ABS Microservices and Ontology Integration for SPL Implementation in Information System

Andri Kurniawan
Faculty of Computer Science
Universitas Indonesia
West Java, Indonesia
Email: andri.kurniawan31@ui.ac.id

Iis Afriyanti
Faculty of Computer Science
Universitas Indonesia
West Java, Indonesia
Email: iisafriyanti@ui.ac.id

Ade Azurat
Faculty of Computer Science
Universitas Indonesia
West Java, Indonesia
Email: ade@cs.ui.ac.id

***Abstract*—Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularised in the 1960s with the release of Letraset sheets containing Lorem Ipsum passages, and more recently with desktop publishing software like Aldus PageMaker including versions of Lorem Ipsum.**

I. Introduction

Software Product Line (SPL) gains a huge attention nowadays since it claims that using the paradigm can reduce the cost production, enhance the quality of product, and shorten time of product distribution [1]. The fundamental key of SPL paradigm is feature model thst is to capture the variabilities and commonalities of different product requirements for different stakeholders; some of previous product components can be used for other requirements. This paradigm has successfully adopted in industry area that several success stories can be found in Product Line Hall of Fame[1]. However, to the best of our knowledge, the automated SPL adoption in information system to have the system more adaptable is still inadequate. The main reason is that the feature model as a key part in SPL paradigm can not be executed as it is a model-oriented. Therefore, to realise automated SPL from requirement gathering to deployment phase, we need an executable model such as Abstract Behavioral Specification (ABS).

Abstract Behavioral Specification (ABS) becomes more promising to solve the problem in the SPL since it is a formal modelling language and executable for distributed object-oriented systems [2] [3]. The study of ABS Microservices that .... //Papernya Afifun. The feature model is implemented into ABS Microservices and results a web service including the required business logic. The web service supports software development that fits the domain of feature model.

In addition, the previous study [4] presents the use of OWL ontology as a feature model to produce an application as part of SPL realisation. By using such mechanism, it is possible to conduct inconsistency checking upon the feature model [5]. The feature model is mapped as *classes* and *data property* of OWL ontology and can be transformed into Zotonic-based information system; a CMS which adopt pragmatic semantic web for its data model. The approach has been already been adopted to automate a charity organization information system [6]. However, the business logic for the system is still manually developed.

Since the web service produced by ABS Microservice provides the business logic and can be retrieved via its web service, we can build an adaptor to integrate web service resulted by ABS Microservice and the Zotonic-based system with ontology inside. Therefore, in this paper we attempt to develop the adaptor, in such a way the information system is semantically structured and the business logic is automatically developed and meets the requirement.

II. Adaptor for ABS Microservices and Zotonic Integration

The adaptor is specifically built to call the web service produced by ABS Microservices. However, the adaptor needs a list of rules which contains methods along its endpoint in JSON format (see Fig 1). When accessing a page that has business logic, the template (the template is a template engine[2] generated by zotonic that has been modified so that it contains business logic) will trigger the model abs to work. The model abs read the rules to identify the endpoint and total parameter in order to check whether the total parameter received is same with the web service requirement. After the model abs gets the endpoint and the number of parameters of the rules, the model abs will send the data to the web service for processing. The full source code of this adaptor can be found in https://gitlab.com/andrikurniawan/adaptor-ontology-to-webservices.

A. Adaptor Interface

The adaptor can be called by two options: *template engine* and a *model*.

1) Calling adaptor through template engine: For this reason, we developed a new model inserted in Zotonic installation folder, we named it as m_abs. Since it is a new model for Zotonic, we need to export the required built in models that is `m_find_value`, `m_to_list` and `m_value`. We did not

[1]http://splc.net/fame.html

[2]https://expressjs.com/en/guide/using-template-engines.html

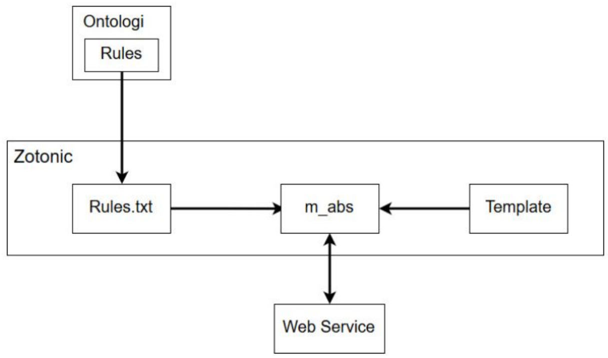


Fig. 1. Architecture Design of Adaptor

define `m_to_list` and `m_value` since they will not be called when the adaptor is running.

However, we specify `m_find_value` to do pattern matching. When the template engine run, it automatically generates some commands structured as `m.abs.functionName[query param=value]` and trigger the adaptor to run. For example `m.abs.totalDonation[query id=id]` means that it calls function `m_abs` in the adaptor for *totalDonation* method or a *key* and `query id=id` is the data which will be delivered to the web service. Both key and data are inputs for the parameters in `m_find_value` function.

2) Calling adaptor by a model: The adaptor is also can be called from other models, such as `m_src` model; one of Zotonic built in model. Regarding that requirement, we implemented a new function in the adaptor that is `call_api_controller` which can be accessed by other models. The function receives two paramaters: Key and Data. The Key indicates the method name in the web service and Data is to pass data to the web service. For example, `m_abs:call_api_controller(deleteDonation, Json);` means that it call function `call_api_controller` in the adaptor for *deleteDonation* method or a key and `Json` is the data which will be delivered to the web service. One example of using an adapter from another model is to be able to store data from multiple sites that have separate database on zotonic to an external centralized database.

B. Adaptor Implementation

Aforementioned, the adaptor needs a table of rules. The rules is a JSON format and structured as in Listing 1. The rules table must be located in root folder of Zotonic.

Listing 1
STRUCTURE OF RULES

```
{
  function name: [endpoint, total of parameter]
}
```

We also implemented `lookup_rules` function to read the rules table then returns a list of endpoint and its total of parameter with *key* as an input. The function calls `read_file` function which read the whole rules file. The adaptor also run `validate_params` function which returns a boolean (see Listing **??**). If *true* then the total parameter in rules table is same in the Query where query is data which will be delivered to the web service. If it returns true, the adaptor runs `fetch_data` to request data from the web service.

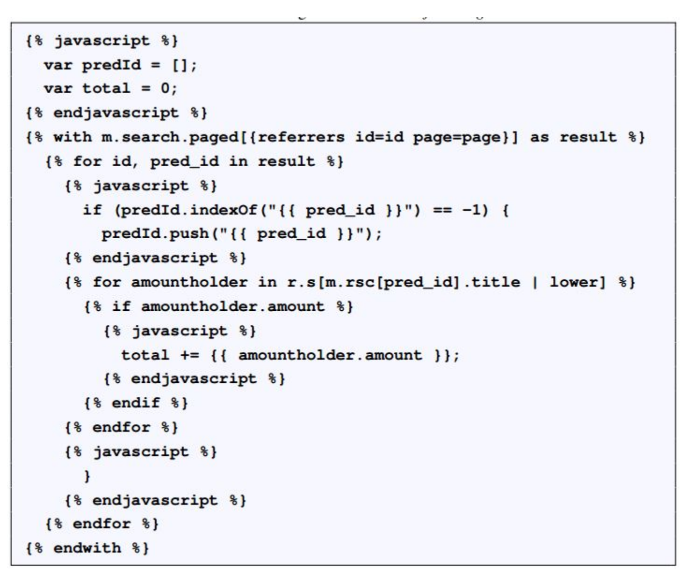
There are two parameters in `fetch_data` function: `Url` and `Query`. The `Url` is the endpoint of web service and `Query` is parameter in JSON format. However, the function needs `post_page_body` to directly interact with the web service (see Listing 2).The function returns data in JSON format and decode with `jiffy:decode/1`

Listing 2
IMPLEMENTATION OF POST_PAGE_BODY FUNCTION

```
post_page_body(Url, Body) ->
 case httpc:request(post, {Url, [],
    "application/json", Body},
    [], []) of
  {ok,{_, _, Response}} ->
    Response;
  Error ->
    {error, Error}
end.
```

C. Adaptor for Business Logic

In the previous study [6] the business logic of the system is implemented manually in Javascript. On the other hand, business logic on one charity organizational with another may be different. By using adapter, business logic will be processed in web service so that it can be more dynamic to change and easy. For example, to count the total donation, the previous work [6] needs to construct a business logic in Javascript manually (see 2)



```
{% javascript %}
  var predId = [];
  var total = 0;
{% endjavascript %}
{% with m.search.paged[{referrers id=id page=page}] as result %}
  {% for id, pred_id in result %}
    {% javascript %}
      if (predId.indexOf("{{ pred_id }}") == -1) {
        predId.push("{{ pred_id }}");
    {% endjavascript %}
    {% for amountholder in r.s[m.rsc[pred_id].title | lower] %}
      {% if amountholder.amount %}
        {% javascript %}
          total += {{ amountholder.amount }};
        {% endjavascript %}
      {% endif %}
    {% endfor %}
    {% javascript %}
      }
    {% endjavascript %}
  {% endfor %}
{% endwith %}
```

Fig. 2. Total Donation function in Javascript

III. Running Example: BSMI

BSMI (Bulan Sabit Merah Indonesia) is a charity organization that works to help others in Indonesia. To make it easier for the community to participate in providing assistance through BSMI, BSMI needs a website that can facilitate them in their operations. In this paper, we employ BSMI as a running example to demonstrate the adaptor and its usage and also linked data.

BSMI has many features, but in this paper we focus on using program features, donation features, target features, and donor features only. Program features are activities undertaken by BSMI. Donation features is a donation given by the community for a program. Donor features are people who donate and target features are target of a program. The features that have been selected have some business logic. For example, between the relationship program and donation is for each program is needed to know how many donations associated with the program. Based on these features, the table mapping rules are generated which contains the endpoint and the number of parameters of the selected feature.

When we open a page containing business logic, it will trigger the adaptor call. In this example, when opening a page with the category of program, total donation of that program will be automatically updated (see fig. 3). If we create a new donation or delete a donation for this program, total donation in this program will be automatically updated.

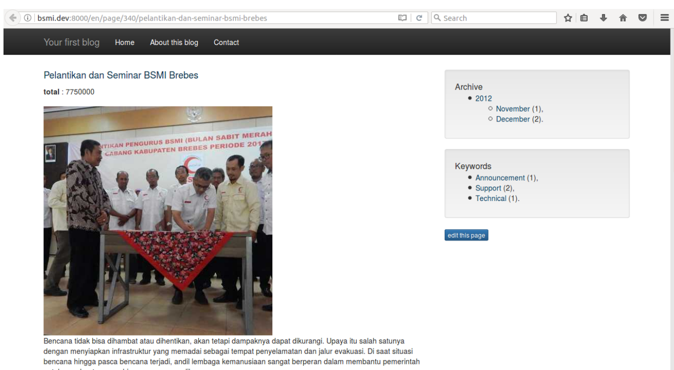


Fig. 3. View of "Pelantikan dan Seminar BSMI Brebes" program

IV. Conclusion

Acknowledgment

This work was fully supported by Reliable Software Engineering (RSE) Laboratory, Fasilkom UI and funded by Universitas Indonesia under Hibah PITTA no. 395/UN2.R3.1/HKP.05.00/2017

References

[1] K. Pohl, G. Böckle, and F. Van Der Linden, *Software Product Line Engineering. Foundations, Principles, and Techniques*. Springer, 2005.

[2] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte and M. Steffen, *ABS: A Core Language for Abstract Behavioral Specification*. Formal Methods for Components and Objects: 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Springer Berlin Heidelberg. Berlin, Heidelberg, 2012.

[3] R. Hähnle. *The abstract behavioral specification language: A tutorial introduction*. In Formal Methods for Components and Objects, pages 137. Springer, 2013.

[4] I. Afriyanti, F.M. Falakh, A. Azurat and B. Takwa, *Feature Model-to-Ontology for SPL Application Realisation*, submitted to The 18th International Conference on Product-Focused Software Process Improvement. URL: https://drive.google.com/file/d/0B5rTAl7jk81Qcno1ZE9WOXhwdTA/view?usp=sharing

[5] H. H. Wang, Y. F. Li, J. Sun, H. Zhang, and J. Pan, *Verifying feature models using OWL*, Web Semantic, vol. 5, no. 2, pp. 117-129, 2007.

[6] B. T. Pangukir, *Pembentukan Otomatis Aplikasi Berbasis Web dengan Masukan Berupa Ontologi: Status Kasus Web BSMI*

[7] M. A. Naily, M. R. A. Setyautami, R. Muschevici, *Development Framework for Microservices-based Software with Software Product Line Approach Using Abstract Behavioral Specification (ABS) Modeling Language*, to appear in Conference ...