DSLI API GENERATOR “EASY API”

Sergio S. Duarte
Daniel R. Cruz
Engineer Faculty, Systems Engineering
Computer Science III
Teacher: Carlos A. Sierra

Abstract - This project is a Domain-Specific Language (DSL) API Generator that automates the creation of Spring Boot applications. It leverages Java CUP and JFlex to define and parse a custom DSL for describing entities, relationships, controllers, and configurations. The parser interprets the DSL syntax and generates fully functional API code in Spring Boot, streamlining the development of RESTful services. The project aims to simplify API generation, enforce consistency, and reduce manual coding effort in enterprise applications.

***Keywords* - Syntax, Grammar, API, Domain-Specific Language (DSL), Spring Boot.**

I. INTRODUCTION

This application introduces a Domain-Specific Language (DSL) designed to revolutionize the development of Spring Boot applications. With a declarative and high-level approach, this DSL allows developers to create robust and scalable backends with significantly reduced effort. Through clear syntax, it is possible to define entities, relationships, validations, and controllers in a structured way, eliminating the need to write repetitive code and accelerating development time.

Key Features of the DSL:

- **Standardized Data Types:** Ensures consistency and reduces common errors in model definitions.
- **Integrated Validations:** Allows specifying validation rules directly within entity definitions.
- **Entity Relationships:** Supports complex associations, such as one-to-many relationships, in a simple and declarative manner.
- **Data Source Configuration:** Facilitates clear and centralized database configuration.
- **RESTful Controllers:** Automatically generates controllers aligned with RESTful best practices, based on JPA repositories.

The DSL utilizes JFlex (v1.9.1) and JCup (v11b-20160615) for syntax analysis and validation, ensuring a high level of accuracy and reliability. Additionally, its integration with GASB provides flexibility to adapt to changing and evolving software project requirements.

By eliminating redundancy and automating repetitive tasks, this DSL not only optimizes backend development but also allows developers to focus on business logic and value creation while maintaining best practices in Spring Boot application design.

II. FUNCTIONALITY

Automatic Code Generation:

- Automatically creates entity classes, JPA repositories, and RESTful controllers from declarative definitions.
- Reduces the need for writing repetitive and error-prone code.

Entity and Relationship Definition:

- Allows specifying data models with standardized types.
- Supports complex entity relationships, such as one-to-many and many-to-many associations.

Integrated Validations:

- Includes a dedicated section for defining validation rules directly within entities, ensuring data integrity.

Simplified Configuration:

- Eases the setup of data sources (databases) and other essential project parameters.

Automatically Generated RESTful Controllers:

- Generates RESTful endpoints based on JPA repositories, following API design good practices.

Integration with Existing Tools:

- Uses JFlex and JCup for DSL syntax analysis and validation.

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Maintaining Good Practices:

- Ensures that the generated code adheres to Spring Boot development standards, promoting project scalability and maintainability.

III. TARGET AUDIENCE

The DSL is aimed at software developers and technical teams working on backend application development with Spring Boot. Specific profiles include:

- **Backend Developers:** Professionals looking to optimize their workflow, reduce development time, and avoid common errors associated with manually writing repetitive code.
- **Agile Teams:** Teams that need to accelerate feature delivery without compromising code quality, allowing greater focus on business logic.
- **Software Architects:** Experts aiming to implement robust and scalable solutions while ensuring applications follow best design and development practices.
- **Startups and Companies with Limited Resources:** Organizations needing to develop high-quality applications with a small team and tight deadlines.
- **Students and Professionals in Training:** Individuals learning Spring Boot who want to understand how to structure backend applications efficiently using modern and declarative tools.
- **Projects with Changing Requirements:** Teams working in dynamic environments where project requirements evolve rapidly and need a flexible and adaptable solution.

IV. MODEL

The EASY API model is designed to streamline the process of API development by providing a structured and intuitive approach to translating high-level specifications into functional backend code. This model leverages a combination of nodes, templates, and automation to bridge the gap between API design and implementation. Below, we explore the key components and workflows of the EASY API model.

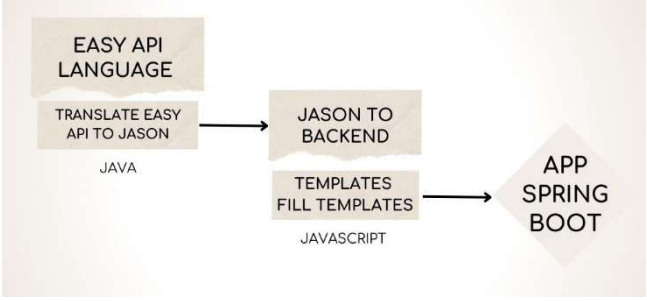

Fig. 1 Model for Easy API

At the core of the Esay API model is the *EASY LANGUAGE*, a domain-specific language that allows developers to define API specifications in a simple and human-readable format. This language abstracts away the complexities of traditional API development, enabling developers to focus on the logical structure and functionality of their APIs. The model includes a translation layer that converts *EASY LANGUAGE* specifications into structured data formats such as JSON. This translation ensures that the API design is machine-readable and can be further processed by the backend system. Also the Backend of Easy Api is build on Java a robust and widely-used programming language.

Next into the workflow, we have *JASON TO BACKEND* or GASB (Generate API with Spring Boot), a tool that translates the structured JSON representation of the API into a fully functional Spring Boot application. The structured JSON format, generated from the DSL, serves as the input for this stage. It includes details about entities, controllers, validations, and database configurations.

GASB uses predefined templates to generate Spring Boot components such as Entities, Controllers, Services and Database Configuration, this node automates the process of converting API specifications into executable Spring Boot code, significantly reducing development time and effort.

V. LEXICAL ANALYSIS

The system employs a multi-phase processing pipeline to convert Domain-Specific Language (DSL) input into structured Spring Boot code. Each stage in this pipeline plays a critical role in ensuring that the DSL code is accurately interpreted and transformed into executable Java code. The process begins with lexical analysis, followed by syntax parsing, and ultimately results in the generation of Spring Boot components such as entities, controllers, and validations.

The lexical analysis phase is the first step in the processing pipeline. It scans the DSL input and breaks it down into meaningful units called tokens. This phase is implemented using JFlex, a lexical analyzer generator for Java. JFlex allows the system to define patterns for recognizing keywords, identifiers, numbers, symbols, and other components of the DSL. The output of this phase is a stream of tokens

UNIVERSIDAD DISTRITAL FRANCISCO JOSÉ DE CALDAS

that is passed to the syntax parser for further processing. Also, uses a set of rules defined using regular expressions (regex) to match patterns in the text input. Each rule is associated with an action that generates a corresponding token. These tokens are then used by the syntax parser to build a parse tree.

Keywords:

- **Definition Keywords:** Words like AS, FOR, and IS are recognized as keywords and mapped to specific tokens.
- **Definable Keywords:** Keywords like ENTITY, CONTROLLER, CRUD, VALIDATIONS, and DATASOURCE define the structure and components of the DSL.
- **Constraints:** Keywords like NOT, NULL, EMPTY, GREATER_THAN, LESS_THAN, EQUALS_TO, and UNIQUE are used for validation and constraints.
- **Controller/CRUD Operations:** Keywords like GET_BY_ID, GET_ALL, UPDATE, DELETE, and SAVE define CRUD operations for controllers.
- **JPA Operations:** Keywords like GET_BY and AND are used for querying and filtering data.
- **Data Source Configuration:** Keywords like DBMS, IP, DATABASE, USERNAME, and PASSWORD are used for database configuration.
- **DBMS Types:** Specific database types like POSTGRESQL are recognized.
- **Data Types:** Keywords like STRING, INTEGER, LONG, FLOAT, and BOOLEAN define data types for entities.

Identifiers:

- Identifiers are recognized using the regex *[a-zA-Z_][a-zA-Z0-9_]**. These are mapped to the *IDENTIFIER* token.

Data:

- **Numbers:** Both integers and floating-point numbers (including negative values) are recognized using regex patterns.
- **String Literals:** Strings enclosed in double or single quotes are recognized and mapped to the STRING_LITERAL token.

Symbols:

- Symbols like { and } are recognized and mapped to their respective tokens (LBRACE and RBRACE).

Ignored Element:

- **Whitespace:** Spaces, tabs, and newlines are ignored.
- **Comments:** Single-line comments starting with // are ignored.

Error Handling:

- Any unrecognized character or pattern is flagged as an error using the ERROR token.

VI. SYNTAX PARSING

Once the lexical analyzer (JFlex) generates a stream of tokens, the next phase in the processing pipeline is syntax parsing. This phase is implemented using CUP (Constructor of Useful Parsers), an LALR (Look-Ahead Left-to-Right) parser generator for Java. The syntax parser ensures that the sequence of tokens adheres to the defined grammar rules of the DSL. It validates the structure of the DSL input and constructs a parse tree, which represents the hierarchical relationships between the tokens.

CUP uses context-free grammar rules to define the structure of the DSL. These rules specify how tokens should be combined to form valid constructs. The parser follows these rules to process the token stream and construct the parse tree. Below is an explanation of them:

- Program = DefinitionList
- DefinitionList = Definition | Definition DefinitionList
- Definition = EntityDef | ValidationDef | ControllerDef | DatasourceDef

The *Program* (non-terminal) represents the entire DSL input. It consists of a *DefinitionList*, which is a sequence of one or more *Definition* constructs.

- EntityDef = ENTITY IDENTIFIER LBRACE AttributeDef RBRACE
- AttributeDef = IDENTIFIER AS DataType | IDENTIFIER AS DataType AttributeDef
- DataType = STRING | INTEGER | LONG | BOOLEAN | FLOAT

An *EntityDef* defines an entity with attributes, wich are defined using the *AttributeDef* rule, which supports multiple of them, separated by commas.

- ValidationDef = VALIDATIONS FOR IDENTIFIER LBRACE ValidationExpr RBRACE
- ValidationExpr = IDENTIFIER IS PreConstrain | IDENTIFIER IS PreConstrain ValidationExpr
- PreConstrain = Constrain | NOT Constrain | UNIQUE
- Constrain = Comparation NUMBER | NULL | EMPTY
- Comparation = GREATER_THAN | LESS_THAN | EQUALS_TO

UNIVERSIDAD DISTRITAL FRANCISCO JOSÉ DE CALDAS

A *ValidationDef* defines validation rules for an entity. These are expressed using constraints like *NOT NULL*, *GREATER_THAN*, etc.

- ControllerDef = CONTROLLER FOR IDENTIFIER LBRACE ControllerExpr RBRACE | CRUD FOR IDENTIFIER
- ControllerExpr = ControllerAction | ControllerAction ControllerExpr
- ControllerAction = GET_ALL | GET_BY_ID | UPDATE | SAVE | DELETE | GET_BY Query
- Query = IDENTIFIER | IDENTIFIER AND Query

A *ControllerDef* defines API controllers, including *CRUD* operations and custom queries.

- DatasourceDef = DATASOURCE LBRACE DatasourceConf RBRACE
- DatasourceConf = DSdbms DSip DSdatabase DSusername DSpassword
- DSdbms = DBMS IS DBMS_SERVER
- DSip = IP IS STRING_LITERAL
- DSdatabase = DATABASE IS STRING_LITERAL
- DSusername = USERNAME IS STRING_LITERAL
- DSpassword = PASSWORD IS STRING_LITERAL

A *DatasourceDef* defines the configuration for the database connection. It includes details like DBMS type, IP address, database name, username, and password.

The parse tree is a hierarchical representation of the DSL input. Each node in the tree corresponds to a grammar rule or a token, capturing the relationships between different components of the DSL. For example, consider the following DSL input:

```
DEFINE User AS ENTITY {
    name AS STRING,
    age AS INTEGER
}
```

This input would look like this in his corresponding parse tree:

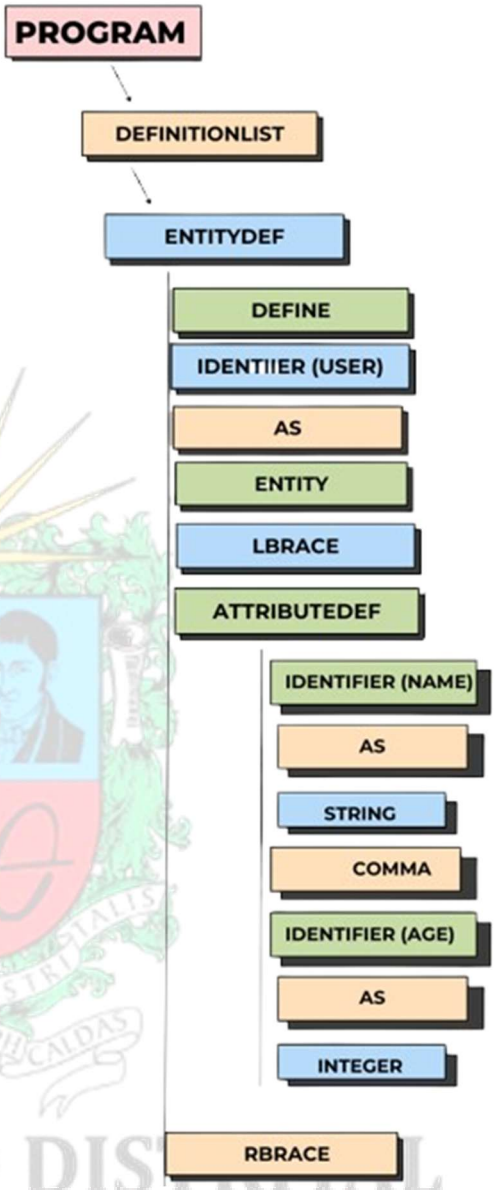

Fig. 2 Parse Tree

VII. CHALLENGES

Besides the robust backend and syntaxis this API has, It might be some troubles the same will experience into the future:

- **Ambiguity in Token Recognition**: Some input sequences may match multiple patterns, leading to ambiguity. For example, the sequence >= could be interpreted as two separate tokens (> and =) or as a single token (>=). The lexer must be designed to handle such cases correctly.

UNIVERSIDAD DISTRITAL FRANCISCO JOSÉ DE CALDAS

- **Handling Complex Patterns**: Certain patterns, such as nested comments or multi-line strings, can be difficult to recognize using simple regular expressions. The lexer must be equipped to handle these complexities.

- **Performance Considerations**: For large DSL inputs, the efficiency of the lexer becomes critical. Poorly designed lexers can become bottlenecks in the processing pipeline.

- **Maintaining Best Practices:** Ensuring that generated RESTful APIs follow evolving industry standards may require frequent updates. Integrating authentication, authorization, and API versioning dynamically within the DSL could be challenging.

- **Integration with Existing Systems:** The generated code might work seamlessly with existing Spring Boot projects, which may have different configurations, dependencies, or architectural styles. Compatibility issues might arise when integrating with third-party libraries or external services.

VIII. REFERENCES

1. S. S. Duarte. “GitHub - Checho019/GASB: Propuesta de ingeniería dirigida por modelos (MDE) para la creación automatizada de un back-end en SpringBoot”. GitHub. Accedido el 15 de febrero de 2025. [En línea]. Disponible: https://github.com/Checho019/GASB

2. S. Duarte. “GitHub - Checho019/dsl-api-generator: A Domain-Specific Language for creating a Spring Boot application using a high-level approach.” GitHub. Accedido el 15 de febrero de 2025. [En línea]. Disponible: https://github.com/Checho019/dsl-api-generator

3. “JFlex - JFlex The Fast Scanner Generator for Java”. JFlex - JFlex The Fast Scanner Generator for Java. Accedido el 15 de febrero de 2025. [En línea]. Disponible: https://jflex.de/

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS