Frontend Architecture Assumptions

- **Framework:** A modern single-page application (SPA) framework like React.js will be used, as mentioned in the tech stack[1].
- **Authentication:** All API requests that require authentication will include a secure token (e.g., JWT) in the request headers[2]. This token is obtained after the user logs in.
- **State Management:** A global state management solution (like Redux, Zustand, or React Context) will be used to hold user information, including authentication status and credit balance, making it accessible across all pages.
- **API Client:** A centralized API client (like Axios) will be configured to automatically attach the auth token to headers and handle standard responses and errors (e.g., 401 Unauthorized, 402 Payment Required).

---

1. Recommended Tenders Page

This page serves as the user's dashboard, displaying a list of tenders that are relevant to their profile.

Objective: To proactively show users tenders they are most likely to bid on, based on their profile data.

Core Components:

- A search bar and filter controls (e.g., by country, state).
- A list of "Tender Cards," with each card showing summary information (title, authority, closing date).
- Pagination controls.

API Interactions & Process Flow:

1. **Initial Load & Profile Fetch:**
    - When the page loads, the frontend first needs to know what to recommend. It retrieves the current user's data from the global state or fetches it via an API call that returns the user's `profile_data`[3].
    - From the `profile_data`, key fields like `primary_industry`[4] and
    - `project_specialization`[5][5] are extracted to be used as keywords.
2. **Fetching Recommended Tenders:**

- The frontend makes a request to the **Tender Recommendation API**: `GET /api/v1/recommendations/tenders`.
- In header we need to pass JWT authentication Token of the user.
- `Authorization: Bearer <user_jwt_token>`
- While waiting for the response, a loading skeleton or spinner is displayed to the user.
- Can also pass query parameters:
  Example GET Request:
  https://api.yourdomain.com/api/v1/recommendations/tenders?country=IN&min_value=10000000&currency=INR&limit=15

3. **Rendering the List:**
   - Upon receiving a successful response, the `data` array containing the list of tenders[7] is stored in the component's state.
   - The frontend maps over this array to render the Tender Cards. Each card is a clickable element.
   - Pagination information from the response (`currentPage`, `totalPages`, etc.[8]) is used to render the pagination controls.
4. **User Interaction (Filtering & Navigation):**
   - If the user types in the search bar or applies filters, the component's state is updated, and a new call to `POST /v1/tenders/search` is made with the updated filter criteria.
   - When a user clicks on a Tender Card, the application navigates to the "Tender Details Page," passing the unique `tenderId`[9] from that tender's data as a URL parameter (e.g., `/tenders/tend_f4a8b1c9`).

---

2. Profile Page

This page allows the user to build and update their dynamic bidder profile by answering a series of questions.

Objective: To capture detailed, structured information about the bidder that can be used by the AI to generate high-quality proposals.

Core Components:

- A display of the user's currently saved `profile_data`[10].
- A dynamic form area to render the current question(s) to be answered.

- "Save and Continue" or "Submit" buttons.

API Interactions & Process Flow:

1. **Fetching the First Question(s):**
    - When the page loads, it calls the Dynamic Bidder Profile API: `GET /api/v1/profile/questions`[11].
    - The backend logic determines if the user is new (returns questions where `is_initial: true`[12]) or has already answered some questions (returns the next questions based on their previous answers[13]).
2. **Rendering the Dynamic Form:**
    - The `questions` array from the API response[14] is stored in the component's state.
    - The frontend maps over this array and dynamically renders the appropriate input field for each question based on its `input_type` (e.g., 'TEXT', 'SELECT', 'FILE_UPLOAD')[15]. Options for 'SELECT' fields are populated from the
    - `options` array[16].
3. **Submitting Answers:**
    - As the user fills out the form, their answers are collected in the component's state, structured as an array of objects, for example: `[{ "question_id": "bidder_type", "value": "COMPANY" }]`[17].
    - When the user clicks "Save and Continue," the frontend calls `POST /api/v1/profile/answers`[18], sending the answers array in the request body.
4. **Handling the Response and Displaying Next Questions:**
    - The response to a successful `POST` request includes the next set of questions in the `data.next_questions` array[19].
    - The frontend updates its state with these new questions, causing the form to re-render with the next part of the profile questionnaire.
    - If the `next_questions` array is empty, it signifies the end of a branch or the completion of the profile[20]. A message like "Profile Complete!" is displayed.

3. Tender Details Page

This page displays all available information for a single, selected tender.

Objective: To give the user a comprehensive view of a tender's requirements and details so they can decide whether to proceed.

Core Components:

- Detailed information fields (Reference Number, Title, Dates, etc.).
- Formatted sections for Procurement Summary and Eligibility Requirements.
- A primary call-to-action button: "Generate Proposal with AI".

API Interactions & Process Flow:

1. **Fetching Tender Data:**
    - The page loads and extracts the `tenderId` from the URL parameter.
    - It then makes a request to the GlobalTender API to fetch the specific tender: `GET /v1/tenders?tenderId=<ID>`[21].
    - A loading indicator is shown while the data is being fetched.
2. **Rendering Details:**
    - On success, the API returns an array with a single tender object. The frontend stores this object in its state.
    - The component then renders the various details from the object onto the page, pulling data from `tenderDetails`[22],
    - `eligibilityRequirements`[23], and other relevant fields.
3. **Navigating to Proposal Generation:**
    - When the user clicks the "Generate Proposal with AI" button, the application navigates to the "Create AI Proposal Page," passing the `tender_id`[24] in the URL so the next page knows which tender to work on.

4. Create AI Proposal Page

This page handles the AI generation process and displays the resulting proposal draft.

Objective: To provide the user with an automatically generated proposal draft by synthesizing their profile with the tender's details.

Core Components:

- A status indicator (e.g., "Generating your proposal, please wait...").
- A structured view of the generated proposal, showing each question and the AI's answer.
- A "Download Proposal" button.

API Interactions & Process Flow:

1. **Initiating AI Generation:**
    - As soon as the page loads, it retrieves the `tender_id` from the URL.
    - It immediately calls the AI Proposal Maker API: `POST /api/v1/proposals/generate`[25].
    - The request body contains only the `tender_id`: `{ "tender_id": "tend_f4a8b1c9" }`[26].
    - The component enters a "loading" or "generating" state, displaying the status indicator to the user. This is crucial as the Gemini API call might take some time.
2. **Displaying the Generated Proposal:**
    - When the API returns a successful 200 OK response, the frontend stores the entire JSON output in its state[27].
    - The "generating" indicator is hidden.
    - The frontend then maps over the `generated_proposal` array[28]. For each item, it displays the
    - `section`[29],
    - `question_text`[30], and the
    - `generated_answer`[31]. The answer's explanation and value are shown clearly[32].
3. **Preparing for Download:**
    - A "Download Proposal" button is prominently displayed. The click handler for this button will trigger the credit consumption and payment flow.

---

5. Download Button to Payment Flow

This is not a page but a critical user flow that integrates the proposal system with the payment and credit system.

Objective: To allow users to download their generated proposal while ensuring they have

sufficient credits, and guiding them through a purchase if they do not.

Core Components:

- The "Download Proposal" button.
- A modal dialog to inform the user of "Insufficient Credits."
- Integration with a payment gateway's frontend modal/widget.

API Interactions & Process Flow:

1. **Download Attempt:**
    - The user clicks the "Download Proposal" button on the Create AI Proposal Page.
    - The frontend makes a call to `POST /api/proposals/download`[33], including the
    - `proposalId` (or a similar identifier) in the body[34].
2. **Handling the API Response (Two Paths):**
    - **Path A: Success (Credits Available):**
        - The API returns a `200 OK` status[35]. The response body is the file stream for the proposal document.
        - The browser automatically prompts the user to save the file.
        - The frontend's global state is updated to reflect the credit deduction (e.g., `user.credits = user.credits - 1`).
    - **Path B: Failure (Insufficient Credits):**
        - The API returns a `402 Payment Required` error with a JSON message[36].
        - The frontend's API client catches this specific status code.
        - It prevents the download and instead triggers a modal to open. The modal displays the message: "Insufficient credits. Please purchase a plan to download proposals."[37] and shows the available subscription plans (e.g., "base", "enterprise").
3. **Purchase Flow:**
    - The user selects a plan (e.g., "enterprise") in the modal and clicks "Purchase."
    - The frontend calls `POST /api/payments/create-order`[38], sending the chosen plan type in the request body[39].
    - The API responds with order details, including a

`gatewayOrderId`[40].

- The frontend uses this `gatewayOrderId` to initialize and open the payment gateway's checkout widget (e.g., Razorpay or Stripe's modal).

4. **Post-Payment and Finalizing Download:**
   - The user completes the payment within the gateway's widget.
   - Upon successful payment, the backend webhook (`/api/payments/verify`[41]) is called by the payment gateway. This updates the user's credits in the database[42].
   - The frontend needs to know about this update. A common pattern is for the payment widget's success callback to trigger a function in the frontend. This function will:
     - Close the "Insufficient Credits" modal.
     - Re-fetch the user's profile to get the updated credit count and store it in the global state.
     - **Automatically re-trigger the original `POST /api/proposals/download` request.**
   - This second attempt will now succeed (Path A), and the user will get the download prompt, completing the seamless flow.