DSL for web application development

Danijela Boberić Krstićev*, Danijela Tešendić*, Milan Jović*, Željko Bajić*
*Faculty of Sciences, University of Novi Sad, Serbia
dboberic@uns.ac.rs, tesendic@uns.ac.rs, milan.jovic@dmi.uns.ac.rs, zeljko.bajic@dmi.uns.ac.rs

***Abstract* – This paper addresses some issues of generative programing. The main aim of this paper is to present a DSL designed for developing web applications. It will be possible to easily generate whole web application by defining its functionalities in proposed DSL. Also, we will present possibilities to implement such DSL in Scala language. In addition, advantages of Scala language for this purpose will be discussed.**

I. INTRODUCTION

Domain specific language (DSL) is a programming language that's designed to solve a specific problem from some domain and it has limited expressive syntax. DSL offers a vocabulary that's specific to the domain it addresses. From the standpoint of an end user, domain specific language is a solution to his realistic problems. DSL provides a clear interface for all the requirements of the end user and should be intuitive for the user who is domain expert rather than programmer. Because nonprogrammers need to be able to use them, DSLs must be more intuitive to users than general-purpose programming languages need to be. Also, syntax of general-purpose programming languages is much richer than the one of DSL. DSLs are widespread no matter whether we brand them as DSLs or not. Some of the most commonly used DSLs are SQL, LaTeX, Ant, CSS or similar to them.

DSLs are mostly used to bridge the gap between business and technology and between stakeholders and programmers. DSLs speaks the language of the domain and provide a humane interface to target users. Domain of a DSL can also relate to specific field of software development. Software developers don't have to take a part in development of every aspect of software. For example, web developers don't have to be expert in parallelism and architecture to properly optimize software for modern heterogeneous hardware. In this case, concepts of parallel programing can be abstracted by DSL. This DSL can be further used by high-level developers.

Usage of DSLs can increase programmer productivity by providing extremely high-level abstractions tailored to a particular domain of software development. If we see development of an information system as a domain, we can develop DSLs, which will in an easy and simple way provide possibility to specify and generate some parts of system. During the development of various information systems it has been noticed that a large amount of time is spent on initial setup and development of basic functionalities. Furthermore, almost all information systems have a lot of similarities in their architecture. For example, if we talk about data layer, systems usually have database and persistence layer with entities, their relations and code lists. Usually, for all entities, it is necessary to provide input and binding data, modifying existing data and searching by various criteria. In the case of large systems, which contain over hundreds of entities, this work takes an enormous amount of time. Similar examples can be found in other layers of systems. Because of these uniformities in structure, we can talk about automation and code generation of some parts of information systems. Those features can be achieved by developing appropriate DSLs.

These DSLs can be useful to speed up initial setup of projects. By using DSLs, we can specify basic system functionalities and automatically generate programming code. This programming code represents skeleton code which can be further enhanced and refined. Also, these DSLs can be used for rapid development of system prototypes used during requirements analysis and design phases. Usage of prototypes simplifies gathering and discovery of requirements from stakeholders and improves quality of final software solution.

This paper addresses some issues of generative programing. The main aim of this paper is to present a DSL designed for developing web applications. It will be possible to easily generate whole web application by defining its functionalities in proposed DSL. Also, we will present possibilities to implement such DSL in Scala language. In addition, advantages of Scala language for this purpose will be discussed.

The presentation of this paper proceeds in 6 sections. At the beginning of the paper, we give brief overview of recent research in the field of DSLs used for specifying and implementing an information system as well as DSLs written in Scala language. Discussion on different approaches for implementing DSLs in Scala in given in Section 3. We continue by describing syntax of proposed DSL. Conclusion remarks as well as some plans for further research are presented in the last sections.

II. RELATED WORK

When an application and its components have well-defined structure and behaviour, developing process can be automated. Programs that perform that automation are called application generators. Application generators can be seen as compilers for DSLs used to define application's functionalities. Application generators supports code generation of applications with similar architecture, but with different semantics. Compared to traditional software development, generators offer a better potential for optimization and can provide better error-checking.

One approach to develop a web application is to start from a formal specification of an application and to generate application code. In this case, specification is used as an input for application generator. This approach is known as Model Driven Development. There are a lot of tools supporting this paradigm and usually they have graphical environment for application's specification. One example of those tools is AndroMDA [1] which transforms UML classes, use cases and state chart diagrams into deployable components for chosen platform. It can

generate code for multiple programming languages such as Java, .NET, PHP and many others. Similar tool is WebRatio [2] which uses Interaction Flow Modelling Language for modelling user interface of an application and generates fully functional application.

Also, it is possible to define database model of an application as a starting point for further code generation of the application. The authors of the paper [3] describes framework that fetches database metadata via JDBC and converts it into an XML document which is further transformed into HTML, XHTML or any other sort of web page using an XSLT. The same approach is used by ASP.NET Dynamic Data [4], but in comparison with previous solution it is more customisable. There are various templates of web pages that can be selected as the starting point of the new web application.

As we previously mentioned, development of a web application is mostly straightforward process. There have been many efforts that have focused on automation of that process by defining specific DSLs. Good example is WebDSL [5]. WebDSL is a DSL for developing dynamic web applications. Applications written in this DSL are translated to Java web applications. The WebDSL generator is implemented using Stratego/XT, SDF, and Spoofax. In addition, DSLs are used for creating mobile applications. MobiClaud DSL [6] is DSL designed for generating hybrid applications spanning across mobile devices as well as computing clouds. MobiCloud DSL closely resembles the MVC design.

Choice of general-purpose language for writing DSLs is substantially determined by language's built-in support. Ruby is an example of languages suitable for writing DSLs. Ruby on Rails [7] is the most popular DSL written in Ruby and designed for web development. Also, authors of the paper [8] combined several other DSLs written in Ruby to create web application. Scala is another language with good built-in support for writing DSLs. There is a lot of examples of DSLs written in Scala. Regarding that Scala allows programmers to create natural looking DSLs, we choose it for implementing DSL presented in this paper.

III. BUILT-IN SCALA SUPPORT FOR DSL DEVELOPMENT

DSLs can be classified regarding the way they are implemented. Generally speaking, DSLs can be divided into internal and external DSLs. Internal DSLs are also known as embedded DSLs because they're implemented within a host language. Host language has to be flexible and to offer the possibility of extension. An internal DSL uses same syntax as a host language. An internal DSL program is, in essence, a program written in the host language and uses the entire infrastructure of the host but it is limited to the syntactic structure of the host language.

A DSL designed as an independent language without using the infrastructure of an existing host language is called an external DSL. It has its own syntax, semantics, and language infrastructure implemented separately by the developer. External DSL is developed as an independent language with its own syntax. Unlike internal, external DSLs provide a greater syntactic freedom and the ability to use any syntax. With an external DSL, it is necessary to learn about parsers, grammars, and compilers, while implementing an internal DSL is just like writing API using facilities of a known language.

Before we delve further into describing syntax of our DSL, it is worth addressing the advantages of Scala language for developing embedded, as well as external DSLs. A number of books and papers have already discuses Scala's features and support for developing DSLs [9-12] and in this section we will summarise that. Scala is an object-functional language that runs on the JVM and has great interoperability with Java. Scala provides functional, as well as object-oriented, abstraction mechanisms that help in design of concise and expressive DSLs.

When an external DSL is designed, usually some external tools, like ANTLR are used. Based on grammar of external DSL, ANTLR will generate all necessary language implementation infrastructure. Another way to develop external DSLs is to use *parser combinators*. Scala offers an internal DSL which consist of library of *parser combinators* (functions and operators) that serves as building blocks for parsers. That internal DSL is used for designing external DSLs without usage of any external tools.

Although Scala supports development of external DSLs, it has even better support for developing embedded DSLs. First of all, Scala has a nice, concise and flexible syntax.

Scala provides flexibility through infix methods which allow to write *a x*, instead of *a.x*, where is *x* method of *a*. Also, some syntactic constraints like semicolon and parentheses are optional in Scala. Also, case classes in Scala enable usage of a shorthand notation for the constructor. It is not necessary to specify the keyword *new* while instantiating an object of the class. In addition, Scala's type inference is another factor that contributes to its conciseness. It is, for instance, often not necessary in Scala to specify the type of a variable, since the compiler can deduce the type from the initialization expression of the variable. Also return types of methods can often be omitted since they corresponds to the type of the body, which gets inferred by the compiler.

Next, Scala is object-oriented language in pure form, which means that every value is an object and every operator is method call. For example, expression *1 + 2* is actually invocation of method named + defined in class *Int*. Control flow statements are also expressed in terms of method calls. For example, the expression *if (c) a else b* is defined as the method call *__ifThenElse(c,a,b)*. As everything is method call in Scala, any built-in abstraction may be redefined by a DSL, just like any other method.

In Scala, functions are first-class values, and a function as an argument can be passed to yet another function. Also a function can return another function. Scala supports so called partial functions. Using partial functions, entire parameter list can be replaced with underscore mark. For example, rather than writing *println( )*, it is possible to write *println _*. It is possible to leave out the last argument of a function by declaring it to be implicit. The compiler will look for a matching argument from the enclosing scope of the function. Existing libraries in Scala can be extended without making any changes to them by using implicit type conversions. The implicit conversion function is automatically applied by the compiler.

All those features offer possibility to write DSLs in Scala which closely resemble natural language and they are sufficient for writing DSLs as pure libraries. Also, Scala offers some more options for developing embedded DSLs. Lightweight modular staging (LMS) [13,14] is a means of building new embedded DSLs in Scala and

creating optimizing domain-specific compilers at the library level. LMS generates code by build an intermediate representation of a program and translating nodes to their corresponding implementation in the target language. Currently, LMS generates Scala, C++ and CUDA code. A number of DSL projects [15- 17] are built in Scala by using LMS.

IV. FEATURES OF DSL FOR DEVELOPING WEB APPLICATIONS

In linguistics, of a particular language, syntax examines the rules that determine how words are combined into sentences. In computer science, the syntax is a set of rules that define the combination of symbols in order to get the correct language structure in a given language. If the syntax would not exist, we would have insignificant structures and we would be without any possibility of validating the language.

In this section, we present features of DSL for developing web applications. Proposed DSL is minimalistic and it strives to take after natural language. This DSL bears a resemblance to entity–relationship model (ER model). It is used to describe entities of an application. Entities have attributes of different types and among entities can exist relationships. Those concepts in the context of our DSL are described in following subsections. Based on entities' definition, DSL will generate database model, application's persistence data layer with CRUD operations on those entities, business layer with controller classes and presentation layer with web pages.

A. Dsl Data Types

Definition of data types used in this DSL is given in this subsection. This DSL supports following data types:

- *text[n]* - Represents the text data type of arbitrary length n. If the length of the text is omitted, the default value is 100. This type of data is equivalent to String in the programming language Scala.
- *number[n]* - This data type represents a numeric value. The parameter n specifies the number of decimal units and when it is present type is interpreted as Double. If the parameter n is omitted then it will be interpreted as Integer in Scala.
- *date* -It describes date and time values and it is equivalent to data type Date in Scala.

These data types are used in the construction of other elements of DSL.

B. Entity

An entity is an abstraction of some aspect of the real world that can be uniquely identified and is capable of an independent existence. Keyword *entity* is used to describe a group of data that has common attributes. In order to define an entity it is necessary to declare name of the entity and list of its attributes. Every attribute has name and data type separated by column. An example of an entity is shown in Listing 1, where we have an entity *Person* which has attributes: *Name*, *Surname* and *Date of birth*. Data type of attributes *name* and *surname* is *text* (implicit length is 100 characters), and data type of attribute *Date of birth* is *date*.

```
entity("Person", "Name": text, "Surname":
text, "Date of birth": date)
```

Listing 1. – Entity example

A. Relationships

Relationships capture how entities are related to one another. In our DSL, keyword *relationship* is used to describe the relationship between entities. This construction consists of entities' names and definition of visibility among them. Visibility can have value from the following set: *include*, *field*, *none*. Value *include* is interpreted that instance of the first entity has connection with more instances of the second entity. Value *field* means that instance of the first entity has connection with only one instance of the second entity. Value *none* means that instance of the first entity has no connection with any instance of the second entity. An example of this construction is given in Listing 2. that describes relationship between entities *Artist* and *Album*. Meaning of this relationship is that entity *Artist* references several entities *Album*, and that entity *Album* reference only one entity *Artist*.

```
relationship("Artist", "Album", {include,
field})
```

Listing 2. – Relationship example

B. Relationships With Attributes

Like entities, relationships can also have attributes and this concept is supported with our DSL. This is an extension of *relationship* construction. To support this concept, firstly, it is necessary to define additional entity that will contain the new attributes associated to relationship. The name of that entity is further used in relationship construction. In this construction, name of the new entity is grouped with definitions of visibility among entities. Definition of visibility has the same meaning as in relationship construction without attributes. An example of this kind of relationship is shown in Listing 3. In this example we have a new entity *Song award* which contains information when the award was given and this entity is used in relationship between entities *Song* and *Award*. Visibility between entities *Song* and *Award* is defined with values *include* and *none*. This means that *Song* reference several entities *Award* and *Award* doesn't have reference to entity *Song*.

```
entity("Song award", "Award received": date)
relationship("Song", "Award", {"Song award",
include, none})
```

Listing 3. – Relationship with attributes example

C. Codebooks

Codebooks are simple entities that don't have relationship with other entities. Other entities just use values from codebooks. Keyword *codebook* is used to define this concept in our DSL. Definition of codebook contains name of codebook and its attributes, which is similar to entities' definition. Relationship with entities is defined by relationship construction but visibility is defined only for entity's side. An example of this concept is presented in Listing 4. Meaning of this example is that entity *Artist* has connection to one instance of codebook *City*.

```
codebook("City", "Name": text)
relationship("Artist", "City", field)
```

Listing 4. –Codebook example

D. Example Of Web Application Specification

In order to better explain usage of proposed DSL, an example of a specification of an simple web application in

that DSL is presented in Listing 5. This application manages music artists and their work and tracks awards which they have received. Artists and their work are defined with entities *Artist*, *Album*, *Song* and corresponding relationships. Regarding the fact that artist can represent musical band, membership to the band is defined with entities *Artist*, *Person*, *Member* and corresponding relationships. Awards received by artists are defined with entities *Artist*, *Award*, *Song award* and corresponding relationships.

```
entity("Artist", "Name": text[60], "Formed":
number, "Active
till":number,"Website":text[100],
"Description": text[2048])
entity("Album", "Title": text[100],
"Production": text[100], "Released": date)
entity("Song", "Song number": number, "Title":
text, "Time": number, "Song text": text[500])
entity("Person", "Name": text, "Surname":
text, "Date of birth": date, "Gender": text)
entity("Award", "Name": text, "Establish
date": date)
entity("Member", "Role": text[100], "Member
since": date)
entity("Song award", "Award received": date)
relationship("Artist", "Album", {include,
field})
relationship("Album", "Song", {include,
field})
relationship("Artist", "Person", {"Member",
include, field})
relationship("Song", "Award", {"Song award",
include, none})
relationship("Artist", "Genre", include)
relationship("Artist", "City", field)
codebook("City", "Name": text)
codebook("Genre", "Name": text)
```

Listing 5. – Web application specification

V. ARCHITECTURE OF GENERATED WEB APPLICATION

The first step in developing DSL is to understand problem domain and according to that to define vocabulary and grammar of new DSL. In the next step it is necessary to choose host language as well as technologies which will be used in implementation of the main concepts of DSL.

In this paper, we presented syntax of our DSL for developing web applications and the next phase in our research is to implement it. In this section we are giving a brief overview how we plan to do that.

Taking into consideration advantages of Scala language for development of DSLs, we chose Scala for implementation of our DSL. We will implement our DSL as embedded DSL. We are planning to embed our DSL in Scala using Lightweight Modular Staging, which will provide a common intermediate representation and basic facilities for optimization and code generation.

Also, generated web applications will be based on Scala web technologies. Generated web applications will have multi-tier architecture consisting of a database layer, persistence layer, business layer, service layer and presentation layer. We will use PostgreSQL database management system for storing data. Persistence layer will be implemented using framework Slick [18] which is a modern database query and access library for Scala. Other layers of application will be implemented using Play web framework [19] which follows MVC architectural pattern.

VI. CONCLUSION

In this paper we presented features of DSL for developing web applications. Proposed DSL is minimalistic and it strives to take after natural language. This DSL bears a resemblance to entity–relationship model. It is used to describe entities of an application. Based on entities' definition, DSL will generate database model, application's persistence data layer with CRUD operations on those entities, business layer with controller classes and presentation layer with web pages.

Presented DSL will be implemented as an embedded DSL in Scala. Paper also gave overview of Scala's built-in support for developing DSLs.

ACKNOWLEDGMENT

This work is partially is supported by the Swiss National Sceince Foundation (SCOPES project IZ74Z0 160453/1, Developing Capacity for Large Scale Productivity Computing).

LITERATURE

[1] AndroMDA Model Driven Architecture Framework, http://www.andromda.org/

[2] WebRatio, http://www.webratio.com/

[3] Elsheh, Mohammed M., and Mick J. Ridley. "Using database metadata and its semantics to generate automatic and dynamic web entry forms in." In *Proceedings of the World Congress on Engineering and Computer Science 2007 WCECS 2007*. 2007.

[4] ASP.NET Dynamic Data, https://msdn.microsoft.com/en-us/library/ee845452.aspx

[5] Visser, Eelco. "WebDSL: A case study in domain-specific language engineering." In *Generative and Transformational Techniques in Software Engineering II*, pp. 291-373. Springer Berlin Heidelberg, 2008.

[6] Ranabahu, Ajith H., Eugene Michael Maximilien, Amit P. Sheth, and Krishnaprasad Thirunarayan. "A domain specific language for enterprise grade cloud-mobile hybrid applications." In *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11*, pp. 77-84. ACM, 2011.

[7] Ruby on Rails, http://rubyonrails.org/

[8] Günther, Sebastian. "Multi-dsl applications with ruby." IEEE software 5 (2010): 25-30.

[9] Odersky, Martin, Lex Spoon, and Bill Venners. Programming in Scala. Artima Inc, 2010.

[10] Ghosh, Debasish. DSLs in action. Manning Publications Co., 2010.

[11] Rompf, Tiark, Nada Amin, Adriaan Moors, Philipp Haller, and Martin Odersky. "Scala-virtualized: Linguistic reuse for deep embeddings." Higher-Order and Symbolic Computation 25, no. 1 (2013): 165-207.

[12] Moors, Adriaan, Tiark Rompf, Philipp Haller, and Martin Odersky. "Scala-virtualized." In Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation, pp. 117-120. ACM, 2012.

[13] Lightweight Modular Staging, https://scala-lms.github.io/

[14] Rompf, Tiark, and Martin Odersky. "Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs." Communications of the ACM 55.6 (2012): 121-130.

[15] Brown, Kevin J., Arvind K. Sujeeth, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. "A heterogeneous parallel framework for domain-specific languages." In Parallel Architectures and Compilation Techniques

(PACT), 2011 International Conference on, pp. 89-100. IEEE, 2011.

[16] Sujeeth, Arvind, HyoukJoong Lee, Kevin Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand Atreya, Martin Odersky, and Kunle Olukotun. "OptiML: an implicitly parallel domain-specific language for machine learning." In Proceedings of the 28th International Conference on Machine Learning (ICML-11), pp. 609-616. 2011.

[17] Vogt, Jan Christopher. "Type safe integration of query languages into Scala." PhD diss., Diplomarbeit, RWTH Aachen, Germany, 2011.

[18] Slick, http://slick.typesafe.com/

[19] Play, https://www.playframework.com/