# Low-code for CRUD Information Systems

Nuno Cardoso
up201706162

Sérgio Dias
up201704889

Tiago Silva
up201705985

## I. Introduction

Over the years, Information Systems have been used to organize and analyze data in a large number of different contexts. Actually, a considerable amount of web applications resemble or are based on an Information System (IS).

Although they all share a very similar underlying structure, it is still very common for organizations to rely on Software Engineers to develop their Information Systems from scratch. This represents a wasted effort, as hundreds of lines of code need to be written even before the specific requirements of an IS are taken into consideration and implemented accordingly.

This work aims to study and implement a code generation approach for this problem that takes advantage of the typical common parts of a system of this nature, and thus provides a simple and practical way to create one without a huge effort or Software Engineering knowledge. In concrete, we propose to prototype a tool that takes a JSON configuration file as an input and is able to build an information system that can be readily used.

## II. Project architecture

Our tool is a Node and TypeScript based application which, once executed, generates an information system as per specified in the `config/config.json` file. The generated project can be found inside the `output` folder, in a sub-directory that is identified with the generation timestamp.

The root folder of the generation app contains not only the source code but also a `template` directory in which the base templates for the generated frontend, backend, and database components are located. These templates include the files that are either common to all projects of same kind or do not require custom generation.

The generated information system is dockerized and includes containers for a React-based client, a Node.js based server, and a non-relacional database (MongoDB).

## III. Specification language

As mentioned in the previous section, the application's behavior is configured through the `config/config.json` file. This file is the only input that the tool needs and its purpose is for expressing the specifications of the generated IS.

The tool intends to target a use case where the end-users configure their own IS, so we assumed that the specification file would be written by someone with very little knowledge about the construction of an IS or programming in general. Therefore, the file specifies only the entities represented in the IS or, more specifically, the classes of information that the system holds and the available pages and features in the user interface. This resulted in a file structure with two main sections.

The first section allows the definition of the entities and their fields of information. Each resource must have a name and a list of attributes. Subsequently, these attributes, which must also have a name, must have a type or the name of the resource that they reference. Additionally, they can have a 'required' restriction that determines whether the field is mandatory or not.

The second section defines the pages and actions that the web application should provide. This section is a little more straightforward as it only specifies the pages and features through a JSON object with two fields. The first field identifies the resource the object refers to, and the other indicates through the values "Get-all", "Get-one", "Add", "Update" and "Delete" the pages and features to include in the generated system. The effect of each option is described in the following sections of this work. A configuration file can be written as the template in the attachments suggests (c.f. fig 1).

## IV. Parsing and modelling approach

Parsing and validating the specification provided by the configuration file are two tasks of our tool's first execution phase.

The parsing task is simple. Attending that the configuration is a JSON file and that we implemented the tool in Javascript and using NodeJS, the language handles the reading and the validation of the file's structure and syntax.

The validation of the values inserted in each field, however, is something that requires a little bit more effort. This task is completed using a class model of the information specified in the file and TypeScript types. Through this approach, we verify if the configuration provided is valid and build a model that will aid the code generation phase. This model stores information regarding the routes and the schema that the database must follow and dictates what code we must generate in the following phase.

## V. Code generation approach

Our tool's second execution phase is responsible for performing the code generation based on the specification received.

As said above, ISs, more precisely, web-based ISs, are structurally very similar. Usually, they contain a database, a server, and an engine that builds the user interface. From a high-level programmatic point of view, these components are always implemented in the same way. The different parts of the code from one case to another are the small fragments specific to the targeted use case. After some consideration and analysis of existing projects, we identified that the sections that change are primarily in the specification of the database schemas and the server routes. So, we concluded that we only needed to generate a limited portion of the code and prepared a template project comprised of a NodeJs server, a ReactJs application, and a MongoDB database capable of receiving the generated code fragments detailed in the rest of this section.

### A. Routes

Inside the server directory of the template folder, we have a directory to save our routes files. The `index.js` file keeps the routes that are available to the general context of the application, such as the routes for getting the home page, pinging the server, modifying the resources and the pages of the information system. These four routes are copied together with the rest of the template to the newly created information system. They are an essential part of our application, providing it the possibility of changing itself. Other routes are added to the `index.js` file during the code generation phase. They are also context-independent but only needed if the user adds pages to the configuration file. Thus, the application generates one route for getting the entire `config.json` file, one for getting the names of the resources of the information system, and another one for getting all the attributes of a specified resource (l. 134-185, `generation/routes.ts`).

We take a more complex approach to code generation for the context-dependent routes (l. 7-132, `generation/routes.ts`). Each page defined by the user in the configuration file has a method type and a resource. A new route is generated for each of those pages, and we organize them in different files for each resource. The type indicated by the user determines the method of the route:

- Get-all: **get** method which returns a list with all the elements of the specified resource;
- Get-one: **get** method which returns all the attributes of a given resource and id;
- Add: **post** method to add a new element of a resource;
- Update: **put** method do edit an element of a resource;
- Delete: **delete** method to remove an element of a resource.

### B. Models

A similar approach to the generation of routes is used to generate models. Since we use MongoDB to store data, a new model file with the resource schema is generated for each resource defined in the configuration file. To achieve that, we implemented three functions to deal with each part of the schema. The "generateModel" function (l. 4-25, `generation/model.ts`) is responsible for generating the mongoose schema name according to the resource name, and calling for the "generateSchemaAttributes" function (l. 27-44, `generation/model.ts`), which will generate

the code for each of the resource attributes and call for the "generateAttributeType" function (l. 46-57, `generation/model.ts`) to deal with the attributes types. Our application schemas support four different types: number, string, bool, and date.

### C. Making it all work

Generating isolated code files is not enough to obtain a working server, as they need somehow to be linked to the original template. The generated models are only required in the route files, so we can just add the import statements on the latter as we generate them. On the other hand, importing the route handlers requires updating the `app.js` file of the template. It is necessary to both add import statements for each of the route handlers (as we are keeping them in separate files), and middleware layers. Having code readability in mind, we apply these modifications in predefined parts of the file (l. 107-115, `generator.ts`), which are identified with commented labels in the server template.

### D. Frontend

The client-side of our application uses dynamic page creation without generating code. It has generic context-independent pages and loops through objects that it obtains from the server to present the data in a user-friendly visual way. It also requests the server for the configuration file to know the needed pages, and loops through them to create the necessary routes. Following this approach, we make the client adaptable to content modification without having to generate new code each time the IS is modified.

Our client comprises seven different template pages. The "Home", "Modify Pages" and "Modify Resources" pages are present in every information system our application generates. They represent the basis of the project that the user can use to access other pages. The "Get-all", "Get-one", "Add" and "Update" pages appear in the information system according to the configuration file and the user's modifications.

### E. Routes to support frontend

In order to know which operations the client should support, some new routes needed to be defined server-side. Five new endpoints were inserted to the code generation in the form of "hasMethodRoute", for each of the five methods types our application supports ("Get-all", "Get-one", "Add", "Update", "Delete"). The server responds with a Boolean value, indicating the existence of the correspondent route method for a specified resource (l. 187-244, `generation/routes.ts`). These make it possible to know when the client needs to display a particular button in a specific part of the interface, like a "Show" button on the home page for a resource or "Edit" and "Delete" buttons for an element page.

## VI. Modifying the IS in real time

One common issue with most software projects is the existence of a discrepancy in the system's functionalities with the actual system requirements. This is due to deficiencies in the definition of the systems requirements during the project development or because, after the software is tested and used in a real-world scenario, the requirements change, and it becomes clear that some additional feature is needed.

ISs are not an exception, and this issue is frequent as well. To minimize this problem with the systems produced with our tool, we evaluated the possibility of generating IS capable of modifying themselves and ultimately integrated a functioning solution.

A key factor for this to be possible in the context of this project was implementing the ReactJs application without any code generation and making it change according to the responses received from the server. Another helpful factor was the database used being non-relational due to its more relaxed behavior in terms of schema definition compared to relational databases.

With this type of database and the user interface built in such a way, the procedure for integrating the feature was clear.

For the IS to be able to modify itself, we changed the IS template used to generate the project to include a slightly modified version of the code generation tool in the server and the configuration file used to create the system. Aside from this, we introduced two new routes to the server and two new pages in the web application.

Hereafter, the used strategy is as follows.

1) The user, through the pages added to the interface, is able to specify modifications to the current configuration and submits a request to the newly added routes.
2) The server verifies if no illegal modifications were requested (for resources and attributes, it is only possible to add to the current configuration). If that is not the case, it modifies the existing configuration file.
3) With the latest configuration file, the server creates a child process that runs its version of the generating tool, produces the new source code files, and updates it.
4) Using a process manager, the server is restarted and begins to execute the new version of the system.

## VII. User manual

After writing the configuration file in `config/config.json`, the user should follow the following steps:

1) Run 'npm install' command on the root, template/client, template/server, and template/server/generator directories and then run 'npm start' on the root of the project;
2) Enter the newly created directory (which follows the format `output/date,time`) and run 'docker-compose up';
3) The user may access the website on `localhost:3000` and visualize a list of all the specified resources, plus the "Show" button for each of the resources with a "Get-all" page associated (c.f. fig 2);
4) By pressing the "Show" button, the user is redirected to the "Get-all" page, with a list of all the resource elements. If an "Add" page was specified in the configuration file, an "Add" button should also appear, and the user can add new elements filling a form. If a "Get-one" page was specified, the user can click on the rows of the table to access them (c.f. fig 3);
5) On the "Get-one" page, the user can check all the attribute values of an element. If an "Update" page was specified in the configuration file, an "Edit" button should also appear, and the user can edit the element. If a "Delete" page was specified, a "Delete" button should appear, and the user can delete the element (c.f. fig 4);
6) On the home page, the user has also access to the "Modify Pages" and the "Modify Resources" by clicking the respective buttons.
7) On the "Modify Pages" page, the user can edit the pages' configuration. For each of the existing pages, two inputs will appear. The user can switch between resources and methods and even add new pages or delete the older ones. The server and the interface update by pressing the "Save" button and waiting just a few seconds (c.f. fig 6);
8) On the "Modify Resources" page, the user can add new elements and resources (always in lower case) to the IS (c.f. fig 7).

## VIII. Conclusions and Future work

The final outcome of this project reflects our initial expectations, considering that we have managed to successfully apply code generation techniques to design a solution capable of building a manageable Information System from a human-friendly configuration file. Throughout the development of this project, and particularly in the late stages, we have realized the potential of code generation approaches, not only as a way of enabling people with no Software Engineering background to create their own systems, but also as a support to the development process, allowing developers to focus on the peculiarities of each system rather than coding the same type of solutions from scratch over and over again.

The implemented tool fulfils its proof of concept purpose, however it could be improved in a number of different ways, of which we highlight input validation, especially during the parsing of the configuration file. Functionality wise, there is also considerable room for improvements, namely the support for more operations and an improved UI in order to enhance the generated IS manageability. Last, the modifying module could also be upgraded to support deeper changes, namely those that involve running migrations in the database, such as renaming or deleting attributes that other resources depend on. If this was to be implemented, an integrated versioning system would be equally welcome, so that it is possible to restore a previous working version in the event of failure.

```
// Names of resources and attributes must be in lower case
{
  "resources": [
    {
      "name": "<name_of_the_resource>",
      "attributes": [
        {
          "name": "<name_of_the_attribute>",
          "required": true/false [OPTIONAL]

          "type": "number"/"text"/"date"/"bool"
          [OR]
          "references": "<name_of_a_resource>"
        },
        ...
      ]
    },
    ...
  ],
  "website": {
    "pages": [
      {
        "method": "get-all"/"get-one"/"add"/"update"/"delete",
        "resource": "<name_of_a_resource>"
      },
      ...
    ]
  }
}
```

Fig. 1. Configuration file template.

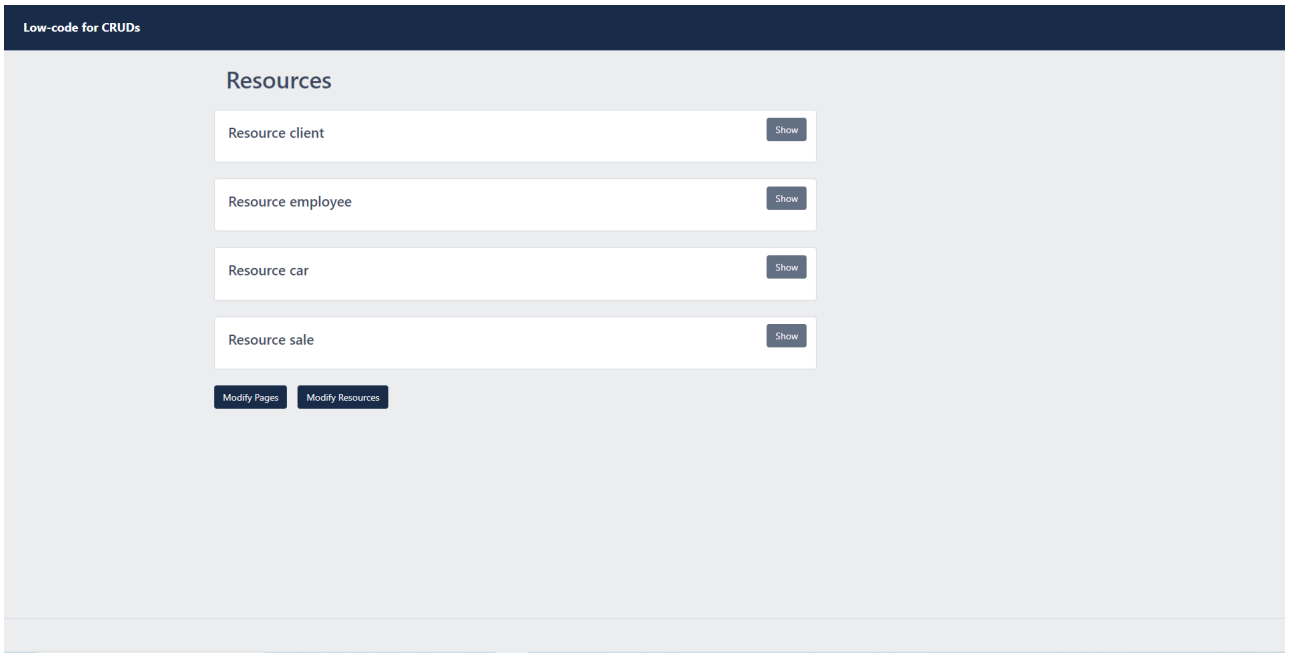


Fig. 2. Home page.

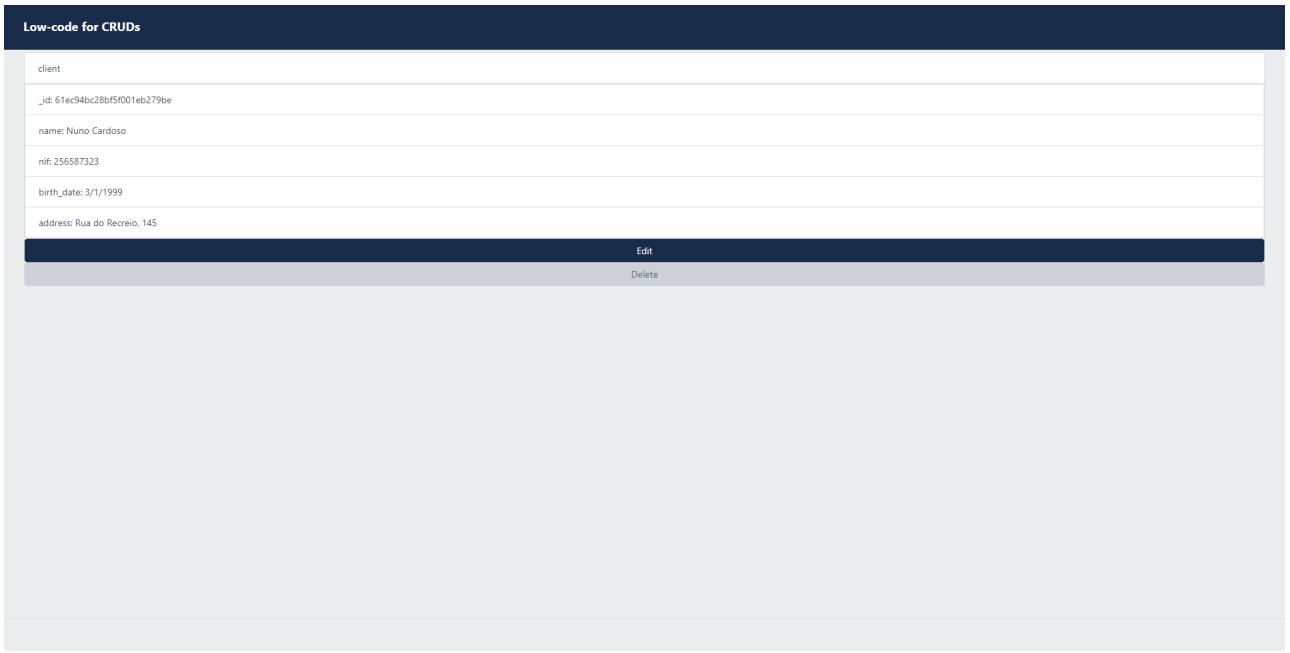


Fig. 3. Get All page.

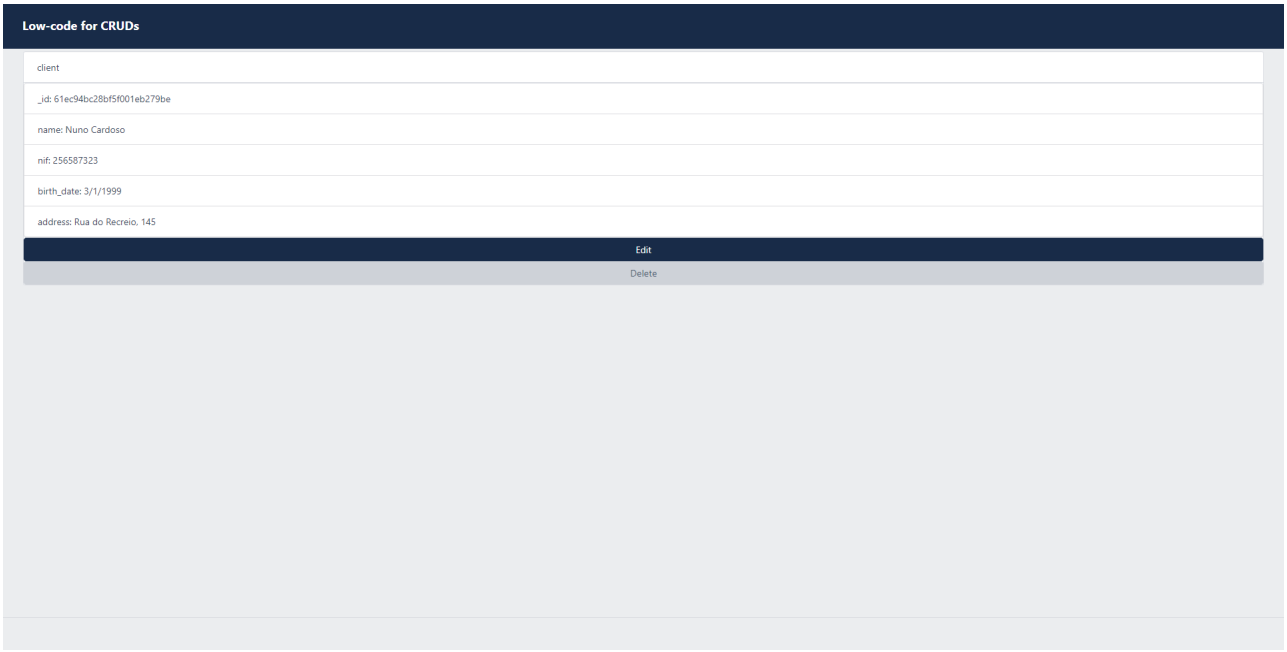


Fig. 4. Get One page.


Fig. 5. Add page.

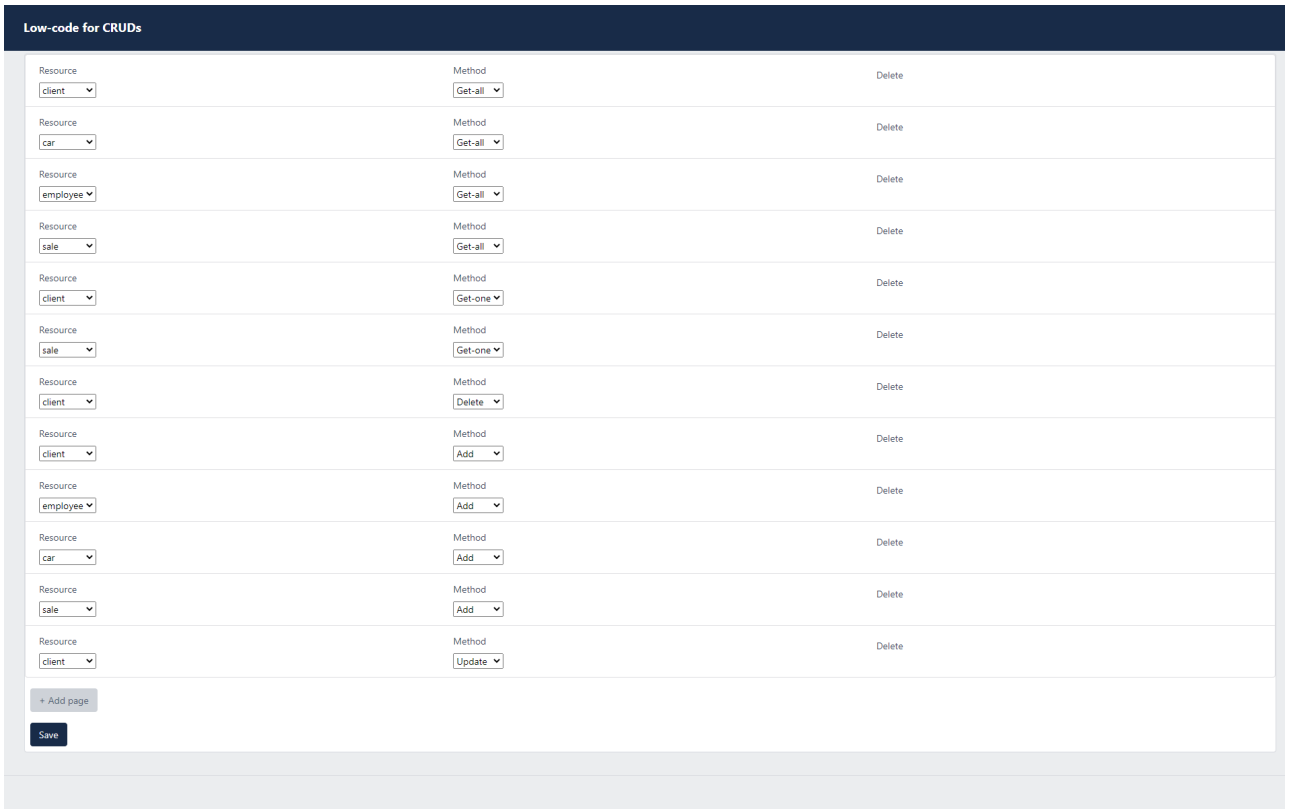


Fig. 6. Modify Pages page.

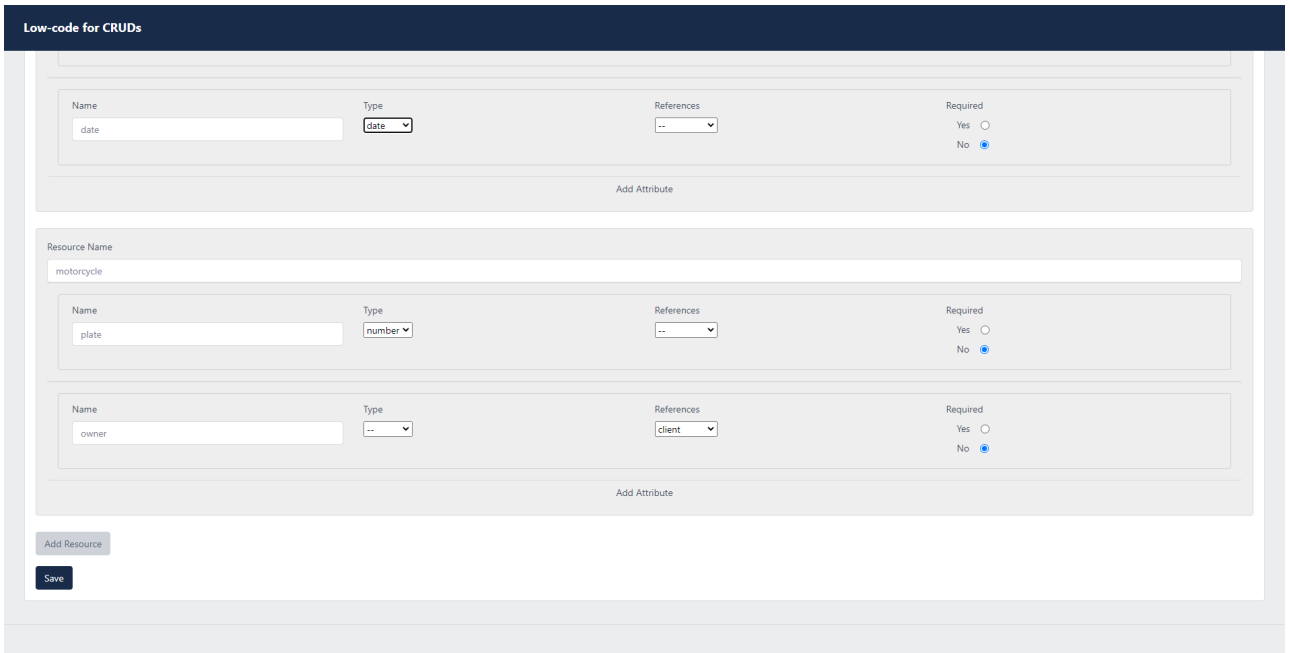


Fig. 7. Modify Resources page.