THE DESIGN OF A TEMPLATING LANGUAGE TO EMBED DATABASE QUERIES INTO DOCUMENTS

Sarah Hussein Toman
Roads and Transport Department
College of Engineering/ Unversity of Al-Qadisiyah
Sarah.toman@qu.edu.iq

Abstract

Presenting information from a database to a human readership is one of the usual tasks in software development. Commonly, an imperative language (such as: PHP, C#, Java, etc.) is used to query a database system and populate with the desired information the application's GUI, a web page or a printed report (referred from now on as Presentation Media).

Virtually all database systems are now capable to format, sort and group the data stored in a database, and last but not least to perform calculations against it. These are most of the time enough to prepare the information that is going to be shown on screen or paper. Thus it leaves just one role for the imperative code: to glue the query results to the Presentation Media. This code tends to become repetitive and grows proportionally with the complexity of the Presentation Media. The need for software developers to write this imperative code can be eliminated thought. Instead, the markup code (HTML, LaTEX, etc) can have the ability to bind its elements directly to the database system. To achieve this ability, I propose mixing the Presentation Media's markup code with a Templating Language.

This paper elaborates the design of a Templating Language, a declarative language that adds annotations to any markup code regarding what data will be queried and how should it be integrated in document. For this markup code to be consumed, there won't be necessary to implement any database query abilities in the process that renderers it. Instead, a preprocessor will be invoked to interpret the Templating Language, connect to the database system and query the desired data, respectively to generate the final markup code.

After analyzing several scenarios of GUIs and printed reports with a different range of presentation complexity, the design was aimed to handle from simple value insertions and markup code loops for multiple row results, to nested loops with interdependent queries, variables to share data between queries and automatic list numbering. The data queries are specified in a subset of SQL, taking advantage of the data selection, formatting, grouping and sorting capabilities of database systems. As most programming languages supports RegEx, to facilitate a faster implementation of the preprocessor, the Templating Language is parsed and validated using a RegEx set provided in this paper.

The method described in this paper simplifies the development and maintenance of a software by reducing the boilerplate code and adhering to the Separation of Concerns design principle, as the code will be organized in operational distinct layers (logic and presentation). Also, it enhances teamwork by allowing a separation of work based on skill set, a designer being able for example to modify the structure of a data report without needing the intervention of a programmer too.

Keywords: *Templating Language, Preprocessor, SQL, Database, Data Report, User Interface, Separation of Concerns*

1. Introduction

A Templating Language or Template Language, is the language that describes how a software system should combine a document template with a data model to produce a result document [1]. The result document may be any kind of formatted output such as a text document or a web page fragment. This software system is called a template processor [2].

In the context of presenting information from a database system through a templated report or user interface, there are a series of advantages that push the use of templates towards an efficient practice [3,4,5]:

- There is less effort involved in filling a document with the desired information, as this task would imply using repetitive imperative code, with few or even no logic branching;
- Separates the program logic and presentation in two loosely coupled distinct layers, achieving the separation of concerns principle;
- Since the presentation layer is separated from the program logic, as a template, its layout may be maintained by the user with no programming skills (such as a designer) without the risk of altering the program's logic. In such a dynamic society, the requirements of information presented in the user interface or report may vary over time: as an example the clauses of a contract model may get modified often, being required in this case only the modification of the contract document template without the software's developer intervention.

This paper presents the design semantics and syntax of a templating language specialized in pulling information from a relational database system, and integrating it into the source template, which may be in any arbitrary formatted text or markup code combined with the templating language elements.

This templating language should be able to be combined with a document markup language such as TeX, LaTeX for report generation purposes, resulting: receipts, invoices, contracts, letters or any other kind of paperwork that references information from a database system. Also combined with HTML, it might serve for web page fragment acting as a web templating engine.

As Structured Query Language (SQL) is nowadays the most widely used relational database system query language, and also capable enough to sustain complex data retrieval operations such as: conditional branching, value aggregation and formatting
[5]; SQL represents the ideal model for a database-centric templating language.

A query performed with SQL returns a result set, which effectively is the object model of a database table, reassembling the retrieved data in a set of rows with column names and values. The challenge in allowing templating using SQL is to introduce a set of semantic elements that provide information regarding the conversion of a result set in order to fit the common requirements of a template.

The proposed language is an adaptation of a subset of SQL, tailored for templating. This subset is limited to data retrieval, since data insertion, updating and deletion is out of the scope of a templating language. The adaptation provides the semantic and syntactic constructs for the template processor to identify and evaluate the SQL queries residing in a source template, then to fill the template accordingly to the result set.

2. Related Works

I have identified in the academic literature few previous approaches of preprocessing documents for the purpose of populating them with data retrieved from a database:

- **A Document-driven Approach to Database Report Generation** (*Chan*)[6] – presents a document transformation language that uses as report model a SGML Document Type Definition;
- **SuperSQL: an extended SQL for database publishing and presentation** (*Toyama*) [7] – extends SQL to be able to render its query results as various media for publishing and presentations.
- **Enforcing Strict Model-View Separation in Template Engines** *(Parr)* [8]: Presents Text-based template engines for object-language-agnostic in the sense that they solely process strings.
- **Language Oriented Programming: The Next Programming Paradigm** *(Dmitriev)* [9]: describes an automatic templating language generator that copies the object language and adds template concepts to it.

- **An Architecture for an XML-Template Engine Enabling Safe Authoring** (*Hartmann*) [10] – presents a safe template engine for XML language described by XML schemes.

3. Design

All components involved in the transformation process of a template to the final result is depicted in Figure 1. The **Template Source** is the portion or an entire document of plain text or any markup code that follows to be transformed. It may be HTML, RTF, TeX, LaTeX, Office Open XML Document, etc. The template source combined with one or more template tags defines a **Template.**

This template is meant to become the input of the **Template Processor,** that is the computer program that parses the tags of a template, and merges the template source with the data obtained from a database system according to the instructions found in those tags. The result of the template processor represents the **Template Output.**

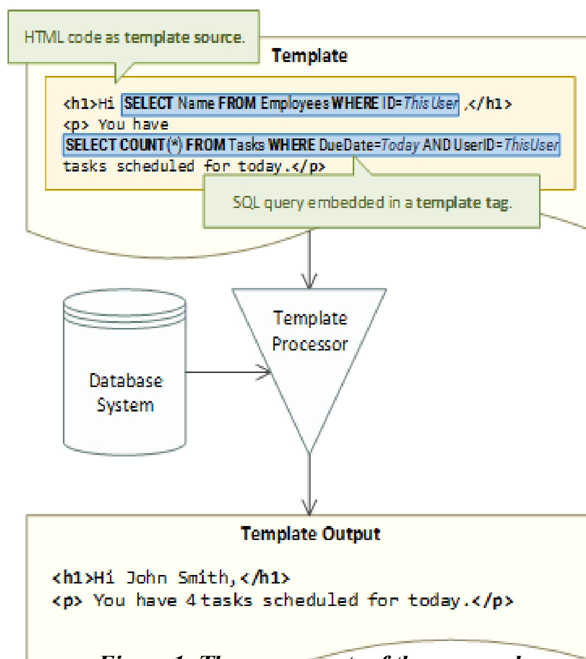

Figure 1: The components of the proposed Templating Language transformation process.

3.1. Identifying the Requirements

The purpose of this section is to outline a minimum set of capabilities that this templating language should feature in order to achieve its desired practicality.

As the mission of this language is to embed SQL queries into a Template and facilitate a Template Processor to substitute them with the corresponding data, those substitution ways should fit the whole range of a SQL query result types.

A SQL query may return two types of result:

- **scalar** – a single value, that can be the result of a SQL function (such as COUNT, SUM, AVERAGE, MIN, MAX, etc.) or a single data row containing a single data column;

- **table** – a set of multiple data rows and columns.

A SQL expression that returns a scalar value will be simply substituted in a template with the respective value. But in the case of a table result, this would be unpractical as the programmer will need a finer control over how those data rows and columns will be integrated in the template. The values may need to be inserted in different locations of a template fragment that reassembles the markup for a list, table or menu, for example.

As a set of table rows can be seen as a set of tuples where all items of the set shares the same data structure (columns), it denotes that the template fragment where the query result should be integrated defines the prototype of a data row. Thus, a copy of this template fragment will be evaluated and inserted in the template output for each row instance of the query result. This template fragment will be called a **sequence block.**

For a greater flexibility, the template processor should allow input arguments to be passed to the SQL expressions embedded in a template. This mechanism lets the programmer to adjust the queries to operate according to a particular context, without the need to effectively modify the template tags. As an example, in the case of a receipt template, the transaction ID could be passed as an input argument, so the SQL queries can use this argument in its clauses to retrieve the corresponding information from database.

An input parameter would be accessible through a **Template Variable**. A Template Variable is essentially a macro
that follows to be expanded to a corresponding value during the template evaluation. The value of a Template Variable is unknown at the moment of defining the template.

Also a Template Variable would be a convenient way to store query results that are used more than once in a template, or to pass the result of a previous query in a SQL expression.

4. Syntax

The only syntactic elements of a template that are significant for a template processor, are the Template Tags. Before parsing those tags, they should first be located in the template, as they are scattered within fragments of Template Source.

To provide a clear distinction between Template Source and Template Tags, a pair of delimiter character sequence will be used, called **Tag Delimiters**.

Template Tags once located, will be parsed by the template processor and eventually substituted with data retrieved from a database system according to the semantics of the respective Template Tags. One of the aims is to create a lightweight Template Processor, featuring as less as possible logical flow for an easy implementation in a different range of programming languages. For this purpose, the Template Tags will be located using Regular Expressions (RegEx).

Furthermore, the semantics of a Template Tag will be extracted also via Regular Expressions capture groups, making RegEx patterns solely responsible for syntax interpretation and validation. Thus, the syntax of Template Tags should be designed in a manner that overcomes the potential limitations of RegEx.

RegEx with an evolution of nearly 60 years, became a powerful standard in string searching and processing; simple parsing, data validation, scraping and wrangling; being nearly omnipresent in software development platforms. Programming languages such as C#, Java, C++, VB.NET, JavaScript, Perl, PHP, Python provides built-in support for RegEx, or via a standard library [11].

Note: Those Regular Expressions will use named capture groups for the sake of readability, although not supported by all RegEx engines. Those RegEx patterns may require to be converted to indexed capture groups prior a production useThe grammar of the Template Tags will be described in the next sections in the form of syntax (railroad) diagrams.

4.1 Tag Delimiters For the Template Tags to be identified in the template source, they must be separated with a distinctive character sequence, that will be called a **Tag Delimiter**. This sequence should have an incidence rate as small as possible in natural languages and most common markup languages.

As the Template Tag is a sequence of characters of an arbitrary length, its boundary should be easily inferred with RegEx by using two delimiters: **Tag Opening Delimiter**, respectively **Tag Closing Delimiter**.

For the Template Tag Opening Delimiter, an open bracket followed by a question mark "(?" was chosen. This decision was driven by the goal for the delimiter to be easy to remember: the open bracket marks a start of a boundary while the question mark is a common analogy of querying. Even if those two characters are commonly used in both natural and markup languages, an open bracket to be proceeded by a question mark is an uncommon case.As the Template Tag Closing Delimiter, "?)" was chosen. This sequence has a frequent incidence in natural and computer languages, but its meaning is to mark the end of the body of a Template Tag which mainly consists of SQL code, where this incidence is negligible, but remaining on the attention of the user to be escaped.

In the case when a template contains a sequence identic to a Tag Delimiter which is not intended to denote a Template Tag delimitation, it should be replaced with an escape sequence. This sequence will be "(??" for the Tag Opening Delimiter, respectively "??)" for the Tag Closing Delimiter. The Template Processor should guarantee de-escaping of those sequences. The Template Tags are matched via the RegEx pattern presented in Expression 1. The body of the Template Tag will be returned in the capture group named "body". All RegEx patterns from the further sections of the paper will operate on the value returned by this capture group. The de-escaping procedure of the Tag Delimiters consists in replacing the match of Expression 2 with the value of "replacement" capture group.

```
\(\?(?!\?)\s*(?<body>.*?)\s*(?<!\?)\?\)
```

Expression 1: RegEx Matching Template Tags

```
(?<replacement>\(\?)\?|\?(?<replacement>\?\))
```

Expression 2: RegEx Matching the Tag Delimiters Escaping Sequences

4.2 Variables

A Template Variable may be present in any place of the body of a Template Tag. To be easily identifiable via RegEx, a variable identifier will be decorated with a prefix and suffix symbol. As SQL code is the predominant content of a Template Tag, it is ideally for this prefix/suffix to avoid being one of the SQL reserved characters or one commonly used in queries. The dollar sign ("$") has the previously mentioned attributes, making it the choice for both: prefix and suffix symbol characters.If a dollar symbol is present in the body of a Template Tag for other purposes than marking a variable, it should be escaped by prefixing it with another dollar symbol ("$$"). The same rule applies for the sequences of multiple such signs also. As an example "$$$" needs to be escaped as "$$$$", as the de-escaping process resumes in removing one symbol when the sequence contains more than one symbols. To lower the chance of a false positive Tag Variable identification in a body with potential dollar signs left without escaping, a set of rules will be enforced for a stricter validation of the variable identifier strings:

- It may contain lower and uppercase letters. Like SQL, there will be no letter casing distinction;
- It may also contain digits. A digit cannot be the first character of the identifier, to avoid occurrences such as "$25" that is one of the most common form of use for this symbol;
- It can contain the following symbols "-", ".", "_". The minus symbol cannot start the identifier for the same reason as in the case of a digit;

Expression 3 will match Template Variables from the body of a Template Tag.

```
\$(?<identifier>[a-zA-Z_.][a-zA-Z0-9\-_.]+)\$
```

Expression 3: RegEx Matching Template Variables

```
(?<replacement>\|+)\||(?<val>\$+)\$
```

Expression 4: RegEx Matching Template Variables Escaping Sequences

مجلة كلية التربية جامعة واسط
ISSN: 4217-1994
Education College Journal

The Template Variable's identifier is returned via the "identifier" capture group. The de-escaping is performed by replacing the match of Expression $_4$ with the value of its capture group named "replacement":

5. Queries

As the sole purpose of the SQL query from a Tag Template is to interrogate a database system, the use of SQL statements is redundant, "SELECT" being the only allowed statement. To simplify those queries, the "SELECT" statement should be omitted from all SQL queries. In this way "SELECT FirstName, LastName FROM Users WHERE ID=1" will become "FirstName, LastName FROM Users WHERE ID=1". Using a SQL statement in queries should result in an error during template processing, to prevent the evaluation of potentially unwanted statements such as "UPDATE", "DELETE", "DROP", etc. A query may not have multiple SQL interrogations separated by ";" as the Template Tags will operate on a single result set. Thus, the ";" statement terminator should result in an invalid query to prevent multiple interrogations.

5.1 Scalar Queries

A scalar query describes a query that returns a single value, opposite to a set of rows and columns that can be individually

accessed. In a template, a scalar query can be introduced by using the following Template Tag described in Figure 2.

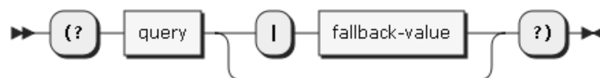

Figure 2: The syntax diagram of the Scalar Query Tag

مجلة كلية التربية
جامعة واسط
ISSN: 4217-1994
Education College Journal

This tag, called Scalar Query Tag will be substituted in a template with the value returned by the SQL interrogation specified in the "query" argument. If the respective query returns no value (null), the value of the optional "fallback-value" argument will be used instead if it is specified, otherwise the tag will be replaced with an empty string.

<table>
<tr><th>TEMPLATE SOURCE</th><th>TEMPLATE OUTPUT</th></tr>
<tr><td><table>
<tr>
<td>Current Sales Agent:</td>
<td>
(? Name FROM ShiftRegistry
WHERE ShiftStart >= NOW()
AND ShiftEnd <= NOW()
| N/A (store is closed) ?)
</td>
</tr>
<tr>
<td>Completed Orders:</td>
<td>
(? COUNT(ID) FROM Orders
WHERE Date=CURDATE()
AND Completed=False ?)
out of (? COUNT(ID) FROM Orders ?)
</td>
</tr>
<tr>
<td>Total Sales:</td>
<td>
$(? SUM(Price FROM Sales
WHERE Date=CURDATE())?)
</td>
</tr>
</table></td><td><table>
<tr>
<td>Current Sales Agent:</td>
<td>
Abra Amina
</td>
</tr>
<tr>
<td>Completed Orders:</td>
<td>
8
out of
14
</td>
</tr>
<tr>
<td>Total Sales:</td>
<td>
$348
</td>
</tr>
</table></td></tr>
</table>

If the query results in a set of rows or columns, it will be transformed to a scalar value. That value will be a string that contains all values of the first column delimited by a comma character (","), while the values from the other columns will be discarded. This feature can be useful for creating simple enumerations like in the following example.

TEMPLATE SOURCE	TEMPLATE OUTPUT
You are currently subscribed to the following television services: (? Name FROM Services WHERE CustomerID=$Customer$?).	You are currently subscribed to the following television services: Digital Base Channel Pack, Digital Documentary Channel Pack, Online Access.

```
\A(?<query>.+?)\s*(?(?<!\|)\|(?!\|)\s*(?<fallback-value>.+?))?\Z
```

Expression 5: RegEx Matching Scalar Query Tags

The Scalar Query Tag is matched and parsed with Expression 5. The SQL query code is returned in the "query" capture group, while the fallback value argument in the "fallback-value" group. The entire match should be replaced with the value resulted from the query.

5.2 Sequence Queries

Queries that return multiple rows and columns most likely requires a finer control over the formatting of the result than the simple comma separated enumeration provided by the Scalar Query Tag.

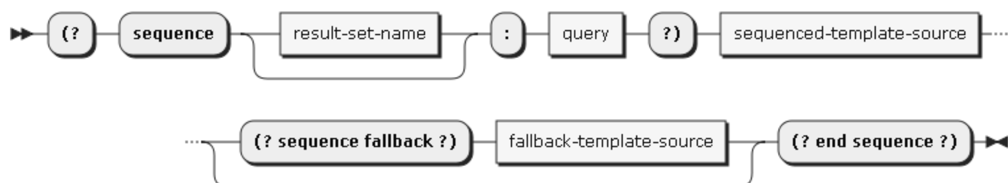

Figure 3: The syntax diagram of a Query Sequence Block

The sequence query provides an iteration of the result set by using a Template Source fragment to describe the formatting prototype of the data rows. That Template Source fragment will be called **Sequenced Template Source**.

The Sequenced Template Source will be enclosed between "sequence" and "end sequence" Template Tags in a **Query Sequence Block**, with the syntax detailed in Figure 3.

As in the case of scalar query, the SQL interrogation code is specified in the ”query” argument. For each row of the result set, a new instance of the Sequenced Template Source fragment will be inserted in the Template Output, after its inner tags are evaluated.

A " (?sequence fallback ?) " Template Tag may be used after the Sequenced Template Source to indicate that the Template Source fragment located between this tag and respectively "(? end sequence ?)" tag (presented as "fallback-sequence-source" in Figure 3) should substitute the current Query Sequence Block if the interrogation doesn’t return any row. The presence of this Template Tag is optional, an empty string being its default value.

In a Sequenced Template Source, the value of a column can be accessed through a **Field Accessor**.

The Field Accessor inserts the value of a column from the current row of a Sequence Block. Its syntax is presented in Figure 4. The name of that column is specified via the "column-name" argument. The lookup will be performed on the result set. Thus in the case of queries that uses column

aliases via the "AS" keyword, the column name mentioned in a Field Accessor should match its alias.

Figure 4: The syntax diagram of a Field Accessor Tag

If the column specified in the Field Accessor has no value (null) or it doesn't exist, the value of "fallback" argument will be used instead if it is set, otherwise the tag will be substituted with an empty string.

For convenience, a fictive column named "#" is added to a result set to introduce a simple row numbering method. This column will contain the 1-based index of the current row of a Query Sequence Block. The following is an example of using Query Sequence Blocks and Field Accessors to fill a HTML table:

```
<body>
 <h1>Book Inventory</h1>
 <table>
  <thead>
   <tr>
    <th>№</th>
    <th>Title</th>
    <th>Author(s)</th>
    <th>Publisher</th>
    <th>Year</th>
   </tr>
  <thead>
<tbody>
   (? sequence: * FROM Books ?)
    <tr>
     <td>(? # ?)</td>
     <td>(? Title ?)</td>
     <td>(? Authors ?)</td>
     <td>(? Publisher ?)</td>
     <td>(? Year ?)</td>
    </tr>
   (? fallback ?)
    <tr>
     <td colspan="5">
      Sorry, there are no available
      books right now.
     </td>
    </tr>
   (? end sequence ?)
  </tbody>
 </table>
</body>
```

For a greater flexibility, the Query Sequence Blocks can be nested. A Sequenced Template Source can contain one or more

Query Sequence Blocks that can access values from its ascendant Query Sequence Blocks.

To be able to share values between nested blocks, each Query Sequence Block can have a named result set specified in the "name" argument of the Opening Tag of a Sequence Query Block.

```
\A(?(?=.*\/.*)(?<result-set-name>[a-zA-Z0-9\-_.]+\/))(?<column-name>(?:[a-zA-Z0-9\-_.]+|\#))(\s*\|\s*(?<fallback-value>.+)\s*)\Z
```

Expression 6: RegEx Matching a Field Accessor Template Tag

The Field Accessor Template Tag parsed using Expression 6, that returns the following capture groups:

- **-set-name** – the identifier of the target result set. The result set corresponding to the current Query Sequence Block will be assumed if there is no value specified;
- **column-name** – the name of the target table column;
- **fallback-value** – the default value.

There is no direct way to access ancestor's column values in SQL code as Field Tags cannot operate inside of tag bodies. However, it can be overcome by using a Variable Assignment Template Tag to copy the column value into a variable.

In the right side can be seen an example of nested Query Sequence Blocks.

```
<body>
 <h1>Client Information</h1>
 (? store info: *
   FROM Clients WHERE Id="$ID$" ?)
 (? store phones: Number
   FROM PhoneNumbers
   WHERE ClientId="$ID$" ?)
 (? store emails: Address
    FROM Emails
    WHERE ClientId="$ID$" ?)
 <table>
  <tr>
   <td>First Name:</td>
   <td>(? info/FirstName ?)</td>
  </tr>
  <tr>
   <td>Last Name:</td>
   <td>(? info/LastName ?)</td>
  </tr>
  <tr>
   <td>Address:</td>
   <td>(? info/Address | N/A ?)</td>
  </tr>
  <tr>
   <td>Phone:</td>
   <td>(? phones/Number | N/A ?)</td>
  </tr>
  <tr>
   <td>E-mail:</td>
   <td>(? emails/Address | N/A ?)</td>
  </tr>
 </table>
</body>
```

The opening tag of a Query Sequence Block is matched by Exp. 7, capturing the following groups:

- **result-set-name** – the name of the result set, if it is specified;
- **query** – the SQL interrogation;

The Sequence Query Fallback Tags and Sequence Query Closing Tags are matched via RegEx using the Exp. 8 respectively Exp. 9 pattern.

5.3 Stored Queries

When different values from the same data row needs to be inserted in different locations of a template source, each value should be obtained by querying each time the RDBMS via a Scalar Query Tag. Consequent calls to the RDBMS to retrieve another data column value from

a certain data row which is frequently used in that template, would degrade the performance of the template processing stage.

```
\Asequence\s*?(?<result-set-name>[a-zA-Z_.][a-zA-Z0-9\-_.]+)\s*:\s*(?<query>.*?)\Z
```

Expression 7: RegEx Matching the Sequence Query Opening Tags

One possible work-around would be to use a Query Sequence Block to wrap all those values and iterate the only one data row returned by the query while inserting the values with Field Accessors. But this approach would introduce unnecessary complexity in writing templates.

For this purpose, the Stored Query Tag will perform a query and store its result set for the entire life time of the template processing stage, allowing the column values to be accessed from any part of the document via Field Accessors.

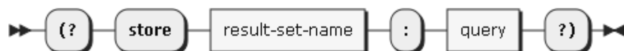

Figure 5: The syntax diagram of a Stored Query Tag

The syntax of the Stored Query Tag is described in Figure 5

The name of the result set is specified through the "name" parameter. If another result set with the same name is defined either by a Query Sequence Block or Stored Query Tag, it will get replaced with the succeeding one.

The SQL interrogation is specified in the "query" argument, similar to the argument with the same name of the Scalar Query Tag. If the query results in multiple rows, they will be merged in a single row: each column will contain a comma separated enumeration of all its values across all rows from the result set. An example of using Stored Query Tags can be found in the right side.

```
<body>
 <h1>Event Details</h1>
 (? store event: * FROM Events
                  WHERE ID="$ID$" ?)
 <table>
  <tr>
    <td>Name:</td>
    <td>(? event/name ?)</td>
  </tr>
  <tr>
   <td>Date and Time:</td>
   <td>(? event/datetime ?)</td>
  </tr>
  <tr>
    <td>Location:</td>
    <td>(? event/location ?)</td>
  </tr>
 </table>
</body>
```

The Stored Query Tag is parsed via RegEx with Expression 10 that returns the following capture groups:

- **result-set-name** – the identifier of the result set that will hold the query result;
- **query** – the SQL interrogation.

```
\Astore\s*(?<result-set-name>[a-zA-Z0-9\-_.]+)\s*:\s*(?<query>.+?)\Z
```

Expression 10: RegEx Matching the Stored Query Tags

This template tag will be substituted with an empty string in a Template Source.

6. Variables

There may be scenarios where two or more queries may depend on the result of a previous query, using those results in their SQL clauses. As an example, in two nested Query Sequence Blocks, the inner block may need to filter the items according to the current row of its ancestor block.

6.1 Assigning Variables

A variable may be assigned by using the Variable

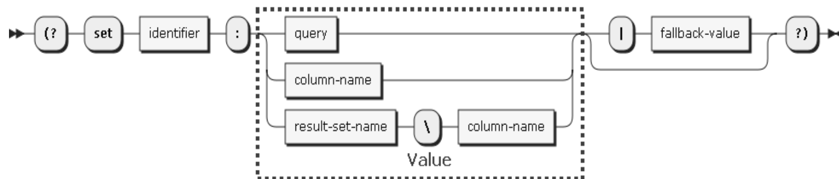

Figure 6: The syntax diagram of a Variable Assignment Tag

Assignment Tag with the syntax detailed in Figure 6.

The identifier of a variable is specified via the "name" argument and

\Asequence\sfallback\Z	\Aend\ssequence\Z
Expression 8: RegEx Matching the Sequence Query Fallback Tag	***Expression 9: RegEx Matching the Sequence Query Closing Tag***

should follow the naming constraints described in section

4.2 Variables If there is a previous value assigned to a variable with the same identifier, that variable will get updated with the new value.

There may be two types of assignment:

1. **Query-based** – the result of the SQL interrogation specified in "query" argument is assigned as the variable's value;
2. **Field-based** – the value of the variable is retrieved from a result set. The name of the table column should be specified in the "column-name" argument. If no result set identifier is specified via the "result-set-name" argument,
3. the result set corresponding to the current Sequence Query Block will be assumed.

The default value of "fallback-value" argument is an empty string, unless it's explicitly set. A variable will not be considered as assigned with an empty (null) value.

Next is an example for using a variable to pass the current category to a descendant Sequence Query Block's SQL interrogation, in order to group items in a list:

```
<body>
 <h1>Flowers By Families</h1>
  (? sequence group: DISTINCT Family FROM Flowers ORDER BY Family ?)
   <h2>(? group/family ?)</h2>
   <ul>
    (? set family: group/family ?)
    (? sequence: Name FROM Flowers WHERE Family="$family$" ORDER BY
Name ?)
     <li>(? Name ?)</li>
    (? end sequence ?)
    </ul>
  (? end sequence ?)
</body>
```

The Variable Assignment Tag is parsed with Expression 11, that returns the following capture groups:

- **identifier** – the variable identifier;

- **query** – the SQL interrogation whose result should be assigned to the variable. A non-empty value indicates a query-based assignment;
- **column-name** – the identifier of the table column. A non-empty value indicates a field-based assignment;
- **result-set-name** – the identifier of the target result set. Empty if not specified or the assignment type is not field-based;
- **fallback-value** – the fallback value.

```
\Aset\s*?(?<identifier>[a-zA-Z_.][a-zA-Z0-9\-
_.]+)\s*:\s*(?:(?(?=.*\/.*)(?<result-set-namet>[a-zA-
Z0-9\-_.]+\/))(?<column-name>(?:[a-zA-Z0-9\-
_.]+|\#))|(?<query>.*?))\s*(?(?<!\|)\|(?!\|)\s*(?<fallback-
value>.+?))?\Z
```

Expression 11: RegEx Matching the Variable Assignment Tag

Note: Expression 11 will also infer the intended assignment type by validating the arguments supplied in the "Value" section of Figure 6. Firstly,

a field-based assignment is implied. Eventually the "column-name" and "result-set-name" validation will fail, indicating a query-based assignment. A SQL expression would never validate as a column name or result set name since it will contain at least one blank space in its body. This Template Tag will be replaced with an empty string in the Template Output.

6.2 Introducing Variables in Template Source

In the body of a Template Tag, a variable is introduced by surrounding its identifier with the "$" symbol as mentioned earlier. But in a Template Source, a variable can be introduced via the Variable Tag.

A Variable Tag will get substituted in a Template Source with the value of the variable with the specified identifier, using the syntax described in Figure 7.

Figure 7: The syntax diagram of a Variable Tag

The value of the optional "default" argument will be used if the variable is not assigned, otherwise if not set, the Template Tag will get substituted with an empty string.

```
\Avar\s*?(?<identifier>[a-zA-Z_.][a-zA-Z0-9\-_.]+)\s*(\|\s*(?<fallback-value>.*?))?\Z
```

Expression 11: RegEx Matching Variable Tags

Expression 11 will serve for parsing the Variable Tag, and returns as capture groups:

- **identifier** – the identifier of the target variable;
- **fallback-value** – the default value.

The entire match will be substituted to the variable's value if is available, otherwise the contents of "fallback" capture group.

7. Conclusion

One of the common use of template engines, noticeably in web development, is to preprocess documents in order to populate them with information stored in a database. This involves a software developer to write code to query the database system to obtain the desired information, then to pass it to a template engine. In this paper I have elaborated the design of a templating language that directly binds the template with the data source, eliminating the need of writing any imperative code and providing a proper separation of concerns.This templating language represents a foundation to interpolate markup code with database queries. It provides means to introduce scalar values; or to manipulate blocks of markup code in order to iterate sequences of data rows in the final document.Relying on SQL expressions to query the desired information from database, this templating language takes advantage of powerful abilities of filtering, sorting, aggregating and formatting data. This also facilitates a short learning curve for those familiarized with database systems.Template Variables may be used to pass parameters from the template processor caller or values from preceding queries to SQL expressions. Also Stored Result Sets may improve template processing times by reducing the database roundtrips, caching the query results whose values are referenced in multiple locations.With a simple syntax, this language can be easily implemented, being parsed with regular expressions that I detailed in this paper.

8. Future Work

The current design assumes the template processor to be aware of the connection string that points to the target data source, no mechanism being provided to embed the connection string into a template. Moreover, there is no support for interrogating multiple databases from a single template. An eventual support of alternative tag delimiters would be ideal for allowing template tags to be obscured from the syntax checking of code editors, providing a better editing experience by disguising Template Tags as comments or any other ignorable annotations for common markup languages.

References

[1] Terence P., “Enforcing Strict ModelView Separation in Template Engines”, ACM, New York, USA, May 17–20, 2004.

[2] Molham K., “Design and Implementation of an Efficient Approach for Custom fields and Formulas with SAP HANA”, MSc. Thesis, Department of Data and Knowledge Engineering, University of Magdeburg, July 20, 2015.

[3] Florian H. and et al, “Generating Safe Template Languages”, ACM, October 4–5, Denver, Colorado, USA,2009.

[4] Alan B., “Learning SQL, Second Edition”, O'Reilly & Associates Inc., 2009.

[5] Andy O. and Robert S., “SQL A Beginner’s Guide”, Third Edition, McGraw-Hill, 2009.

[6] Daniel K.C.,” A Document-driven Approach to Database Report Generation” IEEE, Pages 925 – 930, AUG-1998.

[7] T. Motomichi,” SuperSQL: An Extended SQL for Database Publishing and Presentation”, ACM SIGMOD International Conference on Management of Data, Seattle, Washington, USA, June 2-4, 2000.

[8] T. J. Parr, " Enforcing Strict Model-View Separation in Template Engines", the 13th International Conference on World Wide Web, pages 224–233. ACM, May 2004.

[9] S. Dmitriev, " Language Oriented Programming: The Next Programming Paradigm", White Paper, JetBrains, 2004. URL http://www.onboard.jetbrains.com/is1/articles/04/10/lop.

[10] F. Hartmann, " An Architecture for an XML-Template Engine Enabling Safe Authoring", In Proceedings of the 17th International Workshop on Database and Expert Systems Applications (DEXA 2006), pages 502– 507. IEEE Computer Society, 2006.

[11] S. Yu, “Regular Languages”, in Handbook of Formal Languages, G. Rozenberg and A. Salomaa eds. pps. 41-110, Springer, 1998.