A Template Engine for Parsing Objects from Textual Representations

Milan Rajković, Milena Stanković, and Ivica Marković

Citation: AIP Conference Proceedings **1389**, 825 (2011);
View online: https://doi.org/10.1063/1.3636860
View Table of Contents: http://aip.scitation.org/toc/apc/1389/1
Published by the American Institute of Physics

A Template Engine for Parsing Objects from Textual Representations

Milan Rajković, Milena Stanković and Ivica Marković

Faculty of Electronic Engineering, Aleksandra Medvedeva 14, 18000 Niš, Serbia

Abstract. Template engines are widely used for separation of business and presentation logic. They are commonly used in web applications for clean rendering of HTML pages. Another area of usage is message formatting in distributed applications where they transform objects to appropriate representations. This paper explores the possibility of using templates for a reverse process - for creating objects starting from their representations. We present the prototype of engine that we have developed, and describe benefits and drawbacks of this approach.

Keywords: Template engines, scripting languages, object serialization, REST services
MSC: 68N15

INTRODUCTION

One of the biggest concerns for developers of web applications is a clear separation of controller and presentational parts of the program. The first one is a server side language that implements needed business logic, and the second one is usually HTML mixed with JavaScript that is processed on the client side. Although it is possible to create a custom solution to this problem, for most of programming platforms there are specialized template engines that can help addressing this issue. For example, PHP developers can use *Smarty* engine [1], for Java developers there is *String template* [2], [3] and .Net has built in template mechanism that processes aspx pages [4].

Template engines are not limited to web applications – they are useful whenever a text or code generation is needed [5]. For example, Microsoft's T4 can be used for both text and code generation [6].

These engines can also be exploited when a textual representation of an object is needed - for example in distributed applications where an object needs to be transferred from one application node to another. The benefit of using a template engine is the same as for web applications – they allow separation of logic that generates the output from the objects being serialized. This also means that if the representation of the object needs to be changed, the template can be modified without altering the object itself, and usually without any need for recompiling the program.

But, what is missing for these engines to fully support distributed scenario is possibility to create objects from their representations based on a given template – this is a reverse process to the one described above, and it could be applied to the application node that receives a message and needs to map it to a certain object model. Although there are some attempts to implement this in Perl, most of the modern programming languages don't support the idea to parse input text based on a template.

This paper presents a design of a template engine we are developing that aims to support the "reverse" template scenario – creating objects from text input. Although this is an on-going work, the prototype is already implemented and tested. It is written in Java and can be used to create Java objects from their textual representations.

In the second chapter we will shortly present *String* template engine, and describe common template parts. The third chapter shows the design of our "reverse" template engine with some simple examples and notes on syntax. In the fourth chapter we discuss drawbacks and benefits of this approach, together with possible applications. The last chapter is the conclusion.

A SIMPLE TEMPLATE EXAMPLE

In order to demonstrate how a typical template engine works, we will present a simple template created for *String template engine*. It is displayed in Fig. 1 and contains a short HTML page that displays information about users –

Numerical Analysis and Applied Mathematics ICNAAM 2011
AIP Conf. Proc. 1389, 825-828 (2011); doi: 10.1063/1.3636860

their first name, last name, and known telephone numbers. The figure shows that a template is just an ordinary text file which contains template variables that are separated from surrounding text using special delimiters – in this case '$' signs. During the runtime, controller logic (in this case Java code) will assign values to template variables, and transform it to text output (Fig. 2). Template variable *phones* is an array, so a special syntax is used to iterate through its elements. Templates can also contain *if* statements, loops and other control structures, and the exact syntax can vary depending on the engine. They can also have special statements that allow formatting of user input.

```
<html><body>
First Name: $user.firstName$ <br/> Last Name: $user.lastName$ <br/>
$user.phones:{
    $it.phone$
}$
<body></html>
```

FIGURE 1. A simple template example for *String template* engine

```
StringTemplateGroup stg = new StringTemplateGroup("templates", rootDir);
StringTemplate st = stg.GetInstanceOf(templateName);
st.SetAttribute("user", userObject);
String text = return st.ToString();
```

FIGURE 2. Java controller logic for transforming the template

In Fig. 2, *userObject* is Java object that contains fields *firstName, lastName* and *phones,* which can be accessed through appropriate get methods. This object is passed to the template engine and assigned to template variable *user*. When processing is finished, Java variable *text* will contain the final result.

THE REVERSE TEMPLATE SCENARIO

In the reverse scenario, objects are created based on input text, and the processing is determined by templates. This is shown in more details in Fig. 3. First, the template is converted to a pattern list, which is an object representation more suitable for validating and parsing the input text. This is a process we call template compilation. Then, the pattern list is compared against input text – if the text conforms to the patterns, objects are extracted from it, and if not, an error is generated.

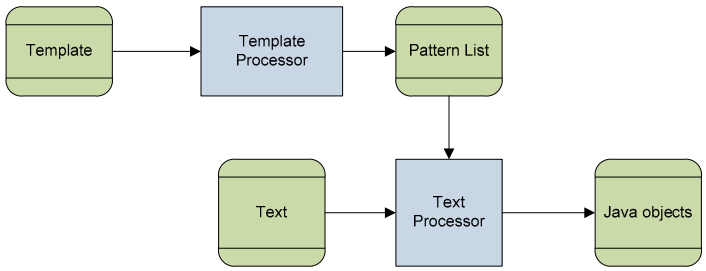

FIGURE 3. Parsing process overview

Figure 4 shows an example of a template that we use. Although the input text doesn't have to be fully structured, we will use xml text in our examples, as it is widely used as an object text representation. The syntax of our templates is similar to the one of a *String template,* and the reason is that they could be used in conjunction – *String template* for creating text outputs, and our engine for parsing text inputs. The template describes how input data is processed and loaded to object *user*. The object contains string typed field *name* and field *addresses* with type *List<Address>*. The template part that is between ":{" and "}$" will map to an individual item of the list, and *city* and *street* are two fields of class *Address.* In order for the fields to be accessible, they should have appropriate setter methods defined in corresponding classes.

Figure 5 contains input text that should be processed, and Fig. 6 displays Java code that uses the template to process input text. *Parser* is the class containing most of the engine logic, and the template itself is given as a constructor parameter. The template is first compiled resulting in the pattern list being created. Then, the *userObject*

is added as a target object, meaning that it is going to be created from the text input. It is possible to add multiple target objects, and each of them can be referenced in the template. At the end, the parsing process is invoked, and input text is submitted as the method parameter. Here is where the actual processing is done, and objects are initialized with proper values.

```
<user>
        <username>$user.name$</username>
        <addresses>$user.addresses:{
                <address>
                   <city>$it.city$</city><street>$it.street$</street>
               </address>
            }$
        </addresses>
</user>
```

FIGURE 4. Example of a template that controls text parsing process

```
<user>
        <username>John</username>
        <addresses>
                <address><city>City1</city><street>Street1</street>
                <address><city>City2</city><street>Street2</street>
        </addresses>
</user>
```

FIGURE 5. Input text that matches the template

```
Parser parser = new Parser(file);
parser.compile();
parser.addTargetObject("user", userObject);
parser.parse(FileReader("testinput.txt"));
```

FIGURE 6. Java code that invokes text processing

Figure 7 displays pattern list that is created upon the template compilation. There are two most important pattern types – *text patterns*, which contain text that should be matched and *variable patterns*, which contain info about variables that should be read from user input. Text processor (Fig. 3) goes through the pattern list and tries to match each pattern to a portion of the input text. The pattern list can contain branches that correspond to loops and conditional template parts. When the processor comes to a branch, it chooses one of the possible options, and continues matching. If a matching error occurs, processor goes back to the last branch, and tries another option. If there are not any more options left, an error is generated. The matching is successful if processor manages to traverse the pattern list and the input text at the same time. The result of this process is a sequential list of objects (pattern responses) that reflects the traversal path.

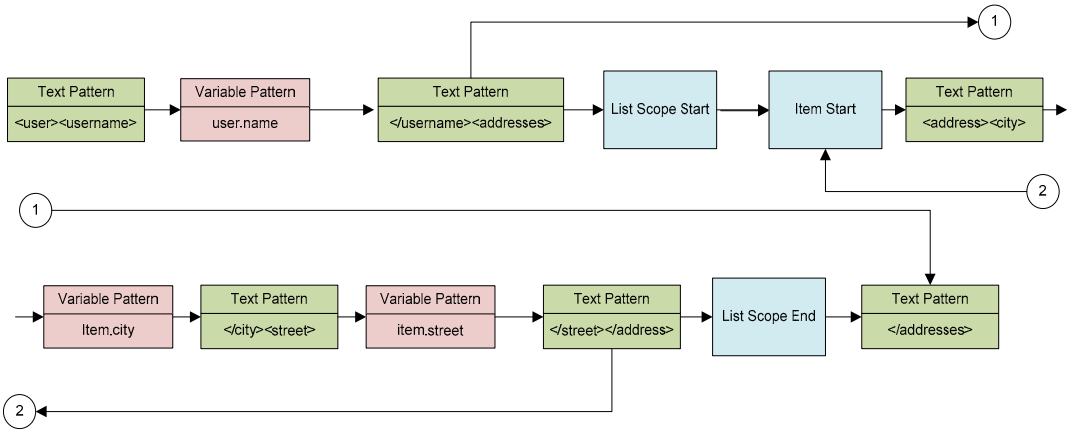

FIGURE 7. Pattern list created for defined template

In our example in Fig. 4, the template contains a part that corresponds to a list of items, which can be repeated many times in the input text. This is reflected in pattern list (Fig. 7) through corresponding branches. Some of the connections can have a processing priority over others – for example, in a loop, backward connection has a priority over the connection that exists out of the loop, because it is more probable.

In the final stage, processor goes through the list of pattern responses and uses reflection to set values of Java objects. Here, controller nodes like *List Scope Start, Item Start* and *List Scope End* are important, because they determine when a new item should be added to the list. If a part of the input text that is matched to a variable node cannot be converted to the proper variable type, a parsing error is generated.

BENEFITS AND DRAWBACKS OF TEMPLATE-BASED TEXT PARSING

Using the templates for creating objects from their representations has the same benefits as the standard template usage – it helps separating the logic that creates the objects from the objects them self. But more importantly, it brings a lot of flexibility to the user, because it is possible to control the way objects are generated in a simple and descriptive way, even without having to recompile parts of Java code.

One of the possible applications is in service oriented applications, especially if they conform the REST service style. Using this approach, services are created as collections of resources, and their states are transferred to the client in a given representation format [7]. Although there are some attempts to describe them in a standard way this is still not done in a widely accepted format, which means that there aren't any automatic tools for proxy generation as for standard SOAP services. As a consequence, developers have to write their own logic both to implement the communication protocol, and to create and parse appropriate resource representations. We think that templates can add needed flexibility and simplicity into this process.

Of course, there are some drawbacks of this approach – object properties are set using reflection, which is slower than setting properties in a standard way. If there are many objects and properties that should be set, and if performance is the key factor, this may not be the appropriate solution. Also, for really complex parsing algorithms, templates may not be powerful enough to fully describe the complete process.

CONLUSION

In this paper we presented the engine we are developing for template-based object creation from their representations. We described the typical applications of this approach, together with some benefits and drawbacks.

At the moment our engine is in a stage of a prototype and the template syntax supports common processing scenarios. Still, some enhancements are needed in order to allow the users to fully control how non-string fields (float values, date-time values etc) are read from their string representations.

We also plan to create template mappings for some popular REST services like Google's GData API. Our final goal is creating a generic service mapping layer, similar to the ORM solutions for data manipulation.

REFERENCES

1. Smarty template Web site, http://www.smarty.net/
2. String template Web site, http://www.stringtemplate.org/
3. T. J. Parr, "Enforcing strict model-view separation in template engines" in *WWW '04*, Proceedings of the 13th international conference on World Wide Web, ACM., New York, NY, USA, 2004, pp. 224–233.
4. The Official Microsoft ASP.NET Site, http://www.asp.net/
5. G. Wachsmuth, "A Formal Way from Text to Code Templates" in *Lecture Notes in Computer Science*, Springer, 2009, vol. 5503, pp. 109–123.
6. MSDN, *Code Generation and T4 Text Templates,* http://msdn.microsoft.com/en-us/library/bb126445.aspx, March 2011
7. L. Richardson and S. Ruby*, RESTful Web Services*, O'Reilly Media, 2007, pp. 79-81