores Vendor ObjectIds.
2. **User — Checklist Item (1:N):**
    - **Relationship:** "Manages".

- **Description:** A single User acts as a parent to multiple Checklist Items (One-to-Many). These items (e.g., "Book Venue", "Finalize Menu") are embedded directly within the User document for efficient retrieval.

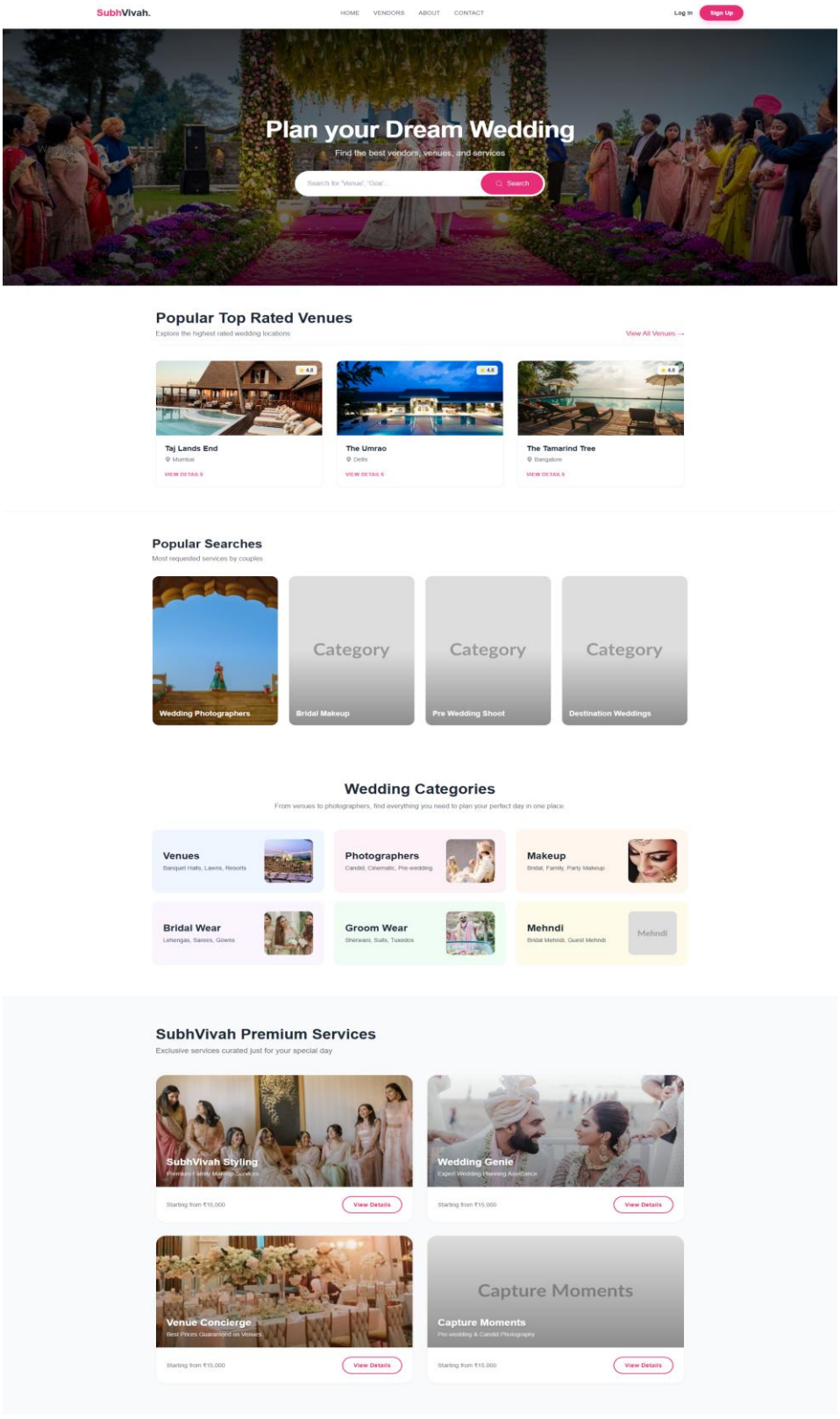
3. **Vendor — Category (N:1):**
    - **Relationship:** "Belongs To".
    - **Description:** Multiple Vendors can belong to a single Category (e.g., many vendors are "Photographers"), but a specific Vendor is assigned to only one primary Category.
4. **User — Contact (1:N):**
    - **Relationship:** "Submits".
    - **Description:** A User can submit multiple Contact inquiries to the system. While `Contact` documents are stored independently, they logically link back to the user via the email address provided.

ENTITY RELATIONSHIP DIAGRAM



INPUT FORMS, PAGES AND CODING

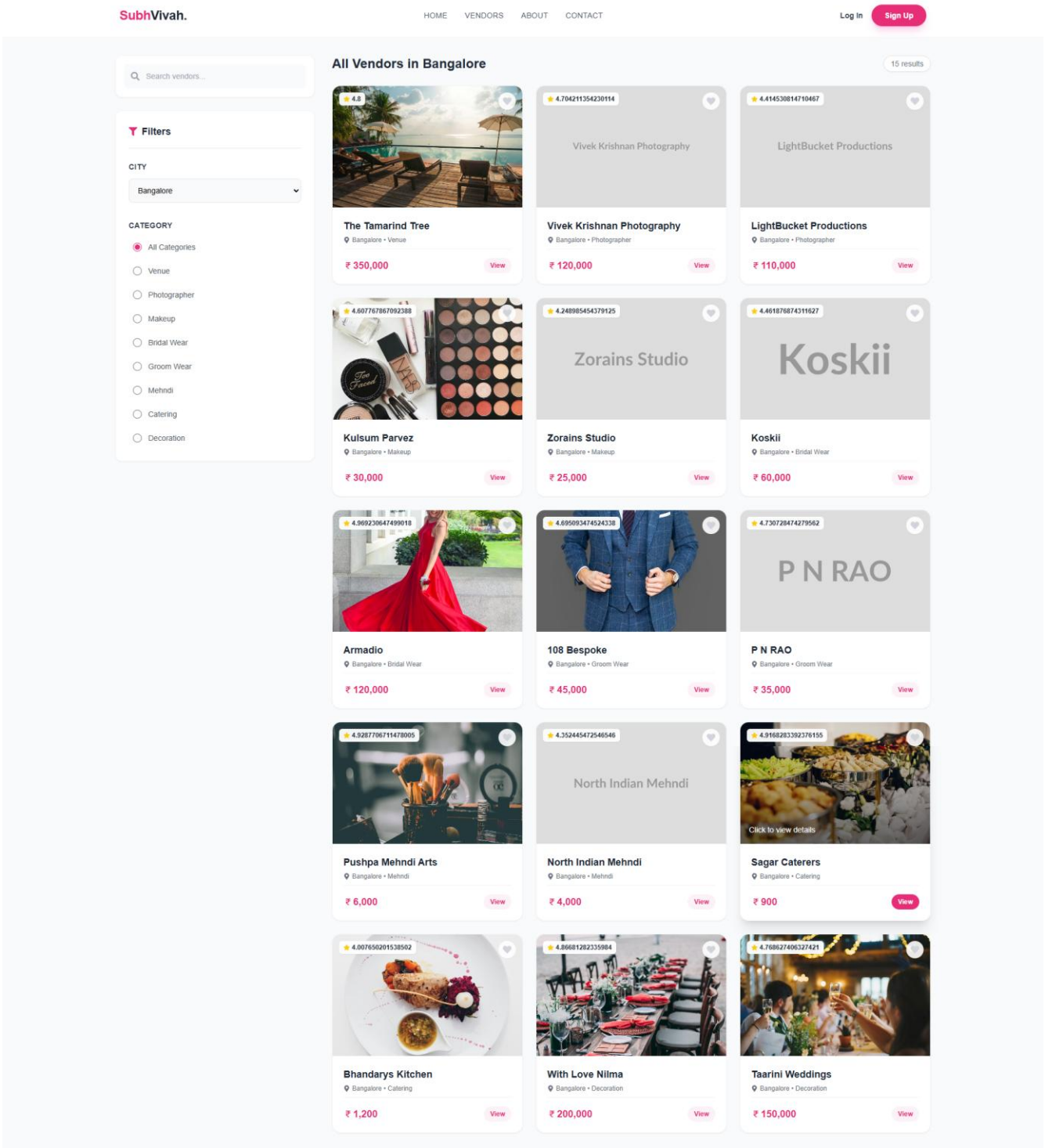
1. Home Page:



Code:

```
import Hero from "./components/Hero";
import InhouseServices from "./components/InhouseServices";
import PopularSearch from "./components/PopularSearch";
import PopularVenue from "./components/PopularVenue";
import WeddingCategories from "./components/WeddingCategories";

const HomePage = () => {
  return (
    <div className='min-h-screen font-sans'>
      {/* Hero Section: Main Banner */}
      <Hero />

      {/* Dynamic Data Sections */}
      <PopularVenue />
      <PopularSearch />

      {/* Static Categories & Services */}
      <WeddingCategories />
      <InhouseServices />
    </div>
  )
}

export default HomePage;
```

2. Vendors Page



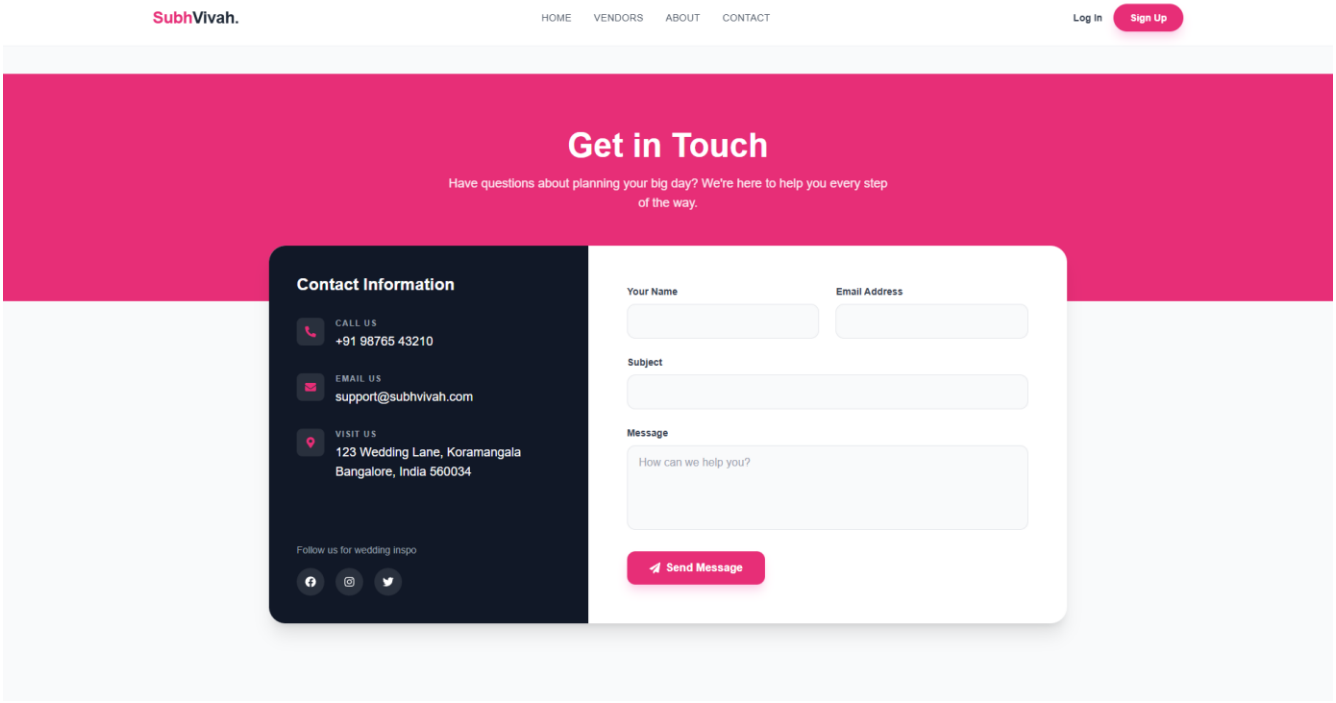
Code:

```
import { useState, useEffect } from 'react';
import { useSearchParams } from 'react-router';
import { AxiosClient } from '../../config/axiosClient';
// ... icon imports

const VendorListingPage = () => {
```

```
  // 1. State Management
  const [vendors, setVendors] =
useState<Vendor[]>([]);
  const [loading, setLoading] = useState(true);
  const [searchParams, setSearchParams] =
useSearchParams();

  // 2. Extract Filters from URL
  const city = searchParams.get('city') || '';
  const category = searchParams.get('category') || '';

  // 3. Fetch Data on Filter Change
  useEffect(() => {
    const fetchVendors = async () => {
      setLoading(true);
      try {
        const query = new URLSearchParams({ city,
category }).toString();
        const response = await
AxiosClient.get(`/vendors?${query}`);
        setVendors(response.data.vendors);
      } catch (error) {
        console.error("Error fetching vendors:",
error);
      } finally {
        setLoading(false);
      }
    };
    fetchVendors();
  }, [city, category]);

  return (
    <div className="container mx-auto flex gap-8">
      {/* Sidebar: Filters */}
      <aside className="w-1/4">
        <input
          type="text"
          placeholder="Search vendors..."
          onChange={(e) =>
handleFilterChange('search', e.target.value)}
        />
        <select value={city} onChange={(e) =>
handleFilterChange('city', e.target.value)}>
          <option value="">All Cities</option>
          {cities.map(c => <option key={c}
value={c}>{c}</option>)}
        </select>
        {/* Category Radio Buttons */}
```

```
            {CATEGORIES.map(cat => (
              <label key={cat}>
                <input type="radio" checked={category
=== cat} onChange={() =>
handleFilterChange('category', cat)} />
                {cat}
              </label>
            ))}
          </aside>

          {/* Main Content: Vendor Grid */}
          <main className="w-3/4">
            <div className="grid grid-cols-3 gap-6">
              {vendors.map((vendor) => (
                <div key={vendor._id} onClick={() =>
setSelectedVendor(vendor)} className="card">
                  <img src={vendor.image}
alt={vendor.name} className="h-56 w-full object-
cover" />
                  <div className="p-4">
                    <h3>{vendor.name}</h3>
                    <p>{vendor.city} •
{vendor.category}</p>
                    <p>Rs. {vendor.price}</p>
                  </div>
                </div>
              ))}
            </div>
          </main>
        </div>
      );
};

export default VendorListingPage;
```

3. Contact Page



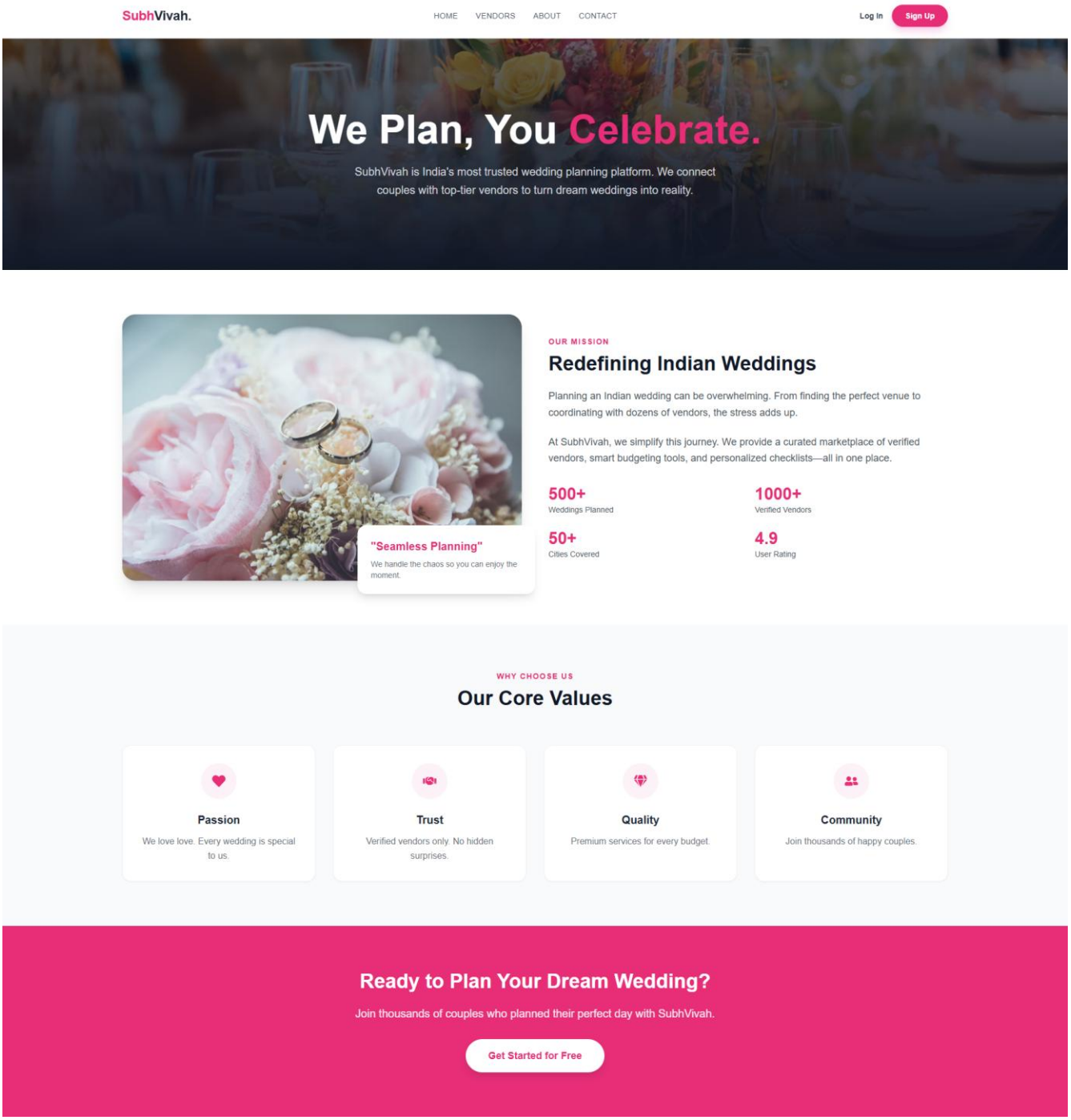
Code:

```
import React, { useState } from 'react';
import { AxiosClient } from '../../config/axiosClient';
import { toast } from 'react-toastify';
// ... imports

const ContactPage = () => {
  // State for form data and loading status
  const [formData, setFormData] = useState({ name: '', email: '', subject: '', message: '' });
  const [loading, setLoading] = useState(false);

  // Handle Input Change
  const handleChange = (e: React.ChangeEvent<HTMLInputElement | HTMLTextAreaElement>) => {
    setFormData({ ...formData, [e.target.name]: e.target.value });
  };

  // Handle API Submission
  const handleSubmit = async (e: React.FormEvent) => {
    e.preventDefault();
    setLoading(true);
    try {
```

```
        await AxiosClient.post("/contact", formData);
        toast.success("Message sent successfully!");
        setFormData({ name: '', email: '', subject: '', message: '' }); // Reset form
      } catch (error) {
        toast.error("Failed to send message.");
      } finally {
        setLoading(false);
      }
    };

    return (
      <div className="min-h-screen bg-gray-50 pt-10 pb-20">
        <div className="container mx-auto">
          <div className="flex flex-col md:flex-row bg-white rounded-3xl shadow-
xl">

            {/* Static Contact Info Sidebar */}
            <div className="md:w-2/5 bg-gray-900 text-white p-10">
              <h2>Contact Information</h2>
              <div className="space-y-8">
                <ContactItem icon={<FaPhoneAlt />} title="Call Us" value="+91
98765 43210" />
                <ContactItem icon={<FaEnvelope />} title="Email"
value="support@subhvivah.com" />
              </div>
            </div>

            {/* Dynamic Contact Form */}
            <div className="md:w-3/5 p-10">
              <form onSubmit={handleSubmit} className="space-y-6">
                <div className="grid grid-cols-2 gap-6">
                  <InputField name="name" value={formData.name}
onChange={handleChange} label="Name" />
                  <InputField name="email" value={formData.email}
onChange={handleChange} label="Email" />
                </div>
                <InputField name="subject" value={formData.subject}
onChange={handleChange} label="Subject" />
                <textarea
                  name="message"
                  value={formData.message}
```

```
                        onChange={handleChange}
                        className="w-full border p-3 rounded-xl"
                      />

                      <button disabled={loading} type="submit" className="bg-primary
text-white px-8 py-3 rounded-xl">
                        {loading ? "Sending..." : "Send Message"}
                      </button>
                    </form>
                  </div>

                </div>
              </div>
            </div>
    );
};

export default ContactPage;
```

4. About Page



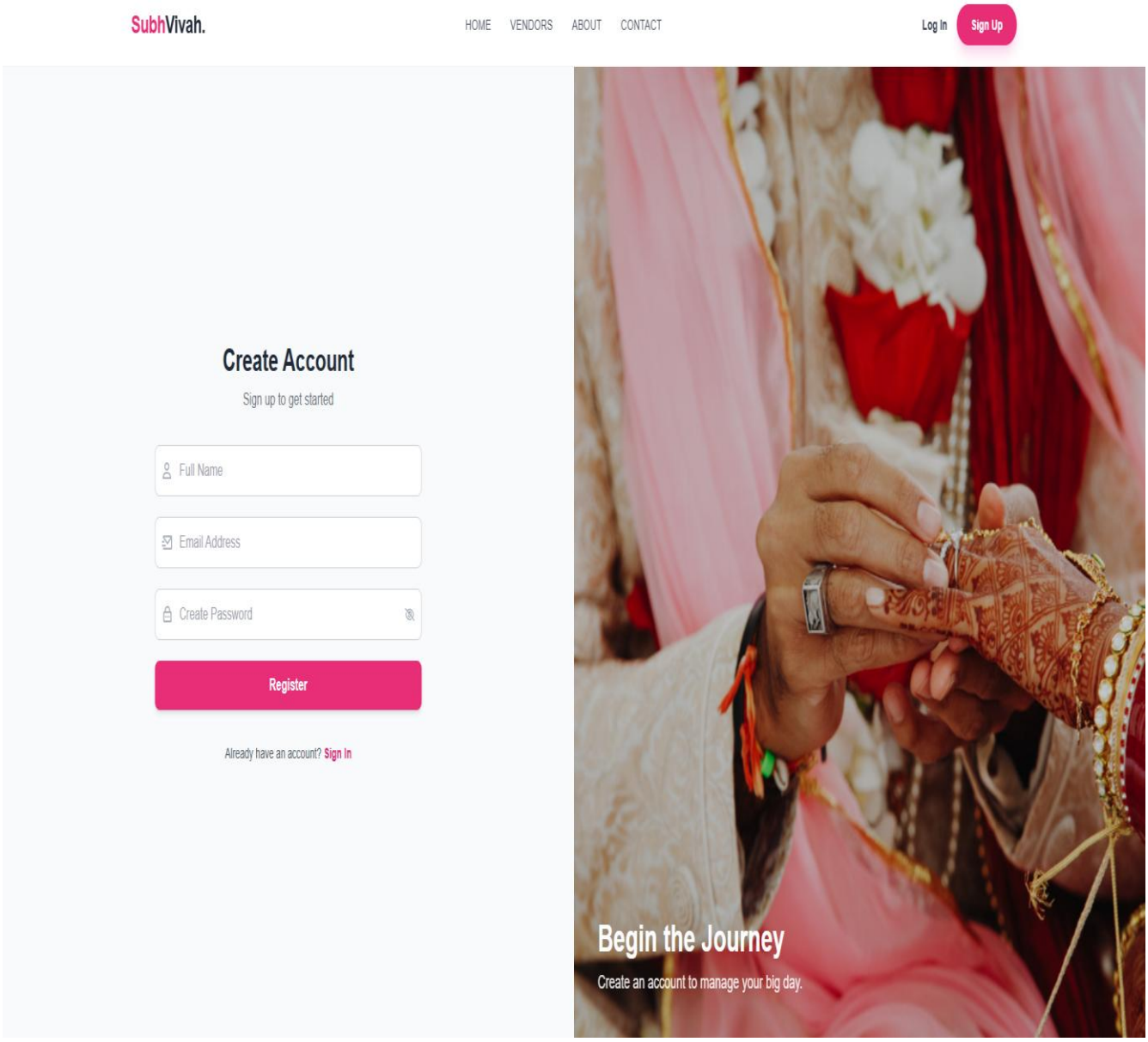
Code:

```
import { motion } from 'framer-motion';
// ... icons and router imports

const AboutPage = () => {
  return (
    <div className="font-sans">
      {/* 1. Hero Section with Animation */}
```

```
        <section className="relative bg-gray-900 text-white py-32">
          <div className="container mx-auto text-center">
            <motion.h1
              initial={{ opacity: 0, y: 20 }}
              animate={{ opacity: 1, y: 0 }}
              className="text-5xl font-bold"
            >
              We Plan, You <span className="text-primary">Celebrate.</span>
            </motion.h1>
            <p className="text-xl">SubhVivah is India's most trusted wedding
platform...</p>
          </div>
        </section>

        {/* 2. Mission & Stats Section */}
        <section className="py-20 bg-white">
          <div className="grid grid-cols-1 md:grid-cols-2 gap-12">
            <img src="/our-story.jpg" alt="Our Story" className="rounded-3xl" />
            <div>
              <h2 className="text-4xl font-bold">Redefining Indian Weddings</h2>
              <div className="grid grid-cols-2 gap-6">
                <Stat number="500+" label="Weddings Planned" />
                <Stat number="1000+" label="Verified Vendors" />
                {/* ... more stats */}
              </div>
            </div>
          </div>
        </section>

        {/* 3. Core Values Component Reuse */}
        <section className="py-20 bg-gray-50">
          <div className="grid grid-cols-1 md:grid-cols-4 gap-8">
            <ValueCard icon={<FaHeart />} title="Passion" desc="We love love." />
            <ValueCard icon={<FaHandshake />} title="Trust" desc="Verified
vendors." />
            {/* ... more values */}
          </div>
        </section>
      </div>
  );
};
```

```
// Reusable Helper Components
const Stat = ({ number, label }: { number: string, label: string }) => (
   <div><p className="text-3xl font-bold">{number}</p><p>{label}</p></div>
);

const ValueCard = ({ icon, title, desc }: any) => (
   <div className="bg-white p-8 rounded-2xl text-center">
      <div className="text-primary text-2xl mb-6">{icon}</div>
      <h3 className="text-xl font-bold">{title}</h3>
      <p>{desc}</p>
   </div>
);

export default AboutPage;
```

5. Register Page



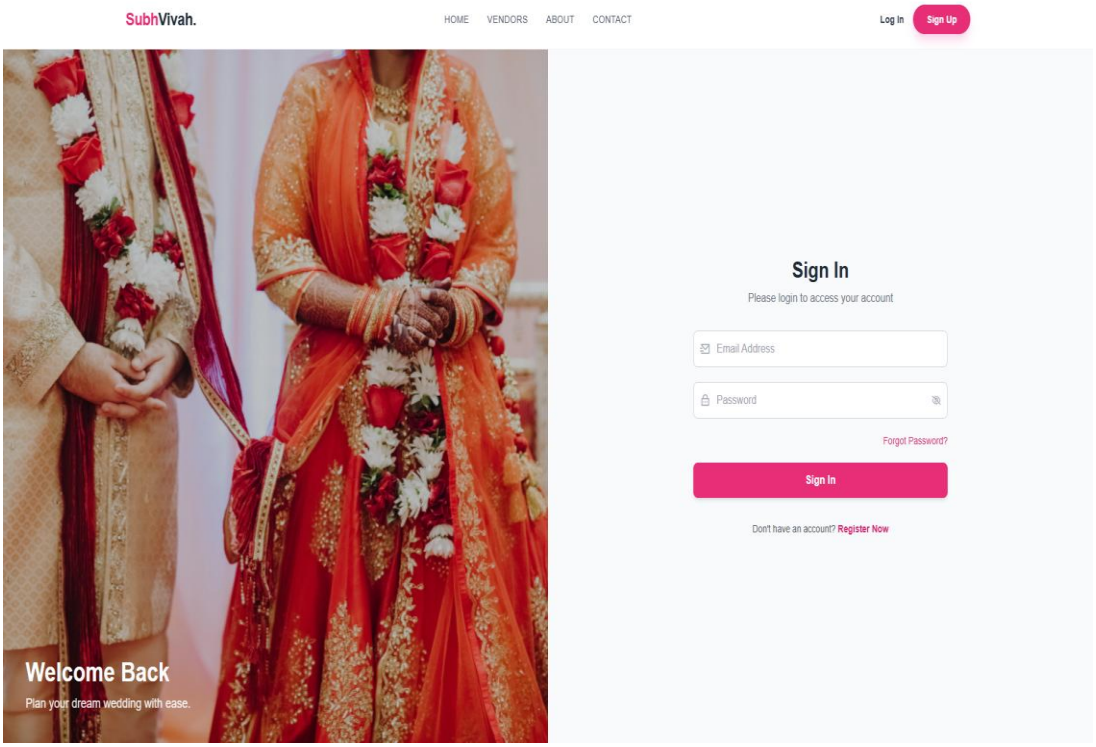
Code:

```
import * as yup from 'yup';
import { Form, Formik } from 'formik';
import { AxiosClient } from '../../config/axiosClient';
import { toast } from 'react-toastify';

const RegisterPage = () => {
   // 1. Validation Schema
   const validationSchema = yup.object({
      name: yup.string().required("Name is required"),
      email: yup.string().email().required(),
```

```
    password: yup.string().min(8).required(),
  });

  // 2. API Submission Handler
  const onSubmitHandler = async (values, helper) => {
    try {
      await AxiosClient.post("/auth/register", values);
      toast.success("Registration Successful! Please Login.");
      navigate('/login');
    } catch (error) {
      toast.error("Registration failed");
    }
  };

  return (
    <div className="flex h-screen">
      {/* Split Layout: Image & Form */}
      <div className="w-1/2 bg-gray-50 flex items-center justify-center">
        <Formik
          initialValues={{ name: '', email: '', password: '' }}
          validationSchema={validationSchema}
          onSubmit={onSubmitHandler}
        >
          {({ isSubmitting }) => (
            <Form className="w-full max-w-md space-y-5">
              <h2>Create Account</h2>

              <Field name="name" placeholder="Full Name" className="input-
field" />
              <ErrorMessage name="name" component="p" className="text-red-
500" />

              <Field name="email" type="email" placeholder="Email"
className="input-field" />
              <ErrorMessage name="email" component="p" className="text-red-
500" />

              <Field name="password" type="password" placeholder="Password"
className="input-field" />
              <ErrorMessage name="password" component="p" className="text-
red-500" />
```

```
                    <button type="submit" disabled={isSubmitting} className="btn-
primary">
                        {isSubmitting ? "Creating..." : "Register"}
                    </button>
                </Form>
            )}
        </Formik>
    </div>
  </div>
 );
}

export default RegisterPage;
```

6. Login Page

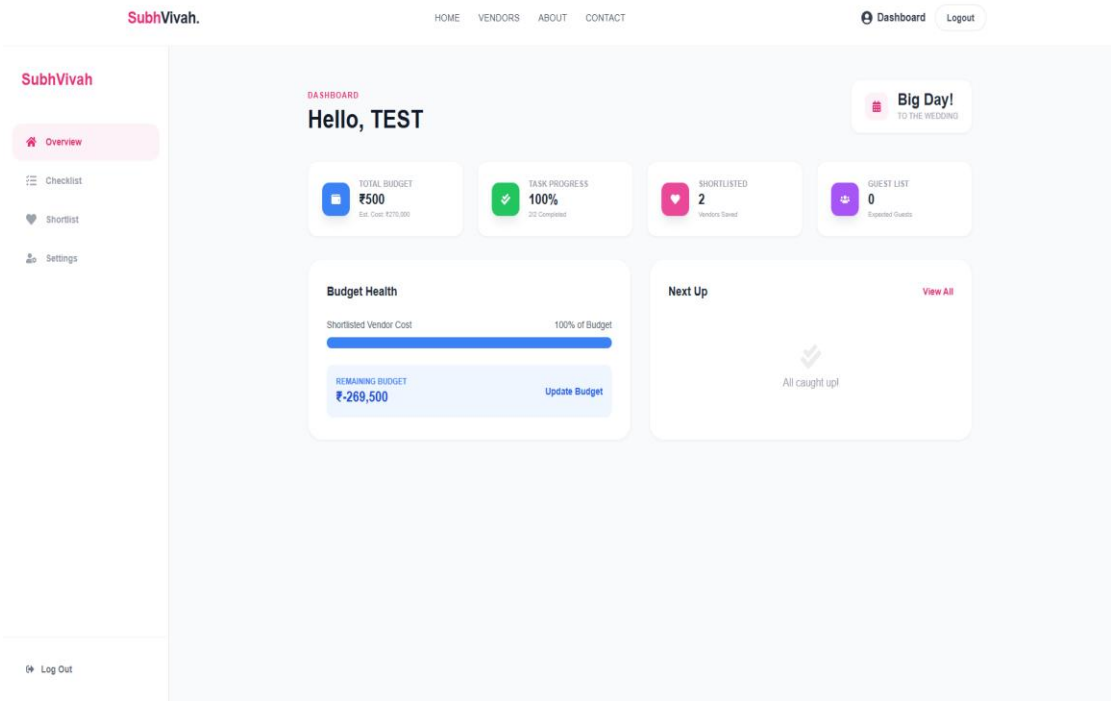


Code:

```
import { useDispatch } from 'react-redux';
import { setUser } from '../../store/slice/User.slice';
import { AxiosClient } from '../../config/axiosClient';

const LoginPage = () => {
   const dispatch = useDispatch();
   const navigate = useNavigate();

   const onSubmitHandler = async (values, helper) => {
      try {
         // 1. API Call
         const response = await AxiosClient.post("/auth/login", values);
         const { user, token } = response.data;

         if (user && token) {
            // 2. Client-Side Persistence
            localStorage.setItem("token", token);
            localStorage.setItem("user", JSON.stringify(user));

            // 3. Update Redux State (Global Auth)
```

```
            dispatch(setUser(user));

            toast.success("Welcome back!");
            navigate('/dashboard');
          }
        } catch (error) {
          toast.error("Invalid Email or Password");
        }
    };

    return (
      <div className="flex h-screen">
        <div className="w-1/2 flex items-center justify-center">
          <Formik
            initialValues={{ email: '', password: '' }}
            validationSchema={/* ... yup schema ... */}
            onSubmit={onSubmitHandler}
          >
            <Form className="w-full max-w-md space-y-5">
              <h2>Sign In</h2>

              <Field name="email" type="email" placeholder="Email Address"
classNamе="input-field" />
              <Field name="password" type="password"
placeholder="Password" className="input-field" />

              <Link to="/forget">Forgot Password?</Link>

              <button type="submit" className="btn-primary">Sign
In</button>
            </Form>
          </Formik>
        </div>
      </div>
    );
}

export default LoginPage;
```

7. User OverviewPage

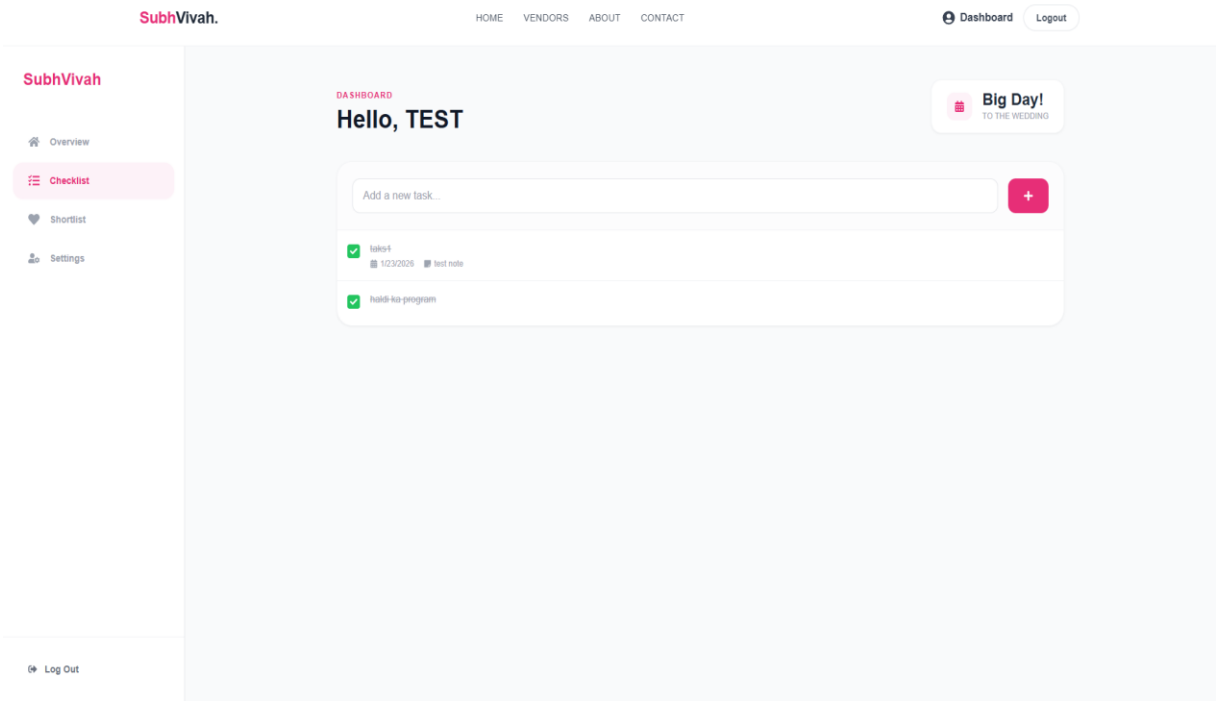


Code:

```
import React from 'react';
import { motion } from 'framer-motion';
// ... icons
// 1. Strict Typing for User Data
interface UserData {
    budget: number;
    checklist: { isCompleted: boolean }[];
    favorites: { price: number }[];
    guestCount: number;
}
const OverviewTab = ({ user, onTabChange }: { user: UserData, onTabChange: any })
=> {
    // 2. Logic: Calculate Statistics
    const totalTasks = user.checklist?.length || 0;
    const completedTasks = user.checklist?.filter(t => t.isCompleted).length || 0;
    const taskProgress = totalTasks === 0 ? 0 : Math.round((completedTasks / totalTasks)
* 100);
    // 3. Logic: Calculate Budget Health
    const estimatedCost = user.favorites?.reduce((sum, v) => sum + (v.price || 0), 0) || 0;
    const budgetHealth = Math.min(Math.round((estimatedCost / (user.budget || 1)) *
100), 100);
    return (
```

```
      <div className="space-y-8">
        {/* Top Stats Grid */}
        <div className="grid grid-cols-4 gap-6">
          <StatCard label="Total Budget" value={`₹${user.budget}`} sub={`Est: ₹${estimatedCost}`} />
          <StatCard label="Progress" value={`${taskProgress}%`} sub={`${completedTasks}/${totalTasks} Done`} />
          <StatCard label="Shortlisted" value={user.favorites?.length} sub="Vendors" />
          <StatCard label="Guests" value={user.guestCount} sub="Expected" />
        </div>
        <div className="grid grid-cols-2 gap-8">
          {/* Budget Visualization */}
          <div className="card">
            <h3>Budget Health</h3>
            <div className="progress-bar">
              <motion.div
                initial={{ width: 0 }}
                animate={{ width: `${budgetHealth}%` }}
                className={`h-full ${budgetHealth > 100 ? 'bg-red-500' : 'bg-blue-500'}`}
              />
            </div>
            <p>Remaining: ₹{user.budget - estimatedCost}</p>
          </div>
          {/* Priority Tasks List */}
          <div className="card">
            <h3>Next Up</h3>
            {user.checklist?.filter(t => !t.isCompleted).slice(0, 3).map(task => (
              <div key={task._id} className="task-item">
                <p>{task.text}</p>
              </div>
            ))}
          </div>
        </div>
      </div>
    );
};

export default OverviewTab;
```

8. Checklist Page

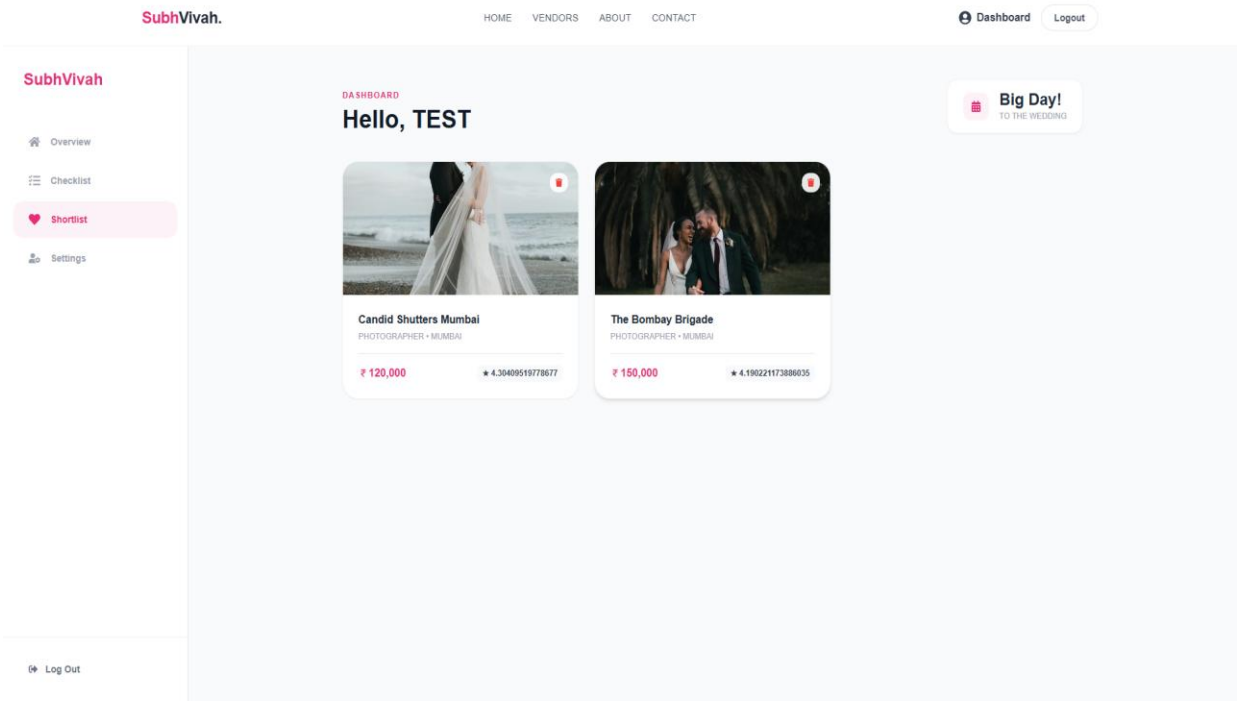


Code:

```
import { useState } from 'react';
import { AnimatePresence } from 'framer-motion';

const ChecklistTab = ({ checklist, onUpdate }: Props) => {
   const [newTask, setNewTask] = useState("");
   const [newDate, setNewDate] = useState("");

   // 1. Add Task Handler
   const handleAdd = () => {
      if (!newTask.trim()) return;
      onUpdate('add', newTask, undefined, { dueDate: newDate });
      setNewTask("");
   };

   return (
      <div className="bg-white rounded-3xl shadow-sm">
         {/* Input Area */}
         <div className="p-6 border-b">
            <input
               value={newTask}
               onChange={(e) => setNewTask(e.target.value)}
```

```
                placeholder="Add a new task..."
              />
              <button onClick={handleAdd}>Add</button>
            </div>

            {/* Task List */}
            <div className="divide-y">
              {checklist.map((task) => (
                <div key={task._id} className="flex justify-between p-4">
                  <div onClick={() => onUpdate('toggle', '', task._id)}
className="cursor-pointer">
                    <span className={task.isCompleted ? 'line-through text-gray-
400' : ''}>
                      {task.text}
                    </span>
                  </div>
                  <button onClick={() => onUpdate('delete', '', task._id)}>
                    <FaTrash />
                  </button>
                </div>
              ))}
            </div>
          </div>
    );
};

export default ChecklistTab;
```

9. Shortlist Page

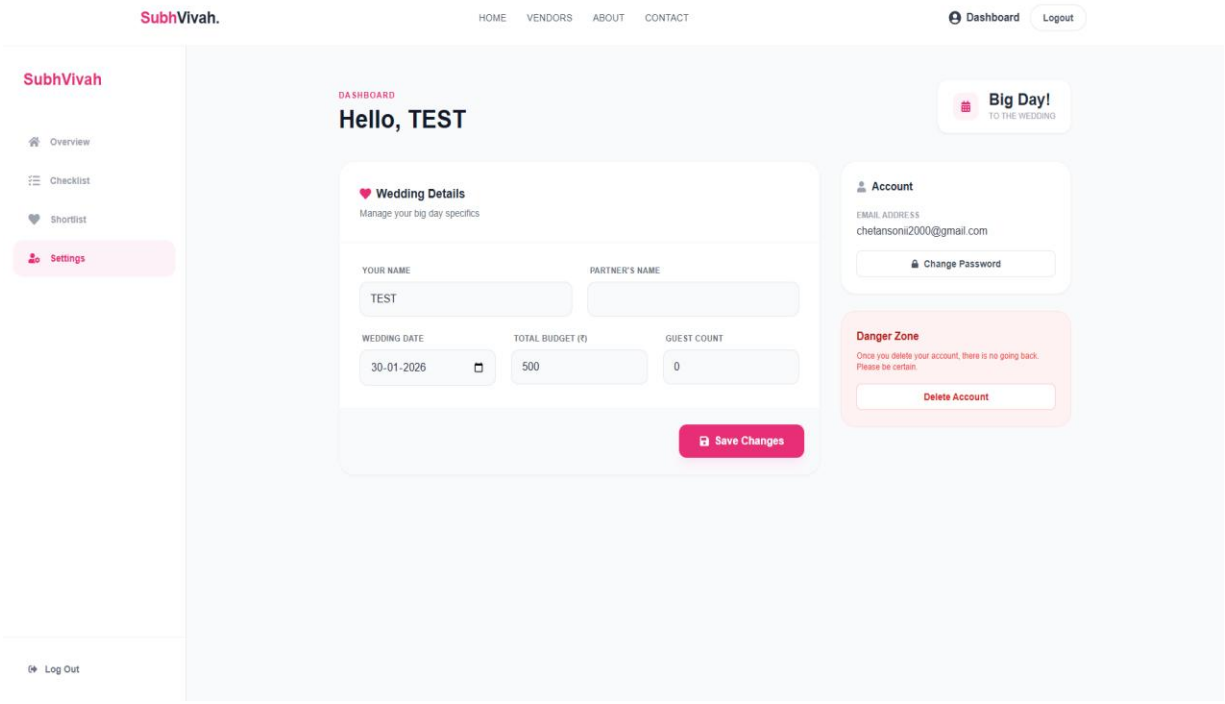


Code:

```
import { useState } from 'react';
import { AnimatePresence, motion } from 'framer-motion';

const VendorsTab = ({ vendors, onRemove }: Props) => {
   const [selectedVendor, setSelectedVendor] = useState<Vendor | null>(null);

   return (
      <>
         {/* Vendor Grid */}
         <div className="grid grid-cols-3 gap-6">
            {vendors.map(vendor => (
               <div key={vendor._id} onClick={() => setSelectedVendor(vendor)}
className="card relative">
                  <img src={vendor.image} alt={vendor.name} className="h-48 w-full
object-cover" />

                  {/* Remove Button */}
                  <button
                     onClick={(e) => { e.stopPropagation(); onRemove(vendor._id); }}
                     className="absolute top-4 right-4 text-red-500"
                  >
                     <FaTrash />
                  </button>
```

```
                <div className="p-4">
                  <h4>{vendor.name}</h4>
                  <p>{vendor.city} • Rs. {vendor.price}</p>
                </div>
              </div>
            ))}
          </div>

          {/* Detailed Modal */}
          <AnimatePresence>
            {selectedVendor && (
              <div className="modal-overlay">
                <motion.div className="modal-content">
                  <button onClick={() => setSelectedVendor(null)}>Close</button>
                  <h2>{selectedVendor.name}</h2>
                  <p>{selectedVendor.description}</p>
                  <div className="actions">
                    <button>Call</button>
                    <button>Email</button>
                  </div>
                </motion.div>
              </div>
            )}
          </AnimatePresence>
        </>
    );
};

export default VendorsTab;
```

10.Settings Page



Code:

```
import React, { useState } from 'react';

const SettingsTab = ({ user, onUpdateProfile }: Props) => {
   // 1. Initialize State with Prop Data
   const [formData, setFormData] = useState({
      name: user.name || "",
      partnerName: user.partnerName || "",
      weddingDate: user.weddingDate ? user.weddingDate.split('T')[0] : "",
      budget: user.budget || 0,
      guestCount: user.guestCount || 0
   });

   const handleChange = (e) => {
      setFormData({ ...formData, [e.target.name]: e.target.value });
   };

   const handleSubmit = (e) => {
      e.preventDefault();
      onUpdateProfile(formData);
   };

   return (
      <div className="grid grid-cols-3 gap-8">
```

```
        {/* Main Form */}
        <form onSubmit={handleSubmit} className="col-span-2 card p-8">
          <h3>Wedding Details</h3>

          <div className="grid grid-cols-2 gap-6">
            <InputField name="name" value={formData.name}
onChange={handleChange} label="Name" />
            <InputField name="partnerName" value={formData.partnerName}
onChange={handleChange} label="Partner" />
          </div>

          <InputField name="weddingDate" type="date"
value={formData.weddingDate} onChange={handleChange} />
          <InputField name="budget" type="number" value={formData.budget}
onChange={handleChange} />

          <button type="submit" className="btn-primary">Save Changes</button>
        </form>

        {/* Account Actions */}
        <div className="space-y-6">
          <div className="card p-6">
            <h3>Account</h3>
            <p>Email: {user.email}</p>
            <button>Change Password</button>
          </div>
          <div className="card p-6 border-red-200">
            <h3 className="text-red-600">Danger Zone</h3>
            <button className="text-red-600">Delete Account</button>
          </div>
        </div>
      </div>
    );
};

export default SettingsTab;
```

LIMITATIONS

While **SubhVivah** successfully achieves its primary objectives of fostering a transparent and efficient wedding planning environment, there are certain limitations in the current version of the system:

- **Lack of Multi-Language Support:** The platform currently operates solely in English. Given the diverse linguistic background of families in India, this limits accessibility for parents or relatives who might prefer navigating vendor profiles in their native regional languages (e.g., Hindi, Gujarati, Tamil).
- **Absence of Real-Time Chat:** The system currently facilitates communication via inquiry forms and simulated email notifications. It does not yet offer a real-time private messaging (WebSocket) feature. Couples wishing to discuss pricing negotiations or availability instantly with vendors must rely on external tools like WhatsApp, which breaks the continuity of the platform experience.
- **Basic Vendor Recommendation Algorithm:** The current "Vendor Listing" feeds rely on explicit filtering logic (City, Category, Price). The platform lacks an intelligent, AI-driven recommendation engine that could suggest vendors based on a couple's specific aesthetic style, hidden budget constraints, or past shortlisting behavior.
- **Absence of Direct Booking & Payments:** While users can view pricing and send inquiries, the platform does not support direct financial transactions. Couples cannot pay booking deposits or manage vendor installments directly through the website, necessitating offline payment methods.

FUTURE SCOPE OF SUBHVIVAH

The current implementation of SubhVivah serves as a robust foundation. However, there is significant potential for expansion and enhancement in future iterations:

- **Real-Time Chat & Negotiation:** Integrating **Socket.io** to enable real-time features such as private messaging between couples and vendors. This would allow for instant availability checks and price negotiations without leaving the platform.
- **Mobile Application Development:** Developing a native mobile application using **React Native**. Since the backend is already built as a RESTful API (Node.js/Express), it can easily serve a mobile app, providing couples with push notifications for checklist reminders, vendor replies, and budget alerts directly on their smartphones.
- **AI-Powered Budget Assistant:** Integrating AI algorithms to offer a "Smart Budget Allocator." Users could input their total budget, and the system would automatically suggest how much to allocate to each category (Venue, Catering, Photography) based on current market rates in their selected city.
- **Virtual Venue Tours:** Enhancing the "Venue" category by integrating 360-degree virtual tours or video walkthroughs. This would allow couples to scout locations remotely before scheduling physical visits.
- **Verified Review System:** Implementing a robust "Verified Couple" review system where users can upload photos of their wedding to prove authenticity before reviewing a vendor. This would build higher trust compared to open text reviews.

CONCLUSION

The **SubhVivah** project successfully addresses the critical need for a centralized, structured, and visually appealing digital marketplace for the Indian wedding industry. By bridging the gap between disorganized local directories and modern digital planning, the platform provides a dedicated space where services can be discovered, compared, and organized efficiently.

The development process demonstrated the power and flexibility of the **MERN Stack** (MongoDB, Express.js, React.js, Node.js). The use of **React** and **Tailwind CSS** ensured a dynamic, single-page user interface that is modern and responsive across devices, while **Redux Toolkit** provided efficient state management for complex features like authentication and wishlist management. On the backend, **Node.js** and **MongoDB** proved to be a highly scalable combination, capable of handling diverse data types—from complex vendor profiles to user-specific checklists.

Key features such as **Smart Filtering** for vendor discovery, **Checklists** for task management, and the **Shortlisting** mechanism have created a personalized environment where couples can plan their big day with confidence. The implementation of secure authentication using **JWT** and **Bcrypt** ensures that user data remains safe and trustworthy.

In conclusion, SubhVivah is not just a directory; it is a scalable, modern solution for event management. With its solid architectural foundation and clear path for future upgrades, it is poised to become an indispensable tool for couples seeking a stress-free journey to their perfect wedding.

BIBLIOGRAPHY

The following resources, documentation, and literature were referred to during the design, development, and implementation of this project:

Books:

1. *Pro MERN Stack: Full Stack Web App Development with Mongo, Express, React, and Node* by Vasan Subramanian.
2. *JavaScript: The Good Parts* by Douglas Crockford.
3. *Software Engineering: A Practitioner's Approach* by Roger S. Pressman.

Official Documentation & Websites:

4. **React.js Documentation** - https://react.dev

5. **Node.js Documentation** - https://nodejs.org/en/docs

6. **MongoDB Manual** - https://www.mongodb.com/docs

7. **Redux Toolkit** - https://redux-toolkit.js.org

8. **Tailwind CSS Documentation** - https://tailwindcss.com/docs

9. **MDN Web Docs (Mozilla)** - https://developer.mozilla.org

Tools & Libraries Used:

10. **Cloudinary** (Image Management) - https://cloudinary.com

11. **JWT** (JSON Web Tokens) - https://jwt.io

12. **Axios** (HTTP Client) - https://axios-http.com

13. **React Icons** - https://react-icons.github.io/react-icons

14. **Framer Motion** (Animations) - https://www.framer.com/motion/

15. **React Toastify** (Notifications) - https://fkhadra.github.io/react-toastify/