Tutorial:
Developing own modules and extensions

version: 0.1

Author: Anja Beuth

Table of contents

1 Checking necessity 2
2 Setting up a project in Visual Studio 2
2.1 Tidying up 3
2.2 Exchanging web.config 4
2.3 Adding links 4
2.4 Configuring project 5
3 Basic approach 6
4 Creating module 6
4.1 Method »Dispose()« 7
4.2 Setter/Getter »Extensions« 8
4.3 Methode »Configure()« 8
5 Creating extension 9
5.1 Defining methods 9
6 Integrating module 10
6.1 Porting file 10
6.2 web.config 10

Requirements

For this tutorial, the developer needs access to the installation directory of the onion.net Render Engine. A current distribution should be available on the developer computer for the local development.

Again, knowledge of programming with C# and experience working with a suitable development environment such as Visual Studio is beneficial.

-
-
- C# Tutorials (MSDN)
- Galileo Open Book "C#" von Eric Gunnerson
- Visual Studio

Description

This tutorial describes the development and use of an own module with an extension for the project-specific extension of the XSLT default language scope using the development environment “Microsoft Visual Studio 2010”.

Signs and symbols

Boxes marked with an arrow symbol and a green border contains instruction of what to do next.

This kind of boxes contains tips and tricks.

```
Source code is shown in blue boxes.
```

1 Checking necessity

Before an extension is programmed in C#, it should be checked whether this is really necessary. Although extendability is no problem, experience shows that XSL transformations that can be changed directly via the editor are much better to maintain. Later extensions or bugfixes of C#-extensions always require a developer with the appropriate knowledge.

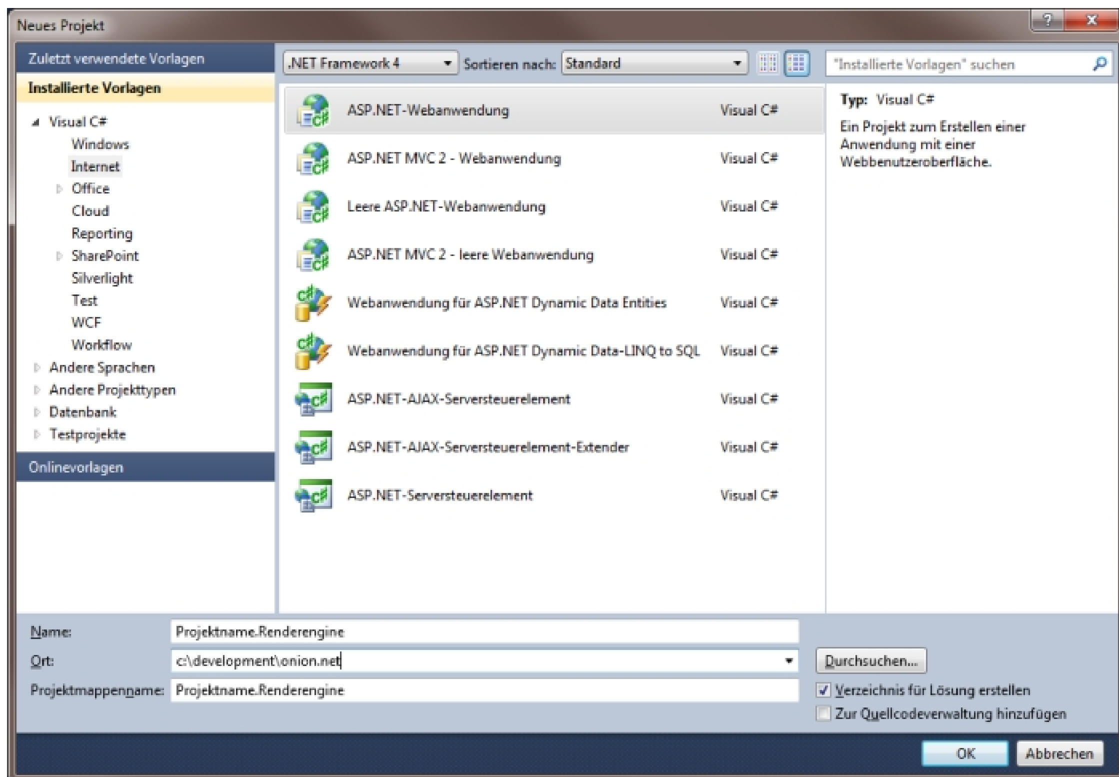
Moreover, a deployment of the appropriate files must take place on the production system for loading the updates. If changes to the “web.config” are also necessary, this therefore results in the restarting of the web application. Updating is therefore more time-consuming than checking-out, editing and returning a method in the onion.net Editor.

Only if the function to be implemented cannot be implemented with XSLT or only with a lot of effort, using an extension is worthwhile. Before self-development however, it should be checked whether an extension is already available which fulfills the requirements or makes implementing with XSLT possible in a simple manner. An overview of the extensions already available can be found on the page extensions (reference).

2 Setting up a project in Visual Studio

For the onion.net development, an appropriate Visual Studio project should be set up first. This should be an “ASP.NET web application”. Since extensions for the onion.net Renderengine are developed, the combination "{Projektname}.Renderengine" makes good sense for the project name.

In this way, other projects can be added to the project folder later on, which extend the editor for example or contain a WebService or WindowsService for the project.



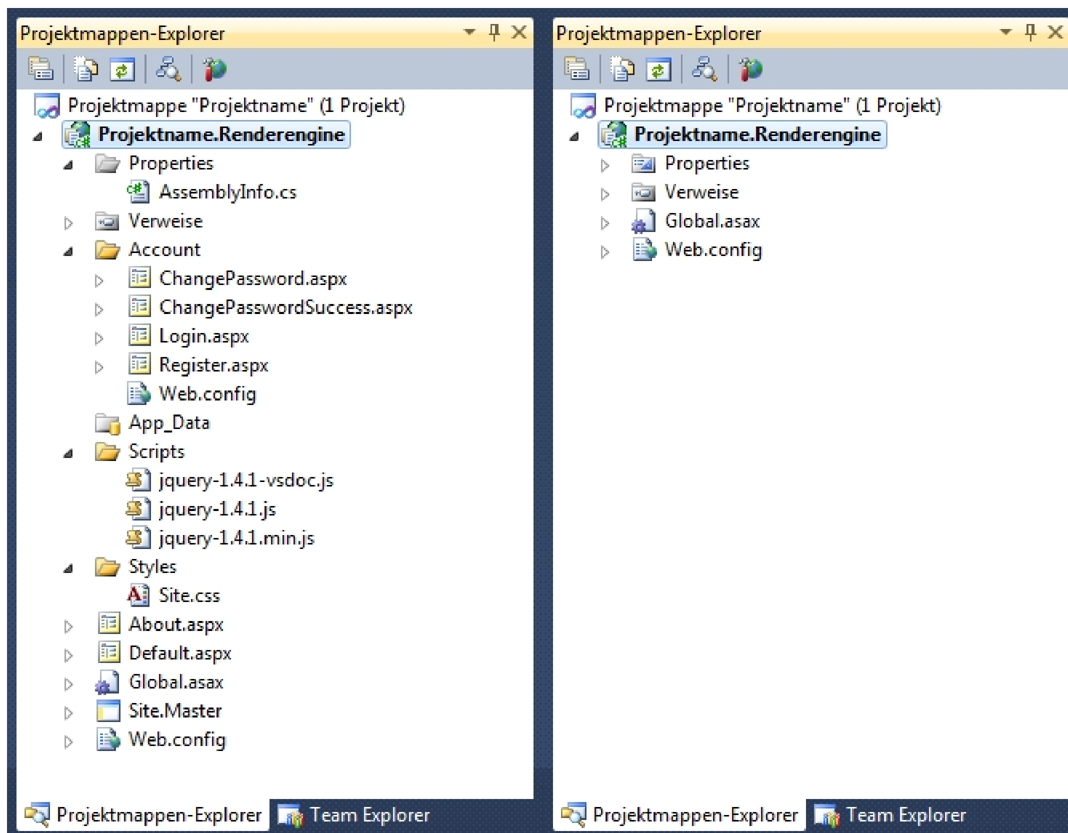
Description: Dialogue "New Projekt" in Visual Studio for creating an "ASP.NET web application" named "{Projektname}.Renderengine"

2.1 Tidying up

After creation, Visual Studio automatically creates a series of files that are needed in a typical web project. Most of these files are not needed in the case of development for onion.net. You can continue to keep the files in the project as templates. We recommend however deleting the unnecessary elements. This makes the project much tidier and also better to maintain.

Only the following elements are needed: Properties, Links, Global.asax, web.config.

The following graphic shows the project folder explorer before and after tidying up.



Description: The project folder before and after tidying up

2.2 Exchanging web.config

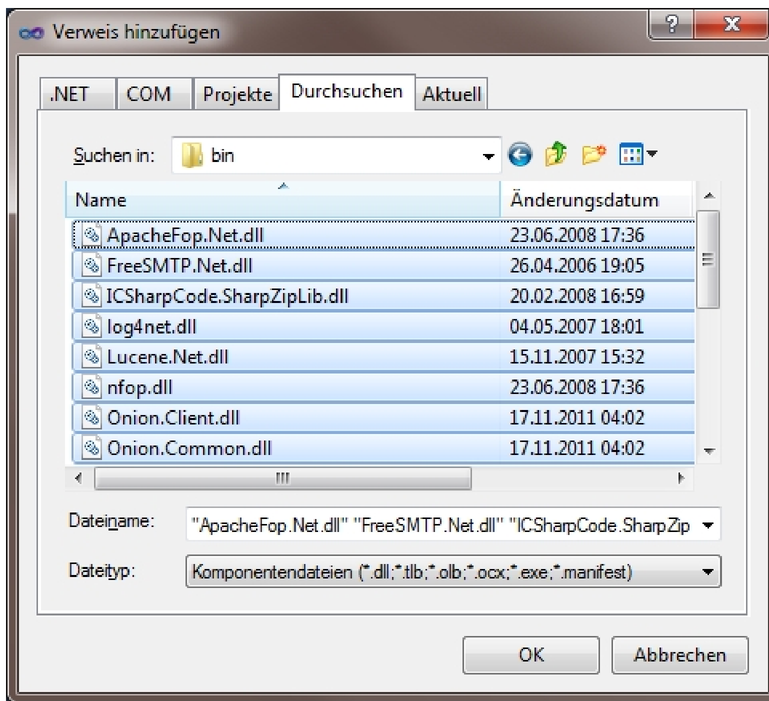
It is a good idea to have a version of the actual web.config of the Renderengine in the local development environment. This may have to be adapted in the case of development for onion.net. It can then be copied into the actual onion.net application together with the extension assembly at the time of updating.

Moreover, it is also possible to connect a Renderengine to the onion.net Information Server of the project in the local development environment. You can then test and debug your development locally with the “real” configuration.

2.3 Adding links

Since we want to extend onion.net, we also need the appropriate classes.

For this purpose, right-click on “Links” on the right-hand side of the project folder explorer and select “Add link”. Then go to the tab “Search” in the opened dialogue and navigate to the folder of a current distribution on your computer. Navigate into the structure “onion.net Render Engine” > “bin” there. You will be displayed all libraries of the onion.net Renderengine. Select all of them and confirm by clicking on “OK”.

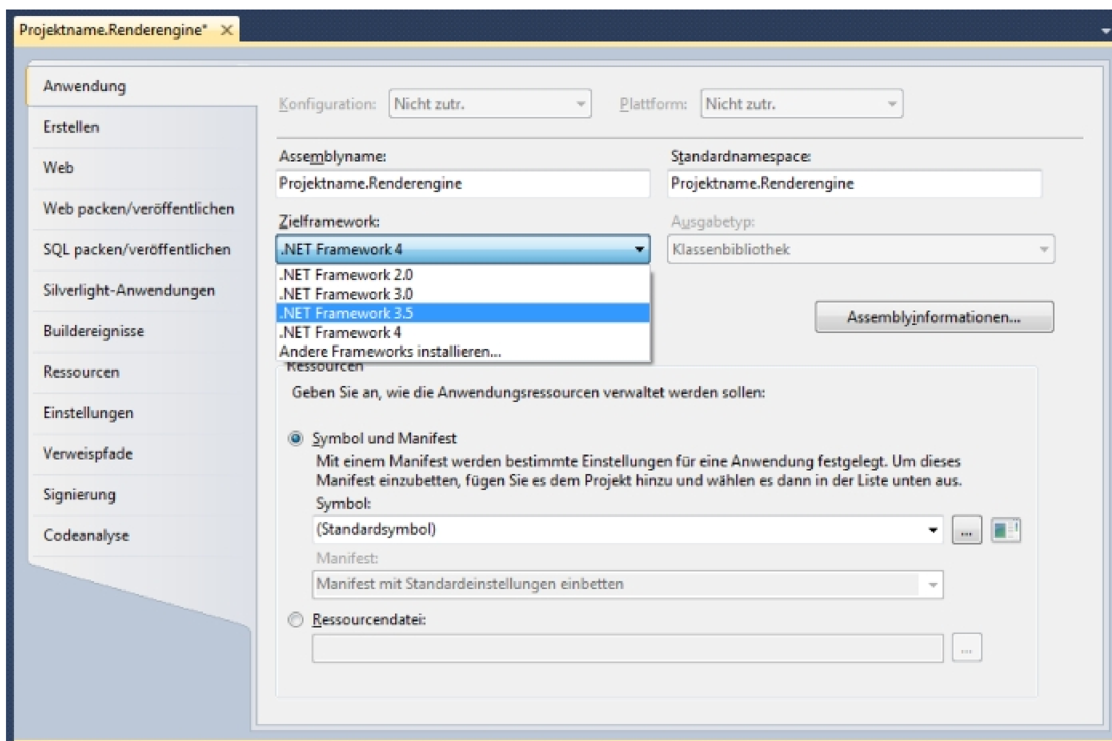


2.4 Configuring project

In order to correctly interact with your onion.net project later on and ensure the updating of the onion.net links, two minor settings are now necessary.

To do this, go to the properties of the project via its context menu.

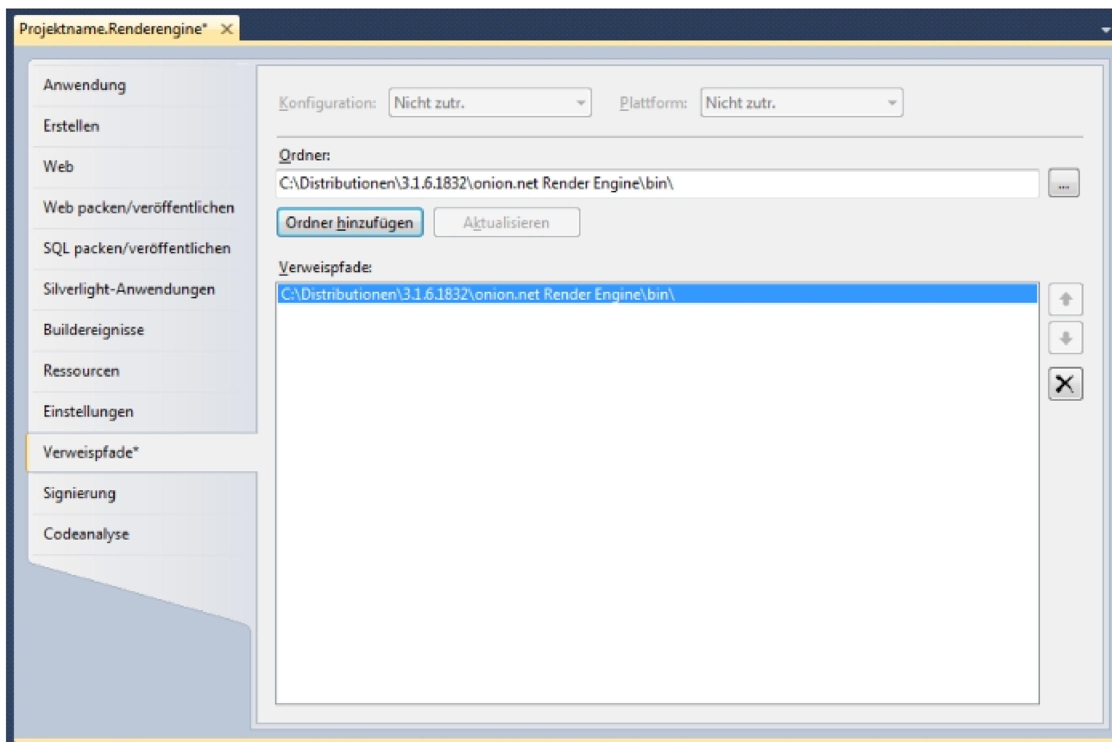
In the tab “application”, change the target framework to the version 3.5.



Confirm the following confirmation prompt by clicking on “yes”.

The properties are automatically closed. In the project folder explorer you will see that the link “Microsoft.CSharp” is flagged with a warning symbol. This assembly is only present from Framework 4.0 onwards and is not needed by us. Delete the link.

Then go to the project properties again and to the tab “link paths” there. Add the “bin” folder of the Renderengine to a current distribution.



After you have saved the properties, you can close the tab.

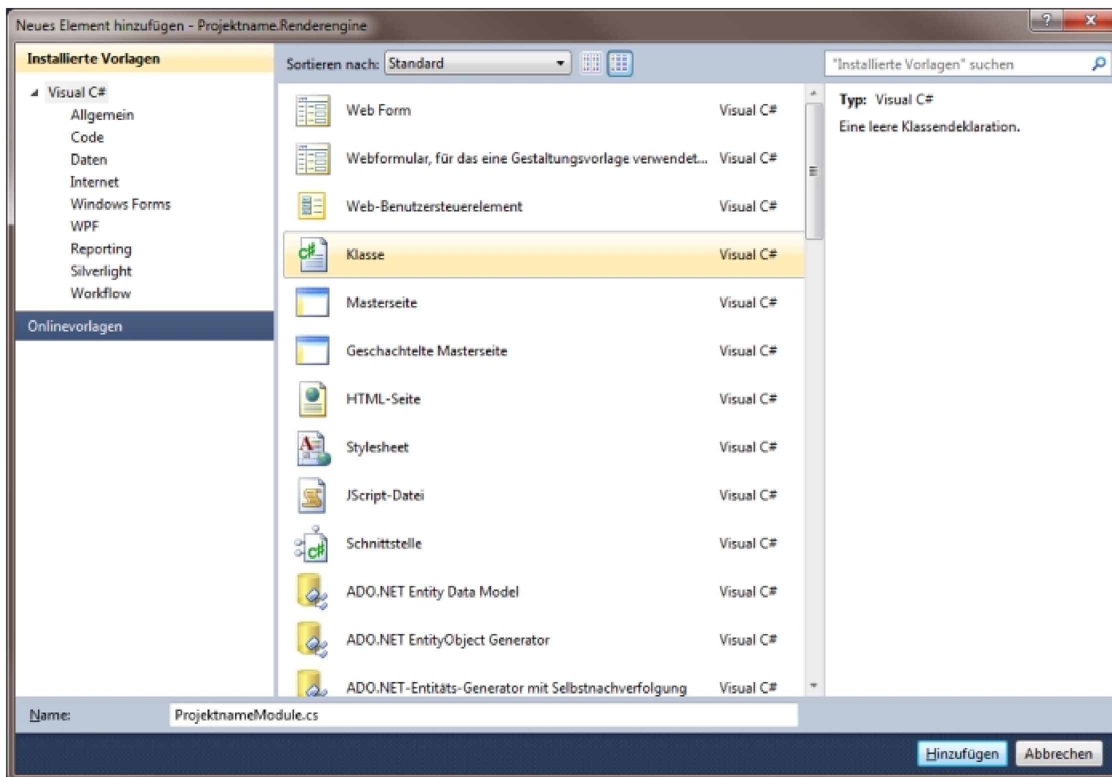
3 Basic approach

In order to extend the range of functions of XSLT, a module must be created first. This sort of serves as a container for different extensions, which in turn then provide the actual function extensions in the form of methods.

As you may have already read in the documentation for the Renderengine configuration, the modules are integrated and configured via the web.config.

4 Creating module

Add a new class to your project and call it “{Project_name}Module”.



Now you implement the interface “IModule”, which you will find under “Onion.RenderEngine.Modules”. When you have implemented all members, your class should look as follows:

```
public class ProjektnameModule : IModule
{
  public void Dispose()
  {
    throw new NotImplementedException();
  }
  public void Configure(IRenderEngine engine, XmlElement configuration)
  {
    throw new NotImplementedException();
  }
  public IExtension[] Extensions
  {
    get { throw new NotImplementedException(); }
  }
}
```

4.1 Method »Dispose()«

For the method »Dispose()«,no special instruction is usually necessary. You can therefore go ahead and remove the content.

4.2 Setter/Getter »Extensions«

Since the functions that can be called from XSLT are embedded in extensions, it must be communicated to the module which extensions are allocated to it. This is done in a global variable “Extensions”, which is of the type “IExtension[]” and which has the visibility “private”.

Add the global variable and extend the getter so that the field can also be queried outside of the class.

```
public class ProjektnameModule : IModule
{
  private IExtension[] extensions;
  public void Dispose() {}
  public void Configure(IRenderEngine engine, XmlElement configuration)
  {
    throw new NotImplementedException();
  }
  public IExtension[] Extensions
  {
    get { return extensions; }
  }
}
```

4.3 Methode »Configure()«

In the method »Configure()«,necessary configurations are made for the module. In most cases and to begin with, this is just initialising the field »extensions«.

```
public void Configure(IRenderEngine engine, XmlElement configuration)
{
  extensions = new IExtension[] { new ProjetktnameExtension() };
}
```

The method is transferred two parameters however, which can be used for further configurations.

IRenderEngine engine

This parameter contains the current RenderEngine, in the context of which the module is embedded or called. It can also be transferred into the extension via the constructor, in order to access data objects there for example.

XmlElement configuration

When integrating the module in the web.config you can make additional configurations via subordinated XML elements. This can be for example access data for a mailserver or a connection string to an additional database in which you would like to save.

5 Creating extension

We allocated an extension to the module in the previous step. We still need to create it however.

In a similar way to the module, create a class named “{Project_name}Extension”. This class must implement the interface “IExtension”. Your class will then look roughly as follows:

```
public class ProjektnameExtension : IExtension
{
  public string Namespace
  {
    get { throw new NotImplementedException(); }
  }
}
```

With the getter you specify the namespace under which the extension is to be integrated in the XML later on.The domain of the later live page is usually taken here with the current year as an addition if necessary.

```
public string Namespace
{
  get { return "http://www.homepage.com"; }
}
```

A namespace is usually linked with a prefix. Having seen the assistant for adding new namespaces, you already know the functionality whereby a certain prefix is automatically linked with the namespace in order to ensure a consistent use. You can also define such a prefix for your own extension. This is done via an “ExtensionDocumentationAttribute”, which you add to the class as follows:

```
[ExtensionDocumentation(ExamplePrefix = "ext")]
public class ProjektnameExtension : IExtension
{
  ...
}
```

5.1 Defining methods

Now you can define any methods for your extension. Similarly to the “ExtensionDocumentationAttribute”, there is an “ExtensionMethodAttribute”. Irrespectively of the C#-compliant code, you can define with this a name under which to call the method in XSLT.

```
[ExtensionMethod(Name = "helloWorld")]
public string HelloWorld()
```

```
{
  return "Hello World!";
}
```

6 Integrating module

6.1 Porting file

When creating the project in Visual Studio, a .dll file ("Project_name.Renderengine.dll") is created in the project folder. This must be copied into the "preview/bin" directory of the onion.net project. The extension is then available.

6.2 web.config

So that your extension is available in the XSL transformations, the module must be added in the web.config.

For this purpose, extend the section <modules> to include the following line:

```
<module type="{Projektname}.Renderengine.{Projektname}Module,
{Projektmame}.Renderengine">
```

The part before the comma indicates the module directly, whilst the part behind the comma indicates the appropriate namespace.

If your extension does not yet pop up in the assistant to update the namespaces, restart the editor web application through changing and saving the web.config, so that the configuration is updated.