# How To Implement A Generic Template Engine For SharePoint 2010 Using DotLiquid


Jürgen Bäurle

Mai 2011

Parago Media GmbH & Co. KG

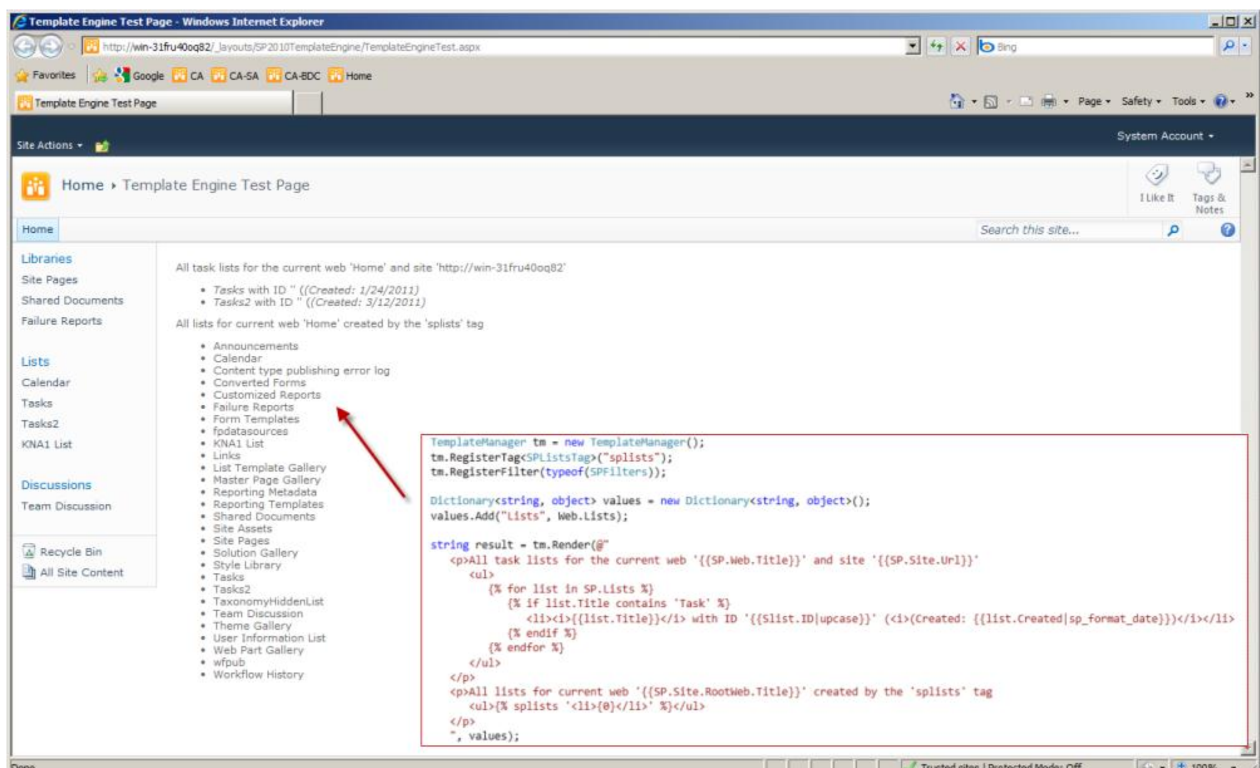

## Introduction

During the process of creating a complex SharePoint application you often need to send mails and create text files based on SharePoint data elements like SPListItem or SPWeb. Mail templates for instance mostly contain specific list item data. It would be helpful sometimes if the text generation itself is template-driven.

This article shows how to implement a generic template manager based on the free DotLiquid templating system with SharePoint specific extensions. This allows you for example to iterate through all SharePoint lists available within a SiteCollection and render only details for lists which contain Task in their title:

```
<p>All task lists for the current web '{{SP.Web.Title}}' and site '{{SP.Site.Url}}'
  <ul>
    {% for list in SP.Lists %}
      {% if list.Title contains 'Task' %}
        <li><i>{{list.Title}}</i> with ID '{{Slist.ID|upcase}}' (<i>(Created:
                                                {{list.Created|sp_format_date}})</i></li>
      {% endif %}
    {% endfor %}
  </ul>
</p>
<p>All lists for current web '{{SP.Site.RootWeb.Title}}' created by the 'splists' tag
  <ul>{% splists '<li>{0}</li>' %}</ul>
</p>
```

The screenshot below shows the result of the rendered template sample:



Of course the technique implemented in this article can also be used in conjunction with other technologies or applications, it's not only SharePoint related.

**DotLiquid Template Engine**

The DotLiquid template engine is a C# port of the Ruby's Liquid templating system and is available for .NET 3.5 and above. DotLiquid is open source and can be downloaded at dotliquidmarkup.org. The software is also available as NuGet package for Visual Studio.

The templating system includes features like variable, text replacement, conditional evaluation and loop statements that are similar to common programming languages. The language elements consists of tags and filter constructs.
The engine can also be easily extended by implementing and adding custom filters and/or tags. This article actually shows how to extend the DotLiquid and implement SharePoint specific parts.

The following sample shows a Liquid template file:

```
<p>{{ user.name | upcase }} has to do:</p>

<ul>
{% for item in user.tasks -%}
  <li>{{ item.name }}</li>
{% endfor -%}
</ul>
```

Output markup is surrounded by curly brackets {{…}} and tag markup by {%...%}. Output markup can take filter definitions like upcase. Filters are simple static methods, where the first parameter is always the output of the left side of the filter and the return value of the filter will be the new left value when the next filter is run. When there are no more filters, the template will receive the resulting string.

There are a big number of standard filters available to use, but later on we will implement a custom filter method for SharePoint. The result of the above rendered template looks like:

```
<p>TIM JONES has to do:</p>

<ul>
  <li>Documentation</li>
  <li>Code comments</li>
</ul>
```

To pass variables and render the template you first need to parse the template and the then just call the Render method with the variable values:

```
string templateCode = @"<ul>
{% for item in user.tasks -%}
  <li>{{ item.name }}</li>
{% endfor -%}
</ul>";

Template template = Template.Parse(templateCode);

string result = template.Render(Hash.FromAnonymousObject(new {
                                    user = new User
                                    {
                                      Name = "Tim Jones",
                                      Tasks = new List<Task> {
                                        new Task { Name = "Documentation" },
                                        new Task { Name = "Code comments" }
                                      }
                                    }}));

public class User : Drop
{
       public string Name { get; set; }
       public List<Task> Tasks { get; set; }
}

public class Task : Drop
{
       public string Name { get; set;        }
}
```

The User and Task classes inherit from the Drop class. This is an important class in DotLiquid. The next sections explains the class in more detail. It is out of scope of this article to discuss all the features for DotLiquid in detail. For more information please see the homepage of DotLiquid (dotliquidmarkup.org) or the website of the original creator of the Liquid template language at liquidmarkup.org. You will find there a lot of manuals and sample code.

**Template Manager**

The TemplateManager class is a wrapper over the DotLiquid template engine and provides SharePoint support. The class allows to cache parsed templates, to register tags and filters and render them using a top-level custom Drop class named SharePointDrop:

```
internal class TemplateManager
{
  public Dictionary<string, Template> Templates { get; protected set; }

  public TemplateManager()
  {
    Templates = new Dictionary<string, Template>();
  }

  public void AddTemplate(string name, string template)
  {
    if(string.IsNullOrEmpty(name))
      throw new ArgumentNullException("name");
    if(string.IsNullOrEmpty(template))
      throw new ArgumentNullException("template");

    if(Templates.ContainsKey(name))
      Templates[name] = Template.Parse(template);
    else
      Templates.Add(name, Template.Parse(template));
  }

  public void RegisterTag<T>(string tagName) where T : Tag, new()
  {
    Template.RegisterTag<T>(tagName);
  }

  public void RegisterFilter(Type type)
  {
    Template.RegisterFilter(type);
  }

  public string Render(string nameOrTemplate, IDictionary<string, object> values)
  {
    Template template;

    if(Templates.ContainsKey(nameOrTemplate))
      template = Templates[nameOrTemplate];
    else
      template = Template.Parse(nameOrTemplate);

    SharePointDrop sp = new SharePointDrop();

    if(values != null)
    {
      foreach(KeyValuePair<string, object> kvp in values)
        sp.AddValue(kvp.Key, kvp.Value);
    }

    return template.Render(new RenderParameters { LocalVariables =
             Hash.FromAnonymousObject(new { SP = sp }), RethrowErrors = true });
  }
}
```

The Render method is using the SharePointDrop class to support objects like SPListItem or SPListCollection. The Drop class as key concept of DotLiquid must be explained in detail. The DotLiquid template engine is focusing on making templates safe. A Drop is a class which allows you to export DOM like objects. DotLiquid, by default, only accepts a limited number of types as parameters to the Render method. These data types include the .NET primitive types (integer, float, string, etc.), and some collection types including IDictionary, IList and IIndexable (a custom DotLiquid interface).

If DotLiquid would support arbitrary types, then it could result in properties or methods being unintentionally exposed to template authors. To prevent this, DotLiquid templating system uses Drop classes that use an opt-in approach to exposing object data.

The code following shows the SharePointDrop implementation:

```
internal class SharePointDrop : Drop
{
  Dictionary<string, object> _values;

  public SharePointDrop()
  {
    _values = new Dictionary<string, object>();

    if(SPContext.Current != null)
      _values.Add("Site", SPContext.Current.Site);
    if(SPContext.Current != null)
      _values.Add("Web", SPContext.Current.Web);
    if(SPContext.Current != null)
      _values.Add("User", SPContext.Current.Web.CurrentUser);

    _values.Add("Date", DateTime.Now);
    _values.Add("DateISO8601",
                  SPUtility.CreateISO8601DateTimeFromSystemDateTime(DateTime.Now));

    // TODO: Add more default values
  }

  public void AddValue(string name, object value)
  {
    if(string.IsNullOrEmpty(name))
      throw new ArgumentNullException("name");

    if(_values.ContainsKey(name))
      _values[name] = value;
    else
      _values.Add(name, value);
  }

  public override object BeforeMethod(string method)
  {
    if(!string.IsNullOrEmpty(method) && _values.ContainsKey(method))
      return DropHelper.MayConvertToDrop(_values[method]);

    return null;
  }
}
```

The SharePointDrop class main objective is to solve the problem of casting unsupported data types like SPListItem or SPListItemCollection and other SharePoint related types. Therefore the class is overriding the BeforeMethod method of the Drop class to analyse the requested variable value. If the variable is available in the value context the method will try to cast the data type to a known Drop type by calling the MayConvertToDrop method of the DropHelper class:

```
public static object MayConvertToDrop(object value)
{
  if(value != null)
  {
    // TODO: Add your own drop implementations here

    if(value is SPList)
      return new SPPropertyDrop(value);
    if(value is SPListCollection)
      return ConvertDropableList<SPPropertyDrop, SPList>(value as ICollection);
    if(value is SPListItem)
      return new SPListItemDrop(value as SPListItem);
    if(value is SPListItemCollection)
      return ConvertDropableList<SPListItemDrop, SPListItem>(value as ICollection);
    if(value is SPWeb)
      return new SPPropertyDrop(value);
    if(value is SPSite)
      return new SPPropertyDrop(value);
    if(value is SPUser)
      return new SPPropertyDrop(value);
    if(value is Uri)
```

```
      return ((Uri)value).ToString();
    if(value is Guid)
      return ((Guid)value).ToString("B");
  }

  return value;
}
```

The SPListItemDrop class for instance is returning the value of the requested field:

```
internal class SPListItemDrop : SPDropBase
{
  public SPListItem ListItem { get { return DropableObject as SPListItem; } }

  public SPListItemDrop()
  {
  }

  public SPListItemDrop(SPListItem listItem)
  {
    DropableObject = listItem;
  }

  public override object BeforeMethod(string method)
  {
    if(!string.IsNullOrEmpty(method))
    {
      StringBuilder sb = new StringBuilder();
      string name = method + "\n";

      for(int i = 0; i < name.Length; i++)
      {
        if(name[i] == '\n')
          continue;
        if(name[i] == '_')
        {
          if(name[i + 1] != '_')
            sb.Append(' ');
          else
          {
            i++;
            sb.Append('_');
          }
        }
        else
          sb.Append(name[i]);
      }

      name = sb.ToString();

      if(ListItem.Fields.ContainsField(name))
        return DropHelper.MayConvertToDrop(ListItem[name]);
    }

    return null;
  }
}
```

The method parameter (field name of the SPListItem) of the BeforeMethod method can contain underscores which are replaced by spaces. So, field names with spaces like Start Date of the Task item must be defined in the template as {{task.Start_Date}}.

The SPPropertyDrop class, also part of the solution of this article, is a generic Drop implementation which exposes all properties of an object and may cast them if needed into an Drop objects again. For implementation details see the source code.

**Filters and Tags**

The solution is also providing a custom filter and tag implementation. The filter called sp_format_date (see template above) is implemented by the method SPFormatDate and is calling the FormatDate method of the class SPUtility form the SharePoint API:

```
internal static class SPFilters
{
```

```
  public static object SPFormatDate(object input)
  {
    DateTime dt = DateTime.MinValue;

    if(input is string)
    {
      try
      {
        dt = SPUtility.ParseDate(SPContext.Current.Web, input as string,
               SPDateFormat.DateOnly, false);
      }
      catch { }
    }
    else if(input is DateTime)
      dt = (DateTime)input;

    if(dt != DateTime.MinValue && dt != DateTime.MaxValue && SPContext.Current != null)
      return SPUtility.FormatDate(SPContext.Current.Web, (DateTime)input,
               SPDateFormat.DateOnly);

    return input;
  }
}
```

The custom tag named splists is returning a formatted list of all SPList object names of the current web (see template above):

```
internal class SPListsTag : Tag
{
  string _format;

  public override void Initialize(string tagName, string markup, List<string> tokens)
  {
    base.Initialize(tagName, markup, tokens);

    if(string.IsNullOrEmpty(markup))
      _format = "{0}";
    else
      _format = markup.Trim().Trim("\"".ToCharArray()).Trim("'".ToCharArray());
  }

  public override void Render(Context context, StreamWriter result)
  {
    base.Render(context, result);

    if(SPContext.Current != null && !string.IsNullOrEmpty(_format))
    {
      foreach(SPList list in SPContext.Current.Web.Lists)
        result.Write(string.Format(_format, list.Title));
    }
  }
}
```

**Summary**

Using a template engine in conjunction with SharePoint helps to simplify the creation process of dynamic text files or mails. It can be used in different areas of development, not only in SharePoint applications.

**Contact Information**

If you have any feedback or suggestions, please feel free to contact me:

Jürgen Bäurle
jbaurle@parago.de
http://www.parago.de/jbaurle

Parago Media GmbH & Co. KG
Im Wengert 3 | 71336 Waiblingen, Germany | Phone +49.7146.861803 | Internet http://www.parago.de