How To Implement A Generic Entity List Repository And Business Logic For SharePoint 2010 Using The T4 Templating Engine

Jürgen Bäurle

August 2010

Parago Media GmbH & Co. KG

Introduction

The SharePoint API offers developers great possibilities to access all kind of information, especially data stored in SharePoint lists. But in real-world scenarios and complex SharePoint projects you often need a more efficient way to access list data. Therefore I've developed a solution that is implementing a generic entity list repository which will be partially created using the T4 templating engine, also known as the Text Template Transformation Toolkit from Microsoft. The template engine is part of Visual Studio 2008 and 2010. For more information about T4 toolkit check out the following link: http://msdn.microsoft.com/en-us/library/bb126445.aspx.

The entity list repository provides a type-safe and extensible business logic layer (BAL) and data access layer (DAL) for SharePoint lists based on content types. Each list repository is associated with a specific SharePoint content type, but multiple list repositories can point to the same list. For the associated content type the T4 template files will automatically generate an entity class that will be used for the list repository.

The SharePoint solution we will create with the Visual Studio 2010 Tools for SharePoint Development and C# contains three projects. The repository project containing the entities and repository code, a T4 templates helper project with code to support and simplify the template generation process as well as the SharePoint project itself, that will use the repository.

The SharePoint solution creates a new content type named *Mail Template* and a list called *Mail Templates* associated with the content type. The mail template itself contains the type name, subject, body and BCC information. The following code fragment shows an example how to use the repository:

```
MailTemplateRepository mr = new MailTemplateRepository();
MailTemplate mt = mr.GetByType("INVITATION");
```

The *GetByType* method is a repository method that internally queries the SharePoint list by executing a CAML query. The query result is automatically converted into the *MailTemplate* entity and returned to the caller.

For each repository the developer can define an execution scope. The scope allows to define if an entity update or deletion shall be executed with the *SystemUpdate* or *Update* method of the *SPList* class or if the task shall be executed with privileged rights. There are also other setting options available. The following sample defines a entity repository scope which uses the *SystemUpdate* method, does not trigger any list item events and executes the update with privileged rights (using *RunWithElevatedPrivileges*):

```
MailTemplateRepository mr = new MailTemplateRepository();
mr.Scope = new EntityRepositoryScope(true);

MailTemplate mt = mr.GetByType("INVITATION");
mt.Subject = "...";
mt.Body = "...";
mr.Update(mt);
```

Scoping is very useful in many scenarios where developers have to update or delete entities without programming special logic themselves. It just helps unifying this process.

The Repository Project

There are basically three main classes in the repository project: *Entity*, *EntityRepository* and *EntityRepositoryScope*. The classes *Entity* and *EntityRepository* are used to create specific entity and repository classes, like the *MailTemplate* or *MailTemplateRepository* class. The *EntityRepositoryScope* class is used to control the update and deletion process of the SharePoint list items (see above).

The repository project contains a couple of helper and automatically created classes. Those files are all organized in special folders within the project:

Folder *Attributes*

This folder contains a class called FieldIDAttribute which is internally used by the EntityRepository class to read the field ID of each public entity property. The repository class is using the field ID to access the field object. The ID is a GUID value defining the field within the SharePoint list item. The template *MailTemplate.tt* is adding the FieldID attribute automatically to the generated entity class code.

To use system entities like the content type *Item*, the entity class must be created manually and the FieldID attribute needs to be added to each public property (see the Item.cs file in code). Instead of using the GUID of build-in fields, it's possible to use a special notation: `[FieldID("[Title]")]` for the Title field.

Folder *Constants*

The *Constants* folder contains classes with list (*Lists* class) and enumeration (*Enums* class) constants. In addition the template file *ContentTypes.tt* in the subfolder *Generated* generates a *ContentTypes* class that contains a partial subclass with ID and name properties for each content type defined in the template file. The *fileNames* variable in the template file itself defines the content types for which code will be produced. The *FieldIDs.tt* template generates similar to the *ContentTypes.tt* template file a class with all field IDs.

Folder *Contracts*

All the repository interfaces are defined in this folder. The generic interface *IEntityRepository* is the main interface and defines all the basic methods of a repository. Each custom repository contract must be inherited from this interface, e.g. the *IMailTemplateRepository* interface implemented by the *MailTemplateRepository* class.

Folder *Entities*

The folder *Entities* is containing all automatically generated and all manually created entity classes. The template-based entity classes are all defined as partial classes and stored in the subfolder *Generated*, so they can be extended with attributes, methods (overrides) and base classes. The *MailTemplate* class generated by the T4 template (in the subfolder *Generated*) will look like this:

```
[CompilerGenerated]
public partial class MailTemplate
{ {
  [FieldID("{B0D7F541-0F86-4794-88AE-9CEEA9BC6F02}")] public string Type { get; set; }
  [FieldID("{31C4A2D3-2928-47D3-A534-9C92993538D7}")] public string Body { get; set; }
  [FieldID("{D41A10DE-5A90-4904-8758-1C6EC9EB54EB}")] public string BCC { get; set; }

  protected override string AssociatedList
    { get { return Lists.MailTemplates; } }
  protected override SPContentTypeId AssociatedContentTypeID
    { get { return ContentTypes.MailTemplate.ID; } }
}
```

This class will be extended with the partial code from the *MailTemplate.cs* file in the folder *Entities*:

```
[Serializable]
public partial class MailTemplate : Item
{ {
  public override bool Validate()
  {
    // TODO: Implement specific object validations!

    return base.Validate();
  }

  public void SetPlaceholders(Dictionary<string, string> placeholderValues)
  {
    // TODO: Replace placeholders in subject and body texts if available!
  }
}
```

In order to support object serialization the *Serializable* attribute will be added to the class. The *MailTemplate* class as well as the content type inherits from the *Item* entity class and content type in SharePoint. To reflect the inheritance situation the base class will be the *Item* class. The definition for the *Item* class looks like this:

```
[Serializable]
public partial class Item : Entity
{ {
  [FieldID("[Title]")]
  public string Title { get; set; }
}
```

Since the base class of the *Item* class is the *Entity* class, it is possible to override a couple of methods (overrides). Those overridable methods are *Validate*, *PrepareBeforeUpdate* and others. The *Validate* method is overridden by the *MailTemplate* class to allow object validation, e.g. to check if a given BCC email address is valid or does exist in the system. The *PrepareBeforeUpdate* method can be used to change values before the list item update is executed.

The *SetPlaceholders* method for instance can be used to replace placeholders in the subject and/or body text of a mail template record. The method is not implemented yet. It only shows the many options developers have when using such a repository system.

Folder *Helpers*

The *Helpers* folder contains utility classes used by the repository.

Folder *Repositories*

The *Repositories* folder contains all custom repository classes like the *MailTemplateRepository*. To create a new class the *EntityRepository* class is used as base class. For *MailTemplateRepository* the base class is *EntityRepository<MailTemplate>*. The *EntityRepository* class is implementing the *IEntityRepository* interface and the custom *MailTemplateRepository* class is implementing the *IMailTemplateRepository* interface.

The specific repository class should implement at least one constructor that is calling the base class constructor with the SharePoint list name and the associated content type ID for initialization. The implementation of the *GetByType* method of the *MailTemplateRepository* class shows an example of creating a CAML query and executing it by calling the base class method *Get*:

```
public IList<MailTemplate> GetByType(string type)
{ {
  if(string.IsNullOrEmpty(type))
    throw new ArgumentNullException("type");

  return Get(string.Format("<Eq><FieldRef ID='{0}'/><Value Type='String'>{1}</Value></Eq>",
    FieldIDs.MailTemplate.Type, type));
}
```

The *Get* method is transforming the CAML statement into a final query, executes the query, materializes the list items received from SharePoint and returns a list of entities. The execution is running with the default or given execution scope. Another option is to use methods that are executing CAML queries and returning a list of entities within the same repository class like the *GetByType* method as input for LINQ to Objects in order to sort, change or transform the list.

This article can only give you an overview of the repository logic. The best way to find out more about how to use it the system efficiently and create new repository classes is to look at the source code and do tests by yourself. The SharePoint project includes an application page (Test.aspx) which will use the *MailTemplateRepository* repository class to display the list data:

```
protected void Page_Load(object sender, EventArgs e)
{ {
  _templateRepository = new MailTemplateRepository();

  if(IsPostBack)
    return;

  var types = _templateRepository.GetTypes();

  if(types == null || types.Count == 0)
    WriteMessage("No mail templates found.");
  else
```

```
  {
    MailTemplateTypeList.DataSource = types;
    MailTemplateTypeList.DataBind();

    SetTemplateData(types[0]);
  }
}

...

protected void SetTemplateData(string type)
{ {
  MailTemplate mt = _templateRepository.GetByType(type).FirstOrDefault();

  if(mt == null)
    WriteMessage(string.Format("No mail templates of type '{0}' found.", type));
  else
  {
    SubjectLabel.Text = mt.Title ?? string.Empty;
    BodyLabel.Text = HttpUtility.HtmlEncode(mt.Body ?? string.Empty);
    BccLabel.Text = mt.BCC ?? string.Empty;

    Views.SetActiveView(DefaultView);
  }
}
```

The next section will shortly describe how the T4 templates helper project will work.

The T4 Templating Project

This project is only used to simplify the process of template generation. It is not part of the production output nor is it deployable on a SharePoint system. Creating a custom class library which can be called during template processing is currently the only way for developers to implement reusable template helper methods. Within the template files the class library will be referenced (in Visual Studio 2010 only):

```
<#@ assembly name="$(SolutionDir)MailTemplate.T4\bin\MailTemplate.T4.dll" #>
```

All the templates used in this article or project are first checking which Visual Studio project is containing the project item (the content type definition). The file names of the content type definitions are defined in the template files in a variable called *fileNames*. The helper methods will open the files and analyze the XML/CAML definition to find out about the defined field types and IDs. Those information are returned to the template code to generate the final source code.

The classes generated by the template files are decorated with the *CompilerGenerated* attribute. Feel free to change the template and adapt the template code to your specific needs. It is also possible to debug template code, but this goes beyond the scope of this article. The T4 templating engine or Text Template Transformation Toolkit from Microsoft is a great option to get certain things done automated.

Summary

This article gave an overview about several interesting topics in the area of SharePoint development. The repository system has helped me to develop large SharePoint project more efficient and with clean code. The T4 templating engine helped to automate annoying tasks and reduced errors by forgetting to adjust entity classes. The power of the system will come up if you have to handle a lot of different entities.

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