CISL - Core Insurance Service Layer

Michael Ilger and Michael Halwax

AMOS Austria, 1140 Wien, AUSTRIA,
WWW home page: https://www.allianz.at

Abstract. Core Insurance Service Layer (CISL) is a project to create a common but extensible service layer catalog for insurance processes. It follows the REST principles to define services and a datamodel which are exposed by new or existing insurance backends and that can easily be consumed by front end applications. The project provides a REST Meta-Model and tools to facilitate the adaption of CISL and the reuse across organizational units within Allianz.

Keywords: API, REST, Insurance, Model

1 Introduction

Large corporation, such as Allianz Insurance, want to standardise processes, software and infrastructure to benefit from synergies [11]. Allianz wants to give customers continuity in the user interfaces they see and to enable sharing of user interface components among different countries. To achieve this a project to create an abstraction layer called Core Insurance Service Layer (CISL) was initiated. The project has the goal to define reusable data objects and operations focussed on the insurance world, while also offering concepts and tooling which can be applied in other scenarios.

Unlike many other standards which are based on SOAP, this service layer uses REST services. The idea behind this is to use some features which are important aspects of HTTP and REST, therefore making the whole process more suitable for the main goal: providing a fast and easy way to provide fresh user interfaces to existing back end applications and scaling them out vertically.

To get a better perspective on the difference between this approach and existing solutions such as BiPro [13] or Acord [14], it is very important to focus on the goals. While the two aforementioned standards rely on SOAP and the concept of larger service calls, the concept of REST allows us to put a much stronger emphasis on smaller resources and the connection between those. This leads to a scenario, where we can have reuse of individual resources and still provide simple, dynamic integration.

2 Project Organization and Outcomes

The original idea for the project started in 2013, when a couple of Allianz entities were faced with a problem: They had to provide a new user interface based on

some new corporate identity and they had to already prepare the migration to a new insurance system. The new insurance system would already provide a user interface, but this would not be usable with their legacy systems, so this would mean two separate migration steps and possibly also three separate user interfaces in a very short time.

Shortly after that, in early 2014, the project was started with a CISL implementation based on the Allianz Business System (ABS). ABS is a software platform for insurance backends and is developed by Allianz in Austria. ABS is used world wide, as part of a plan to provide standardised software and processes for insurances. The focus of the project is allow to have a separation between the user interface and the underlying logic. Originally the project team was staffed with three developers and analysts from the pool of AMOS (Allianz Managed Operations and Services) Austria resources working on the insurance system ABS. Soon the project was split into a separate entity dealing with the standardisation of the interfaces and therefore also giving other systems more influence on the development of the standard. A second unit, part of the ABS development team, continues to implement the interfaces defined by the CISL team.

As of 2016 many different countries, including Austria, Switzerland, Italy and many more, have CISL in their road map, while Austria already has an implementation based on CISL in production.

3 The Integration Scenario

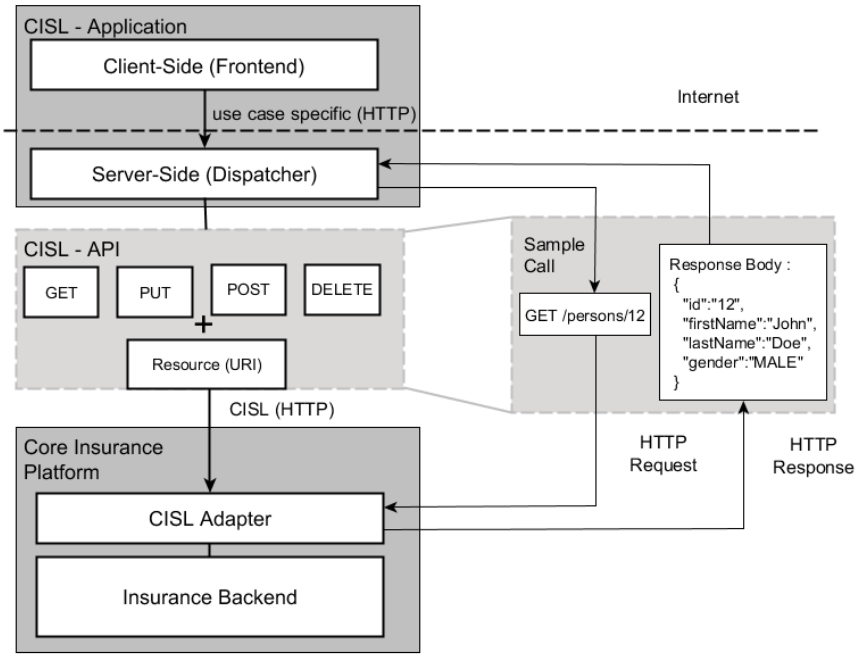
The general integration scenario surrounding CISL is relatively simple. It will separate a back end system from the front end system. One of the main ideas behind this is, that those two systems will evolve at different speeds. Especially in banking or insurance companies the underlying 'systems of record' generally evolve rather slowly, the reason for that being the stability of the system and the fact that the sheer size of systems do not allow an quick rewrite.

While it generally is not considered a problem if the underlying logic runs on and older system (as long as software and hardware updates are still available), this is not true for user-facing front ends. While there are still some expert systems around using traditional host-style text input, these systems are getting fewer and acceptance for such solutions is going down. End customers expect even more than that and modern client side javascript applications provide usability and customer experience which is muss better than anything which was possible in the past - and all that across a variety of different types of devices.

Being able to focus on the front end alone and not having to deal with details of specific business logic allows you to create new user interfaces at a fraction of the cost and faster than previously. CISL allows you to do exactly that, as can be seen in figure 1.

The central part of figure 1 represents the core of the API, creating an abstraction between front end and back end systems. While the implementation of the services on the insurance platform is out of scope for now, we want to focus

Fig. 1. CISL Consumer Producer



on the UI side. The first approach for a solution like this would be to carry the structure provided by the CISL interfaces all the way to the user interface, but this creates a couple of new problems: Unnecessary data transfer to the client, potential security problems due to wide open public interfaces and other security or performance concerns.

The solution for this is to provide a complete application, called CISL application in figure 1, which consists of a server side element and a client side element. Those two elements are tightly coupled and are not using any standardised communication format, besides the general concept of HTTP and REST which is also typically followed here.

4 Challenges

The project deals with a couple of different topics which have been around for a long time in academia and in practice. Starting with the data standardisation point of view, there is the important question about how individual resources and services are cut and then the follow up question regarding how they can be extended. These problems existed back when EDIFACT (with a syntax based on separator characters) was created, existed when XML-based standards were

created and still exist today. The goal is to create elements which are highly reusable, but still leave room for customisations if needed - even if too much customisation leads to fragmentation and therefore can destroy a standard.

One very simple example of cases where you need to be able to deal with different systems which handle data differently would be 'gender'. Traditionally supporting exactly two values here would be enough, but there are movements which go towards allowing more than those two traditional values. Even if the underlying standard supports more than the two traditional values, it does not mean that all the systems implementing this standard also support all the values and that a front end system should support all those values.

5 The REST Metamodel

As CISL is used in the context of multiple platforms and shared among stakeholders with various skill levels. The project decided to use a metamodel that serves as a common base for communication between business analysts and developers. This model defines the abstract syntax, the possible elements and the relations between them and is used as a base for model driven development [7].

Many frameworks that support the definition of REST APIs evolve. Prominent ones, that have also influenced CISL and it's REST meta model, are Swagger [4], RAML [3] and WADL [5]. The CISL Project using an early version of of RAML in 2014 [1]. Due to limitations in tool support, modularisation, expressing extensions and complex object oriented representations the project decided to introduce the meta models RADL (REST API Definition Language) and RCORE (based on EMF Ecore).

RADL and RCORE are based on the modelling framework EMF [9] and also provide a textual representation based on Xtext [8]. The CISL model provides all the information necessary to support the structural modelling of a REST API [1]. In addition to the IDE integration (highlighting, code completion, validation) the project provides a generator that is used for server-code generation and to create API user documentation.

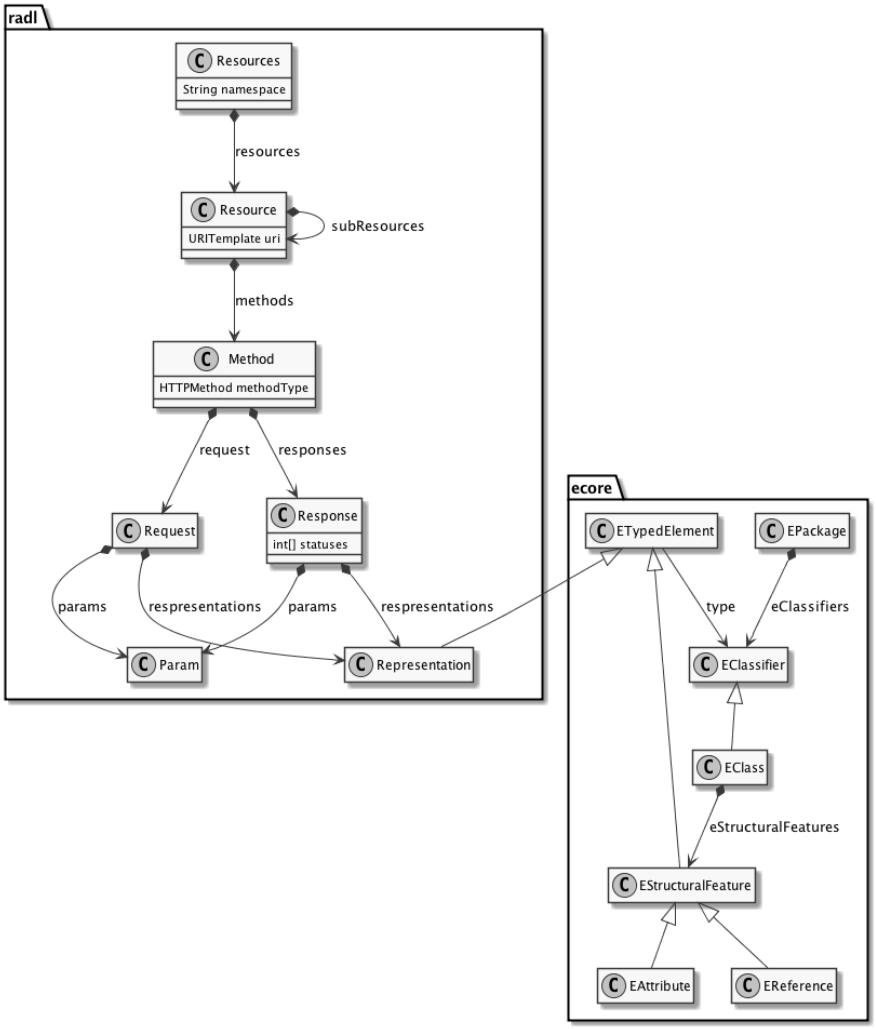
With the use of Xtext it is possible to support IDE Integration like highlighting, code completion, validation and code generation. To extend distribution possibilities an export of the model to Swagger and Microsoft Excel is supported as well in order to reach also developers as well as non-technical stakeholders.

Figure 2 describes a class diagram that gives an simplified overview of the most relevant RADL and RCORE model.

RCORE is an extension of Ecore and in its textual representation built on top of Xtext. It provides better support for annotations that may be used to specify constraints like datalists. A datalist is a list of options that are valid for a specific field, that is determined via a REST call at runtime.

[1] Since the RAML 1.0 (released in 2016), it provides initial support for object oriented representations and new extension concepts.

Fig. 2. RADL and RCORE



The listing 1.1 provides a sample in the RCORE textual representation that describes EMF Ecore company package with a Company class.

Listing 1.1. Company package

```
package com.allianz.direct.company

import com.allianz.cisl.base.Datalist
```

```
class Company {
        String name
        @Datalist(Degree)
        String[] degrees
}
```

RADL is based on WADL [5] but optimized for the integration with RCORE. It provides the means to define resources, methods, parameters and representations that are based on RCORE.

Listing 1.1 provides a sample RADL textual definition to get companies (based on an optional query parameter company name) and post a new company on the collection resource defined by the path /companies .

```
package com.allianz.direct.rest

import com.allianz.direct.company.Company

resources company

/companies | Companies {
    get | getCompanies(query String companyName){
        return Company[]
    }
    post | postCompany(
        Company company
    ){
        return Company
    }
}
```

6 Workflows

Basically everything we do is based on workflows and interaction. Most data exchange formats are based on the assumption that they are used for the integration of (automated) systems and therefore also expects all the information to be available from the start. When we think of a system based on user interaction this assumption does not work. Taking the relatively simple case of a car insurance shows us what types of information are required: The minimum information consists of at least one person (the owner; optionally also a driver or other roles), a vehicle and an underlying insurance product. In a user interface-driven scenario this means that you could end up first selecting a product, then providing the owner's personal data and at the end choosing a car from a list. Having all this functionality combined in one single back end call would result in limited customer experience as the user would have to fill out all the information

first, before finding out that he is not allowed to buy this product due to age restrictions. It would still be possible to duplicate the logic which checks for the age restrictions, but this would again cause heightened software maintenance costs and reduce reusability. The approach taken here is to rely on something which has been around as long as the internet: Hyperlinks. While the anchor tag in HTML is a very central part of the web, there are no such standards for APIs, even though the concepts have been around for a while now in HATEOAS, or Hypermedia as the Engine of Application State. This basically allows you to put meta information into REST resources which points the consuming application into different directions for a workflow. In case of our above example this means that the workflow could be specified in the following (simplified) way from a REST point of view

POST /quote - creates a new quote; this returns a quote object which could also be retrieved using a GET requests and the given ID (e.g. GET /quote/4711)

The resource representing that quote would also contain a section pointing to different other URLs specifying certain functionality. In our case this section would also include the hint that a POST to /owner or a POST to /vehicle would be allowed.

POST /quote/4711/owner - adds an owner to the quote created above. This can subsequently be read or modified using other HTTP verbs.

A subsequent GET to the quote resource would now return a different result. The owner is already present, therefore the POST link to the owner would not be offered anymore.

The same concept would be used to create a vehicle in the next step, which would then leave the quote in a different state, where neither of the two links would be available anymore. Instead the resource could now show a couple of different links: Buy and Save.

It is important to remember here, that this is all done on the level of the underlying API and not directly in the user interface. It would be better if the user interface designer was already aware of the possible different actions that are available, but generally it is not required to know all the possible steps in advance. It is also worth noting that this would allow you to react to subtle differences in workflows in different scenarios, where for example one system would require the owner to be entered first and one system would require the vehicle to be entered first.

7 Reusability and Extensibility

Though CISL API should be used as is, it important that it remains extensible (for example to adhere to national regulatory). CISL describes the common parts that are designed and shared by the project team, but a partner may introduce an extension module to support non global requirements. To extend the core there are two possiblities:

- add new (sub)resources - introduce resource in parallel to the existing ones (the URI has to be unique)

– inheritance - use inheritance in the representation, typically this approach would be chosen when adding properties and having an is-a relation ship

The CISL API defines REST resources to expose core business functions (as described in globally defined processes) that can be used in CISL Applications. A major challenge lies in finding the right granularity for the definition of resources:

> If we design the API around fine grained resources, we would end up with a chattier API for consumer applications. On the other hand, if we design the API based on very coarse grained resources (de-signed to do everything) there will not be enough variations to support all the API consumers' needs and the API may become too difficult to use and maintain.[6]

Since various frontend specific requirements are expected, tailored interfaces, that would be hard to specify globally, are rarely used. This aligns with the goal to define a general interface that separates the fast evolving CISL applications from the stable and slow adapting core insurance platforms.

8 Conclusion

Overall the approach in CISL is to combine multiple different approaches into one new framework, combining the advantages and prior art into one coherent solution. This includes the definition of service calls based on REST instead of the more traditional SOAP-based web services which have been around for a long time. Using REST brings two advantages with the typical format (JSON) being usually much smaller than XML messages [12] and with the full support for HTTP verbs and hyperlinking.

Two more topics, which still need more thorough work, are reuse and tooling. While it is very simple to define services which return some data, it is not that easy today to describe their functionality as a model and use this as the basis for actual implementations. In the recent past many new attempts were made to find a good solution, but this is nowhere near the quality of such features in standards like XML Schema. These concepts of reuse of existing components as well as the possibility to extend the functionality, if the standard does not provide everything that is needed, are the most important aspects of the future work in this area.

References

1. Silvia Schreier. Modeling restful applications. In *Proceedings of the Second International Workshop on RESTful Design*, WS-REST '11, pages 15–21, New York, NY, USA, 2011. ACM.
2. Roy Thomas Fielding. *REST: Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000.
3. RAML - RESTful API Modeling Language, `http://raml.org/`, May 2016.

4. Swagger, http://swagger.io/, May 2016.
5. Marc J. Hadley. Web application description language (wadl). Technical report, Mountain View, CA, USA, 2006.
6. Prakash Subramaniam, REST API Design - Resource Modeling, http://www.thoughtworks.com/insights/blog/rest-api-design-resource-modeling
7. Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.
8. Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. 2013.
9. David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.
10. Küster, Jochen M. and Ryndina, Ksenia and Gall, Harald Generation of Business Process Models for Object Life Cycle Compliance Business Process Management: 5th International Conference, BPM 2007
11. Martin Grossman and Richard V. McCarthy and Jay E. Aronson E-Commerce Adoption in the Insurance Industry Issues in Information Systems, Volume V
12. Pavan Kumar Potti, Sanjay Ahuja, Karthikeyan Umapathy, and Zornitza Prodanoff Comparing Performance of Web Service Interaction Styles: SOAP vs. REST 2012 Proceedings of the Conference on Information Systems Applied Research
13. Brancheninstitut Prozessoptimierung, http://www.bipro.net
14. Acord Data Standards http://www.acord.net