Overview of the flimey Application Architecture, Design and Runtime Modeling Concepts

KARL KEGEL, flimey Project

Every small business is different, varying in their organization structure as well as in their goals and workflows. In consequence it is impossible to target all those requirements with a single software tool to help them managing their teams and workloads. To fit their particular needs, a software system must be modified to every customer individually. This is done in software product lines which can be easily fitted on implementation level. Another concept is configuration over implementation where different approaches try to build software systems which are as flexible as possible and can be configured for every customer individually and without further need for implementation.

flimey is an open-source software tool developed for managing and supervising teams, organizing workloads and structuring data. It employs the configuration over implementation concept in a way that even the experienced user on customer side can reconfigure the underlying (meta-)models at runtime without causing downtimes or restarts. Additionally flimey's most important design goal is to be as powerful as needed while being as simple to use as possible.

1 INTRODUCTION

In this report we want to explain the flimey system in detail on implementation and architecture level. First we explain the underlying architecture. Afterwards we will go in detail how the runtime modeling is implemented by looking at our implementation models. At the end we discuss possible ways to extend flimey in future development and compare our implementation to the concept of Clabjects as well have a look at Deep Modeling. The flimey source code is available under open-source license on our Github repository [1]. There also a detailed user level documentation can be found.

2 ARCHITECTURE

2.1 Technologies

To better understand the architecture and design choices, we first take a look at the used technologies and frameworks influencing our implementation significantly.

The used programming language is Scala [2]. As a powerful modern language, Scala enables object oriented as well as purely functional programming. Because flimey is a multi-user web application we additionally use the Playframework [3] as an established web framework enabling REST-APIs as well as including a template engine for server side web programming and rendering just like with the well known Spring framework in Java. To persist our data we use a PostgreSQL [4] database. To access the data from within our application we do not use an object-relational-mapping (ORM) because this would not fit in our overall programming concept. Instead we use Slick 3 [5] as a flexible and functional JDBC based framework which acts more like a SQL request builder.

Author's address: Karl Kegel, flimey Project, karl.kegel@flimey.io.

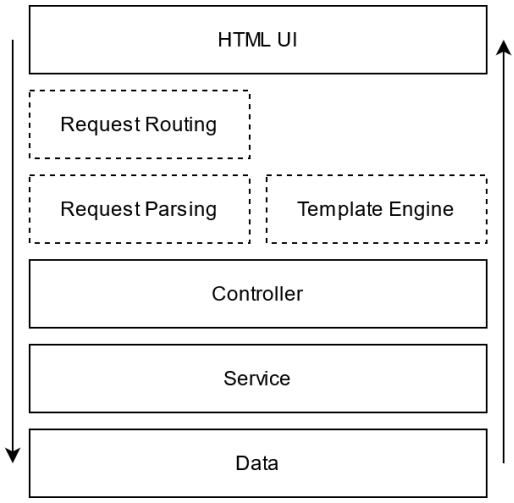

Fig. 1. flimey architecture layers

2.2 Fundamental Concepts and Architecture

2.2.1 Fundamental Concepts. flimey is implemented using the functional features of Scala. Additionally we consequently use the Immutability Pattern [6]. This results in a pipeline-like data and control flow where data is never mutated but always mapped to new instances.

By default, Scala also uses object-oriented concepts such as classes and traits and of course we have to use them despite our purely functional ambitions. Therefore we introduced some implementation restrictions.

- All model/data classes are *case classes* (Scala classes without methods, immutable if not specified otherwise). They are not allowed to receive traits.
- All classes containing functions of the base control flow like controllers, services or repositories are specified as singletons and used by dependency injection. Such classes have no mutable attributes.
- Shared pieces of static functionality are defined as traits and mixed in the classes or *objects* (Scala static instances) which need them.
- If a piece of functionality aligns semantically more to a model class than to a class handling the control flow, an object is used containing the static functionality (no attributes) and is named corresponding to the model case class it is related to.

Those restrictions were not consequently defined beforehand but have condensed themselves by trying incrementally to find a good way to enforce a clean, testable and extensible code structure as well as keep an easy framework integration.

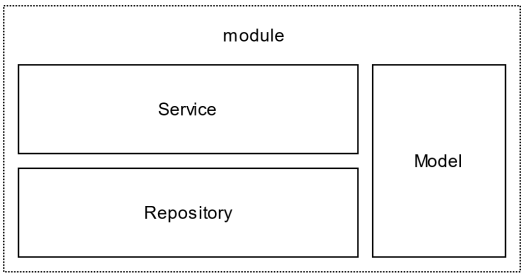

Fig. 2. Horizontal module structure

2.2.2 Application Architecture. flimey uses a layered and monolithic architecture. This allows for division of concerns within the application as well as an easy deployment in the target environment. Figure 1 shows a simplified overview of the application layers. Dashed layers are mostly controlled by the framework. The arrows indicate the control and data flow on every request.

Although the top level module structure is aligned horizontally, we use a vertical packaging in the the two bottom-most layers. Therefore we merged the *Service* and *Data* layers and instead introduce a vertical module for every semantically group of functionality as shown in Fig. 2. This enables us to group the business functionality by high cohesion into modules easy to distinguish. Between those modules then only a very low coupling is present with the additional aim to couple the different modules without cyclic dependencies to drastically improve the maintainability.

The Playframeworks template based HTML request engine we use works on top of the Model-View-Controller Pattern. Request handling and parsing is also mostly done by the framework. The server application acts REST-full and therefore is able to handle each request independently and only with the knowledge provided by the request. In this overview, we will not go in further details regarding the framework usage and especially the implemented authentication. However those parts also fit well in the here described architecture and programming concepts.

3 RUNTIME MODELING

In this section we will describe our runtime modeling approach in detail. However in this overview we will not provide any sourcecode. The reference implementation can be found in our project repository [1]. We will also omit solely technical aspects.

Our main goal while providing a flexible runtime environment is to avoid changes of the low-level class structure. In consequence we decided not to use runtime compilation or reflections to construct classes. Instead we want to express type and model information only by interpreting the object properties of a defined class model to represent types.

3.1 A Flexible Instance Datastructure

The first step towards our flexible runtime modeling is to define a datastructure/class structure which is able to represent every possible instance of the defined types only by changing its object net. Therefore we define two things: first the actual datastructure only with case classes and based on a Multi-Bridge Pattern with

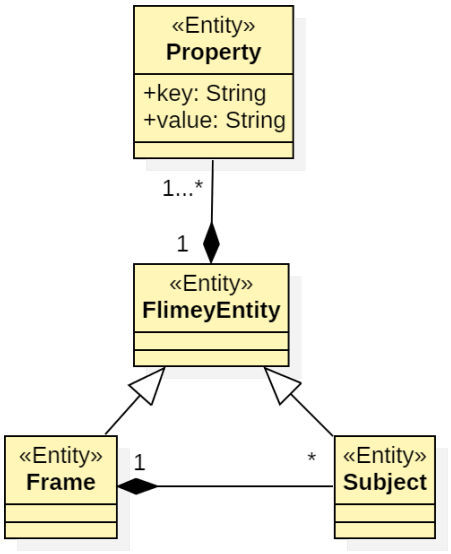

Fig. 3. The flexible instance core datastructure

a core facet and multiple attribute-facets (we will refer to such an object net as *flexible instance* in the following) and second a static group of functions which can map such a flexible instance to a modified one by removing, adding or mapping its facets. Because every object of a flexible instance has an invariant ID attribute, the processing of a mapped flexible instance in the persistence layer will result in an update of the original objects and not the insertion of new objects even if they are different in the view of the runtime environment. Of course actual insertions and deletions must take place if a new flexible instance was generated or whole facets where removed or added (indicated by differing ID sets).

Figure 3 shows a simplified class diagram of the explained structure. The *Entity* stereotype indicates that objects of the class are persistable. Note that inheritance is not actually implemented but simulated by indexed OneToOne compositions. The *key* and *value* attributes of the *Property* class can be anything as of right now. This will be only limited by the constraint model introduced in the next section.

Now all other application parts can be designed to handle flexible instances only. That is very easy because independently from their ability to express every business entity, they still have a fixed class structure. So the only thing we have to take care of is the dynamic amounts of facets we need to process, for example during view generation. This again is simple because all facet instances are of the same *Property* class.

3.2 The Constraint Model

With our flexible instance model as well as an infrastructure handling it, a user could create every possible configured instance, which appears in the UI to be a business class consisting of properties with values. However providing no restrictions limits the

a meaningful usage significantly. What we need is a type system which will be represented by a constraint model. The naming may be a bit misleading because such a model does not really employ constraints such as known from OCL but is more meant as a model defining how our flexible instances are widened or restricted. The user will see such a set of constraints as a type.

Figure 4 shows the simplified class diagram of the constraint model attached to a *FlimeyEntity*. There, we use basically the same pattern as in our flexible instance datastructure. To enable flexible types, an *EntityType* object serves as the core facet and *Constraint* facets are attached dynamically using a Multi-Bridge. The three String type attributes of the *Constraint* class are a constraint-type c (one of a fixed number of options), and two values v1 and v2. The content of these values is dependent of the constraint-type. Basically all constraints of a type form a set of triples $(c, v1, v2)$. This set is the input of two important parts of the application: The implemented type-checking algorithm to verify if a flexible instance fulfills all restrictions of its type; The view generator to generate forms and labels in a type-responsive way to optimize the user experience.

Type Checking. The type-checking algorithm can tell if a given flexible instance fulfills the restrictions given by its type/constraint set. The type checking happens always in the service layer and follows a zero trust strategy. This results in the following execution plan.

1. A user provides a somehow modified flexible instance of a specified type
2. The system fetches the original instance from the repository
3. The system fetches the *EntityType* and constraint set from the repository
4. The system checks if the provided type is also the type of the original instance. flimey allows no type changes (avoid type corruption)
5. The system checks if the provided instance is indeed a modification of the original instance (avoid ID/reference corruption)
6. The system checks if the modified instance fulfills all constraints (avoid model corruption)
7. Feedback with an optional error or success information is returned to the user

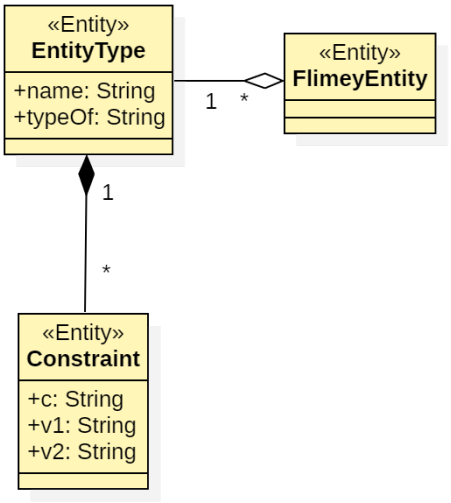

Fig. 4. Simplified type and constraint model

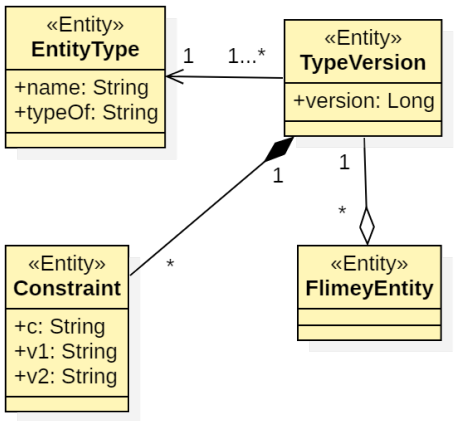

Fig. 5. Simplified versioned type and constraint model

3.3 Type Versioning

With our defined modeling environment, it is now possible to define constraint sets which act as types and flexible instances which are restricted by those types. A further step towards a dynamic and evolutionary system would be the possibility to also edit types at runtime. The simplest approach is to directly couple type changes with changes of the types instances. For example if a constraint which allows (widens) the instance to have a certain attribute is removed, all those attribute fields are removed from all instances immediately. If such a widening constraint is added, new attribute fields need to be added to each instance. The other way around if a restricting constraint is removed, for example a range restriction of an attribute, nothing needs to be changed at instance level. Instead the modification problem would occur by adding such a constraint.

flimey implements this concept by the default edit type functionality which immediately influences all instances. However because of the explained drawbacks in certain cases, we decided to extend this very basic approach by an additional type versioning. Therefore the class structure from Fig. 4 is actually implement as stated simplified in Fig. 5. There, the *TypeVersion* class is used to open the association between *FlimeyEntity* and *EntityType*. Also the constraint set is decoupled from the actual *EntityType* class and instead associated with a *TypeVersion* only. The *EntityType* class remains just to give the type a name and specifies if it is either describing subjects or frames. Now the *FlimeyEntity* has exactly one *TypeVersion* which specifies the constraint set and is associated to an *EntityType* which basically gives only the name and if the correct subclass is instantiated.

This refined concept allows to copy, fork and create new TypeVersions as one would do with types before. However all those

TypeVersions are still in the same familiy of their EntityType which is not interchangable. With this the user can create new versions of a type instead of editing only one constraint set. Because flexible instances are linked to their version, flexible instances of other versions will stay invariant if another version is edited. In consequence flexible instances of the same type but different versions can live in parallel without restricting each other in this given context.

4 KNOWN LIMITATIONS AND FUTURE IDEAS

With the current state of the actual implementation, some limitations and yet not solved problems became known to us. In the following we will name them and already propose solution ideas if existing.

- It is not possible to migrate instances between type versions or between "before and after" version edit states. For example if a property of datatype string is changed to datatype datetime, the user would first need to delete the property and then add it again with another datatype. However this would delete all data of this property on the way. Instead it would be better to somehow define the constraint model in a more formal way that could allow predictable model transitions.
- The user is by default overwhelmed by the used model editor which is based on configuring constraints in a more technical manner. It may be possible to migrate the constraint format for example to a ontology based representation which could allow a more natural modelling. Also other representations may be possible.
- The existing constraint possibilities are not enough to powerfully model more complex business classes
- The existing constraint model only allows for static modeling. It is not possible to to model dependencies between instances or statechart-like transition handling. It is planned to implement a signal-slot based event handling. However, controlling those events also needs more thoughts about the underlying (meta-)model.

From very few and informal test sessions with possible users and the current version of the flimey application, we inferred that no matter how reasonable a modeling concept is implemented, it needs a broad evaluation of both end users as well as technically educated maintainers to ensure actual usefulness.

5 COMPARISON TO EXISTING APPROACHES

5.1 Clabjects

Clabjects are in general the characteristic of an model element derived from a metamodel to have a type information facet as well as a instance facet [7]. In other words a model element knows its own type and how it can be configured as well as its actual configuration. The flimey implementation employs this concept in principle. Following our previous explanations we use a type facet (constraint set represented by TypeVersion) and the flexible instance as the instance facet (represented by a *FlimeyEntity* and their flexible instance environment).

The given *Frame* class could be described as such a Clabject. Note that the extends relations shown in previous sections are implemented by indexed associations here. The *Frame* class has two important attributes: *entityId* references a *FlimeyEntity* object which serves as the instance facet; *typeVersionId* references a *TypeVersion* which serves as the type facet.

But as indicated before, we do not belief to have Clabjects implemented completely as intended by their authors because the fully assembled Clabject never actually exists. Instead the referenced facets are mostly handled independently. Even if they are grouped together in some helper classes, they play no central role. Our design is more refined to handle type and instance facets in parallel and keep the actual Clabject a conceptual but virtual existence.

```
case class Frame(
    id: Long,

    //instance facet indexed reference:
    entityId: Long,

    //type facet indexed reference:
    typeVersionId: Long,

    state: SubjectState.State,
    created: Timestamp
)
```

5.2 Deep Modeling

Deep Modeling using multiple clabject levels as explained in [8] and introduced in [9] is in our opinion not part of the flimey system as of right now. At least we provide no way to vary the dimensionality of our implemented metamodel which are structured as in the following.

1. Implemented M3 metamodel defining meta classes such as constraints and properties
2. Logical M2 metamodel defining types using M3 which is created and varied by the user at runtime
3. Logical M1 model which unifies a type model from M2 as well as a flexible object from M3

As named *Logical* those layers never actually exist as physical classes but live only as data (instances of M3 and are interpreted as more concrete layers).

It may be worth considering to implement Deep Modeling in a flexible concept such as type templates. However we do not belief that such complexity should be introduced in flimey. Instead it would be more reasonable to providing more horizontal functionality on a restricted type model instead of vertically deepen the model and loosing the ability to make easy and concrete assumptions.

REFERENCES

[1] flimey source repository. https://github.com/flimeyio/flimey-core.
[2] Scala programming language. https://www.scala-lang.org.
[3] Playframework. https://www.playframework.com.
[4] Postgresql. www.postgresql.org.
[5] Slick 3.0 framework. https://scala-slick.org/doc/3.0.0/#.
[6] John Hunt. *Scala Design Patterns*, chapter 4, page 53 et seq. Springer, 2013.
[7] Colin Atkinson and Thomas Kühne. Meta-level independent modelling. In *International Workshop on Model Engineering at 14th European Conference on Object-Oriented Programming*, volume 12, page 16, 2000.

[8] Colin Atkinson, Ralph Gerbig, Katharina Markert, Mariia Zrianina, Alexander Egurnov, and Fabian Kajzar. Towards a deep, domain specific modeling framework for robot applications. *MORSE@ STAF*, 242, 2014.

[9] Colin Atkinson, Matthias Gutheil, and Bastian Kennel. A flexible infrastructure for multilevel language engineering. *IEEE Transactions on Software Engineering*, 35(6):742–755, 2009.