CPP-Admin: An Engine and DSL to Automate the Development of Web CRUD Applications in C++

Samuel da Silva Feitosa
Federal University of the Southern Border
Chapecó - SC - Brazil
Samuel.feitosa [at] uffs.edu.br

ABSTRACT

A web application is usually developed as a set of simple and complex concepts. For the simple part, such as the implementation of CRUD operations, most of the job is repetitive and can be highly automated. For the complex part, the full power of the language is desirable for the developers. A DSL is a language designed specifically to solve problems in a given domain, aiming to improve quality and bring productivity for their users. CGI-based web applications are still in use nowadays, mostly running on devices with a small amount of resources, to provide easy-to-use configurations. With all these concepts in mind, this paper presents a DSL to automate the generation of CRUDs in a CGI-based web application in C++ and its processing engine, which can be used to foster application development through the use of model driven software engineering.

Author Keywords

Domain-specific languages; CRUD generation; Embedded systems; C++.

1. INTRODUCTION

The area of web programming is evolving constantly. Today, this area is the first choice for developing new applications, being essential for several organizations. However, the task of developing a web application can become complex and time consuming. A CRUD (acronym for Create, Read, Update, and Delete) contains the four fundamental components to manage a web application, where the ``create'' component allows adding new data to the database, the ``read'' is used to retrieve items from the database and present into a web page, the ``update'' enables the user to edit an existing item, and the ``delete'' allows the user to remove an item record from the database [13]. These components are implemented widely in web applications, providing basic code and defining how the data is related in an information system. However, implementing CRUD operations for such a system is boring, and this task can be highly automated [9].

To approach this problem, automatic generators of CRUDs emerged, aiming to provide a way to construct such operations in databases rapidly, reducing considerably the time and effort spent when developing a web application [1]. Modern and popular frameworks such as Ruby on Rails and Django provide such tools, with a simple and easy-to-use interface to model a database and generate web pages automatically. These tools help the developer to have a web application with minimal interface and functionality faster than building it from scratch. However, usually these tools generate code that is not easy to maintain and also need a heavy setting of components to work on a web server.

With that in mind, this paper presents a JSON-like DSL to specify the components and behavior of an application, automating the generation of CRUDs together with an engine developed in C++ in a cgi-based setting. For tasks not covered by the CRUD paradigm, the full power of C++ can be employed to deal with the web requests, allowing the developers to focus on more complex and strategic features for their systems. Besides the benefits of having this automation, this engine implements a lightweight framework to develop C++ web applications, which can be employed for different contexts. It can be specially used in embedded systems, which usually lack resources to provide a full web server to run user applications.

The main contributions of this paper are:

- A DSL to specify and automate the processing of CRUD operations.
- C++ libraries to deal with HTTP requests, different databases, and generation of code.
- An engine to process the HTTP requests generating the corresponding code to communicate to the user.

The rest of this paper is organized as follows: Section 2 presents the background concepts for this paper, such as model-driven engineering, domain-specific languages, and CRUD applications. Section 3 presents the syntax and keywords, an example of a CRUD application using the proposed DSL, and the main components of the framework. Section 4 explains how the engine processes the language and some details about its implementation. Section 5 brings some discussion about the results. Section 6 shows some related work. And finally, Section 7 concludes the paper.

2. BACKGROUND

In this section, we introduce the main topics covered by this paper. The model-driven engineering (MDE), which allows the description of a system through abstract models; the domain-specific languages (DSL), which can be used specifically to solve a certain problem; and some extra information about the architecture of a CRUD application.

2.1 Model-driven Engineering

The Object Management Group (OMG) published in 2001 the first version of the model-driven architecture specification, which emphasized the use of models as artifacts for software development, giving rise to the model-driven engineering. More specifically, these models should be complete in a way that a software could be automatically derived from them [19].

Model-driven engineering (MDE) is a software development approach which focuses on the creation of models capturing the behaviors and requirements of a system. These models can be specified through a modeling language such as UML or SysML, or by using a DSL capturing the static and dynamic aspects of the system being developed [17]. Then, the models can be used to automatically generate the source-code, the documentation, or other software artifact.

Although the concept is interesting, there remains a lack of clarity on whether a general-purpose model is a good approach to be the core of software development [19]. Sometimes, creating a model requires more knowledge (and time) than coding software directly. There are cases of success and failure in industry, which makes it difficult to evaluate the application of the MDE in practice [19]. Despite that, the MDE has been shown to be a promising approach for developing software in several areas, including web development.

2.2 Domain-specific languages

A Domain-Specific Language (DSL) is a programming language developed and adapted for solving problems in a specific area (or domain), i.e., a more abstract language developed exclusively to tackle a domain of problems [14]. In contrast to General Purpose Languages (GPL) (such as Java, C, or Python), which are designed to be generic and to offer support for developing several kinds of applications, a DSL is developed with an specific aim~\cite{langlois2007dsl}. For example, a DSL can be used to query a database, to filter a network packet, to describe a CRUD, etc.

DSLs are used to simplify the development process, often making the code easy to read and to maintain. They usually offer some benefits, such as productivity, lower learning curve, less errors, etc [8]. It is common to have a DSL based on a model, where the language is able to describe data and operations. In the context of MDE, a DSL is a specialized language which uses a transformation (or generation) function, aiming to abstract the software and facilitate the development [14]. Features like the discussed earlier are specially important for this paper, since we aim to facilitate the development of web applications by using a DSL that specifies the data and behavior for common operations.

2.3 CRUD Applications

CRUD is a well-known acronym in web development which means "Create, Read, Update and Delete". It is a broadly used approach to perform basic operations when integrating databases and information systems. It consists of an essential part of the development of software, being used in several domains, including finances, health, sales, management, among others [18].

As mentioned, there are four operations in a CRUD. The "Create" operation refers to creating or inserting new registers in a database. The "Read" operation is used to retrieve data already inserted. It can be used to retrieve all data of a given entity, or to obtain information from a single register. The "Update" operation is used to modify a given register in a database system. And the "Delete"' operation is used to remove a given register from the database [9]. All operations together form the basis of a CRUD which is used in a similar form in several systems with different databases and programming languages. The approach is so common that some popular frameworks (such as Ruby on Rails and Django) have built-in support for creating CRUDs, providing easy and fast implementation of these operations.

3. A DSL TO GENERATE CRUD APPLICATIONS

This section presents the JSON-like[1] DSL used to describe the data and behavior of a web application, which is stored in a configuration file for a CRUD. A web application in the proposed framework is composed of several CRUD configuration files, which are used as input to the engine to be interpreted and to generate the HTML code for the frontend side.

Next we explain the syntax, the reserved keywords, and some predefined web components, which are used in a configuration file to define the CRUD options, and can be expanded to serve different domains.

3.1 Syntax and Keywords

A configuration file defines a CRUD using a JSON-like format, i.e., a key-pair setting of values for describing the data and behavior for a Graphical User Interface (GUI) of a given entity with possible relationships. Here we explain how the document should look like and how each keyword is interpreted by the engine. A syntactically valid key-pair configuration can be seen next.

```
label="Corporate Client Registration"
```

Some keywords expect a list of values to be interpreted. The list of values is just a regular string separated by commas, as we can see next.

```
mandatory="descr,realname,document"
```

[1] We are calling it a JSON-like DSL because we are not using the full power of JSON [2], relying on a key-paired list of possibly nested strings.

And some keywords are nested with their specific information. For example,

```
id.label="ID"
id.fieldtype="text"
```

shows specific information (*label* and *fieldtype*) linked directly to an *id* keyword.

The configuration file to specify a CRUD application can be divided in some parts: (1) general keywords; (2) form fields and table options; and (3) specific information for form fields. We describe each valid keyword in the next subsections.

3.1.1 General keywords

The syntax has keywords which are general for the whole CRUD application and are shared by different CRUD operations. We show next a list of the valid keywords.

- *label*: A label to be shown on the title of the web page.
- *info* (optional): A description about the entity or the CRUD itself.
- *options*: A possibly empty list with CRUD options (create, search, etc.).
- *dbtable*: The main database table to be managed.
- *prefix* (optional): An optional prefix for each database table.
- *orderfields* (optional): An optional list of column names to order the result set.

The presented keywords are shared by the CRUD operations, and are used for configuring title, additional screen description, whether the CRUD allows creating and/or searching for information, which database table is linked, etc.

3.1.2 Form fields and table options

These configurations are used to manage the CRUD operations, and how they should relate to the database. The valid keywords are listed next.

- *datafields*: Defines which column names should be retrieved from the database.
- *formfields*: Describes the fields that should be collected from the user by a web form and sent to the server to be saved on the database.
- *tablefields*: Expects the column names to be shown on the listing page.
- *tableoptions*: Defines which options should be allowed for each register (show, edit, delete).
- *tableoffset*: The offset to be used on the pagination.
- *mandatory* (optional): Defines which fields are mandatory for validation.

The keywords presented in this subsection are used specifically for some CRUD operations. For example, the *formfields* keyword is used on the "Create" and "Update" operations of a CRUD, since they are responsible to render and collect information from the user, which should be saved on the database on the server side. The *datafields*, *tablefields*, and *tableoffset* keywords are used on the "Read" operation of a CRUD, being responsible to define what should be retrieved from the database and shown to the user on a listing page. And, finally, the *tableoptions* keyword is used for both the "Update" and "Delete" CRUD operations, since they are responsible to allow the user to proceed with these actions.

3.1.3 Specific information for form fields

Each form field should be described in more detail, since they can define the field type, whether it is related to another database table (foreign key), and some extra personalization on the user interface (label, icon, etc.). For that, there are still some keywords, which should be nested on the form field item.

- *label*: The label to be shown on the form for a given field.
- *fieldtype*: The description of a type for a given field. We offer a set of predefined visual components for field types, both for simple types (Text, TextArea, Select, Radio, etc.) and composite types (InputToList, Se-lectToList, ListToList, etc.).
- *icon* (optional): An optional icon to be shown next to the field on the form.
- *relation* (optional): Used when the field represents a relation with another table (one to one, one to many, or many to one).
- *multirelation* (optional): Used when the field repre-sents a many to many relation between two tables.
- *reltable* (optional): Allows the user to inform the name of a join table for a many to many relationship.

With the presented keywords, the engine is able to link the form fields with the database fields, proceed with validations, and render the HTML/CSS for the browser. As we could note, the DSL allows the user to manage different kinds of relationships between entities, including one to one, one to many, many to one, and many to many, and the engine is capable of interpreting and generating the code automatically by using the configurations.

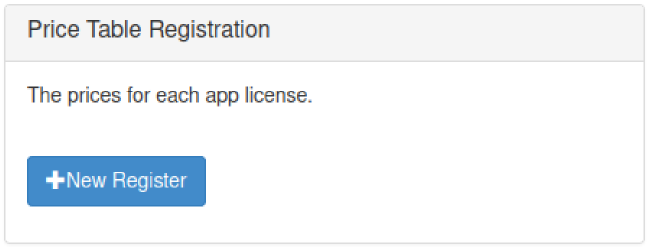
3.2 CRUD configuration example

This section shows an example of a CRUD configuration file, which is used to manage information on an information system for selling software licenses. We present the configuration file named `price.conf` divided in three parts, showing the results on the generated screen together with each part to make the explanation easier.

The first part describes the general CRUD information, with a label, options, and the database table responsible to store the data.

```
# General CRUD information
label="Price Table Registration" info="The prices
for each app license." options="create"
dbtable="software_license"
prefix="swm" orderfields="id"
```

This piece of code is responsible for rendering the upper part of the user interface, and to keep basic information for the CRUD application. Figure 1 shows the results of processing the code.


Price Table Registration

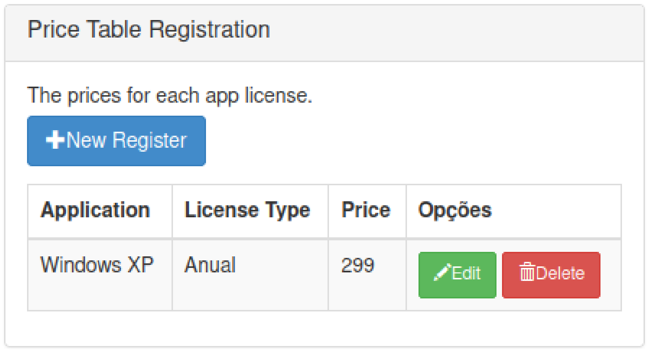
The prices for each app license.

Figure 1: Rendered user interface for the first piece of code.

Next, we show the form fields and table options information, that is responsible for describing which data should be retrieved from the database, the fields that should be shown and collected on the generated form, and the columns that should appear on the listing table.

```
# Form fields and table options information
datafields="id,descr,price"
formfields="descr,software,license,price"
tablefields="software,license,price" tableoptions="edit,delete"
tableoffset="10"
mandatory="price"
```

The table rendered through the specification on the presented piece of code can be seen in Figure 2. The reader can note that the generated code is showing the buttons "Edit" and "Delete" as expected.


Price Table Registration

The prices for each app license.

Application	License Type	Price	Opções
Windows XP	Anual	299	Edit Delete

Figure 2: Rendered user interface for the second piece of code.

The last part of the configuration file contains specific information about the form fields.

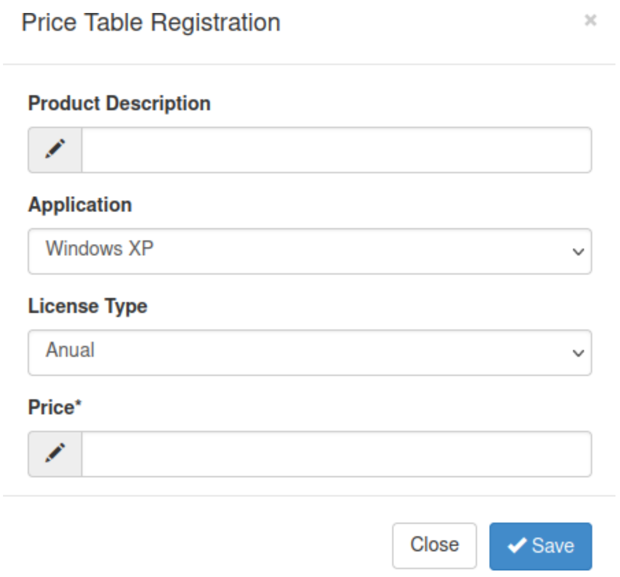
```
# Form fields information
id.label="ID"
descr.label="Product Description"
descr.fieldtype="text"
descr.icon="glyphicon-pencil"

software.label="Application"
software.fieldtype="SelectList"
software.relation="software"

license.label="License Type"
license.fieldtype="SelectList"
license.relation="license"

price.label="Price"
price.fieldtype="text"
price.icon="glyphicon-pencil"
```

Here we can note that, for each form field, we have a fieldtype (except for the *id*, which is created as a hidden field by omission). Also, the fields *descr* and *price* have icons to be shown, and the fields *software* and *license* have additional information about the relation with another table. For example, one can note that the field *software* is related to the database table *software*, and the *license* field is related to the database table *license*. Figure 3 shows the form rendered accordingly to its configuration.


Price Table Registration

Product Description	
Application	Windows XP
License Type	Anual
Price*	

Figure 3: Rendered user interface for the last piece of code.

The reader can note that the *software* and *license* fields are related to other tables, and that they use a "SelectList" field type. This is a special component which allows one to select one register coming from a related table. In our framework, we have several predefined components that can be used for simple and for composite types.

Although we offer some visual components to be used, the proposed lightweight framework is developed to be extensible, since the user/programmer itself can create new field type components using our template language to integrate with the server-side processor. In the next sections we will explain in detail how the server and client-side communicate, and how we can extend them.

3.3 Server-side Components

The DSL described in the previous subsections relies on several components on the server-side, which together allow a user to generate a CRUD application using a no-code approach. These components include libraries to integrate with different databases, profiling rules to improve application security, and the processing itself of the actions performed by the user.

3.3.1 Database Integration

This component is responsible for handling the business information managed by the system. We provide, on this first version, drivers for the databases PostgreSQL[2] and SQLite[3], and a factory pattern allowing a programmer to extend the support for new database drivers.

3.3.2 Users and Groups

With this component we provide access control mechanisms for system security. A user can be part of zero or more groups, and the permissions can be set both to users or groups. These permissions grant or remove access to the application's CRUD operations for each user. Obviously, this module is only useful together with an authentication mechanism.

3.3.3 Menus and Options

To be able to access the CRUDs for each entity, any application needs to provide a main menu with links to them. We provide an extra JSON-like configuration file for the admin page, which allows one to list all the links of the application. As we mentioned, the proposed approach is to automate the CRUD operations for each entity, and the complex business rules should be implemented using the full power of the C++ language. Considering this, the admin page is able to link both the generated CRUDs and the custom made CGI programs.

3.4 Client-side Components

The frontend of the CRUD is also automatically generated, and it uses some components to proceed with the user interface generation. Predefined tables, forms, and fields are filled by the engine, and the actions trigger the processing for showing information to the user, or to create, update or delete registers. Also, to provide a no-code experience for the user, we have a set of useful visual components to be used on the forms.

3.4.1 Tables, Forms, Fields and Actions

Tables and Forms are used to manage data, having a close relationship with the database model. This component generates all user interfaces automatically and is responsible to process the requests and manage the database by retrieving and saving data. As usual, the CRUD operations are performed when the user requests such actions through buttons or links.

3.4.2 Visual Components

We provide the most used visual components for a CRUD. They are divided into two categories: (1) the components to individual database columns, such as Text, TextArea, Password, Date, Time, etc.; and (2) the components which can be related to another table through a foreign key, such as CheckBoxList, RadioList, InputToList, SelectList, ListToList, etc. These components are available as regular HTML files with template variables, which are filled by the processing mechanism. As already mentioned, these form template fields can be extended according to the user needs.

4. ENGINE ARCHITECTURE

The DSL presented in the previous section is responsible for describing how a CRUD application should work. From the language perspective, the engine can be seen as an interpreter, which parses the CRUD configuration files, performs validation and outputs HTML code as response. However, a software to generate CRUDs automatically needs to perform other activities.

Thus, the problem tackled by the engine can be divided in four general steps: managing HTTP requests, interpreting the proposed DSL, providing access to databases, and rendering HTML/CSS/Javascript to be shown by the browser. That way, the engine is responsible for working with both the backend and the frontend. The engine implements the architecture shown in Figure \ref{fig:arch}.

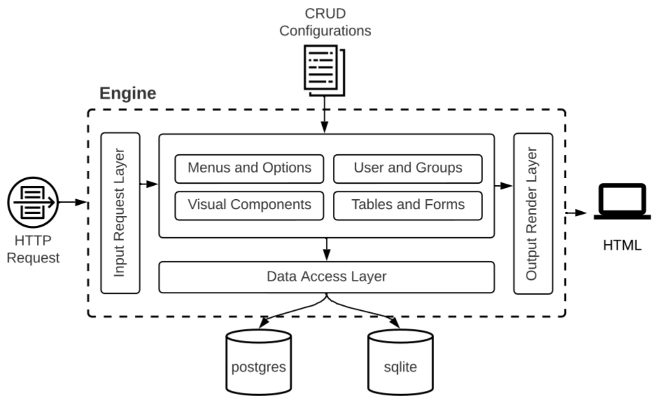


Figure 4: The engine architecture.

As we can see, the engine work starts by receiving a HTTP request, which contains the usual HTTP readers and the parameters, which can come via GET, POST, PUT, etc. To deal with that, we have the *Input Request Layer* which parses the HTTP packet and fills an object of the class *HTTPRequest* we developed. In our implementation, the CRUD operations only use GET and POST requests, where GET parameters are used mostly to control the route and the

[2] https://www.postgresql.org

[3] https://www.sqlite.org

register *ids*, while the POST parameters are used to pass form data.

Knowing which route should be generated, the code written using the DSL describing the *CRUD operations* is loaded and validated, preparing all the internal structures for the generation process. To process this step, we implemented classes to deal with profiling, menus, items, registers, forms, tables, etc. These classes are multipurpose, since each of them parse specific source code of the DSL and also output the results to the user.

Linked to the previous step, we have the *Data Access Layer*, which provides a common interface to access the data stored on the database. We used the object-oriented design pattern Factory to make it easily extensible for other database drivers. As mentioned, in the current version, we provide common access methods for both the *postgres* and *sqlite* databases.

The last step lives on the *Output Render Layer*, which is responsible for generating the actual HTML/CSS/Javascript code to interact with the user by means of a web server. It also invokes the actions requested by end-users (insert, delete, update on the database). The render is responsible for filling the visual components template with data, which will automatically generate tables, forms, fields, buttons, etc. It means that the user of the DSL does not need to care about this process, since it is performed automatically.

It is important to remember that the engine is implemented in C++, with a minimal use of external libraries and resources. That way we could keep the executable small enough to run them in low performance hardware, such as embedded systems.

4.1 Source Code

All the source-code for the proposed framework was compiled and tested with gcc/g++ version 11 using a Linux machine running Ubuntu 24.04.1 LTS. We avoided showing C++ code to not distract the reader from understanding the high-level structure of the DSL. The curious reader can access the source code of this project together with instructions to run an example application on our github repository[4].

5. DISCUSSION

Code generators, no-code programming, and CRUD generators have gained attention in the last years and have been increasingly used in the software development industry to accelerate the development of applications [4, 3]. Although those approaches have some differences, they all share the common goal of simplifying the development process.

Code generators are programs used to create code automatically by following a set of requirements or specifications provided by a developer [6]. Usually, they are used to automate the creation of applications with some general format or structure, such as CRUD web applications. By using such generators, developers can save time and avoid common mistakes. On the other hand, no-code programming is an approach to allow end-users to develop their own application without coding. The platforms of no-code usually offer a GUI which allows the user to create forms, listings, and other features without the need to understand a complete programming language [3]. This approach is useful for people without a background in programming, allowing them to create software focused on the business they know.

The result of this paper tries to merge both approaches, by offering an easy-to-understand DSL, using the style of configuration files, to allow an user to describe the common operations of a web application. We believe that a person without a background in programming can be able to create CRUD applications using the presented framework. Another important aspect is the way the engine works. Differently from code generators, the engine is an interpreter of the configuration files, which accelerates the process of development and test of the system, since it is not necessary to recompile the application for each new CRUD that is added.

Furthermore, we developed the engine to be a lightweight framework, taking into account embedded systems, which usually have to provide some web user interface for the users to configure basic parameters. The idea is to allow the embedded developers to focus on their area of expertise and offer a tool to generate the repetitive tasks of CRUD operations even for such low resourced devices. It is becoming even more important nowadays with the increasing number of IoT devices and industrial controllers.

6. RELATED WORK

In this section we present some related work on the automatic generation of CRUDs by using model-driven engineering or domain-specific languages.

Gomez et al. [9] describe in their paper a domain-specific language called CRUDyLeaf, which is used to generate RESTful APIs using the Spring Boot framework. This DSL allows developers to specify through a configuration file the CRUD operations they want to provide as a RESTful API. Then, from the configuration file, the CRUDyLeaf generator exports Java code automatically, including all the necessary models, controllers, and other classes to be consumed online. The authors discuss that their DSL can save time and effort from developers by generating complete RESTful APIs from a configuration file, without the need to code all the Java code necessary. This paper differs from ours because it only generates the backend part of the software, and is encoded in Java, where our DSL is used to generate both, the backend

[4] https://github.com/sfeitosa/cpp-crud-gen-dsl

and the frontend for a CRUD application using the C++ language. Also, we provide an engine that interprets our DSL instead of generating regular language files which have to be later compiled.

Livraghi [12] presents a system to generate CRUD applications automatically. The system uses a model-driven approach, which uses metadata to generate a complete application, including the backend and frontend part. The metadata can be a Java class, a XMI model, or a database schema. The author provides a tool which receives the metadata as input and generates the necessary code to run a RESTful API in Java, and an Angular script to provide the UI and to consume de generated API. The author states that the solution reduced drastically the development time, reducing costs for when developing an application. The approach proposed by the author is very complete, but it differs from ours in the sense that it generates Java code using Spring, which usually is not appropriate for devices with few resources, while our paper describes a lightweight engine that works with C++ and can run in such environments.

The work of Rodriguez-Echeverria et al. [16] proposes an approach to generate CRUDs using an Interaction Flow Modeling Language (IFML) developed as an Eclipse Plug-in in the Java language. The paper uses existing IFML models, and detects data entities, which are used to derive CRUD operations, and then generates the respective code for this purpose. The authors evaluate their approach in a joint effort by academia and industry to assess its validity in a real scenario, showing that it is effective to generate CRUDs from IFML models, reducing the time necessary to develop such tasks, and avoiding common errors. The difference from our work is that it is focused on scalable applications, which requires processing power from the server, while ours is implemented considering systems with low resources in mind.

There are still a bunch of papers describing different techniques and approaches to generating CRUD applications [15, 10, 7, 5]. Additionally, some frameworks such as Ruby on Rails[5], Django-Admin[6], among others, also provide some tools to automate the task of generating CRUDs. Since they are complete frameworks for web development, the automation of CRUDs is indicated for simple application configuration user interfaces, where the CRUDs for the real information system is usually written directly on the framework. Again, the essential difference from our work is that we focus on generating complete CRUDs (backend and frontend) in a setting that does not require many resources.

7. CONCLUSION

This paper described a JSON-like language to automate the process of encoding CRUDs in web applications, together with a lightweight engine developed in C++ to interpret the DSL, and responsible for generating both the backend and frontend, aiming to accelerate the development of software, especially for low resourced hardware, such as for embedded systems. With a tool like the one presented here, the developer can automate the repetitive tasks and focus on other important parts of the project. In such scenarios, the CRUDs are fully automated, and for specific parts of the system, the user can use the full power of the C++ language.

As future work, we can expand the generator to embed a whole lightweight web server, similarly to Spring, Node and others, to be able to run directly on small devices. Another possibility is to separate the client and server-side using REST principles and modern libraries, which would improve the quality of the generated code. We could also add new predefined components to serve for different application domains. One adaptation of the language could be made to allow the engine to generate and manage the database schema. Furthermore, the DSL itself could be improved by using JSON objects to describe the nested structures of the syntax. The authors believe that the results of this paper can be transformed into an open-source project as a framework for developing web applications in the domain of embedded systems.

8. REFERENCES

[1] A. W. Anuar, N. Kama, A. Azmi, H. M. Rusli, and Y. Yahya. Re-crud code automation framework evaluation using desmet feature analysis. International Journal of Advanced Computer Science and Applications, 13(5), 2022.

[2] L. Bassett. Introduction to JavaScript object notation: a to-the-point guide to JSON. ” O’Reilly Media, Inc.”, 2015.

[3] T. Beranic, P. Rek, and M. Heriˇcko. Adoption and usability of low-code/no-code development tools. In Central European Conference on Information and Intelligent Systems, pages 97–103. Faculty of Organization and Informatics Varazdin, 2020.

[4] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al. Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374, 2021.

[5] A. Delgado, A. Estepa, and R. Estepa. Waine-automatic generator of web based applications. In International Conference on Web Information Systems and Technologies, volume 2, pages 226–233. SCITEPRESS, 2007.

[5] https://rubyonrails.org/

[6] https://www.djangoproject.com/

[6] S. E. V. and P. Samuel. Automatic code generation from uml state chart diagrams. IEEE Access, 7:8591–8608, 2019.

[7] H. Ed-Douibi, J. L. C. Izquierdo, A. Gómez, M. Tisi, and J. Cabot. Emf-rest: generation of restful apis from models. In Proceedings of the 31st Annual ACM Symposium on Applied Computing, pages 1446–1453, 12016.

[8] M. Fowler. Domain Specific Languages. Addison-Wesley Professional, 1st edition, 2010.

[9] O. S. Gómez, R. H. Rosero, and K. Cortés-Verdín. Crudyleaf: A dsl for generating spring boot rest apis from entity crud operations. Cybern. Inf. Technol., 20(3):3–14, sep 2020.

[10] T. Karungu, L. Nderu, and D. Kaburu. An enhanced automatic generation of crud operations in react-js. 2022.

[11] B. Langlois, C.-E. Jitia, and E. Jouenne. Dsl classification. In OOPSLA 7th workshop on domain specific modeling, 2007.

[12] M. Livraghi. Automatic generation of web crud applications. 2016.

[13] P. McFedries. Web Coding & Development All-in-One For Dummies. John Wiley & Sons, 2018.

[14] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. ACM computing surveys (CSUR), 37(4):316–344, 2005.

[15] O. M. Pereira, R. L. Aguiar, and M. Y. Santos. Crud-dom: a model for bridging the gap between the object-oriented and the relational paradigms. In 2010 Fifth International Conference on Software Engineering Advances, pages 114–122. IEEE, 2010.

[16] R. Rodriguez-Echeverria, J. C. Preciado, J. Sierra, J. M. Conejero, and F. Sanchez-Figueroa. Autocrud: Automatic generation of crud specifications in interaction flow modelling language. Science of Computer Programming, 168:165–168, 2018.

[17] D. C. Schmidt et al. Model-driven engineering. Computer-IEEE Computer Society-, 39(2):25, 2006.

[18] C.-O. Truica, F. Radulescu, A. Boicea, and I. Bucur. Performance evaluation for crud operations in asynchronously replicated document oriented database. In 2015 20th International Conference on Control Systems and Computer Science, pages 191–196, 2015.

[19] J. Whittle, J. Hutchinson, and M. Rouncefield. The state of practice in model-driven engineering. IEEE software, 31(3):79–85, 2013.