Implementation and Application of GUI Model-Driven Low-Code Platform in Energy Industry

Jun Zhu, Xinyang Pan(✉), Zhihua Zhong, Wenbin Mao, and Likui He

Shanghai Shine Energy Information Technology Development Co., Ltd., Shanghai, China
panxy@shineenergy.com

Abstract. Nowadays, with the rapid development of science and technology, the business requirements of the internal digital platform of enterprises are becoming more and more diverse, so it needs to provide more and more functional points. As the complexity of functions increases, the operation and maintenance costs of the platform increase and the productivity of enterprises decreases. If an enterprise uses a low-code platform for development, it will be able to solve or alleviate these problems to a large extent, including reducing the development cost of the enterprise and improving productivity by lowering the technical threshold for development. Compared with traditional code development, the low-code platform supports rapid development and one-click deployment of applications, and combines model-driven and generative programming in design concepts.

Keywords: GUI development · LCDP · rapid development · automation · code generation

1 Introduction

As an important super-large energy enterprise in China, State Grid proposes to adhere to one industry as the mainstay in information construction and digital transformation, speed up the upgrading of power grid to energy Internet, and ensure to basically build an international leading energy Internet enterprise with China characteristics by 2025. It has become an important link to promote the digitalization of power grid production and strengthen the digital management and control of power grid planning, construction, dispatching, operation and maintenance. In terms of technical research, it has also increased the research efforts of new technologies such as power chips, artificial intelligence, blockchain and electric Beidou.

For digital transformation, enterprises often have two ways to transform: traditional code development and low-code platform development. For most enterprises, the traditional code development method will still be used to develop digital platforms, but there are some disadvantages in using this method. Although using low-code platform to develop can not provide a perfect solution, it can still solve many problems of existing methods to a great extent.

D. Kumar et al. (Eds.): IEIT 2023, AHSSEH 10, pp. 296–313, 2023.
https://doi.org/10.2991/978-94-6463-230-9_36

Low-code platform was first mentioned in Forrester Research in 2014 [1]. Up to now, it has attracted the attention of many large software companies and developed their own software applications based on it as a conceptual model, and the discussion on it has become more and more enthusiastic [2]. Although the rise of low-code platform is relatively late, it doesn't mean that the low-code platform originated only recently. In fact, we are no strangers to the "low-code" platform, which may have originated from 1990 to 2000. As early as 2000, there were related documents about Generative Programming [3], in which how to assemble reusable components for pipeline programming was expounded. Nowadays, the usage rate of low-code platform is gradually increasing around the world, but most audiences keep a wait-and-see attitude towards it, and users generally think that its ease of use still has room for improvement [4]. On the whole, the market of low code platform is on the rise [5].

This design will focus on the development method of low-code platform, which is a cloud service platform that supports the development and deployment of software applications with little or no code, and debugs its Graphical-User-Interface (GUI) and other graphical elements [6]. The core of low-code platform development lies in rapid application development, automatic application deployment and execution, and the use of model-driven design principles. Low-code platform lowers the user's use threshold with its low learning cost, thus improving the overall production efficiency of the platform, and also reducing the cost of personnel training for enterprises [7]. Its highly automated characteristics also ensure the stability of development and application, and enterprises will not need to spend too much money on the operation and deployment of the platform [8].

This paper will implement a low-code platform design for GUI and other graphic elements. The services it supports include GUI generation, development and automatic deployment, etc. The platform will follow the principle of WYSIWYG (What-You-See-Is-What-You-Get). By abstracting the information and data sources of the graphical interface into static data, and using the platform's parser to convert the data into the graphical interface, the platform can change the interface by editing the static data, and finally build a complete low-code platform for GUI.

2 Design the Overall Architecture

The project saves the interface data by transforming the front-end GUI interface into persistent static abstract data, and uses the corresponding parser to complete the reverse transformation from static abstract data to GUI interface in the actual rendering stage to generate the GUI interface of the corresponding application. The specific research contents are as follows:

(1) Design the visualization scheme of interface data, and realize the transformation of interface data into GUI interface.
(2) Design the visualization scheme of GUI component style data, realize the visual style configuration of a single component in the interface, and design the transformation specification between configuration data and style.
(3) Design the binding scheme between interface and data, realize the visual configuration of data displayed in the interface, and realize the layout and generation of visual interface combined with data model.

(4) Design the visualization scheme of jump logic, realize the automatic data generation of jump logic, and realize the data visualization of jump logic among all interfaces of the application platform.

The system framework of this paper is divided into six steps: user-defined data model, user-defined atomic service, user-defined composite service, visual interface layout, interface data binding and multi-interface integration, as shown in Fig. 1. Among them, custom atomic service and custom composite service are the platform support contents, which will not be explained in this paper.

In the user-defined data model stage, after the user changes the data model, the database will update the table structure of the data model synchronously, and if the data model is newly generated, the corresponding CRUD service will be generated for it. If the associated data model is set, the corresponding associated database in the database is updated. The data of user-defined data model will be able to be integrated into the interface in the form of data source in the interface data binding stage.

In the visual interface layout stage, users will visually layout the DOM structure and CSS data of the interface by dragging and dropping, which will be synchronous in the design stage and after the final launch, and will conform to the principle of "what you see is what you get".

In the data binding stage of the interface, users can choose to integrate dynamic data and static data into the interface. Dynamic data can be bound to data sources in the platform, including data model data sources, jump data sources and other data sources that exist in the running stage of the platform. The data model data source in the data source is obtained through the database, so the data in the platform can be integrated into the interface for use.

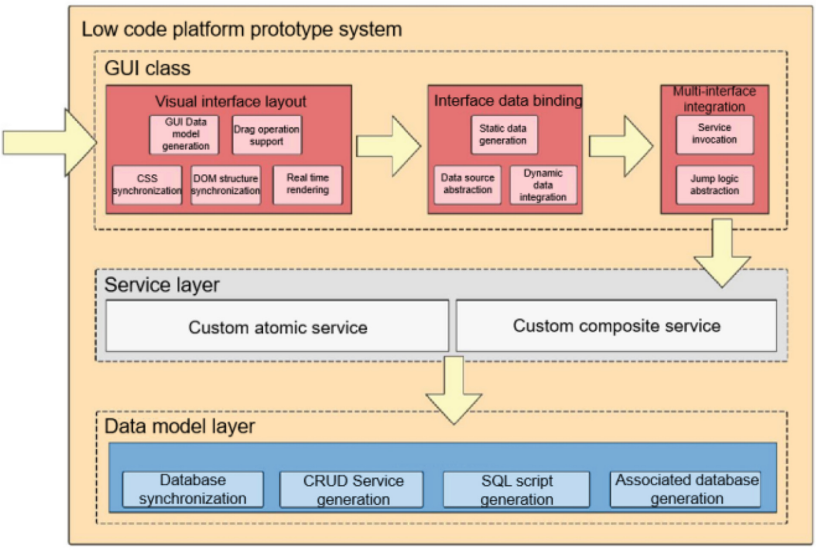

Fig. 1. GUI model-driven low-code platform system frame diagram

In the stage of multi-interface integration, users can choose to set the jump logic for interfaces, and realize the jump between interfaces in the actual operation stage by binding the data source carried during the jump.

Functional requirements: After consulting relevant literature, according to the business scenario of low-code platform design driven by GUI model, we analyzed the functional requirements of the platform, and obtained the use case diagram as shown in Fig. 2. We will explain the functional requirements of the platform in detail based on the use case diagram.

For the low-code platform, in addition to the basic services of adding, deleting and modifying data, it also needs to support users to use other types of services, so the platform should support user-defined generation services. After analysis, it is concluded that the platform's requirements for custom services include the development of atomic services and the development of generated atomic services. This part is the technical support of this design, which is not within the scope of this design, so it will not be explained in detail.

We then analyze the use cases of data service developers and GUI developers, and conclude that the main functional requirements of the platform involved in this paper are management data model, visual interface layout, interface data binding and multi-interface integration. Data service developers can manage the data models used in the platform, generate data models, and then manage the relationships between the models. GUI developers can visually lay out the interface by dragging and dropping components, and can selectively modify the style of the interface. Users can bind the data in the interface, which can be static or dynamic. Dynamic data supports the predefined data model data, atomic service data and composite service data in the display platform. Users can also choose whether to set jump logic for this interface.

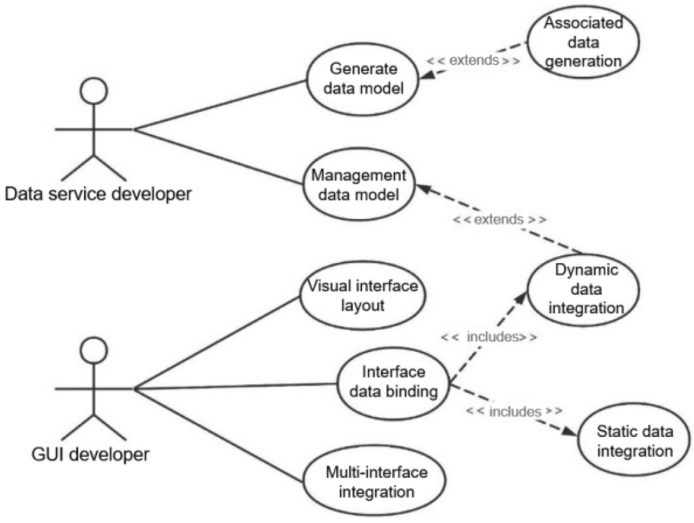

Fig. 2. Use Case Diagram of Data Service Developers and GUI Developers

3 Design Scheme

This chapter will describe the whole process of interface generation, including the concrete realization process of four steps: user-defined data model, visual interface layout, interface data binding and multi-interface integration. The service layer in the middle is the platform support content, so it will not be described:

(1) User-defined data model: users can define data models. After publishing the data models, the system database will generate corresponding data tables, and can also manage the relationship between data models.
(2) Visual interface layout: Layout the components in the interface by dragging and dropping, or freely set the style of the components. After the interface is published, the system will generate an interface with the corresponding style, which is consistent with the design interface and conforms to the principle of "what you see is what you get".
(3) Interface data binding: In the design stage, users can bind the data used by components. After publishing the interface, the system will automatically obtain the required data for all components.
(4) Multi-interface integration: In the design stage, users can manage the jump logic of components, including data transmission in the jump process. After the interface is published, the system will automatically transmit the corresponding data when the interface jumps, thus realizing the integration among multiple interfaces in the platform.

3.1 Custom Data Model

In order to manage the data model used in the low-code platform, we designed a user-defined data model for the platform, which allows users to generate data tables by defining the attributes and basic information of the data model. After publishing the data model, the system will generate the corresponding data table in the database.

1) Data structure

The data model in the platform is managed by application, and the data structure of the user-defined data model is shown in Fig. 3.

The DataInfo class will store all the relevant information of the data model, including its name, attributes, project, attribute association and publishing status. All data models stored in the database have their own projects, and the Project class will save all relevant information of the project, including the User id and user name of the creator, corresponding to the data in the User class. For the attribute information in DataInfo, we designed a Property class to store his attribute name, type and whether null is allowed or not. Attribute types support five common types: char, int, double, bool and string. The association information between data models is saved by Relation class, and the information to be saved includes association attribute names and association types, including one-to-one and many-to-many associations.

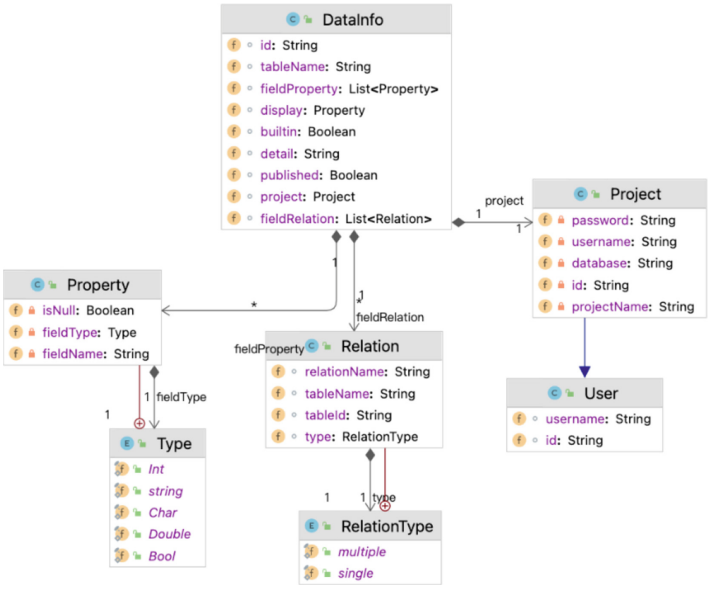

Fig. 3. Data Structure of Custom Data Model

2) Database connection

In order to automatically generate data tables through data models, we need to create a connection with the database. After the user confirms to publish the data model, the system will establish a link with the database, and generate a script according to the data of the data model, including all attribute information and associated information. After the script is generated, you can create a data table corresponding to the data model data in the database by running the script on the database connection.

The association information between data models is stored in the Relation class, and when creating a data table, if the corresponding data model has an association, the corresponding data table will be generated according to all the associations stored in it. The data table will store the id of the original table and the id of the associated table.

Because the data tables are automatically generated, the management of the structure of the data table itself needs to be realized by running SQL scripts, and the management of the data in the data table also needs to be realized by running corresponding SQL scripts. Compared with the traditional code development, all data tables generated by the platform need to be directly linked to the database, instead of using the existing API directly.

For the data in the data table, the platform will automatically generate data services based on Restful semantics, and these services are realized by running corresponding SQL scripts generated according to the data of the data model. Figure 4 shows the difference between the service layer of the platform's custom data model and the traditional architecture design.

From the original development of Controller, Service, Repository classes for all data models, it has evolved into a corresponding class that only needs to be designed

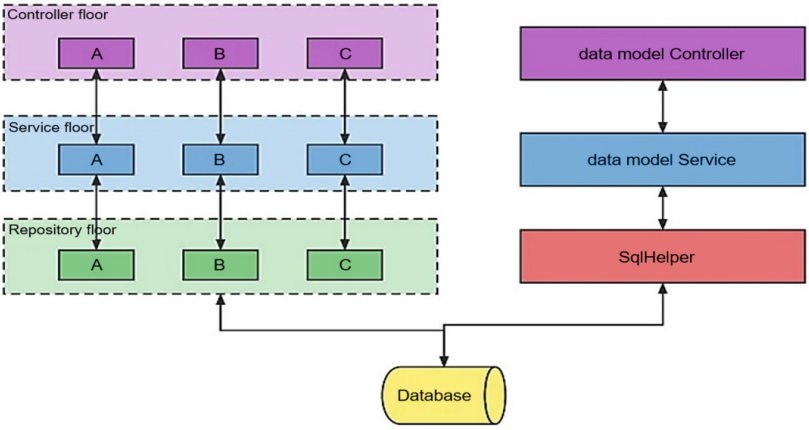

Fig. 4. Comparison of Service Layer Architecture of Custom Data Model

to manage all data models. Because the platform needs to connect directly with the database, we no longer use the Repository layer, but use a SQLHelper class to process data requests, including changing the structure of data tables, data services and all other database-related operations.

3) Presentation layer

In the presentation layer of the customized data model, users can view the attribute information and associated information of the data model. The published data model can view the specific data stored in the data table in the database at this time. In addition, the platform also provides the function of querying a single data model by using query statements. Users can add query statements for all the attributes of the data model. Finally, the platform will summarize all the query statements, generate corresponding SQL scripts, and return the query results after running.

3.2 Visual Interface Layout

Considering the ease of use of the platform, we need to make the use process of the platform as friendly as possible, so in the visual interface layout stage, we hope that users can browse the actually generated interface in real time when editing the interface. Therefore, we put forward that the layout of visual interface should follow the principle of "what you see is what you get". In order to simplify the user's design operation, we choose drag and drop as the main operation method to edit the interface layout. For editing the style of interface elements, we allow users to edit the CSS data of corresponding components directly in the detailed panel, thus realizing the custom editing of the style.

1) Interface data model structure

In the process of using the low-code platform, we will store the data model and interface information on a project basis, so the first step users need to do when using the low-code platform is to create the project. After consulting the relevant literature, the interface

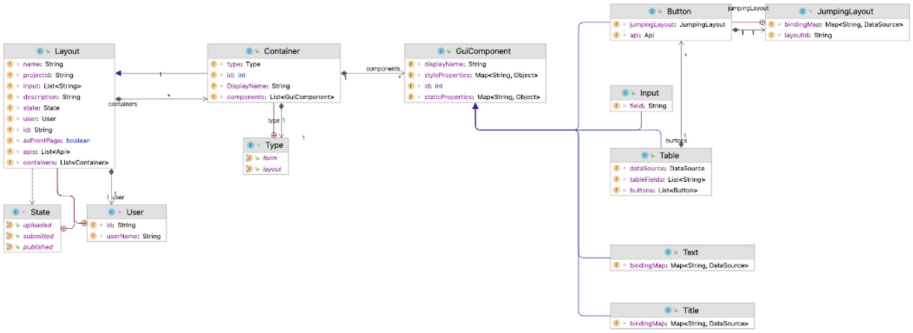

Fig. 5. Data Structure of Interface Model

model data structure as shown in Fig. 5 is obtained. It should be noted that this diagram is only a subset of the whole architecture class diagram, and the classes related to this part of the function are screened out here in order to focus on the visual interface layout.

(1) Layout class

Starting from the interface, the Layout class stores abstract data of an interface, and an interface should have basic information such as its project id, detailed description, release status, etc. at Front Page indicates whether the interface should appear on the home page. In addition to these basic information, the Layout class also needs to save the DOM structure of the interface and its corresponding style. In order to systematically manage the DOM structure of an interface, we stipulate that the Layout class corresponds to the pages in the DOM structure, while the Container class saved in the Layout class corresponds to the block elements in HTML.

(2) Container class

The Container class can be regarded as an abstract class of block elements. Because this design uses the React + Antd framework, the Type types in the Container class and GuiComponent class correspond to the interface elements in the Antd framework. After analyzing the functional requirements of interface elements in the interface, we found that the main functions of block elements in the use of the interface are to display data and input data, so we designed two common Container types, Layout and Form. The former will be responsible for displaying data, while the latter will be responsible for collecting user input data.

(3) GuiComponent class

GuiComponent class can be regarded as all other elements embedded in block elements in the interface. GuiComponent class can only exist in Container class, which is also in line with the nesting relationship between them during actual rendering. Considering that different elements of HTML have many common attributes, and all HTML elements can apply CSS, we design the GuiComponent class as a general abstract class of HTML elements. StaticProperties will correspond to the attribute information of the element,

while styleProperties will correspond to the style information of the component, and displayName will represent its name displayed in the editor.

On the basis of GuiComponent class, we design unique classes for different elements to describe their corresponding features and functions. At present, we have designed five abstract classes of representative elements, all of which inherit from GuiComponent class:

(a) Button: the corresponding button element, because it usually executes its bound callback function after clicking, obviously it is not ideal to use the button element to display data, so it should only exist in a Container of Form type. For the callback function called by the button element, we can analyze two types of callback functions according to the platform requirements. The first is to call the service in the platform, and its abstract data is stored in the api attribute, and its type is Api class. This class is a general class of API that can be called inside the platform. Because this part is related to data binding of the interface, we will describe it in detail later. Secondly, it is also possible to call a function that jumps to other interfaces. The abstract data of this behavior is saved in the jumpingLayout property, and its corresponding class is JumpingLayout class, which stores the id of the target interface to be jumped to and the data source information it needs to carry when jumping. Considering that API calls and interface jumps often occur at the same time in actual use scenarios, they are allowed to exist at the same time in the Button class.
(b) Input: corresponding to the input box elements. Like Button, it should only exist in the Container class of type Form.
(c) Table: corresponding to table elements. After summing up, we found that the data displayed by it is the data obtained from the application at runtime. This data has many possible sources, and we abstract it as a DataSource class, and its principle will be explained in the section of interface data binding. The use requirements of table elements are often accompanied by certain operations on the display data, such as allowing users to click buttons to view the detailed data of the display data or other operations. To support this behavior, we have saved the buttons attribute in the Table class to provide these operations. All data in a table element should support the same operation. For example, each column of data in a dataset should have its corresponding view button, but the buttons themselves should be responsible for the same behavior and style, and the only difference is that the corresponding data are different. Therefore, for these operations, we can save their abstract information in table elements, and because these operations generally exist in the form of buttons, we save multiple button elements in table elements. These Button elements are completely consistent with the running logic of the button class, that is, they also support API calls and interface jump calls of applications, which is also in line with the usage scenarios of table elements. Considering that when displaying data, users may not want to show all the attributes of the data model, we provide the showFields attribute to save the data that users want to show.
(d) Text: corresponding text element.
(e) Title: corresponding title element.

2) Interface editor

Obviously, the ordinary form filling and editing methods can not meet the high usability requirements of visual interface layout. To this end, we need to design an editor to edit the interface. Figure 6 shows the GUI model-driven interface editor of low-code platform.

We divide it into seven regions:

(1) Page header: Area A provides the function of editing part of the overall information of the interface, including modifying the name of the interface, whether it is displayed on the home page or not, and publishing status. The button of submitting modification will synchronize the editing information of the current interface to the server.
(2) Component panel: Area B provides the function of adding a Container class or a GuiComponent class, and the user only needs to drag the corresponding icon to area C to generate a new corresponding class to the interface.
(3) Canvas: Area C is a real-time preview of the interface, which has the same layout as the actual online interface, and will be rendered in equal proportion when the interface is online, thus achieving the effect of "what you see is what you get".
(4) InputPropertiespanel: D area is used to manage the data that this interface must carry when jumping from other interfaces.
(5) The API Propertiespanel: E area will be used to manage API data that needs to be called in advance when the interface is running.
(6) Heritage Panel: Area F will be used to show the element architecture of the interface, including all the existing Containers in the interface and all the GuiComponents it contains. By clicking the corresponding graphic elements, the ComponentProperties-Panel can focus on the corresponding container class or GUI component class.
(7) Component Properties panel: The G area will be used to edit the related information of the component. In addition to clicking the graphic elements in HierchyPanel, the user can also directly click the graphic elements in the Canvas to focus on the corresponding elements.

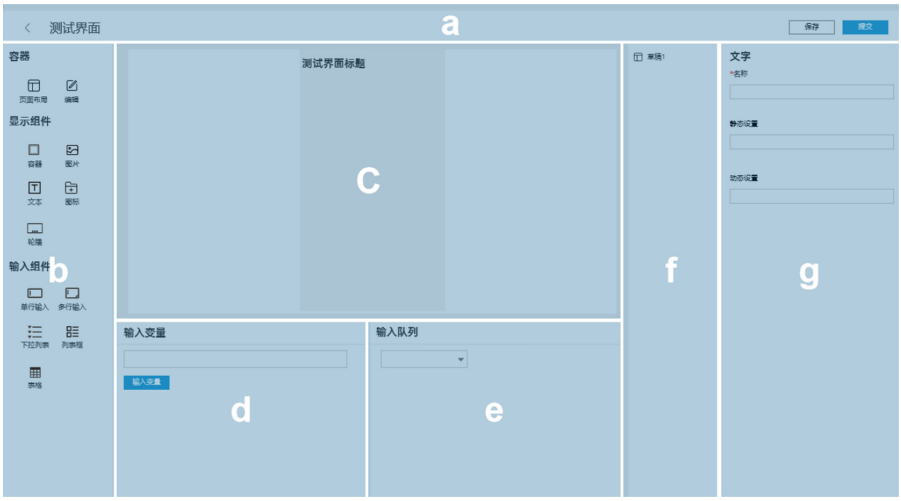

Fig. 6. GUI model-driven interface editor

3) Rendering method of interface layout

After determining the main operation method of using dragging interface elements as visual interface layout, we need to determine the actual scheme and save the obtained location information in the database. We also need to ensure that the interface follows the principle of "what you see is what you get", so we also need to determine how to make the canvas and the online interface render the same interface when facing the same data.

(1) Element drag and drop

The principle of dragging an element is to mark the draggable attribute of the element in ComponentPanel as true, and bind the setData function of DataTransfer object for the onDragStart event of the selected element, and bind the type of the selected element to the setData function.

At the same time, bind the getData function of the DataTransfer object for the onDrop event of the target element. After the target element successfully obtains the type data of the generated element from the getData function, the corresponding abstract class is generated for it.

(2) DOM structure is synchronized with CSS data

To ensure that the platform follows the principle of "what you see is what you get", we need to ensure that the Canvas area and the final online interface have the same DOM structure and CSS data.

The data used by the Canvas area and the final online interface are the same when rendering the interface, so to keep their DOM structure consistent with CSS data, it is only necessary to adapt their renderers. The renderer of Canvas area should use Canvas as the root of DOM structure, while the interface renderer needs to use the root element of the page as the root of DOM structure, and other things should use the same rendering logic.

On the basis of the same DOM structure, it is only necessary to keep the CSS data synchronized to ensure the same interface style, and the renderer only needs to apply the corresponding CSS data to the elements when rendering.

It should be noted that for the CSS data of the components mentioned earlier, because the CSS data relative to the Canvas area is stored in the database, if we want to synchronize some CSS data with absolute values, such as CSS data saved with px, we need to multiply the original data with the size ratio of Canvas and interface to get the correct display result.

3.3 Interface Data Binding

In this section, we will explain in detail how to specify the data used by the components in the interface. After analyzing the usage scenarios of the platform, we found that users will need to bind predefined static data and dynamic data that can only be obtained at runtime. For static data, users can edit it directly in ComponentPropertiesPanel. For dynamic data, we need to sort out all the dynamic data that the platform can obtain, and analyze their sources and acquisition methods.

1) Platform service call binding

By analyzing the usage scenarios of the platform, we can find that the final online interface obviously needs to integrate the data in the platform, and conversely, we also need to provide a display interface and an editing interface for the data model inside the platform. And we can abstract the behavior of this service call into data, so as to get the class diagram as shown in Fig. 7.

The Api class is an abstract class of calling behavior, and its derived classes are DataModelApi, MultiServiceApi and ASServiceApi, which respectively represent Restful data service call, composite service call and atomic service call to the data model, and the latter two are not explained here. For the calling behavior, it is characterized by the possibility of input data and output data, which needs to be determined according to the specific type of service. For example, the Post service in the data service only has input data, while the Get service only has output data. Another feature is that when elements call these services, they need to bind their corresponding elements for these data. For input data, we need to specify the data sources of all input data in the interface. For the output data, if necessary, we need to specify where the output data is displayed or used. The bindingMap attribute of Api class corresponds to the binding relationship of input data, while the binding relationship of output data is saved in the element that needs the corresponding data.

For the DataModelApi class, in addition to specifying the basic information needed for service invocation such as data model id, the platform also provides the binding of conditional statements, which enables users to bind data query services in the visual interface layout stage.

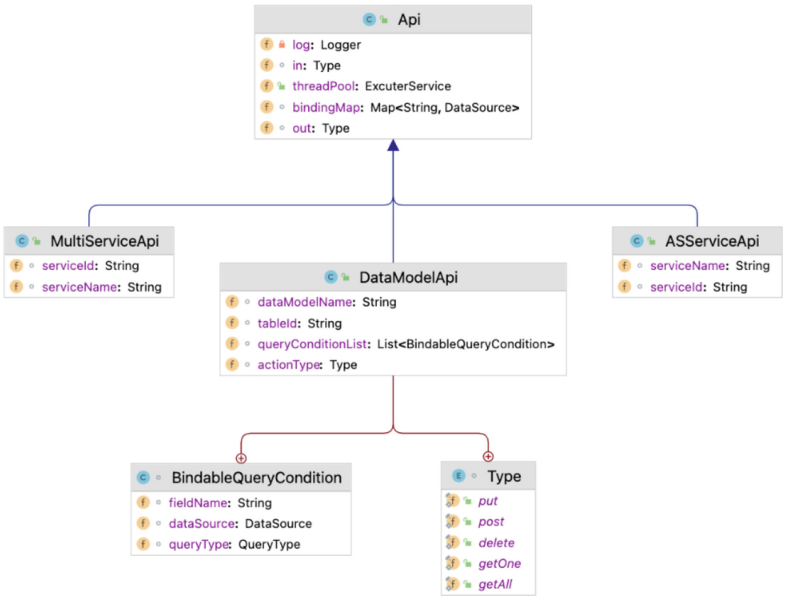

Fig. 7. API Class Diagram

The Api class exists in the Layout class and the Button class, because for the interface, it is often necessary to call some services in advance before loading, while for the button component, combined with the use scenario, we find that only the button component may have the behavior of service calling. In the running stage of the platform, the Api class saved in the Layout class will be parsed when the interface is loaded, and then the corresponding service will be called, while the Api class saved in the Button class will be processed when the click event occurs.

2) Binding of platform data sources

Earlier, we mentioned that there will be input data and output data in the call of services in the platform, and for input data, the data source of input data should be bound. Before explaining how each component saves the binding relationship, we need to consider the saving form of the data source itself.

After summarizing all the data that the platform can get at runtime, we get the data source class as shown in Fig. 8.

The DataSource class is the base class of all data sources. Field has different usage methods in different data source types, but it essentially saves data similar to key values in the corresponding data source. For example, the field in the DataModelApiDataSource class saves the attribute names of the data model. We have summed up a total of eight data sources, which are explained in detail below:

(1) inputDataSource: it corresponds to the value of the input attribute in the Layout class, and this data source represents the data passed to this interface by other interfaces when they jump to this interface.
(2) ConstantDataSource: it corresponds to the data source when users want to bind static data. The reason for designing this data source when staticProperties exist in the GuiComponent class is to ensure the unity of binding relationship.
(3) TableElementDataSource: It corresponds to the situation when the button element of each data existing in the table element needs to bind the data, and the data source can only be bound in this situation.
(4) EmbeddedDataSource: it corresponds to the built-in data in the application, including the basic data of users.
(5) ComponentDataSource: corresponding to the Input data of the component, such as the user input data obtained by the input class at runtime.
(6) DataModelApiDataSource: corresponding to the data obtained from the preloaded custom data model defined by the interface.

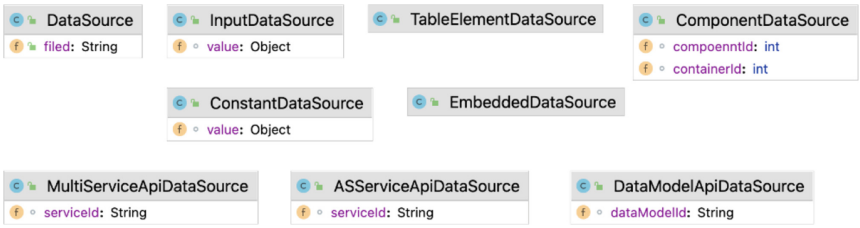

Fig. 8. Data Source Class Diagram

(7) Asset API data source: corresponding to the data obtained by the preloaded custom atomic service defined by the interface.
(8) MultiserviceApiDataSource: corresponding to the data obtained by the preloaded custom composite service defined by the interface.

After summarizing all the data sources of the platform, we can bind these data sources to different components of the interface, and there are different binding methods for different components. For all components except Table class, there may be multiple data to be bound. For example, suppose a Button class saves an Api class of DataModelApi class, and its actionType is post, then we need to bind a DataSource class for all non-empty attributes of the corresponding data model, otherwise the service cannot be successfully called in the actual running stage. Therefore, we designed the property bindingMap for these components, which is used to save all binding relationships. For the Table class, because the Table class itself is used to display data, and secondly, when the Table class displays data, the information about whether to display a certain attribute has been saved in showFields, so it is possible to directly save its dataSource with a single data source class attribute. Because the Table class shows a data set, the dataSource to which it is bound can only be DataModelApiDataSource, ASServiceApiDataSource or MultiServiceApiDataSource, so we set this restriction in ComponentPropertieSpan.

3.4 Multi-interface Integration

In the previous section, we have mentioned how the jump logic is saved as a JumpingLayout class in the Button class. In fact, this is the only use case in which the interface jumps to other interfaces, including the Button class saved in the Table class. For the Button class in the actual running stage, we bind an interface jump function to the onClick event. The layoutId in the JumpingLayout class will save the id of the target page, and get the corresponding layoutclass when visiting the link with the id, and then use the interface renderer to render the information of the layoutclass into the interface.

In this process, we should pay attention to the way of data transmission between interfaces when jumping. After considering various implementation schemes, we decided to let the Layout class save the input data needed when it is used as the target interface and the binding information of the data needed when it is used as the initial interface. The input data when the interface jumps is defined in InputPropertiesPanel, where users can freely edit the relevant information of the input data, which will appear when other interfaces choose this interface as the target boundary. At this time, users need to bind the DataSource class for these data to ensure that the jump can be executed correctly at runtime.

Figure 9 illustrates how the jump logic of the interface is parsed from data into jump functions in the Layout class. First of all, the JumpingLayout class in interface A has two properties, layoutId and bindingMap. LayoutId will save the id of the target interface. In Fig. 9, Interface A will specify Interface B through layoutId.

After interface B is specified, interface A will access the Input attribute of the Layout class of interface B, and bindingMap will need to bind a DataSource class as the data source for each value of Input at this time. After completing this step, interface A has successfully saved the jump logic as data.

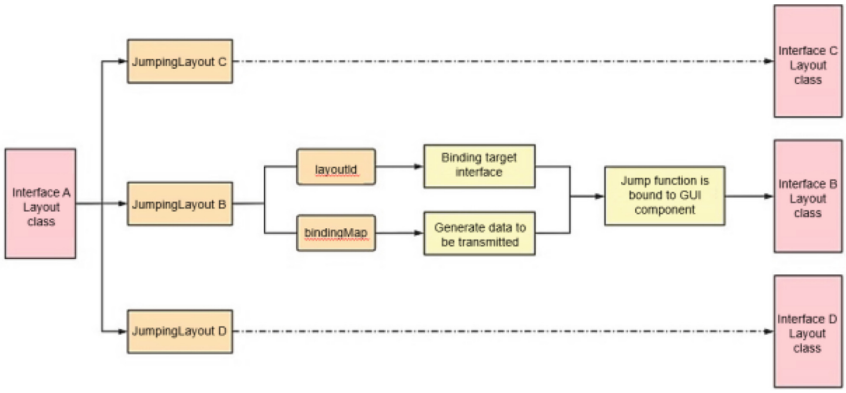

Fig. 9. Jump Binding of Multi-interface Integration

In actual operation, for the component with this JumpingLayout class, it will parse the bindingMap, and pass the data obtained by processing all the DataSource classes to the interface B when jumping. When transferring data, the binding relationship between the data and the Input variable will also be transferred. The reason for this is that all data in the interface needs to be obtained through the data source, and the transferred data needs to be saved as the InputDataSource class in the new interface to meet this demand. Multiple JumpingLayout classes are allowed in the same interface, and we can realize the integration of multiple interfaces in the platform by saving multiple JumpingLayout classes.

Verification effect:

In this chapter, a prototype system is established according to the proposed system framework and the described implementation method, and then a specific example of enterprise digital transformation is analyzed as experimental data, which is realized on the prototype system through four steps: user-defined data model, visual interface layout, interface data binding and multi-interface integration. After that, we compare the prototype system with other low-code platforms according to these four steps, and analyze the experimental results of the two, and finally make the effect evaluation of the prototype system. The evaluation results are as follows:

In order to compare and analyze the effectiveness of GUI model-driven low-code platform, we compare its advantages and disadvantages with other types of low-code platforms from four aspects: user-defined data model, visual interface layout, interface data binding and multi-interface integration. By consulting the related literature [4], we choose Quickbase low-code platform which belongs to different types from GUI model-driven low-code platform for experimental comparison. Their advantages in different aspects are shown in Table 1:

On the whole, we can draw a conclusion that the low-code platform driven by GUI model is better than Quickbase in functionality, which is mainly reflected in three aspects: visual interface layout, interface data binding and multi-interface integration. However, the GUI model-driven low-code platform needs to be improved in ease of use. First,

Table 1. Advantages of Low Code Platform Driven by GUI Model and Quickbase

procedure	GUI Model Driven Low Code Platform	Advantages of QuickBase
Custom data model	Meet functional requirements.	Meet the functional requirements and support the use of visual methods to edit data models.
Visual interface layout	Support the use of component layout interface as opposed to HTML, and also point out editing CSS data.	Support the use of HTML and JavaScript to edit the interface.
Interface data binding	Abstraction of interface data sources allows users to specify all possible data sources for components, which improves the fluidity of data in the interface.	Figurating the data source of components does not have the concept of data source, but supports specifying specific data sources for different components.
Multi-interface integration	Users can set the jump logic of the interface and specify the data to be transmitted.	Users can only set the jump logic of the interface, and cannot transfer data.

QuickBase can be referenced in the customized data model to support visual editing of the data model. We can also refer to Quickbase to support components with larger granularity when designing components, and supporting components with larger granularity and smaller granularity will further improve the functionality and ease of use of the platform.

4 Conclusion

This paper is based on the scientific research project of Shanghai Xinneng Information Technology Development Co., Ltd. "Research on Key Technologies of Low-code Development Platform Based on Micro-service" (project number: R22-003). The research content is the design of low-code platform driven by GUI model, which aims to provide a stable solution that can support rapid deployment, rapid development and save the human resources and time resources of enterprises in this process.

For the concept of low code platform and low code itself, it is almost inevitable that its follow-up will attract more scholars to study it. As long as the current development speed of science and technology is maintained, the data and informatization of enterprises will continue to be popularized, so it is not difficult to see that the development of low-code platform as a fast and stable solution is also guaranteed [9]. The design of GUI model-driven low-code platform proposed in this paper is verified by experiments, and it is found that there is still room for improvement and perfection.

For the components provided by the interface, besides the basic component implementation, we can consider providing users with more complex components to lay out

the interface. If we can provide both basic and advanced components at the same time, we will be able to further improve the functionality of the GUI model-driven low-code platform.

For the presentation layer of custom data model, we can change the editing mode from form submission to visual editing, thus further improving the usability of low-code platform driven by GUI model.

In addition to these two aspects, in the general direction, we can consider the subsequent improvements needed to be compatible with non-universal business scenarios [10]. At the same time, we also need to pay attention to ensure that the system still meets the software requirements mentioned earlier when the system is expanded in this process.

Acknowledgement. The study was supported by the project of Shanghai Shine energy Information Technology Development Co., Ltd. Which is named Research on key technologies of low-code development platform based on microservices in 2022, Grant No. R22-003.

About the Author. Zhu Jun, a senior engineer, has in-depth research in the fields of digital transformation, big data and artificial intelligence. He has presided over large-scale projects such as the smart supply chain project group of Shanghai Electric Power Company, the business center of Shanghai Electric Power Company and the integration of power grid resources.

Pan Xinyang, who has in-depth research in the field of Internet of Things and digital modeling, has been responsible for the research and development of large-scale projects such as *Lean Management System for Equipment (Assets) Operation and Maintenance* of State Grid Corporation of China (allocation and dispatching), business center of Shanghai Electric Power Company and supplier service platform of Shanghai Electric Power Company.

References

1. Vincent P, Iijima K, Driver M, et al. Magic quadrant for enterprise low-code application platforms [J]. Gartner report, 2019.
2. Al Alamin M A, Malakar S, Uddin G, et al. An Empirical Study of Developer Discussions on Low-Code Software Development Challenges[C]//2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR). IEEE, 2021: 46-57.
3. Czarnecki K, Eisenecker U W. Generative programming[J]. 2000.
4. Beranic, Tina, Patrik Rek, and Marjan Heričko. "Adoption and usability of low-code/no-code development tools." Central European Conference on Information and Intelligent Systems. Faculty of Organization and Informatics Varazdin, 2020.
5. Vincent P, Iijima K, Driver M, et al. Magic quadrant for enterprise low-code application platforms[J]. Gartner report, 2019.
6. Sahay A, Indamutsa A, Di Ruscio D, et al. Supporting the understanding and comparison of low-code development platforms[C]//2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). IEEE, 2020: 171-178.
7. Frank U, Maier P, Bock A. Low code platforms: promises, concepts and prospects. A comparative study of ten systems[R]. ICB-Research Report, 2021.
8. Sanchis R, García-Perales Ó, Fraile F, et al. Low-code as enabler of digital transformation in manufacturing industry[J]. Applied Sciences, 2020, 10(1): 12.

9. Bock A C, Frank U. Low-Code Platform[J]. Business & Information Systems Engineering, 2021, 63(6): 733-740.
10. Bock A C, Frank U. In search of the essence of low-code: an exploratory study of seven development platforms[C]//2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C). IEEE, 2021: 57-66.