Frontiers in Computing and Intelligent Systems
ISSN: 2832-6024 | Vol. 16, No. 2, 2026

Research on Configuration-Driven Dynamic Form Generation

Qiankun Ji [1], Ying Ding [2], Ming Feng [1], Yushan Xu [1, *]

[1] School of Big Data and Artificial Intelligence, Anhui Xinhua University, Hefei, Anhui, 230088, China
[2] Hefei Ruilin Huyu Network Technology Co., Ltd., Hefei, Anhui 230041, China

*** Corresponding author:** Yushan Xu

Abstract: Traditional static forms have two clear problems. They are not flexible enough. And they cost a lot to keep. They do not work well for complex business situations. And they cannot fit different devices well. These problems make traditional static forms hard to satisfy the needs of complex applications. This paper gives a configuration-driven way to make and improve dynamic forms. This method has two core technical parts. First, we build a standard form metadata model with JSON Schema. This model fully separates business logic from the front-end interface. It cuts the strong link between the two parts in traditional development. Second, we design a weight-driven adaptive layout algorithm. This algorithm lets forms change their layouts on their own. Forms can fit screens of different sizes well. We did comparative tests between this new method and traditional hard-coded forms. The test results show two clear advantages of the new method. It makes form rendering much faster. And it cuts down the time users need to fill out the form.

Keywords: Dynamic Form; Weight-Driven Adaptive Layout; JSON Schema.

1. Introduction

Business forms need updates often. Work flows inside a business change over time. Input fields must change too. Traditional hard-coded forms have clear problems. They are not flexible enough. And they cost a lot to keep. A small structure change has a clear problem. Developers must edit the front-end code. Then they must put the whole system online again. This raises development costs. It slows down iteration speed. This paper gives a configuration-driven way to solve these problems. The core idea is simple. Forms are described with metadata. Form details are not put into the program code. The interface builds itself at runtime. This system has two core parts. First, we build a form metadata model with JSON Schema. This model fully separates the form description from the front-end code. Second, we design a responsive layout system. It places input fields based on field importance and screen size [1] [2] [3]. This work does not aim to create a new theory. It gives a practical solution for common web development [4].

2. Related Technological Foundations

This is a system which includes regular front-end software. We describe forms using JSON Schema. It specifies that data type, limits and validation rules to be applied on each field. This provides a standard form. It is parsed in the same manner by different systems [5] [6]. Front end is developed using React or Vue. They are used to make reusable UI elements [7]. Here every form field is an individual component. Picked and demonstrated as required. We use CSS Grid and Flexbox for the page layout. Fixed layouts are not used in the system. Determines the number of columns that will fit onto a screen. Distributes the form fields over these columns [8] [9].

3. Requirement Analysis

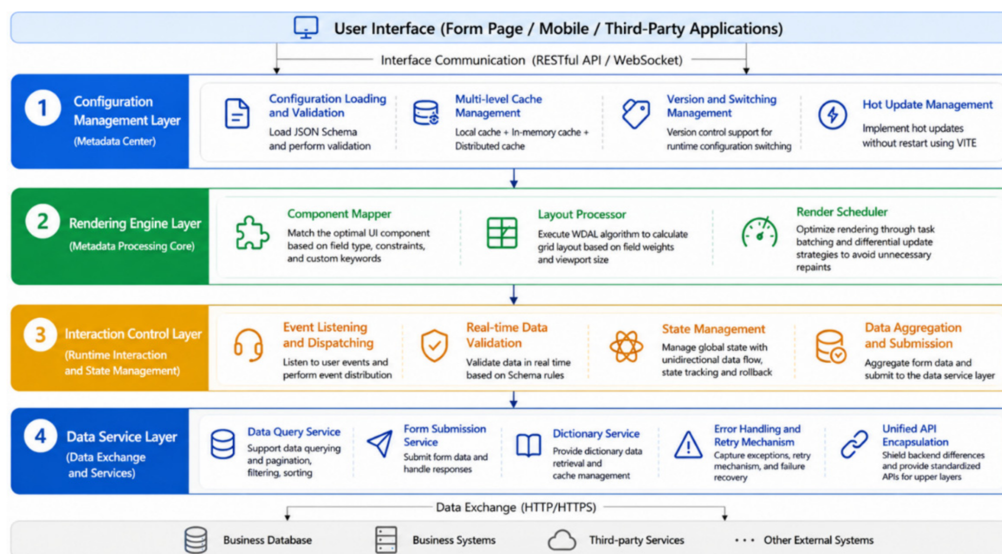

Fig 1. System Framework Diagram

The shape of structures varies greatly. They must work well in static code. It is for these situations that this system is for. In practice a form might require multiple versions. May have to be adapted for various business situations. And users use the system from a large number of devices. Thus, the configuration has to be flexible. In such instances the system should be capable of:

(1) It builds forms out of configuration data.

(2) It's consistent with the correct UI elements for every field definition. Supports simple rules for interaction

(3) It determines whether a field can be seen or not. Carries out checks for validity.

(4) Supports the typical CRUD functions: Create, Read, Update and Delete.

(5) Responsive–changes layout on the fly for different screen sizes.

3.1. Configuration Management Layer

The configuration management layer has clear tasks. It is the metadata center. It manages the whole life cycle of JSON Schema configuration files. It has four core tasks. It can loads form configurations, checks configuration structure, keeps version records, and switches between versions at runtime. Someone updates a form configuration. The system switches to the required version. It does not change how content is displayed. The system uses a multi-level caching method. This way, when the form starts up, the system does not have to parse the same data over and over, or read from the disk too many times. While the code is being developed, a tool named Vite is used. Its hot module replacement, or HMR for short, feature works with real-time configuration updates. What this means is that after you change a configuration, the change goes into effect immediately. And you do not need to restart the whole application.

3.2. Rendering Engine Layer

There are well-defined duties for the rendering engine layer. Converts parsed schema metadata to interface parts. They are executed during runtime. The main component of it is the component mapper. Selects appropriate user interface elements. It decides at run-time. This takes into account the type of fields, validation rules and additional keywords provided in the schema. This selection is done at run time. So it does not require to have the system set up with page templates beforehand. And it provides flexible form structures. Once the mapping has been completed, the control is handed over to the layout processor. Executes the WDAL algorithm. It is a grid that displays all form fields. It works out the location of each field. This calculation is based on two considerations. The weight of each field is one. The second is the size of the screen that is being used. This will allow the form layout to adjust to different screen sizes. And it maintains a well-balanced look of the shape. The rendering scheduler has special rendering methods to improve rendering speed. It does task batching and differential updates. It doesn't update the entire page for each change of state. It just updates the concerned components. This decreases the amount of re-rendering work. And it also cuts down unnecessary repaint work.

3.3. Interaction Control Layer

The interaction control layer controls the operation of the forms. It manages the interactions with the user during runtime of the program. Uses a one event system. It is able to capture all user actions. Users enter text into text boxes. They work in turn on the input components. Manual data checks are performed by them. Passes these events on to the relevant handlers. Validates data instantly as it is entered. It provides a complete status of the form. All parts of the form react alike. Has a one way data flow model. This model is used to control the shared state. An input field in the local area is changed. The update is gradual, step by step, to each piece that is connected. All data from the final form is gathered in one go, immediately prior to the user submitting.

3.4. Data Service Layer

The data service layer has two main jobs. First, it makes sure upper modules get the same data structure from any backend. It uses a single service interface for all backend responses. It wraps external APIs into standard data access methods. This removes the need for a specific backend. And it avoids problems from different service structures. Second, it supports data movement and key tasks for the form system. It creates forms. It responds to user actions. It saves data. It fetches data when needed. It submits form data back. It loads dictionary items. It processes error messages. And it retries failed requests automatically.

4. Key Function Implementation

4.1. Data Model Design

This data model adds 'ui:'-prefixed keywords to the standard JSON Schema. Convert both standard and extended keywords into the same internal intermediate form. The above is a layout calculation and component map. The model has kept backward compatibility. It is as follows. First, standard parsers do not recognize the extension keyword. They are not normal validations. Second, in areas without these extensions, the model will still function normally as standard JSON Schema. This Design will be cross-compatible. It does not affect the regular operation.

Table 1. Custom Keywords

Keyword	Type	Semantic Description	Example Value
ui:widget	string	Field UI component type	‘text’, ‘date’, ‘select’
ui:options	object	Passed configuration parameters	{‘placeholder’:’Please input’}
ui:priority	integer	Field weight (1-5)	5
ui:grid	object	Grid layout configuration	{‘span’:12,’offset’:0}
ui:step	integer	Field step number	1
ui:dependency	object	Inter-field dependencies	{‘fieldA’: {‘equals’:’ value1’}}

4.1.1. Metadata Parsing and Validation

The two steps for metadata validation in the system. First, Syntax Validation. General-purpose validation libraries are used. Verify whether the configuration file is a valid JSON Schema. The Second Stage is Semantic Validation. Syntax validation passed; the range of the keyword value is within bounds, the component type is valid, and field dependencies make sense.

4.2. Design and Implementation of the Weight-Driven Adaptive Layout (WDAL) Algorithm

The theoretical architecture of the WDAL algorithm combines three main knowledge systems. These are grid system theory, information architecture, and visual perception theory. It keeps a balance between aesthetic rules and user experience improvement. It also creates adaptive layout plans that improve visual consistency.

4.2.1. Formal Definition of the Algorithm

Definition 1 (Dynamic Form Layout):

Given a field set $F=\{f_1, f_2, ..., f_n\}$, each field f_i has a business weight w_i in [1,5] and an estimated height h_i, alongside a viewport width V_w. The objective is to find a layout scheme L: $F \rightarrow N^2$,such that:

maximize$\sum(w_i \times visibility_score(f_i))$

Subject to:

(1)$\forall f_i, f_j \in F, L(f_i) \cap L(f_j) = \emptyset$(No-overlap constraint)

(2)$\sum h_i \leq V_h$(Height constraint, where V_h is the viewport height)

(3)Column_balance(L) ≤ threshold(Column balance constraint, where threshold is a preset column balance threshold)

Definition 2 (Grid System):

Let $G = (C, V_W)$ be a responsive grid system, where $C \in N^2$represents the number of equal-width columns, and $V_W \in R^2$represents the current viewport width (unit: pixels). The number of columns C is dynamically determined as a piecewise function of V_W, and its mapping relationship is defined as:

$$C(V_w) = \begin{cases} 1, if V_w < 768\ (mobiledevicebreakpoint) \\ 2, if 768 \leq V_w < 1200\ (tabletdevicebreakpoint) \\ 4, if V_w \geq 1200\ (desktopdevicebreakpoint) \end{cases}$$

Definition 3 (Layout Scheme):

A layout scheme L is a function that maps each field f_i to a position (p_i,r_i) in the grid system, where f_i represents the column index and r_i represents the row index.

$$L(f_i) = (p_i, r_i)$$

4.2.2. Detailed Implementation of the Algorithm

The WDAL algorithm implementation adopts a greedy strategy. Its core idea is to prioritize placing high-weight fields in visually prominent positions, and subsequently process lower-weight fields. The detailed steps are as follows:

Algorithm 1: Weight-Driven Adaptive Layout (WDAL) Algorithm

Input: Field set F, Viewport width V_w

Output: Layout scheme L

Steps:

(1) Initialization:

1)Calculate the number of grid columns C based on V_w (see Definition 2);

2)Initialize C empty columns columns[1...C] to store fields in each column;

3) Initialize the current column height array heights[1...C] (length C) with all initial values set to 0, used to record the current total height of each column;

4) Initialize the layout scheme L as an empty set to store the final layout results.

(2) Field Priority Sorting:

1) Sort the field set F in descending order according to the business weight wi to obtain an ordered queue Q;

2)For fields with identical weights, arrange them according to their field order in the original JSON Schema configuration (maintaining sorting stability) to ensure the consistency of layout results.

(3) Greedy Placement Strategy:

Iterate through each field f_i in the ordered queue Q:

1) Calculate the grid span s_i: based on the 'ui:grid.span' configuration value; if unconfigured, the default is 1; if it exceeds the [1,C] range, correct it to the corresponding extreme value;

2) Find the optimal column j: it must satisfy that the column and its rightward continuous s_i columns are unoccupied and do not exceed the grid range, while the current height heights[j] of this column is the minimum. If there are multiple shortest columns, select the one with the smallest index to ensure left alignment;

3)Place f_i and update data: add f_i to columns[j], synchronously update the height of the s_i columns it occupies to heights[j]+h_i, and record the layout information of f_i (p_i=j, r_i, s_i, h_i) into the layout scheme L.

(4) Return Results:

1)Calculate the total number of layout rows R= max(heights), and determine the row index r_i of each field based on its position in the column and the column height;

2) Organize the fields in each column and their layout information (p_i=j, r_i, s_i, h_i) to form a standardized layout scheme L and return it.

The algorithm consists of the main function 'WDAL' and the sub-function 'FindOptPos'.

Algorithm 1: WDAL

Input: Field set F, Viewport width V_w

Output: Layout scheme L

Initialization: Determine the value of C based on V_w, $heights[C] \leftarrow \{0\}, L \leftarrow \emptyset$.

Sorting: Stably sort F in descending order by w_i to obtain queue Q.

Iteration: for each f in Q do:

Get height h and span s of f (constrain s to [1, C]).

j*←FindOptPos(s,heights) // find the optimal starting column

top←max(heights[j*...j*+s-1]) // calculate foundation height

heights[j*..j*+s-1]←top+h // update column heights

L.add({f,j*,s,top}) // record layout

end for

Return: L

Algorithm 2: FindOptPos

Input: Span s, Column heights heights

Output: Optimal column j*

minH←∞,j*←1,C←len(heights)

For j from 1 to C-s+1 do:

currH←max(heights[j..j+s-1])

if currH<minH

then

minH←currH,j*←j

return j*

5. Experimental Verification

To evaluate the proposed method, experiments were conducted using three types of form complexity: simple, medium, and complex. The tests were carried out under the same hardware and software conditions. Three methods were compared: traditional static forms, existing dynamic form solutions, and the proposed approach. Each experiment was repeated multiple times, and the average values were recorded for analysis. The evaluation focused on rendering performance, interaction response, completion efficiency, and configuration adjustment cost.

5.1. Experimental Process

The experimental procedure is as follows:

(1) Construct simple, medium, and complex form samples, and configure the corresponding metadata for the proposed method;

(2) Deploy Method A, Method B, and Method C respectively, and measure the first screen rendering time and linkage response latency;

(3) Organize testers with similar experience levels to complete the filling tasks and record the completion time;

(4) Simulate requirement changes and calculate the configuration adaptation time for each method.

5.2. Experimental Results

The results show that all methods perform similarly in simple scenarios. However, differences become more obvious in medium and complex cases.

The proposed method shows better performance in terms of rendering time and adaptation cost. This is mainly because configuration-based rendering reduces the need for repeated code-level modifications.

Table 2. Rendering Time Comparison

Scenario	Method	First Screen Rendering Time / (ms)	Linkage Response Latency / (ms)	Form Completion Time / (s)	Adaptation Time / (m)
Simple	A	118	42	51.4	28
	B	126	47	49.6	12
	C	121	40	46.8	8
Medium	A	246	96	112.3	65
	B	221	88	103.5	24
	C	198	71	92.7	14
Complex	A	482	183	241.6	132
	B	401	156	218.4	47
	C	312	109	172.1	23

6. Conclusion

This study brings up a new method for making and improving dynamic forms, which puts together configuration-driven architecture and a weight-driven adaptive layout. It actually works well to fix key problems that old static forms have, like not having enough flexibility and being hard to adjust to different device screens when handling complicated business tasks. After we expanded the JSON Schema standard, we put together a full metadata model, and this gives basic support for the configuration-driven architecture idea. The WDAL algorithm that we worked out mixes together theories from several different fields really well, and builds a layout optimization mechanism that matches how people think more closely. The results we got from this research can give technical help to people who want to build complex dynamic forms more quickly, and also supports their actual use in real work.

Acknowledgments

This study was supported by the Youth General Program of Natural Science Research of Anhui Xinhua University [Grant Number 2025zrqyb01], the Anhui Provincial Natural Science Research Program: Research on Key Technologies of Network Intrusion Detection Based on High-reliability Cluster Analysis [Grant Number 2024AH050617], the Anhui Provincial Teaching Research Program: Research on Curriculum Ideological and Political Construction of Data Structure Course Based on OBE Philosophy under the Background of Emerging Engineering Education [Grant Number 2023jyxm0883], the 2025 Anhui Provincial College Students' Innovation and Entrepreneurship Program: Design of Short Link Generator Based on Spring Boot [Grant Number S202512216126].

References

[1] Chen, G. (2023). Design and implementation of dynamic form functional modules based on SpringBoot + Thymeleaf + MySQL. *Changjiang Information & Communication*, *36*(9), 100-102.

[2] Yang, E. M. (2023). *Development of dynamic form workflow engine based on Flowable* [Master's thesis]. China University of Geosciences (Beijing).

[3] Huang, X. P., Liu, Z. Y., & Yan, S. B. (2024). Design and implementation of strongly typed JSON data exchange interface generator. *Information Technology and Informatization*, (5), 80-84. https://doi.org/ 10.3969/j. issn. 1672-9528.2024.05.017.

[4] Yu, X., Li, Q. H., & Liu, X. H. (2026). Adaptive component layout algorithm for low-code platforms under changing business processes. *Computer Technology and Development*, *36*(3), 44-48.

[5] Wei, X. (2026). Research on JSON-configured front-end development platform based on Avue. *Computer Programming Skills & Maintenance*, (1), 10-13.

[6] Meng, L. D., Lin, N. B., & Wei, W. Q. (2025). Design of lightweight workflow engine based on JSON. *Popular Science & Technology*, *27*(2), 20-24.

[7] Yang, F. W., & Lin, S. R. (2025). Design of dynamic validation mechanism for Web forms based on jQuery framework. *Popular Standardization*, (21), 46-48.

[8] Li, M., Chen, X. N., & Li, K. (2024). Design and application practice of form interaction on low-code platforms. *Scientific and Technological Innovation and Application*, (34), 52-62.

[9] Gao, K. (2023). Research on rendering optimization of front-end framework based on MVVM pattern [Master's thesis]. Southwest University of Science and Technology. https://doi. org/ 10. 27415/ d. cnki. gxngc. 2023.000971.