

Einstellungssache

Der Configuration Application Block der Enterprise Library 1.0 im Detail

von Jochen Reinelt und Stefan Zill

Mit dem .NET Framework in der Hinterhand ist die eigene Anwendungskonfiguration schnell erledigt: XML-Serialisierung und *PropertyGrid* übernehmen dabei die mühsamsten Aufgaben und ermöglichen ein Lesen, Schreiben und Editieren der Daten mit wenig eigenem Code. Die Themen Caching und Verschlüsselung werden dann schon etwas aufwändiger. Und wie flexibel und wieder verwendbar ist der eigene Schnellschuss hinterher? Der Configuration Application Block steht Ihnen nun mit Rat und Tat zur Seite.

Die Enterprise Library mit ihren sieben aufeinander abgestimmten Application Blocks wurde in einem Überblick [1] bereits vorgestellt. Die einzelnen Blocks erleichtern viele Routineaufgaben in der Anwendungsentwicklung. Ihre große Stärke liegt aber vor allem in einem durchgängigen Provider- und Konfigurationsmodell, das es ermöglicht, die grundlegende Funktionalität mit geringem Aufwand nachträglich anzupassen oder auszutauschen. Der Configuration Application Block steht im Zentrum der Enterprise Library und wird von allen anderen Blocks benötigt. Neben den funktionalen Klassen zur eigentlichen Verwaltung der Konfiguration besteht die vorliegende Lösung durch ein GUI-Konfigurationswerkzeug, welches die Änderung der Daten besonders komfortabel ermöglicht.

Erleichterung für eigene Projekte

Das Lesen und Schreiben eigener Konfigurationsdaten gestaltet sich mit dem Con-

figuration Application Block recht einfach. Dank einer statischen Fassade genügt eine Zeile Code, um ein Objekt der eigenen Konfigurationsklasse (siehe Listing 1) mit Werten zu füllen:

```
TestConfig config = ConfigurationManager.GetConfiguration
    (TestConfig.SectionName) as TestConfig;
```

Die folgende Zeile schreibt dagegen die Konfigurationswerte wieder zurück in den zugrunde liegenden Datenspeicher:

```
ConfigurationManager.WriteConfiguration
    (TestConfig.SectionName, config);
```

Der Configuration Application Block muss allerdings selbst konfiguriert werden, damit dieser einfache Zugriff funktioniert. Er benötigt dabei zumindest Informationen über die Art des Datenzugriffs, den Speicherort und die Verwaltung der Konfigurationsdaten. Das Provider-Modell unterstützt als zentrale Erweiterungspunkte die beiden Basisklassen *StorageProvider* und *TransformerProvider*. Ein *StorageProvider* ermöglicht das Lesen und Schreiben von Objekten in einen Datenspeicher. Mitgeliefert ist lediglich die Klasse *XmlFileStorageProvider* für den Zugriff auf XML-Dateien. Die Community hat aber

kurz & bündig

Inhalt

Enterprise Library 1.0 - Configuration Application Block

Zusammenfassung

Der Configuration Application Block steht im Zentrum der Enterprise Library und führt ein flexibles Framework zur Konfiguration von Anwendungen vor, das auch in eigenen Projekten deutlichen Mehrwert bietet



Quellcode: www.dotnet-magazin.de

Listing 1

```
[Serializable]
public class TestConfig
{
    public const string SectionName = "TestConfig";

    private int limit;
    public int Limit
    {
        get { return limit; }
        set { limit = value; }
    }

    private string emailAddress;
    public string EmailAddress
    {
        get { return emailAddress; }
        set { emailAddress = value; }
    }

    private string filePath;

    public string FilePath
    {
        get { return filePath; }
        set { filePath = value; }
    }

    private ColorEnum color;
    public ColorEnum Color
    {
        get { return color; }
        set { color = value; }
    }

    public enum ColorEnum
    {
        Red,
        Green,
        Blue
    }
}
```

zum Beispiel auch entsprechende *Storage-Provider* für SQL Server oder die Windows Registry bereitgestellt [2]. Ein *Transformer-Provider* bereitet dagegen die vorliegenden Daten so auf, dass sie in den korrespondierenden Datenspeicher geschrieben werden können – in diesem Fall etwa durch XML-Serialisierung (*XmlSerializer-Transformer*).

Über das Metadatenkonzept der Enterprise Library werden der zu verwendende Provider ausgewählt und Zusatzinformationen bereitgestellt. Die *app.config*- bzw. *web.config*-Datei (Listing 2) stellt somit lediglich einen Einstiegspunkt für die Konfiguration dar. Die eigentlichen Konfigurationsdaten befinden sich bei dem gewählten Provider in einer externen XML-Datei (Listing 3).

Wer jetzt angesichts der gezeigten XML-Dateien dankend abwinkt, hat völ-

lig Recht: Solche Dateien möchte für eine Konfiguration niemand mehr von Hand pflegen müssen. Es wird daher höchste Zeit, das GUI des Configuration Application Block für diese Arbeit einzuspannen. Im Zentrum eines Konfigurationsknotens steht eine einfache Klasse, abgeleitet von *ConfigurationNode*, welche für die Zuordnung von Properties im GUI und Properties in der eigenen Konfigurationsklasse verantwortlich ist. Durch .NET-Designer-Attribute und zusätzliche Attribute der Enterprise Library lassen sich das Verhalten und die Darstellung der einzelnen Properties einfach beeinflussen. In Listing 4 wird der eigene *ConfigurationNode* in Auszügen dargestellt. Besonders angenehm ist dabei die Umsetzung der einzelnen Properties. Enumerationen werden vom *PropertyGrid* ohne jeden weiteren Aufwand als DropDown-Feld zur Auswahl angebo-

ten. Integer- und String-Werte lassen sich dagegen als Textfeld editieren. Wann immer ein komplexerer Eingabedialog notwendig ist, lässt sich ein externer Editor an die Property binden – wie etwa ein Dateiauswahldialog (z. B. *SaveFileDialog*) für die Auswahl des Dateipfades. In diesem Fall ergänzt ein Attribut der Enterprise Library (*FilteredFileNameEditor*) die Beschränkung auf spezielle Dateiendungen. Weitere .NET-Designer-Attribute steuern Meta-Informationen bei, wie die Beschrei-

Listing 2

```
<configuration>
  <configSections>
    <section name="enterpriselibrary.configurationSettings" type="Microsoft.Practices.EnterpriseLibrary.Configuration.
      ConfigurationManagerSectionHandler, Microsoft.Practices.EnterpriseLibrary.Configuration, Version=1.0.0.0, Culture=
        neutral, PublicKeyToken=null" />
  </configSections>
  <enterpriselibrary.configurationSettings xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://
    www.w3.org/2001/XMLSchema-instance" applicationName="TestApp" xmlns="http://www.microsoft.com/
    practices/enterpriselibrary/08-31-2004/configuration">
    <configurationSections>
      <configurationSection xsi:type="ReadOnlyConfigurationSectionData" name="TestConfig" encrypt="false">
        <storageProvider xsi:type="XmlFileStorageProviderData" name="XML File Storage Provider" path="TestConfig.config" />
        <dataTransformer xsi:type="XmlSerializerTransformerData" name="Xml Serializer Transformer">
          <includeTypes />
        </dataTransformer>
      </configurationSection>
    </configurationSections>
    <keyAlgorithmStorageProvider xsi:nil="true" />
    <includeTypes />
  </enterpriselibrary.configurationSettings>
</configuration>
```

Listing 3

```
<?xml version="1.0" encoding="utf-8"?>
<TestConfig>
  <xmlSerializerSection type="TestConfiguration.TestConfig, TestConfiguration, Version=1.0.1989.15001,
    Culture=neutral, PublicKeyToken=null">
    <TestConfig xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/
      XMLSchema-instance">
      <Color>Blue</Color>
      <Limit>20</Limit>
      <EmailAddress>jochenr@avanade.com</EmailAddress>
      <FilePath>C:\test.xml</FilePath>
    </TestConfig>
  </xmlSerializerSection>
</TestConfig>
```

Listing 4

```
[Image(typeof(TestConfigNode))]
public class TestConfigNode : ConfigurationNode
{
  private TestConfig config;

  [Description("Pick one of the color values.")]
  [Category("Test Settings")]
  public ColorEnum Color
  {
    get { return config.Color; }
    set { config.Color = value; }
  }

  [Description("Enter the limit between 1 and 99")]
  [Category("Test Settings")]
  [AssertRange(1, RangeBoundaryType.Inclusive, 100,
    RangeBoundaryType.Exclusive)]
  public int Limit
  {
    get { return config.Limit; }
    set { config.Limit = value; }
  }

  [Description("Enter a valid e-mail address")]
  [Category("Test Settings")]
  [Regex(@"^\w+([+]\w+)*@\w+([-]\w+)*\.\w+
    ([-]\w+)*$")]
  public string EmailAddress
  {
    get { return config.EmailAddress; }
    set { config.EmailAddress = value; }
  }

  [Description("Pick a file path")]
  [Category("Test Settings")]
  [Editor(typeof(SaveFileDialog), typeof(UITypeEditor))]
  [FilteredFileNameEditor("Configuration files
    (*.xml)*.xml|All files|*.*")]
  [FileValidation]
  public string FilePath
  {
    get { return config.FilePath; }
    set { config.FilePath = value; }
  }

  // (...)
}
```

bung und die Kategorie einer Properties oder ein eigenes Icon für den Konfigurationsknoten. Den größten Mehrwert bieten aber sicherlich die vielfältigen Attribute, die dazu verwendet werden können, eine Validierung der eingegebenen Werte durchzuführen, die etwa gegenüber einem Wertebereich, einem regulären Ausdruck oder einem gültigen Pfad zu einer Datei auf die Schreibrechte bestehen. Wer von den gezeigten Möglichkeiten noch nicht sonderlich beeindruckt ist, sollte sich z. B. den Blog von *Olaf Conijn* etwas genauer anschauen [3]. Er zeigt unter anderem das Editieren der *Machine.config* mithilfe des GUI-Konfigurationswerkzeuges der Enterprise Library.

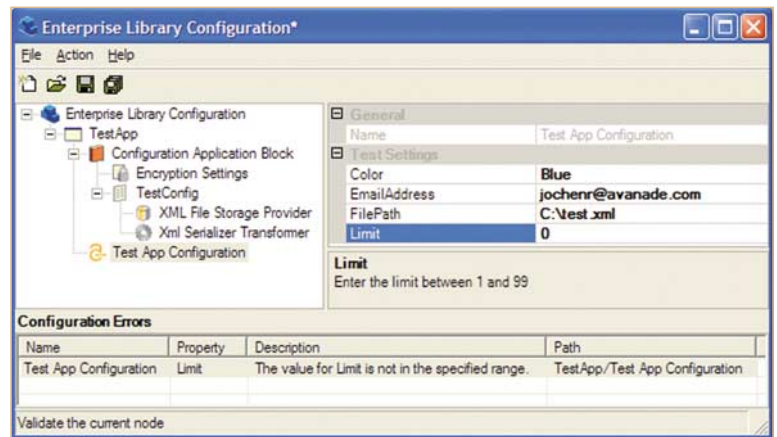
Mit den gezeigten Klassen *TestConfig* und *TestConfigNode* ist der größte Teil der eigenen Konfiguration bereits erledigt. Für die Verwaltung des Konfigurationsknotens benötigt das GUI noch eine spezialisierte *IConfigurationDesignManager*-Klasse. Deren Implementierung ist zwar nicht gerade als intuitiv zu bezeichnen, aber mit den bestehenden Klassen der Enterprise Library finden sich genügend Vorlagen. Auf www.dotnet-magazin.de ist zudem das in diesem Artikel gezeigte Beispiel vollständig verfügbar. Um die eigenen Konfigurationsklassen inklusive GUI-Unterstützung in die Konfiguration der Enterprise Library einzubinden, genügt es, die entsprechende Assembly ins */bin*-Verzeichnis der Enterprise Library zu kopieren. Das Konfigurationswerkzeug kontrolliert beim Laden einer neuen Konfiguration sämtliche DLLs im eigenen Verzeichnis und überprüft dadurch, ob sie einen *ConfigurationDesignManager* enthalten. Kennzeichnend gemacht wird dies durch ein Assembly-Attribut, das im eigenen Projekt gesetzt werden muss:

```
[assembly: ConfigurationDesignManager(typeof
    (TestConfiguration.TestConfigDesignManager))]
```

Der Lohn der Mühe ist dann hoffentlich die perfekte Integration der eigenen Konfigurationsklasse in das Konfigurationswerkzeug der Enterprise Library (Abbildung 1). Die gesetzten Attribute zur Darstellung, Editierung und Validierung werden von dem GUI entsprechend umgesetzt.

Ein besonderer Anreiz zur Verwendung des Configuration Application Blocks ist ganz sicher, dass alle weiteren Features der

Abb. 1: Enterprise Library Configuration mit eigener Konfigurationsklasse



Enterprise Library nun einfach genutzt werden können, ohne weiteren Code zu erzeugen. Zur Verschlüsselung der Konfigurationsdaten wird im Konfigurationswerkzeug zu den *Encryption Settings* ein Verschlüsselungsverfahren hinzugefügt. Der *File Key Algorithm Storage Provider* lässt den Benutzer aus verschiedenen Algorithmen wählen, erzeugt einen neuen Schlüssel und speichert diesen auf Wunsch per Data Protection API (DPAPI) gesichert ins Dateisystem ab. Ein gegen Multithreading abgesicherter Cache puffert die Konfigurationsdaten automatisch um einen schnellen Zugriff zu gewährleisten. Und schließlich stellt die Klasse *ConfigurationManager* dem Benutzer Events bereit, mit denen sich Konfigurationsänderungen benachrichtigen bzw. verwalten lassen.

Fazit

Wer sich den Configuration Application Block etwas genauer ansieht, stellt schnell fest, dass auch hierbei das Rad nicht neu erfunden wurde. Die Autoren setzten vielmehr auf die bereits bewährten Technologien XML-Serialisierung und *PropertyGrid*. Auch die bekannten Attribute aus *System.XML.Serialization* und *System.ComponentModel*, wie sie etwa der Visual Studio Designer verwendet, kommen daher erneut zum Einsatz. Auf der anderen Seite muss sich der Application Block den Vergleich mit .NET Framework 2.0 gefallen lassen. Im Gegensatz zu den bisherigen Klassen aus *System.Configuration* unterstützt die kommende Version nicht nur endlich einen Schreibzugriff, sondern bietet obendrein die Sicherung sensibler Daten per DPAPI- und RSA-Verschlüsselung. Selbst ein vergleichbares Provider-Modell und Events zur Benachrichtigung von Kon-

figurationsänderungen werden mit .NET 2.0 Einzug halten. Schließlich wird für ASP.NET 2.0 sogar ein rudimentäres Management-Consolen-Snap-in zur Änderung einer *Web.config* zur Verfügung stehen – für alle anderen Anwendungsarten ist jedoch keine integrierte Lösung in Sicht. Der Mehrwert des Configuration Application Block wird im Vergleich mit .NET 2.0 zwar kleiner, ihn trotzdem einzusetzen ergibt aber Sinn: Zum einen hat man heute schon die Möglichkeit, wichtige Features rund um das Thema Anwendungs-konfiguration einfach einzusetzen. Zum anderen wird eine zukünftige Version der Enterprise Library bemüht sein, unter Verwendung der neuen Features von .NET 2.0 größtmögliche Kompatibilität zur jetzigen Version zu wahren. Der Migrationsaufwand wird daher gering sein, verglichen mit einer ähnlich optimalen Umstellung der selbst gestrickten Lösung. Und schließlich bietet der Application Block nicht zuletzt mit dem für alle Anwendungsarten verwendbaren GUI auch noch ein wichtiges Merkmal, das .NET 2.0 fehlen wird. ●

Jochen Reinelt arbeitet als Consultant bei Avanade im Bereich .NET Anwendungsentwicklung – Sie erreichen ihn unter jochenr@avanade.com.

Stefan Zill ist Architekt und Leiter der EAI-Gruppe bei Avanade – Sie erreichen ihn unter stefanz@avanade.com.

● **Links & Literatur**

- [1] Jochen Reinelt / Stefan Zill, Sieben auf einen Streich, in: *dot.net magazin* 7/8.2005
- [2] workspaces.gotdotnet.com/entlib/
- [3] spaces.msn.com/members/olafconijn/?partqs=cat%3DEntLib