**Building an Domain Specific Language for a micro-Service Oriented Architecture problem**

A File Paper

Presented to

**Department of Computer Science**

**Midwestern State University**

In Partial Fulfillment

Of the Requirements of the Degree

**Master of Science**

By

**Krishna Bakka**

August, 2016

**Building an Domain Specific Language for a micro-Service Oriented Architecture problem**

by

**Krishna Bakka**

Approved:

Chair, Graduate Advisory Committee

Member

Member

Program Chair

Date of Approval

**Abstract**

Domain specific languages (DSL) are programming languages that are specially designed to solve a particular problem in a specific area. Embedded DSLs, eDSLs, are comparatively easy to implement and are usually considered as an extension for a host language. The host language must support a variety of language constructs to make the eDSL easy to program. This is because one cannot get all the required resources from a single language, as they are built upon a single programming paradigm, such as functional, object oriented, imperative or logic paradigm. Hybrid languages come in handy when building better eDSLs. The purpose of my research is to construct an eDSL using Scala as host the language. The eDSL is related to the domain of service oriented architecture. By using this eDSL, the user can generate the skeletal structure of service components by simply writing the scripts in the eDSL. This DSL can become a powerful tool in the hands of a domain expert and can precisely control the software development without having much expertise in programming.

**Contents**

1. Introduction

2. What is a DSL

2.1. Types of DSLs

2.2. Design patterns in DSLs

2.3. Comparison of design patterns of DSLs

3. Understanding service oriented architectures

3.1. Components of service oriented architectures

4. Structure of the proposed DSL

4.1. Scala language

4.2. Implementation details

4.2.1. Service trait

4.2.2. Module trait

4.2.3. Action trait

4.2.4. Input trait

4.2.5. Output trait

4.2.6. Data Object

4.2.7. CodeGenerator

5. Conclusion

6. Future work

7. References

**1. Introduction**

A domain specific language, DSL, is a special purpose programming language specifically designed to solve problems in a particular domain. They are different from general-purpose languages in several ways. First, their syntax is restricted to a domain vocabulary. Additionally, their functionality is restricted and the overall language is light. Moreover, they are not used to solve problems in other domains. They are more advanced and elegant than the general-purpose languages in solving problems related to the specific domain.

The core of this paper is divided into two parts. In the first part, literature on DSLs is reviewed and mainly focuses on DSLs, understanding the design patterns, comparing and analyzing the design patterns. The second part covers the implementation of DSL in Scala. It consists of descriptions for design pattern, implementation approach, the techniques and tools used and the problems solved.

**2. What is a DSL?**

Domain specific languages, DSLs, are tailored to solve a specific problem related to the domain. For example, Lex, used to develop lexical analyzers, and Yacc, used to build parsers, are also domain specific languages and are useful in developing DSLs. DSLs are more concise, can be written quickly and are easier to maintain. The most important fact is that they can be used by a non-programmer, who is an expert in a particular domain. The domain expert already knows the semantics of the domain well and he just needs the syntax provided to express the semantics. This reduces the gap between the developer and the user, which is a major cost. DSLs are easy to learn and they are very powerful in developing applications in a particular domain when compared to general-purpose languages. For example, HTML in developing web application, Yacc tool in developing parsers and SQL for querying the relational databases.

Developing a DSL requires three steps: defining the domain, designing the DSL which implements the domain semantics accurately and building some tools to support the DSL by which an infrastructure is formed [3].

**2.1. Types of DSL**

There are mainly two types of DSLs: internal and external. Internal DSLs are built upon another language, more often a general purpose language. These internal DSLs inherit the tools and other resources of the base language and hence they are easy to build. The main disadvantage is that they are restricted by the syntax of the base language and as a result, sometimes it becomes very hard to express the domain vocabulary more concisely. On the other hand, external DSLs are built from the scratch. They have their own parsers and compilers and thus will have greater independence in language design. The only disadvantage is that they are relatively hard to build, since everything has to be built from scratch.

**2.2. Design patterns in DSLs**

Several important design patterns in DSLs exist.

The Piggyback is a structural pattern, which uses a base high-level language/general purpose language, to build a DSL. In this pattern, a DSL can inherit the useful characteristics in the form of linguistic elements such as expression handling, variables, or compilation. The DSL language processor passes the linguistic elements that are expressed in the base language, to the language processor of the base language. Examples of this pattern are Yacc and Lex DSLs. The input grammar and input strings are expressed in a DSL that are then converted into C language.

In the pipeline pattern, multiple DSLs are represented. The DSL will get the input from the other DSL and its output may become the input for the other DSL. In a complex domain, each simple structure/part is represented by a single DSL.

Lexical processing is a creational pattern, a design pattern that deals with creation mechanism. In this pattern, a DSL is prepared by using simple lexical substitution, without tree based syntax analysis. This is used together with the piggyback pattern. This pattern offers the simplest way to build DSLs. Language extension is another good creational pattern and it proposes the extension of already existed language within its syntactic and semantic framework. Language specialization is also a creational pattern and works by removing the unnecessary specifications of a base language, so that it serves the purpose of a specific domain. Data structure representation is another creational pattern. Complex data structures can be represented using the DSLs. This minimizes the chances of initializing the data structures with wrong or inconsistent data as the DSL compiler or translator checks for errors before passing it to base language.

Source to source transformation is a pattern that allows the efficient use of DSL translators. By using various tools, the source code of a DSL is directly translated into source code of the base language. The tools include compilers optimizers, linkers and native code instruction schedulers. Utilizing this pattern makes troubleshooting easy because the resulting code is easy to read and reason about.

System front end is a pattern that is used for building interface DSL between the complex software system and the users. As the time passes, the complex systems are updated with additional features and implementations by which it becomes less user friendly. In such cases, a DSL is used for programming such complex systems. It acts as a front end for that system. The systems configuration parameters and variables are exposed as the elements of the DSL. DSLs can also be used to provide communication between different systems [4].

**2.3. Comparison of DSL design patterns**

Comparing various implementations of a DSL is a complex task, since they are implemented for different domains for different problems. Five different categories of implementation patterns of DSL are present. The first category is preprocessing, which includes macro-processing where domain constructs are defined by using base language constructs; source-to-source transformation, where the domain source code is translated into base language source code; pipeline, where a series of DSLs is used to solve domain problems; and lexical processing where lexical scanning is done rather than tree-based analysis. The second implementation category is embedding, where a base language is used to build a library of domain-specific operations. The base language's compiler or interpreter is used. The third implementation category is compiler generator, where compiler generator tools are used to generate the compiler of a DSL. The fourth implementation is extensible compiler/interpreter, where a base language compiler is extended by using domain specific optimization rules or domain-specific code generation. The fifth implementation category is commercially off-the-shelf, COTS, where existing tools or notations are used to solve the problems in a domain, for example, XML is used in processing querying documents.

Comparison validity, context of the study, measurement framework and presentation of key findings that focus on controversial problems can be used for comparing two DSLs.

The implementer's effort is compared on the basis of the number of lines of code (LOC), that has to be written by the implementer. From the LOC point of view, the increasing order of the implementers effort is implementation by embedding, source-to-source, extensible compiler, compiler generator and COTS. Macro processing, compiler generator and extensible compiler are good implementation techniques for converting an application programming interface, API, into a DSL.

The end user effort in writing a program and debugging it is in the increasing order of interpreter and extensible compiler, compiler generator, macro processing and embedded, source-to-source and then COTS [3].

**3. Understanding Service Oriented Architectures**

The service-oriented architecture is a new software paradigm based on an old concept of applications communication through a network. These software architectures are very useful in some business domains, where services are essential tasks, like banking and ticket booking software. Conventional software is a collection of classes and functions that are tightly coupled. The components in such software talk to each other through function calls. Additionally, the whole software package works as a single entity and is usually called a monolith application. In such software architectures, there will be less scope for extensibility, interoperability and modification. Code reuse is another problem that conventional software has and such software will not support the reuse of its code in another application [2]. Service oriented architectures (SOA) are useful in the above-mentioned scenarios. SOA is not a way of implementing the software, but it is a way of thinking about where every functionality is conceived as service. Therefore, the basic abstraction in SOA is the service. Services are built in such a way that they are aware of their work and communicate with other services over a network without any complications. Sometimes legacy systems can be wrapped with some kind of code that makes them work as services. Their API is exposed as services to other services, thus integrating the legacy system software into the newer ones [5]. SOA helps businesses by streamlining the IT infrastructure. SOA's flexibility is most suitable for today's competitive and ever changing businesses. Business needs change every day and SOA's can be modified accordingly by simply adding or deleting services.

**3.1. Components of Service-Oriented Architecture**

The components of SOA are services and the enterprise service bus. Services are the fundamental building blocks of SOA. Services are an abstract way to represent some business functionality such as retrieving a bank account details, making a transaction, etc. They work to change the state of data from one state to another. The important characteristics of services are: they are loosely coupled, they are limited by contract, they are abstract, they are reusable, they are to be composable and by combining multiple services they have to be able perform complex function, a service must be stateless, attending one service request must not influence another service request and to make use of a service, it has to be discoverable by other services.

The enterprise service bus, ESB, acts a backbone for SOAs. It plays a central role in SOA by controlling all the services and all services communicate through the ESB. The ESB provides a registry of metadata for all the services. Each service has its own metadata by which other services are able to find it. Interoperability is the main goal behind the ESB and it has to support heterogeneous systems [6]. The functions of ESB are routing, security, reliability and service management. The ESB cleverly routes the service requests and includes prioritizing services and requests, balancing the loads and environment dependant routing. The ESB manages the authentication and authorization of services and protect the data. The ESB takes necessary steps when a service request is lost before it reaches the necessary service or if there is a mis-communication between the services. If necessary, it must be able to send a message on behalf of a service. TheESB manages all the services. It adds, modifies or deletes the services. It monitors, logs and maintains the overview of all the services and it oversees the performance of the services and resolves the issues.

**4. Structure of the proposed DSL**

Since the implementation of a DSL for a whole SOA is a complex task, it is out of scope for small academic projects. Therefore, a DSL for a small part of an SOA is built. By using this DSL, the domain expert can describe precisely and programmatically the structure of the service. The user could be a business analyst or a product owner who does not need to know intricate technical details of technology stack that a generated service is going to be implemented. The main idea behind the DSL is to give the domain expert more control over the software development. The domain expert just writes the service structure in the domain language and the DSL creates the skeletal structure of the service which consists of the all the classes and methods involved in the service. To achieve this, the user has to provide the names of the classes and methods, the names can be chosen from the domain vocabulary. The DSL, initiates a complete stand-alone service application with empty classes in it which the developer will implement later. Thus, the domain expert gives the description and functionality of the DSL and the implementation of the logic will be done by the developer. This reduces the communication problems between the domain experts and developers. The name of the proposed DSL is Micro-services DSL, MicsDSL. A service described in MicsDSL generates a service targeting the Play framework. Play framework is a popular open source MVC (Model-View-Controller) framework that is targeted to Java and Scala languages. This service can be started from the command line. Once started it acts as a typical web service, which can be communicated from a web browser. The DSL describes modules of a target service. Each module corresponds to a controller class in Play service application. Controller classes control the data flow between the model and view. It mediates between the model and view classes.

For each module, the DSL describes the actions. Each action corresponds to a service end-point. This action translates into Controller's Action methods. Controller actions direct the state of the controller modules.

Each action expects input parameters that are given by the "input" methods while the "return" method determines the return type.

For example, if the user wants to describe services for a Library management service, he would describe the service in the manner shown in figure-1.

*service("Library") defines "Library service application" at "org.library.dsl" having modules (*

*module("User") defines "User management" having actions (*

*add("user") to "Register a new user" takes (*

*input("userName") as "UserName" carries "Member's user name",*

*input("password") as "Password" carries "user password",*

*input("address") as "Address" carries "user address"*

*) and returns ("User account" as "User"),*

*update("user") to "Modify user account details" takes (*

*input("userName") as "UserName" carries "Member's user name",*

*input("address") as "Address" carries "Member's new address"*

*) and returns ("Updated user" as "User")*

*),*

*module("Catalog") defines "Books and Magazine catalog management" having actions (*

*add("item") to "Add a new catalog item" takes (*

*input("item") as "CatalogItem" carries "Catalog item"*

*) and returns ("Catalog item id" as "CatalogItemDetails"),*

*delete("item") to "Remove a catalog item" takes (*

*input("itemId") as "CatalogItemId" carries "Catalog item id"*

*) and returns ("Removed catalog item id" as "CatalogItemId")*

*)*

*)*

*Figure-1. Client description of the application.*

The DSL script in figure-1 generates a play application called Library. The modules create a new user class with add and update methods in it. The add method takes username, password and address as parameters and returns the user account while update takes username and address as parameters and returns updated user account. The second module creates a catalog of books and magazines with add and delete as the methods. The *Add* method takes "item" as a parameter and returns catalog item details. In the same way, the delete method takes "itemId" as a parameter and returns a modified catalog.

**4.1. Scala language**

Scala is an acronym for "Scalable language" and is pure object oriented language with full blown functional features like first class functions and immutability of data structures. It runs on Java virtual machine, JVM and seamlessly integrates with Java libraries and programs. It has flexible syntax like optional dots in method invocation, optional semicolons, infix operators, optional parenthesis, functions as first class values, open classes through implicit construct, type inference, operators as methods and case classes. Scala is suitable for developing DSLs due to its conciseness. For example, the code snippet figure-2 displays some of the above-mentioned features that support DSL development.

*val**myService = service("Library") defines "Library service application" at "org.library.dsl" having modules (  
  
 module("User") defines "User management" having actions (  
  
 add("user") to "Register a new user" takes (  
 input("userName") as "String" carries "Member's user name",  
 input("password") as "String" carries "user password",  
 input("address") as "Address" carries "user address"  
 ) and returns ("User account" as "User"),*

*Figure-2. Features of Scala.*

From the code snippet shown in figure-2, it can be inferred that a service class can be invoked without using the '*new'* keyword, *'defines'* is a method that is called without using dot operator and methods can be chained. Value *'myService'* is a service object and is inferred automatically without being typed. It can also be inferred that arguments may not be surrounded by parenthesis and modules is method that takes functions, module, as arguments. Moreover, there can be a variable number of arguments.

**4.2. Implementation details**

This section covers the implementation details of the proposed DSL. It briefly explains the Scala units that are used to achieve the current functionality and additionally, some Scala features that were used to build the DSL are discussed in between. The files containing all the code are in the appendix.

**4.2.1. Service trait**

Traits are similar to interfaces in Java. Classes *NameService*, *DescribedService*, *PackagedService* and *ServiceWithModules* inherit *Service* trait. All these classes collect the details of the service user wanted to build. *NamedService* is used to get the service name, *DescribedService* for getting the description of the service and so on. Here method chaining is achieved by designing the methods in the classes that returns another class. The structure of these classes is shown in the figure-3.

***case class*** *NamedService(serviceName: String)* ***extends*** *Service {* ***def*** *defines(serviceDescription: String): DescribedService =  
 DescribedService(serviceName, serviceDescription)  
}****case class*** *DescribedService(serviceName: String, serviceDescription: String)* ***extends*** *Service {* ***def*** *at(packageName: String): PackagedService =  
 PackagedService(serviceName, serviceDescription, packageName)  
}*

*Figure-3. Classes of Service trait.*

For example 'defines' is a method that is defined in the NamedService class, returns DescribedService. So now method 'defines' and method 'at' can be chained and can be called as shown in figure-4.

*NamedService.defines("service name ").at(" location")*

*Figure-4. Method chaining.*

Finally, the above statement can be written as shown in figure-5.

*NamedService defines " service name" at "location"*

*Figure-5. Method chaining without dot operator.*

Case classes are used instead of classes. Case classes are same as regular classes but can be instantiated without using the *new* keyword and are usually used to store data.

Theclass *ServiceWithModules* will store the service name, service description, package name and a sequence of modules.

**4.2.2. Module trait**

Each module mentioned above will collect data related to modules and the type of actions related to each module. There can be any number of modules in a service, like a user related module that contains adding the user, editing the user and deleting the user and books related module like catalog module that might contain any number of actions like getting the book, searching the catalog, deleting the catalog, and so on.

*NamedModule* gets the module name, *DescribedModule* gets the module description and implements the *module* trait. Finally, *ModuleWithActions* stores all module information like name of the module, description of the module and the sequence of *Actions* of each module as shown in figure-6.

***case class*** *ModuleWithActions(moduleName: String, moduleDescription: String, actions: Seq[Action])*

*Figure-6. Class ModuleWithActions.*

**4.2.3. Other traits**

Actions are like methods in a class. They change the state of the module by adding the user to the user module, updating the registry and so on. Like methods, each action accepts parameters and returns some data.

The parameters or input data is collected and stored using the classes derived from *Input* trait. *NamedInput* collects the input name while *TypedInput* collects the type of the input.

Classes that collect the output or return data is stored in classes derived from *Output* trait.

**4.2.6. Data Object**

When methods and values are not associated with any instance of a class, the entity is called a singleton object. They are comparable to Java static classes. In this *Data* object, all the collected data is sorted into small data units. Implicit classes are used to extend the classes without using any relationship like inheritance, association and so on.

This implicit class can be used in the following way as shown in the figure-7. They are a way to extend a class without using any inheritance.

***implicit class*** *IServiceData(service: Service) {* ***def*** *data = service* ***match*** *{* ***case*** *ServiceWithModules(name, description, packageName, modules) =>  
 Some(ServiceData(name, description, packageName, modules))* ***case*** *\_ => None  
 }  
}*

*Figure-7. An implicit class.*

Here method *data* is behaving as if it is a method of class or Object service as shown in figure-8.

*service.data* ***match*** *{* ***case*** *Some(ServiceData(serviceName, serviceDescription, packageName, modules)) =>  
 createProject(serviceName)*

*Figure-8. An implicit class method.*

**4.2.7. CodeGenerator**

The *CodeGenerator* class uses the Data Object to generate all the required classes and files. The Twirl template engine generates the code template and Java libraries are used to write them to the files. Additionally, this class initiates a new service play application and starts running it.

**5. Conclusion**

DSLs are programming languages that are specially designed to solve a problem in a particular domain. There are two types of DSLs and they are internal and external DSLs. Internal DSLs are built up on the a base language while the external DSLs are built from scratch.

There are several design patterns used in building a DSL.DSLs can be very handy in some cases, while in others are of no use. Therefore, the developer has to decide whether a new DSL has to be built or not. If it is needed, he has to analyze the domain problem and available resources and decide what kind of DSL has to be built and what design pattern has to be followed.

Service-oriented architecture, SOA, is a new paradigm where service is the fundamental abstraction and different services communicate over a network. These software architectures are very useful in some business domains, particularly for those where services are fundamental like banking and ticket booking software. SOA's are hard to build and some ready-made models or designs are available in the market to relieve the burden of architects.

This internal DSL in SOA, MicsDSL, will empower the domain expert and enable him to communicate his requirements more clearly to the software developer. Using this DSL, a domain expert will be able to generate the skeletal structure of a service. The generated service contains only class names and the method names in it. The implementation of the these classes can be done by a software developer.

**6. Future work**

The future work is mainly focused on extending the features of the proposed DSL. At present, the DSL is just able to create the controllers and routing files in the application. Therefore, the future work is aimed at developing the DSL in such a way that it can generate the skeleton of the classes and documentation.

**7. References**

[1] Debasish, G., *DSLs in action*, Manning Publications Co. Stamford, 2011.

[2] Gold, N., Mohan, A., Knight, C., Munro, M., "Understanding Service-Oriented Software, IEEE Software, to appear.

[3] Hudak, P., *"Building Domain-Specific Embedded Languages"*, ACM Computing Surveys, 24(4es), 1996.

[4] Spinellis, D., *"Notable design patterns for domain-specific languages",* The Journal of Systems and Software, 56 (2001) 91-99.

[5] Sprott,D., Wilkes,L., "Understanding Service-Oriented Architecture, CBDI Forum, January 2004, https://msdn.microsoft.com/en- us/library/aa480021.aspx

[6] Vegter, W., "Critical success factors for a SOA implementation", 11th Twente Student Conference on IT, Enschede, University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science, June 29th, 2009.

Appendix 1: Client.scala

***package*** *mics.dsl.client****import*** *mics.dsl.generator.CodeGenerator****import*** *mics.dsl.service.Action.\_****import*** *mics.dsl.service.Input.\_****import*** *mics.dsl.service.Module.\_****import*** *mics.dsl.service.Output.\_****import*** *mics.dsl.service.Service.\_****object*** *Client* ***extends*** *App {* ***val*** *myService = service("Library") defines "Library service application" at "org.library.dsl" having modules (  
  
 module("User") defines "User management" having actions (  
  
 add("user") to "Register a new user" takes (  
 input("userName") as "String" carries "Member's user name",  
 input("password") as "String" carries "user password",  
 input("address") as "Address" carries "user address"  
 ) and returns ("User account" as "User"),  
  
 update("user") to "Modify user account details" takes (  
 input("userName") as "String" carries "Member's user name",  
 input("address") as "Address" carries "Member's new address"  
 ) and returns ("Updated user" as "User"),  
  
 get("user") to "get a user" takes (  
 input("id") as "String" carries "User id"  
 ) and returns ("User information" as "User")  
 ),*

*module("Catalog") defines "Books and Magazine catalog management" having actions (  
  
 get("item") to "Get an existing catalog item" takes (  
 input("itemId") as "String" carries "Item id"  
 ) and returns ("Catalog item" as "CatalogItem"),  
  
 add("item") to "Add a new catalog item" takes (*

*input("item") as "CatalogItem" carries "Catalog item"  
 ) and returns ("Catalog item id" as "String"),  
  
 delete("item") to "Remove a catalog item" takes (  
 input("itemId") as "CatalogItemId" carries "Catalog item id"  
 ) and returns ("Removed catalog item id" as "String")  
 )  
 )  
  
 CodeGenerator(myService, "C:\\play\\activator\\projects").generateCode  
}*

Appendix:2 Service.scala

***package*** *mics.dsl.service****sealed trait*** *Service****case class*** *NamedService(serviceName: String)* ***extends*** *Service {* ***def*** *defines(serviceDescription: String): DescribedService =  
 DescribedService(serviceName, serviceDescription)  
}****case class*** *DescribedService(serviceName: String, serviceDescription: String)* ***extends*** *Service {* ***def*** *at(packageName: String): PackagedService =  
 PackagedService(serviceName, serviceDescription, packageName)  
}****case class*** *PackagedService(serviceName: String, serviceDescription: String, packageName: String)* ***extends*** *Service {* ***def*** *having(f: (String, String, String) => ServiceWithModules): ServiceWithModules =  
 f(serviceName, serviceDescription, packageName)* ***def*** *withModules(modules: Module\*): ServiceWithModules =  
 ServiceWithModules(serviceName, serviceDescription, packageName, modules)  
}****case class*** *ServiceWithModules(serviceName: String, serviceDescription: String, packageName: String, modules: Seq[Module])* ***extends*** *Service****object*** *Service {* ***def*** *service(serviceName: String): NamedService = NamedService(serviceName)* ***def*** *modules(moduleList: Module\*)(serviceName: String, serviceDescription: String, packageName: String): ServiceWithModules =  
 ServiceWithModules(serviceName, serviceDescription, packageName, moduleList)  
}*

Appendix:3 Action.scala

***package*** *mics.dsl.service****sealed trait*** *Action****case class*** *NamedAction(actionName: String, actionType: ActionType = Get)* ***extends*** *Action {* ***def*** *to(actionDescription: String): DescribedAction =  
 DescribedAction(actionName, actionType, actionDescription)  
}****case class*** *DescribedAction(actionName: String, actionType: ActionType, actionDescription: String)* ***extends*** *Action {* ***def*** *takes(inputs: Input\*): ActionWithInputs =  
 ActionWithInputs(actionName, actionType, actionDescription, inputs)  
}****case class*** *ActionWithInputs(actionName: String, actionType: ActionType, actionDescription: String, inputs: Seq[Input])* ***extends*** *Action {* ***def*** *and(f: (String, ActionType, String, Seq[Input]) => ActionWithOutput): ActionWithOutput =  
 f(actionName, actionType, actionDescription, inputs)  
}****case class*** *ActionWithOutput(actionName: String, actionType: ActionType, actionDescription: String, input: Seq[Input], output: Output)* ***extends*** *Action****object*** *Action {* ***def*** *get(endPointName: String): NamedAction =* ***new*** *NamedAction(endPointName, Get)* ***def*** *add(endPointName: String): NamedAction =* ***new*** *NamedAction(endPointName, Post)* ***def*** *update(endPointName: String): NamedAction =* ***new*** *NamedAction(endPointName, Put)* ***def*** *delete(endPointName: String): NamedAction =* ***new*** *NamedAction(endPointName, Delete)* ***def*** *action(endPointName: String): NamedAction =* ***new*** *NamedAction(endPointName)* ***def*** *returns(output: Output)(actionName: String, actionType: ActionType, actionDescription: String, inputs: Seq[Input]): ActionWithOutput =  
 ActionWithOutput(actionName, actionType, actionDescription, inputs, output)  
}*

Appendix:4 ActionType.scala

***package*** *mics.dsl.service****sealed trait*** *ActionType****case object*** *Get* ***extends*** *ActionType****case object*** *Post* ***extends*** *ActionType****case object*** *Put* ***extends*** *ActionType****case object*** *Delete* ***extends*** *ActionType*

Appendix:5 Module.scala

***package*** *mics.dsl.service****trait*** *Module {* ***def*** *moduleName: String  
}****case class*** *NamedModule(moduleName: String)* ***extends*** *Module {* ***def*** *defines(moduleDescription: String): DescribedModule =  
 DescribedModule(moduleName, moduleDescription)  
}****case class*** *DescribedModule(moduleName: String, moduleDescription: String)* ***extends*** *Module {* ***def*** *having(f: (String, String) => ModuleWithActions): ModuleWithActions =  
 f(moduleName, moduleDescription)* ***def*** *withActions(actions: Action\*): ModuleWithActions =  
 ModuleWithActions(moduleName, moduleDescription, actions)  
}****case class*** *ModuleWithActions(moduleName: String, moduleDescription: String, actions: Seq[Action])* ***extends*** *Module****object*** *Module {* ***def*** *module(moduleName: String): NamedModule =  
 NamedModule(moduleName)* ***def*** *actions(actionList: Action\*)(moduleName: String, moduleDescription: String): ModuleWithActions =  
 ModuleWithActions(moduleName, moduleDescription, actionList)  
}*

Appendix:6 Input.scala

***package*** *mics.dsl.service****trait*** *Input****case class*** *NamedInput(inputName: String)* ***extends*** *Input {* ***def*** *as(inputType: String): TypedInput =  
 TypedInput(inputName, inputType)  
}****case class*** *TypedInput(inputName: String, inputType: String)* ***extends*** *Input {* ***def*** *carries(inputDescription: String): DescribedInput =  
 DescribedInput(inputName, inputType, inputDescription)  
}****case class*** *DescribedInput(inputName: String, inputType: String, inputDescription: String)* ***extends*** *Input****object*** *Input {* ***def*** *input(inputName: String): NamedInput = NamedInput(inputName)  
}*

Appendix:7 Output.scala

***package*** *mics.dsl.service****trait*** *Output****case class*** *TypedOutput(outputDescription: String, outputType: String)* ***extends*** *Output****object*** *Output {* ***implicit class*** *StringToOutput(outputDescription: String) {* ***def*** *as(outputType: String): TypedOutput = TypedOutput(outputDescription, outputType)  
 }  
}*

Appendix:8 CodeGenerator.scala

***package*** *mics.dsl.generator*

***import*** *java.io.{PrintWriter, File}****import*** *java.nio.file.Paths****import*** *mics.dsl.service.Data.ServiceData****import*** *mics.dsl.service.Data.\_****import*** *mics.dsl.service.Service****import*** *play.twirl.api.Html****import*** *scala.sys.process.\_****import*** *scala.util.{Failure, Success, Try}****case class*** *CodeGenerator(service: Service, directory: String) {* ***val*** *path = Paths.get(directory)* ***def*** *generateCode: Try[Boolean] = {  
 generate.map(\_ =>* ***true****)  
 }* ***private def*** *generate: Try[String] = service.data* ***match*** *{* ***case*** *Some(ServiceData(serviceName, serviceDescription, packageName, modules)) =>  
 createProject(serviceName)* ***for*** *{  
 module <- modules  
 moduleData <- module.data  
 moduleCode: Html = twirl.html.controller(packageName, moduleData)  
 saved = writeToFile(s"****$****path\\****$****serviceName\\app", packageName, "controllers", s"****$****{moduleData.name}Controller.scala", moduleCode.body.trim)  
 }* ***yield true  
 val*** *routesCode: Html = twirl.html.routes(packageName, modules)  
 writeToFile(s"****$****path\\****$****serviceName", "", "conf", "routes", routesCode.body.trim)  
 Success(s"****$****serviceName project successfully generated!")* ***case*** *\_ =>  
 Failure(****new*** *RuntimeException(s"Service is not defined completely"))  
 }*

***private def*** *createProject(projectName: String): Try[String] = Try {* ***val*** *activatorCmd = s"C:\\play\\activator\\dist-1.3.10\\bin\\activator.bat new* ***$****path/****$****projectName play-scala"  
 activatorCmd.!!  
 }* ***private def*** *writeToFile(basePath: String, packageName: String, component: String, fileName: String, fileContent: String): Boolean = {* ***val*** *filePath = basePath + "\\" + packageName.split('.').mkString("\\") + "\\" + component + "\\" + fileName* ***val*** *file =* ***new*** *File(filePath)  
 file.getParentFile.mkdirs()  
 file.createNewFile()* ***val*** *writer =* ***new*** *PrintWriter(file)  
 writer.write(fileContent)  
 writer.close()* ***true*** *}  
}*

Appendix:9 Data.scala

***package*** *mics.dsl.service****object*** *Data {* ***case class*** *ServiceData(name: String, description: String, packageName: String, modules: Seq[Module])* ***case class*** *ModuleData(name: String, description: String, actions: Seq[Action])* ***case class*** *ActionData(actionName: String, name: String, actionType: String, description: String, input: Option[String] = None) {* ***def*** *inputParam = input.map(\_ => "(id)").getOrElse("")* ***def*** *inputRoute = input.map(\_ => "/:id").getOrElse("")  
 }* ***case class*** *InputData(name: String, description: String, inputType: String)* ***case class*** *OutputData(description: String, outputType: String)* ***implicit class*** *IServiceData(service: Service) {* ***def*** *data = service* ***match*** *{* ***case*** *ServiceWithModules(name, description, packageName, modules) =>  
 Some(ServiceData(name, description, packageName, modules))* ***case*** *\_ => None  
 }  
 }* ***implicit class*** *IModuleData(module: Module) {* ***def*** *data = module* ***match*** *{* ***case*** *ModuleWithActions(name, description, actions) =>  
 Some(ModuleData(name, description, actions))* ***case*** *\_ => None  
 }  
 }* ***implicit class*** *IActionData(action: Action) {* ***def*** *data = action* ***match*** *{* ***case*** *ActionWithOutput(name, actionType, description, input, output) =>* ***val*** *action = actionType* ***match*** *{*

***case*** *Get => "get"* ***case*** *Post => "create"* ***case*** *Put => "update"* ***case*** *Delete => "delete"  
 }* ***val*** *actionName = action + name.head.toUpper + name.tail* ***val*** *input = actionType* ***match*** *{* ***case*** *Get | Delete => Some("id")* ***case*** *\_ => None  
 }  
 Some(ActionData(actionName, action, actionType.toString.toUpperCase, description, input))* ***case*** *\_ => None  
 }  
 }* ***implicit class*** *IInputData(input: Input) {* ***def*** *data = input* ***match*** *{* ***case*** *DescribedInput(name, inputType, description) =>  
 Some(InputData(name, inputType, description))* ***case*** *\_ => None  
 }  
 }* ***implicit class*** *IOutputData(output: Output) {* ***def*** *data = output* ***match*** *{* ***case*** *TypedOutput(description, outputType) =>  
 Some(OutputData(description, outputType))* ***case*** *\_ => None  
 }  
 }  
}*