Survey App Architecture Plan/Feedback

Below are the Procurated Engineering team's thoughts on what a MVP, possibly Replit-built or otherwise vibecoded Survey app might look like/need to be able to do.

Functionality

Build a **standalone survey application** with two main areas:

1. Authenticated admin area
    - Create and manage surveys
    - Define criteria for which survey is sent
    - View stats (surveys sent, surveys completed, etc)
2. Unauthenticated survey completion area
    - Recipients complete a survey with no need to authenticate.

API & Survey Delivery Flow

- The application must expose an **API endpoint** that our main application can call when a review is published.
- The payload sent to this endpoint will include:
    - Review ID
    - Contract ID
    - Contract Vehicle ID
    - Reviewer Organization ID
    - Any other identifiers needed to determine survey eligibility
    - Any data needed to display review context in the survey UI
    - Any data needed to send the survey request email.
- Upon receiving this request, the survey application should:

a. Determine whether **any active survey currently applies** based on the provided criteria
b. If no survey applies
    - Respond with a JSON response indicating the review was not eligible, eg `survey_eligible: false, reason: `no_active_surveys`` or equivalent, reason codes are fine
        - Possible reasons, not inclusive: 'no_active_surveys', 'no_matching_criteria', 'already_processed', 'invalid_payload'
c. If a survey applies
    - Store the review information internally and generate an internal ID
    - Respond with a JSON response indicating the review was eligible, something like `survey_eligible: true, survey_review_id: <internalId>`
        - survey_review_id is for logging/debugging purposes only.
    - Generate a **hashed token** derived from that ID
    - Send a transactional email via Hubspot containing a survey link with the hashed token
    - *Note that sending the transactional email should likely be handled by a worker task.*

- IMPORTANT: This endpoint must be idempotent - repeated calls for the same review must not result in duplicate emails or duplicate stored records.

Survey Completion Flow

- When the recipient clicks the survey link
    - The application looks up the stored review using the hashed token
    - Based on the stored review details, determines which survey/questions to display
    - Renders the survey completion UI
- When a survey is completed
    - The completed survey data must be stored in the survey application database.

Architecture & Technology Constraints

Backend

- Ruby on Rails (preferably Rails 7)
- Deployable on Heroku
- Sidekiq + redis for background job processing
- All API endpoints should be versioned (IE `api/v1/...`

Frontend

- Vite + React + Typescript
- Sass (.scss) for styling, compiled via Vite
- Do NOT use Webpacker, importmaps, or Rails Asset Pipeline for new Javascript or CSS

Frontend Structure

- The app should have at least two Javascript entrypoints:
  - One for the authenticated admin area
  - One for the unauthenticated survey completion area
- React components should **not reference rails concepts directly** (helpers, controllers, ERB, etc)
- React components may receive **initial, read-only data** from the backend at page load (for example, review details or survey questions passed from the controller)
- Any **dynamic or subsequent backend interactions** must happen via HTTP calls.
- All HTTP calls must go through one shared API helper, e.g.: `app/frontend/lib/api.ts` .

Styling

- Avoid global CSS
- Prefer component-scoped styles
- Use per-component .scss files.

- Do not assume ownership of the entire page (IE avoid styling `body`, global resets, etc)

UI Organization

- Reusable UI elements/components should be separated from application-specific pages wherever possible
- Focus on keeping reusable components portable

Testing

Backend/Integration

- RSpec
- Cpaybara
- FactoryBot
- Timecop
- Webmock
- End-to-end feature specs that cover React-driven flows which modify backend state

Frontend

- Jest
- React Testing Library
- Jest test files should live under `spec/javascript/components` (rails-style test structure, not colocated with components)

Documentation

- The repository must include a `README.md` that explains:
  - How to run the app locally
  - How to run tests

    How to build for production

Additional Requirements

- Include a CAPTCHA in the survey completion flow
- Completed surveys must be persisted in the application.