Programmatic Plan and Architectural Specification

The JSON Indexing-Sequencer Controller (jisc.php)

1.0 System Characterization and Definition

1.1 Characterization

The .json ISC is characterized as an **Automation and Batch Processing Engine** for the codebase-composer.php environment. It functions as a higher-level controller that programmatically interacts with the Composer's data structures to perform complex, repetitive, or sequence-based tasks that would be inefficient to execute manually.

1.2 Definition

jisc.php is a monolithic PHP application with a multi-page graphical user interface that provides two primary services:

1. **Batch Definition Generation:** The automated creation of foundational Definition Libraries (e.g., character sets) from user-defined inputs.
2. **Batch Composition:** The programmatic assembly of new Compositions by executing a user-defined sequence of references to existing Definitions, with a corresponding animated visualization of the process.

1.3 Primary Purpose

The primary purpose of the .json ISC is to introduce **programmatic automation** to the compositional development workflow. This serves to:

- Drastically reduce the time required to establish foundational Definition Libraries.
- Enable the creation of complex or repetitive code structures through automated sequencing.
- Provide a visual, educational tool for understanding how low-level Definitions are assembled into high-level Compositions.

2.0 Fundamental Components

The jisc.php system will be constituted by three fundamental components, delivered via a multi-page GUI.

2.1 Component 1: The Alphabet Generator

This component addresses the need for user-defined batch definition generation.

- **GUI:** A simple, single-page interface with the following elements:
  - A dropdown to select the target codebase-composer.php Project.

    - A text input for the new Definition Library filename (e.g., my_alphabet.json).
    - A large textarea for the user to input the string of characters to be defined (e.g., abcdefghijklmnopqrstuvwxyz0123456789...).
    - A "Generate Library" button.
- **Backend Logic (batch_generate_definitions action):**
    1. Receives the project, library name, and character string.
    2. Iterates through each unique character in the input string.
    3. For each character, it constructs a fully compliant Definition object (e.g., {"id": "def_...", "name": "char_a_lowercase", "content": "a", "notes": "Auto-generated by JISC"}).
    4. It assembles these objects into an array.
    5. It saves this array as a new .json file in the /composer_data/[project]/definitions/ directory.
    6. It returns a success or error status.

2.2 Component 2: The Indexing-Sequencer Controller

This is the core component for batch composition.

- **GUI:** A control panel with the following elements:
    - Dropdowns to select the target Project and a new Composition name.
    - A dropdown to select the primary Definition Library to visualize on the grid.
    - A textarea for inputting the sequence. The sequence will be defined as a comma-separated list of Definition names (e.g., char_h, char_e, char_l, char_l, char_o).
    - A range slider to control the execution speed (e.g., from 50ms to 2000ms per step).
    - An "Execute Sequence" button.
- **Backend Logic (execute_sequence action):**
    1. Receives the project, new composition name, and the final array of Definition ids from the front-end.
    2. It creates a new .composition.json file.
    3. It invokes the get_composition and file-writing logic (which can be ported from codebase-composer.php) to generate the final .txt build artifact from the sequence of IDs.
    4. It returns a success or error status.

2.3 Component 3: The Animated Visualization Grid

This component provides the visual feedback for the sequence execution.

- **GUI:**
    - A grid of squares, dynamically generated based on the number of definitions

in the selected library.
    - Each square will be programmatically linked to a single Definition and will display the Definition's name or content.
- **Frontend Logic (JavaScript):**
    1. When a user selects a library, an API call fetches all definitions for that library.
    2. The grid is populated, creating one square for each definition and storing the definition's id and name in the square's data-* attributes.
    3. When the user clicks "Execute Sequence":
       a. The JavaScript parses the sequence from the textarea.
       b. It begins a loop that is throttled by the user-defined speed.
       c. In each step of the loop, it takes the next Definition name from the sequence.
       d. It finds the corresponding square on the grid (by its data-name attribute) and applies a "lit-up" CSS class.
       e. It finds the full Definition object (including its id) from its internal state and adds the id to a compositionDefIds array.
       f. After the delay, it removes the "lit-up" class from the square.
       g. After the loop completes, it sends the final compositionDefIds array to the backend execute_sequence action.

3.0 Compatibility and Data Structure Compliance

The .json ISC will operate as a fully compliant peer to the codebase-composer.php system.

- **Read Operations:** jisc.php will only read from the established composer_data directory structure. It will parse existing .json Definition Libraries without modification.
- **Write Operations (Alphabet Generator):** The generated Definition Library .json files will adhere strictly to the established schema: an array of objects, where each object contains id, name, content, and notes keys.
- **Write Operations (Sequencer):** The generated .composition.json files will also adhere to the standard, containing a single key definitions which holds an array of Definition ids. The corresponding .txt file will be a direct concatenation of the content from those definitions.

This ensures that any artifact created by jisc.php is immediately and fully usable within the codebase-composer.php environment, and vice-versa.

4.0 Programmatic and Architectural Plan Summary

The construction of jisc.php will be a monolithic PHP file, consistent with the

established architectural preference.

1. **Backend First:** The core PHP functions for the API actions (batch_generate_definitions, execute_sequence) will be implemented first. These will share file-handling and sanitization logic with codebase-composer.php.
2. **Multi-Page UI Scaffolding:** The HTML structure will be created with distinct div containers for each "page" (Alphabet Generator, Sequencer). JavaScript will be used to toggle the visibility of these containers.
3. **Alphabet Generator Implementation:** The UI and corresponding JavaScript logic for the Alphabet Generator page will be developed.
4. **Sequencer and Grid Implementation:** The UI and JavaScript logic for the Sequencer page, including the dynamic grid generation and animation loop, will be implemented last, as it is the most complex component.

This plan provides a comprehensive and robust foundation for the development of the .json ISC, ensuring it meets all specified requirements while integrating seamlessly into the existing enterprise architecture.