SharpTAL Documentation

Release 2.2.0

Roman Lacko

April 03, 2014

SharpTAL is an HTML/XML template engine for .NET platform, that you can use in any application running on .NET 4.0.

The template engine compiles HTML/XML templates into .NET assemblies.

It contains implementation of the ZPT language (Zope Page Templates). ZPT is a system which can generate HTML, XML or plain text output. ZPT is formed by the TAL (Template Attribute Language), TALES (TAL Expression Syntax) and the METAL (Macro Expansion TAL).

CHAPTER 1

Getting the code

Binaries are provided as a NuGet package (https://nuget.org/packages/SharpTAL).

The project is hosted in a GitHub repository

Please report any issues to the issue tracker.

CHAPTER 2

Introduction

Using a set of simple language constructs, you control the document flow, element repetition and text replacement.

The basic TAL (Template Attribute Language) example:

```
<html>
  <body>
    <h1>Hello, ${"world"}!</h1>
    <table>
      <tr tal:repeat='row new string[] { "red", "green", "blue" }'>
        <td tal:repeat='col new string[] { "rectangle", "triangle", "circle" }'>
           ${row} ${col}
        </td>
      </tr>
    </table>
  </body>
</html>
```

The ${...} notation is short-hand for text insertion. The C# expression inside the braces is evaluated and the result included in the output. By default, the string is escaped before insertion. To avoid this, use the *structure:* prefix:

```
<div>${structure: ...}</div>
```

The macro language (known as the macro expansion language or METAL) provides a means of filling in portions of a generic template.

The macro template (saved as main.html file):

```
<html metal:define-macro="main">
  <head>
    <title>Example ${document.title}</title>
  </head>
  <body>
    <h1>${document.title}</h1>
    <div id="content">
      <metal:tag metal:define-slot="content" />
    </div>
  </body>
</html>
```

Template that imports and uses the macro, filling in the `content` slot:

```
<metal:tag metal:import="main.html" use-macro='macros["main"]'>
  <p metal:fill-slot="content">${structure: document.body}<p/>
</metal:tag>
```

In the example, the statement *metal:import* is used to import a template from the file system using a path relative to the calling template.

Sample code that shows how easy the library is to use:

```
var globals = new Dictionary<string, object>
{
    { "movies", new List<string> { "alien", "star wars", "star trek" } }
};

const string body = @"<!DOCTYPE html>
<html tal:define='textInfo new System.Globalization.CultureInfo(""en-US"", false).TextInfo'>
    Favorite sci-fi movies:
    <div tal:repeat='movie movies'>${textInfo.ToTitleCase(movie)}</div>
</html>";

var template = new Template(body);

var result = template.Render(globals);

Console.WriteLine(result);
```

Here's the console output:

```
<!DOCTYPE html>
<html>
   Favorite sci-fi movies:
   <div>Alien</div><div>Star Wars</div><div>Star Trek</div>
</html>
```

CHAPTER 3

License

This software is made available under Apache Licence Version 2.0.

CHAPTER 4

Contents

4.1 Language Reference

4.1.1 Template Attribute Language (TAL)

The *Template Attribute Language* (TAL) is an attribute language used to create dynamic XML-like content. It allows elements of a document to be replaced, repeated, or omitted.

An attribute language is a programming language designed to render documents written in XML markup. The input XML must be well-formed. The output from the template is usually XML-like but isn't required to be well-formed.

The statements of the language are document tags with special attributes, and look like this:

```
<p namespace:command="argument">Some Text</p>
```

In the above example, the attribute `namespace:command="argument"` is the statement, and the entire paragraph tag is the statement's element. The statement's element is the portion of the document on which this statement operates.

Each statement has three parts: the namespace prefix, the name, and the argument. The prefix identifies the language, and must be introduced by an XML namespace declaration in XML and XHTML documents, like this:

```
xmlns:namespace="http://example.com/namespace"
```

The statements of TAL are XML attributes from the TAL namespace. These attributes can be applied to an XML or HTML document in order to make it act as a template.

The TAL namespace URI is currently defined as:

```
xmlns:tal="http://xml.zope.org/namespaces/tal"
```

This is not a URL, but merely a unique identifier. Do not expect a browser to resolve it successfully. This definition is required in every file that uses ZPT. For example:

```
<div xmlns="http://www.w3.org/1999/xhtml"
     xmlns:tal="http://xml.zope.org/namespaces/tal">
     .... rest of the template here ...
</div>
```

All templates that you use ZPT in *must* include the `xmlns:tal="http://xml.zope.org/namespaces/tal"` attribute on some top-level tag.

Statements

A **TAL statement** has a name (the attribute name) and an argument (the attribute value). For example, a `tal:content` statement might look like `tal:content="string:Hello"`. The element on which a statement is defined is its **statement element**. Most TAL statements are *expressions*, but the syntax and semantics of these expressions are not part of TAL.

Note: *TALES* is used as the expression language for the "stuff in the quotes" typically. TALES is documented separately.

These are the available TAL statements:

- `tal:attributes` - dynamically change element attributes.
- `tal:define` - define variables.
- `tal:condition` - test conditions.
- `tal:content` - replace the content of an element.
- `tal:omit-tag` - remove an element, leaving the content of the element.
- `tal:repeat` - repeat an element.
- `tal:replace` - replace the content of an element and remove the element leaving the content.

Order of Operations

When there is only one TAL statement per element, the order in which they are executed is simple. Starting with the root element, each element's statements are executed, then each of its child elements is visited, in order, to do the same.

Any combination of statements may appear on the same element, except that the `tal:content` and `tal:replace` statements may not be used on the same element.

TAL does not use use the order in which statements are written in the tag to determine the order in which they are executed. When an element has multiple statements, they are executed in this order:

1. `tal:define`
2. `tal:condition`
3. `tal:repeat`
4. `tal:content` or `tal:replace`
5. `tal:omit-tag`
6. `tal:attributes`

There is a reasoning behind this ordering. Because users often want to set up variables for use in other statements contained within this element or subelements, `tal:define` is executed first. `tal:condition` follows, then `tal:repeat` , then `tal:content` or `tal:replace`. Finally, before `tal:attributes`, we have `tal:omit-tag` (which is implied with `tal:replace`).

tal:attributes

Replace element attributes

Syntax

`tal:attributes` syntax:

```
argument              ::= attribute_statement [';' attribute_statement]*
attribute_statement   ::= attribute_name expression
attribute_name        ::= [namespace-prefix ':'] Name
namespace-prefix      ::= Name
```

Description

The `tal:attributes` statement replaces the value of an attribute (or creates an attribute) with a dynamic value. The value of each expression is converted to a string, if necessary.

Note: You can qualify an attribute name with a namespace prefix, for example `html:table`, if you are generating an XML document with multiple namespaces.

If an attribute expression evaluates to `null`, then that attribute is deleted from the statement element.

If the expression evaluates to the symbol `default` (a symbol which is always available when evaluating attributes), its value is defined as the default static attribute value.

If you use `tal:attributes` on an element with an active `tal:replace` command, the `tal:attributes` statement is ignored.

If you use `tal:attributes` on an element with a `tal:repeat` statement, the replacement is made on each repetition of the element, and the replacement expression is evaluated fresh for each repetition.

Examples

Replacing a link:

```
<a href="/sample/link.html"
 tal:attributes="href context.url()">
```

Replacing two attributes:

```
<textarea rows="80" cols="20"
 tal:attributes="rows request.rows();cols request.cols()">
```

tal:condition

Conditionally insert or remove an element

Syntax

`tal:condition` syntax:

```
argument ::= expression
```

Description

The `tal:condition` statement includes the statement element in the template only if the condition is met, and omits it otherwise. If its expression evaluates to a *true* value, then normal processing of the element continues, otherwise the statement element is immediately removed from the template. For these purposes, the value `nothing` is false, and `default` has the same effect as returning a true value.

Note: SharpTAL considers null, zero, empty strings, empty sequences, empty dictionaries false; all other values are true, including `default`.

Examples

Test a variable before inserting it:

```
<p tal:condition="request.message"
  tal:content="request.message">
  message goes here
</p>
```

Testing for odd/even in a repeat-loop:

```
<div tal:repeat="item Enumerable.Range(0, 10)">
  <p tal:condition='repeat["item"].even'>Even</p>
  <p tal:condition='repeat["item"].odd'>Odd</p>
</div>
```

tal:content

Replace the content of an element

Syntax

`tal:content` syntax:

```
argument ::= (['text'] | 'structure') expression
```

Description

Rather than replacing an entire element, you can insert text or structure in place of its children with the `tal:content` statement. The statement argument is exactly like that of `tal:replace`, and is interpreted in the same fashion. If the expression evaluates to `null`, the statement element is left childless. If the expression evaluates to `default`, then the element's contents are unchanged.

The default replacement behavior is `text`, which replaces angle-brackets and ampersands with their HTML entity equivalents. The `structure` keyword passes the replacement text through unchanged, allowing HTML/XML markup to be inserted. This can break your page if the text contains unanticipated markup (eg. text submitted via a web form), which is the reason that it is not the default.

Examples

Inserting the user name:

```
<p tal:content="user.getUserName()">Fred Farkas</p>
```

Inserting HTML/XML:

```
<p tal:content="structure context.getStory()">marked <b>up</b>
content goes here.</p>
```

tal:define

Define variables

Syntax

`tal:define` syntax:

```
argument             ::= attribute_statement [';' attribute_statement]*
attribute_statement  ::= [context] variable_name expression
context              ::= global | local | nonlocal
variable_name        ::= Name
```

Description

The `tal:define` statement defines variables.

When you define a local variable in a statement element, you can use that variable in that element and the elements it contains.

If the expression associated with a variable evaluates to `null`, then that variable has the value `null`, and may be used as such in further expressions. Likewise, if the expression evaluates to `default`, then the variable has the value `default`, and may be used as such in further expressions.

Examples

Defining a global variable:

```
<tal:tag tal:define='global company_name '"My Company"'>
```

Defining a local variable:

```
<tal:tag tal:define='company_name "My Company"'>
```

Defining two local variables, where the second depends on the first:

```
<tal:tag tal:define="mytitle context.title; tlen mytitle.Length">
```

Declare that the listed identifiers refers to previously bound variables in the nearest enclosing scope:

```
<p tal:define="mytitle context.title">
  <tal:tag tal:define="nonlocal mytitle context.new_title">
</p>
```

tal:omit-tag

Remove an element, leaving its contents

Syntax

`tal:omit-tag` syntax:

```
argument ::= [ expression ]
```

Description

The `tal:omit-tag` statement leaves the contents of an element in place while omitting the surrounding start and end tags.

If the expression evaluates to a *false* value, then normal processing of the element continues and the tags are not omitted. If the expression evaluates to a *true* value, or no expression is provided, the statement element is replaced with its contents.

Note: null, zero, empty strings, empty sequences, empty dictionaries are false; all other values are true, including `default`.

Examples

Unconditionally omitting a tag:

```
<div tal:omit-tag="" comment="This tag will be removed">
  <i>...but this text will remain.</i>
</div>
```

Conditionally omitting a tag:

```
<b tal:omit-tag="bold == false">I may be bold.</b>
```

The above example will omit the `b` tag if the variable `bold` is false.

Creating ten paragraph tags, with no enclosing tag:

```
<span tal:repeat="n Enumerable.Range(0, 10)" tal:omit-tag="">
  <p tal:content="n">1</p>
</span>
```

tal:repeat

Repeat an element

Syntax

`tal:repeat` syntax:

```
argument      ::= variable_name expression
variable_name ::= Name
```

Description

The `tal:repeat` statement replicates a sub-tree of your document once for each item in a sequence. The expression should evaluate to a sequence. If the sequence is empty, then the statement element is deleted, otherwise it is repeated for each value in the sequence. If the expression is `default`, then the element is left unchanged, and no new variables are defined.

The `variable_name` is used to define a local variable and a repeat variable. For each repetition, the local variable is set to the current sequence element, and the repeat variable is set to an iteration object.

Repeat Variables

You use repeat variables to access information about the current repetition (such as the repeat index). The repeat variable has the same name as the local variable, but is only accessible through the built-in variable named `repeat`.

The following information is available from the repeat variable:

- `index` - repetition number, starting from zero.
- `number` - repetition number, starting from one.
- `even` - true for even-indexed repetitions (0, 2, 4, ...).
- `odd` - true for odd-indexed repetitions (1, 3, 5, ...).
- `start` - true for the starting repetition (index 0).
- `end` - true for the ending, or final, repetition.
- `length` - length of the sequence, which will be the total number of repetitions.
- `letter` - repetition number as a lower-case letter: “a” - “z”, “aa” - “az”, “ba” - “bz”, ..., “za” - “zz”, “aaa” - “aaz”, and so forth.
- `Letter` - upper-case version of *letter*.
- `roman` - repetition number as a lower-case roman numeral: “i”, “ii”, “iii”, “iv”, “v”, etc.
- `Roman` - upper-case version of *roman*.

You can access the contents of the repeat variable using dictionary, e.g. `repeat["item"].start`.

Examples

Iterating over a sequence of strings:

```
<p tal:repeat='txt new List<string>() { "one", "two", "three" }'>
   <span tal:replace="txt" />
</p>
```

Inserting a sequence of table rows, and using the repeat variable to number the rows:

```
<table>
  <tr tal:repeat="item here.cart">
    <td tal:content='repeat["item"].number'>1</td>
    <td tal:content="item.description">Widget</td>
    <td tal:content="item.price">$1.50</td>
  </tr>
</table>
```

Nested repeats:

```
<table border="1">
  <tr tal:repeat="row Enumerable.Range(0, 10)">
    <td tal:repeat="column Enumerable.Range(0, 10)">
      <span tal:define='x repeat["row"].number;
                        y repeat["column"].number;
                        z x * y'
            tal:replace="string:${x} * ${y} = ${z}">1 * 1 = 1</span>
    </td>
  </tr>
</table>
```

Insert objects. Separates groups of objects by type by drawing a rule between them:

```
<div tal:repeat="object objects">
  <h2 tal:condition='repeat["object"].first.meta_type'
    tal:content="object.type">Meta Type</h2>
  <p tal:content="object.id">Object ID</p>
</div>
```

Note: the objects in the above example should already be sorted by type.

tal:replace

Replace an element

Syntax

`tal:replace` syntax:

```
argument ::= ['structure'] expression
```

Description

The `tal:replace` statement replaces an element with dynamic content. It replaces the statement element with either text or a structure (unescaped markup). The body of the statement is an expression with an optional type prefix. The value of the expression is converted into an escaped string unless you provide the 'structure' prefix. Escaping consists of converting `&` to `&amp;`, `<` to `&lt;`, and `>` to `&gt;`.

If the expression evaluates to `null`, the element is simply removed. If the value is `default`, then the element is left unchanged.

Examples

Inserting a title:

```
<span tal:replace="context.title">Title</span>
```

Inserting HTML/XML:

```
<div tal:replace="structure table" />
```

4.1.2 Expressions (TALES)

The *Template Attribute Language Expression Syntax* (TALES) standard describes expressions that supply *Template Attribute Language (TAL)* and *Macros (METAL)* with data. TALES is *one* possible expression syntax for these languages, but they are not bound to this definition. Similarly, TALES could be used in a context having nothing to do with TAL or METAL.

TALES expressions are described below with any delimiter or quote markup from higher language layers removed. Here is the basic definition of TALES syntax:

```
Expression  ::= [type_prefix ':'] String
type_prefix ::= Name
```

Here are some simple examples:

```
1 + 2
null
string:Hello, ${view.user_name}
```

The optional *type prefix* determines the semantics and syntax of the *expression string* that follows it. A given implementation of TALES can define any number of expression types, with whatever syntax you like. It also determines which expression type is indicated by omitting the prefix.

Types

These are the TALES expression types supported by default in `SharpTAL`:

- `csharp` - execute a C# expression
- `string` - format a string

Note: if you do not specify a prefix within an expression context, `SharpTAL` assumes that the expression is a *csharp* expression.

Built-in Names

These are the names always available to TALES expressions in `SharpTAL`:

- `default` - special value used to specify that existing text or attributes should not be replaced. See the documentation for individual TAL statements for details on how they interpret *default*.
- `repeat` - the *repeat* variables; see *tal:repeat* for more information.
- `template` - reference to the template which was first called; this symbol is carried over when using macros.
- `macros` - reference to the macros dictionary that corresponds to the current template.

`csharp` expressions

Syntax

Python expression syntax:

```
Any valid C# language expression
```

Description

C# expressions are executed natively within the translated template source code.

`string` expressions

Syntax

String expression syntax:

```
string_expression ::= ( plain_string | [ varsub ] )*
varsub            ::= ( '${ Expression }' )
```

Description

String expressions interpret the expression string as text. If no expression string is supplied the resulting string is *empty*. The string can contain variable substitutions of the form `${expression}`, where `expression` is a TALES-expression. The escaped string value of the expression is inserted into the string.

Note: To prevent a `${...}` from being interpreted this way, it must be escaped as `\${...}`.

Examples

Basic string formatting:

```
<span tal:replace="string:${what} and ${that}">
  Spam and Eggs
</span>

<p tal:content='string:${request.form["total"]}'>
  total: 12
</p>
```

Including a dollar sign:

```
<p tal:content="string:$${cost}">
  cost: $42.00
</p>
```

Including operator ${...}:

```
<p tal:content="string:The expression operator: \${cost}">
  cost: $42.00
</p>
```

4.1.3 Macros (METAL)

The *Macro Expansion Template Attribute Language* (METAL) standard is a facility for HTML/XML macro preprocessing. It can be used in conjunction with or independently of TAL and TALES.

Macros provide a way to define a chunk of presentation in one template, and share it in others, so that changes to the macro are immediately reflected in all of the places that share it. Additionally, macros are always fully expanded, even in a template's source text, so that the template appears very similar to its final rendering.

A single Page Template can accomodate multiple macros.

Namespace

The METAL namespace URI and recommended alias are currently defined as:

```
xmlns:metal="http://xml.zope.org/namespaces/metal"
```

Just like the TAL namespace URI, this URI is not attached to a web page; it's just a unique identifier. This identifier must be used in all templates which use METAL.

Statements

METAL defines a number of statements:

- `metal:define-macro` Define a macro.
- `metal:use-macro` Use a macro.
- `metal:define-slot` Define a macro customization point.
- `metal:fill-slot` Customize a macro.
- `metal:define-param` Define macro parameters.
- `metal:fill-param` Fill macro parameters.
- `metal:import` Import macro definitions from external file.

Although METAL does not define the syntax of expression non-terminals, leaving that up to the implementation, a canonical expression syntax for use in METAL arguments is described in TALES Specification.

metal:define-macro

Define a macro

Syntax

`metal:define-macro` syntax:

```
argument ::= Name
```

Description

The `metal:define-macro` statement defines a macro. The macro is named by the statement expression, and is defined as the element and its sub-tree.

Examples

Simple macro definition:

```
<p metal:define-macro="copyright">
  Copyright 2004, <em>Foobar</em> Inc.
</p>
```

metal:define-slot

Define a macro customization point

Syntax

`metal:define-slot` syntax:

```
argument ::= Name
```

Description

The `metal:define-slot` statement defines a macro customization point or *slot*. When a macro is used, its slots can be replaced, in order to customize the macro. Slot definitions provide default content for the slot. You will get the default slot contents if you decide not to customize the macro when using it.

The `metal:define-slot` statement must be used inside a `metal:define-macro` statement.

Slot names must be unique within a macro.

Examples

Simple macro with slot:

```
<p metal:define-macro="hello">
  Hello <b metal:define-slot="name">World</b>
</p>
```

This example defines a macro with one slot named `name`. When you use this macro you can customize the `b` element by filling the `name` slot.

metal:fill-slot

Customize a macro

Syntax

`metal:fill-slot` syntax:

```
argument ::= Name
```

Description

The `metal:fill-slot` statement customizes a macro by replacing a *slot* in the macro with the statement element (and its content).

The `metal:fill-slot` statement must be used inside a `metal:use-macro` statement.

Slot names must be unique within a macro.

If the named slot does not exist within the macro, the slot contents will be silently dropped.

Examples

Given this macro:

```
<p metal:define-macro="hello">
  Hello <b metal:define-slot="name">World</b>
</p>
```

You can fill the `name` slot like so:

```
<p metal:use-macro='master.macros["hello"]'>
  Hello <b metal:fill-slot="name">Kevin Bacon</b>
</p>
```

metal:use-macro

Use a macro

Syntax

`metal:use-macro` syntax:

```
argument ::= expression
```

Description

The `metal:use-macro` statement replaces the statement element with a macro. The statement expression describes a macro definition.

The effect of expanding a macro is to graft a subtree from another document (or from elsewhere in the current document) in place of the statement element, replacing the existing sub-tree. Parts of the original subtree may remain, grafted onto the new subtree, if the macro has *slots*. See `metal:define-slot` for more information. If the macro body uses any macros, they are expanded first.

Examples

Basic macro usage:

```
<p metal:use-macro='other.macros["header"]'>
  header macro from defined in other.html template
</p>
```

This example refers to the `header` macro defined in the `other` template which has been passed as a keyword argument to `ITemplateCache`'s `RenderTemplate` method. When the macro is expanded, the `p` element and its contents will be replaced by the macro.

metal:define-param

Define a macro parameter

Syntax

`metal:define-param` syntax:

```
argument              ::= attribute_statement [';' attribute_statement]*
attribute_statement   ::= param_type param_name [expression]
param_type            ::= Parameter Type
param_name            ::= Parameter Name
```

Description

The `metal:define-param` statement defines macro parameters. When you define a parameter it can be used as a normal local variable in a macro element, and the elements it contains.

The `metal:define-param` statement must be used inside a `metal:define-macro` statement.

Parameter names must be unique within a macro.

Examples

Simple macro with two parameters:

```
<p metal:define-macro="hello"
   metal:define-param='string name; int age'>
  Hello, my name is <b>${name}</b>.
  I'm <b>${age}</b> years old.
</p>
```

You can declare parameters with default values:

```
<p metal:define-macro="hello"
   metal:define-param='string name "Roman"; int age 33'>
  Hello, my name is <b>${name}</b>.
  I'm <b>${age}</b> years old.
</p>
```

metal:fill-param

Fill a macro parameter

Syntax

`metal:fill-param` syntax:

```
argument              ::= attribute_statement [';' attribute_statement]*
attribute_statement   ::= param_name expression
param_name            ::= Parameter Name
```

Description

The `metal:fill-param` statement fills macro parameters.

The `metal:fill-param` statement must be used inside a `metal:use-macro` statement.

If the named parameter does not exist within the macro, the parameter contents will be silently dropped.

Examples

Given this macro:

```
<p metal:define-macro="hello"
   metal:define-param='string name; int age'>
  Hello, my name is <b>${name}</b>.
  I'm <b>${age}</b> years old.
</p>
```

You can fill the `name` and `age` parameters like so:

```
<p metal:use-macro='master.macros["hello"]'
   metal:fill-param='name "Roman"; age 33'>
</p>
```

metal:import

Import macro definitions from external file

Syntax

`metal:import` syntax:

```
argument              ::= import_statement [';' import_statement]*
import_statement      ::= (namespace_name ':') path
namespace_name        ::= Namespace name
path                  ::= Path to file
```

Description

The `metal:import` statement imports macro defintions from external files. Macros can be imported to specific namespace, defined by `namespace` argument part. If the namespace is not specified, macros are imported to default namespace.

Examples

Import macros from file `Macros.html` into default namespace and use imported macro `hello`:

```
<p metal:import="Macros.html">
  <p metal:use-macro='macros["hello"]'
     metal:fill-param='name "Roman"; age 33'>
  </p>
</p>
```

Import macros from file `Macros.html` into custom namespace `mymacros` and use imported macro `hello`:

```
<p metal:import="mymacros:Macros.html">
  <p metal:use-macro='mymacros.macros["hello"]'
     metal:fill-param='name "Roman"; age 33'>
  </p>
</p>
```

Import macros from multiple files into one custom namespace:

```
<p metal:import="mymacros:Macros1.html;mymacros:Macros2.html">
</p>
```

Import macros from multiple files into multiple custom namespaces:

```
<p metal:import="mymacros1:Macros1.html;mymacros2:Macros2.html">
</p>
```

4.1.4 ${...} operator

The `${...}` notation is short-hand for text insertion. The C# expression inside the braces is evaluated and the result included in the output (all inserted text is escaped by default):

```
<div id="section-${index + 1}">
  ${content}
</div>
```

To escape this behavior, prefix the notation with a backslash character: `\${...}`.

4.1.5 Code blocks

The `<?csharp ... ?>` notation allows you to embed C# code in templates:

```
<div>
  <?csharp
    var numbers = Enumerable.Range(1, 10).Select(n => n.ToString()).ToList()
  ?>
  Please input a number from the range ${string.Join(", ", numbers)}.
</div>
```

4.2 Changes

4.2.1 3.0 (2014-03-04)

- Macro parameters can be declared without default values
- Removed runtime dependency on ICSharpCode.NRefactory

4.2.2 2.1 (2013-05-30)

- Improved type definition resolution of variables defined in globals dictionary

4.2.3 2.0 (2013-01-18)

Features:

- Add support for plain text templates
- Create NuGet package

Dependency Changes:

- SharpTAL now relies on ICSharpCode.NRefactory 5.3.0
- .NET 4.0 is now required

4.2.4 2.0b1 (2013-01-04)

Features:

- Added support for code blocks using the *<?csharp ... ?>* processing instruction syntax.
- Enable expression interpolation in CDATA [Roman Lacko]
- The "Template" class now provides virtual method "FormatResult(object)" to enable customization of expression results formatting. [Roman Lacko]

Backwards Incompatibilities:

- Removed "RenderTemplate()" methods from "ITemplateCache" interface (and it's implementations). [Roman Lacko]

4.2.5 2.0a2 (2012-01-05)

Features:

- New "meta:interpolation" command to control expression interpolation setting. [Roman Lacko] To disable expression interpolation: meta:interpolation="false" To enable expression interpolation: meta:interpolation="true"

Internal:

- More code refactoring. [Roman Lacko]

Bugs fixed:

- Tags in the custom tal/metal namespace were not ommited, if the custom namespace was declared on that tag. [Roman Lacko]

Backwards Incompatibilities:

- Rename "tal:define:set" variable context definition to "tal:define:nonlocal" to declare that the listed identifiers refers to previously bound variables in the nearest enclosing scope. [Roman Lacko]
- Removed "<tal:omit-scope>". It was non standart and introduces bad design in template. [Roman Lacko]

4.2.6 2.0a1 (2011-12-20)

Features:

- New HTML/XML template parser. This adds support for HTML5 templates. [Roman Lacko]
- String expression interpolation using ${...} operator in element attributes and in the text of an element. [Roman Lacko]
- New "Template" class that replaces the direct usage of "MemoryTemplateCache" and "FileSystemTemplateCache". [Roman Lacko]
- Allow setting CultureInfo for string formatting, default to InvariantCulture [Petteri Aimonen]
- Added method CompileTemplate() to ITemplateCache to precompile template before knowing the global variable values [Petteri Aimonen]

Internal:

- Code refactoring. [Roman Lacko]
- Add relevant lines of the generated source code to CompileSourceException message [Petteri Aimonen]
- Made template hash calculation more robus [Petteri Aimonen]

Backwards Incompatibilities:

- Removed "Inline Templates" from ITemplateChache.RenderTemplate method. Use "metal:import" command to import macros from external templates [Roman Lacko]

Dependency Changes:

- SharpTAL now relies on ICSharpCode.NRefactory.dll [Roman Lacko]

Bugs fixed:

- In SourceGenerator, fix the handling of newlines in attributes [Petteri Aimonen]

4.2.7 1.2 (2011-01-26)

- Fixed tal:repeat command when using with empty arrays [Roman Lacko]

4.2.8 1.1 (2010-10-25)

- Unit Tests ported to NUnit [Roman Lacko]
- Mono 2.6 with MonoDevelop 2.4 now supported under Linux (tested under Ubuntu 10.10) [Roman Lacko]
- .NET Framework 3.5 and 4.0 with Sharpdevelop 4.0beta3 supported under Windows [Roman Lacko]

4.2.9 1.0 (2010-06-28)

- Initial version [Roman Lacko]