**Subject:** A future-proof, cost-effective strategy for our form infrastructure post-Flowable Enterprise.

**Executive Summary**

This document outlines a strategy to address the upcoming non-renewal of our Flowable Enterprise license. Our current library of forms, defined in Flowable's proprietary JSON format, will become unusable without the licensed software, posing a significant risk to business processes that rely on them.

We propose a one-time migration of our existing form definitions to a modern, open-source standard. The recommended solution is to adopt the **formio.js open-source library**, which provides a complete ecosystem for building, rendering, and managing forms.

This approach will not only ensure **100% backward compatibility** for our existing forms but will also **eliminate licensing costs**, prevent future vendor lock-in, and provide a powerful, highly customizable platform that can be securely hosted within our own intranet. The estimated effort is a one-time development project to script the migration, with significant long-term benefits in cost, control, and flexibility.

**1. Problem Statement**

With the decision to not renew the Flowable Enterprise license, we face a critical issue:

* We possess a valuable asset: a large number of business-critical forms defined as Flowable Page Model JSON files.
* This JSON schema is proprietary and requires the licensed Flowable rendering engine to display, edit, or use.
* Upon license expiration, all existing forms will become inoperable, requiring a complete, manual rebuild of every form, leading to significant cost, time investment, and potential disruption to business operations.

**2. Core Objective**

Our goal is to find a solution that allows us to:

1. **Preserve our investment** by salvaging our existing form definitions.
2. **Ensure backward compatibility** so that forms can still be rendered and submitted.
3. **Establish a future-proof platform** for creating and editing forms.
4. **Eliminate recurring license costs** and avoid vendor lock-in.
5. **Maintain full control and security** by hosting the entire solution on our internal network (intranet).

**3. Analysis of Open-Source Form Solutions**

We analyzed several leading open-source solutions that provide a form schema, a visual editor (builder), and a rendering engine. The formio.js library emerged as the strongest candidate.

**Comparison of Leading Open-Source Form Ecosystems:**

|  |  |  |  |
| --- | --- | --- | --- |
| Feature | form.io (formio.js) | SurveyJS | JSON Forms |
| **Schema Type** | Proprietary (but open & well-documented) | Proprietary (but open) | **JSON Schema (Open Standard)** |
| **Visual Editor** | **Excellent & Free (MIT License)** | Excellent, but **Paid Commercial License Required** | No mature, official editor |
| **Renderer License** | Free (MIT License) | Free (MIT License) | Free (Apache 2.0 License) |
| **Key Strength** | **Complete, free ecosystem** (builder, renderer, complex components, logic). A direct replacement for enterprise systems. | Highly polished UI/UX and powerful conditional logic. | Strict adherence to open standards, clean separation of data and UI. |
| **Best For** | Organizations seeking a powerful, all-in-one, no-cost form solution with full ownership. | Projects where a superior UI is critical and the editor license fee is acceptable. | Programmatic form generation where a visual builder is not a primary requirement. |
| **Recommendation** | **Highly Recommended** | **Not Recommended** (due to editor cost) | **Not Recommended** (due to lack of editor) |

**4. Proposed Solution: formio.js**

We recommend adopting the free, MIT-licensed formio.js library. It is crucial to distinguish this from the *paid Form.io commercial platform*. We will only use the free library and build our own backend.

**How the formio.js Free Version Helps Us:**

* **Form Builder:** It provides a powerful, visual, drag-and-drop editor that we can embed in our own internal web application.
* **Form Renderer:** It can take a form JSON and render a fully interactive and functional HTML form.
* **JSON Schema:** It uses a well-defined JSON format that can support all our existing form complexities, including layouts, validation, and conditional logic.

This means we get all the necessary front-end tools completely free of charge. Our only cost is the internal development effort to host it and build the necessary backend APIs.

**5. Implementation and Integration Strategy**

**5.1. The Migration Path: Converting Flowable JSON**

The migration from Flowable JSON to formio.js JSON is a medium-complexity task achievable via a one-time script.

* **Process:** We will develop a script (e.g., in Node.js or Python) that reads each Flowable JSON file, transforms it according to a defined mapping, and saves the new formio.js JSON file.
* **Tools:** Open-source tools like jq, JSONata, or native language features can be used to perform the mapping.
* **Expression Buttons:** Complex logic like Flowable's "outcome" buttons can be cleanly migrated. The standard pattern in formio.js is to use a hidden field to store the outcome ('approved', 'rejected') and have buttons that set the value of this field before a final "Submit" button is clicked.

**5.2. Secure Intranet Deployment**

We can host the formio.js builder and renderer securely within our intranet, ensuring no data is ever exposed to the public internet.

* **Architecture:** The solution consists of two parts:
  1. **Frontend App (on ECS):** A lightweight containerized web application that serves the Form Builder UI. This will be placed in a **private VPC subnet** and accessed via an **internal Application Load Balancer (ALB)**.
  2. **Backend API & DB (Our Control):** Our internal team will manage the API and database to save/load form definitions and submissions. These will also reside in the private VPC.
* **Result:** The entire system is firewalled from the internet. Users access the form builder via an internal URL, and all data communication happens exclusively within our secure network.

**5.3. Integration with Corporate Design System (Custom CSS)**

The solution provides full support for our organization's mandated design system and styling.

* **Requirement:** All form controls (text fields, buttons, etc.) must match our corporate branding.
* **Solution:** formio.js offers multiple ways to apply custom CSS. The recommended and most robust method is using the **customClass property**. We can assign our design system's classes (e.g., .ds-input, .ds-button-primary) directly to form components in the JSON definition, ensuring perfect and maintainable style alignment.

**5.4. Advanced Extensibility (Custom React Components)**

For unique requirements not covered by standard controls, we can integrate our own custom React components.

* **Requirement:** Ability to use specialized, internally-developed React components (e.g., an interactive chart, a custom address lookup) within a form.
* **Solution:** While an advanced feature, this is fully supported. It involves creating a JavaScript "bridge" that allows formio.js to render a placeholder element and then uses ReactDOM to mount our custom React component into that element. This gives us unlimited flexibility to extend the platform to meet any future business need.

Excellent. Here is the next section of the presentation: a detailed, step-wise action plan that breaks down the project into manageable phases and concrete tasks for the development team.

**6. Detailed Action Plan & Implementation Phases**

This section outlines the four key phases for executing the migration and deploying the new form infrastructure. Each phase has clear objectives, tasks, and deliverables.

**Phase 1: Migration of Existing Form Definitions**

**Objective:** To convert all existing Flowable JSON form definitions into the formio.js schema format with 100% fidelity. This is a one-time, backend-focused task.

**Action Steps:**

1. **Inventory and Categorize:**
   * **Task:** Gather all Flowable form JSON files from our repository.
   * **Task:** Analyze and categorize them by complexity:
     + **Simple:** Basic input fields and layouts.
     + **Medium:** Forms with conditional visibility logic and dropdowns from data sources.
     + **Complex:** Forms with expression buttons, rich text, or other proprietary enterprise components.
   * **Deliverable:** A spreadsheet cataloging all forms and their complexity rating.
2. **Define the Mapping Schema:**
   * **Task:** Create a detailed mapping document that specifies the translation from Flowable component properties to formio.js component properties.
   * **Example:** Flowable.components[].type: "text" -> formio.components[].type: "textfield".
   * **Task:** Define the standard pattern for migrating complex logic, especially "Expression Buttons" (as outlined in Section 5.1).
   * **Deliverable:** A technical mapping document (e.g., on Confluence or in a Markdown file).
3. **Develop the Migration Script:**
   * **Task:** Choose a scripting language (recommendation: Node.js).
   * **Task:** Develop a script that:  
     a. Reads a source directory of Flowable JSON files.  
     b. Applies the mapping rules defined in the previous step.  
     c. Handles errors and logs any components that cannot be automatically migrated.  
     d. Writes the transformed formio.js JSON files to an output directory.
   * **Tooling:** Use a library like JSONata within the script for declarative mapping or write pure functions for more control.
   * **Deliverable:** A version-controlled migration script with unit tests.
4. **Execute and Validate:**
   * **Task:** Run the script on the entire inventory of forms.
   * **Task:** Manually review a sample of the transformed JSON files (simple, medium, and complex) to ensure accuracy.
   * **Task:** Address any forms that failed to migrate and manually adjust or update the script.
   * **Deliverable:** A complete set of formio.js-compatible form definition files.

**Phase 2: Integrating the Form Renderer**

**Objective:** To display the new formio.js forms within our existing applications, allowing users to view and submit data.

**Action Steps:**

1. **Setup Frontend Environment:**
   * **Task:** In the target application (e.g., our main React portal), install the formio.js renderer library: npm install --save @formio/react react-formio.
   * **Deliverable:** Updated package.json file.
2. **Create a Reusable Renderer Component:**
   * **Task:** Develop a generic React component (e.g., <FormioFormRenderer />) that accepts a formJson prop and a submissionUrl prop.
   * **Task:** This component will use the <Form /> component from the @formio/react library to render the form.
   * **Task:** On submission, it will POST the form data to the provided submissionUrl.
   * **Deliverable:** A reusable React component for displaying any form.
3. **Apply Corporate Styling:**
   * **Task:** Import our organization's main CSS file.
   * **Task:** Add specific CSS override rules or use the customClass property (as defined in Section 5.3) to ensure the rendered forms adhere to our design system.
   * **Deliverable:** Visually compliant forms integrated into the application.

**Phase 3: Deploying the Form Builder on ECS**

**Objective:** To provide our internal teams with a secure, web-based tool to create new forms or edit existing ones.

**Action Steps:**

1. **Develop the Builder Application:**
   * **Task:** Create a new, simple React application dedicated to form building.
   * **Task:** Use the <FormBuilder /> component from @formio/react as the core of this application.
   * **Task:** Add UI elements for loading existing forms (from our new backend) and saving new/updated form designs.
   * **Deliverable:** A standalone React application for form management.
2. **Containerize the Application:**
   * **Task:** Write a Dockerfile to package the builder application into a lightweight container (e.g., using Nginx to serve the static files).
   * **Task:** Push the resulting Docker image to our internal Amazon ECR (Elastic Container Registry).
   * **Deliverable:** A versioned Docker image in ECR.
3. **Configure ECS and Networking:**
   * **Task:** Define an ECS Task Definition for the builder application.
   * **Task:** Create an **internal Application Load Balancer (ALB)** within our private VPC.
   * **Task:** Set up an ECS Service to run the builder task in our private subnets, routing traffic from the internal ALB.
   * **Task:** Configure security groups to restrict access to the ALB from our corporate network/VPN only.
   * **Deliverable:** A running, secure, and internally accessible Form Builder application.

**Phase 4: Creating the Backend Service for Form Management**

**Objective:** To create the necessary API and database infrastructure to support the form renderer and builder.

**Action Steps:**

1. **Database Schema Design:**
   * **Task:** Design two simple database tables (e.g., in our existing PostgreSQL or a new instance):
     1. forms: To store the form definitions (e.g., id, name, schema\_json).
     2. submissions: To store the submitted data (e.g., id, form\_id, submission\_data\_json).
   * **Deliverable:** SQL schema definition script.
2. **API Development:**
   * **Task:** Develop a simple RESTful API (e.g., in Node.js/Express or Java/Spring) with the following endpoints:
     1. GET /api/forms/:id: Retrieve a form definition JSON.
     2. POST /api/forms: Save a new form definition JSON (used by the Builder).
     3. PUT /api/forms/:id: Update an existing form definition.
     4. GET /api/forms: List all available forms.
     5. POST /api/submissions/:formId: Accept a new form data submission (used by the Renderer).
   * **Deliverable:** A version-controlled and documented backend API application.
3. **Deployment and Integration:**
   * **Task:** Deploy the API service to our existing infrastructure (e.g., on ECS or EC2).
   * **Task:** Connect the Form Builder (Phase 3) and Form Renderer (Phase 2) to this new API.
   * **Task:** Conduct end-to-end testing: Create a form in the builder, save it, render it in the application, and submit data.
   * **Deliverable:** A fully functional, end-to-end form management system.