NEUTRONIUM

Windows-Apps alternativ

Desktop-Anwendungen mit HTML und C# entwickeln – Neutronium macht's möglich.

Im vergangenen Jahr hatte die dotnetpro bereits zwei weniger bekannte Frameworks für das Entwickeln von Desktop-Apps vorgestellt: OmniGUI und Eto.Forms [1]. Seitdem hat sich das Angebot an Alternativen nicht unbedingt verringert. Zu diesen Alternativen gehört auch die UI-Bibliothek Neutronium [2]. Sie will die Vorteile von WPF mit den Vorteilen von HTML & CSS verbinden.

Über Neutronium lassen sich .NET-Desktop-Anwendungen mit HTML, CSS, JavaScript und C# erstellen. Das klingt erst einmal relativ bekannt und ist nicht nur namentlich an den Editor Electron angelehnt. Dieser baut auf ein Grundgerüst von Node.js und JavaScript auf, während Neutronium eher auf .NET setzt.

Das Versprechen klingt verlockend – zumindest für Entwickler, die sich mit XAML als UI-Stack nie anfreunden konnten: die Geschäftslogik in gewohnter Umgebung mit .NET/C# entwickeln und für die Oberfläche HTML, CSS und JavaScript verwenden.

Damit die Webwelt auch wirklich Spaß macht, setzt Neutronium auf ChromiumFX [3], das einen einfachen Zugriff auf das Chromium Embedded Framework [4] erlaubt und auch immer recht schnell auf die neueste Chromium-Engine aktualisiert wird.

Das Hauptmerkmal von Neutronium ist allerdings nicht die Fähigkeit, einen Webbrowser einfach einzubinden, sondern vielmehr die Verknüpfung von Web und „klassischen" .NET/C#-Objekten mitsamt den bekannten MVVM-Mustern (*INotifyPropertyChanged* lässt grüßen).

Für diesen Brückenschlag mussten die Neutronium-Entwickler Adapter für die Webwelt bauen. Die größte Unterstützung gilt derzeit dem JavaScript-SPA-Framework Vue.js. Es gibt allerdings auch eine Variante für das Web-Framework Knockout.js, das aber im schnelllebigen JavaScript-Zeitalter quasi schon „tot" ist und auch in Bezug auf Neutronium eher ein Schattendasein fristet.

Erste Schritte

Um mit dem Framework Kontakt aufzunehmen, kommt das berühmte „Hello World" zum Einsatz. Dazu wird eine WPF-Applikation benötigt, die das .NET Framework in der Version 4.7.2 oder höher verwendet.

Über NuGet wird nun das Paket Neutronium.ChromiumFX.Vue geladen. Über dieses Package werden neben dem eigentlichen Neutronium alle weiteren Abhängigkeiten wie zum Beispiel ChromiumFX und das Binding Richtung Vue.js ins Projekt aufgenommen.

Listing 1: App.cs

```
using Neutronium.Core.JavascriptFramework;
using Neutronium.WebBrowserEngine.ChromiumFx;
using Neutronium.JavascriptFramework.Vue;

namespace NeutroniumApplication
{
    // Interaction logic for App.xaml
    public partial class App :
        ChromiumFxWebBrowserApp
    {
        protected override
            IJavascriptFrameworkManager
            GetJavascriptUIFrameworkManager()
        {
            return new VueSessionInjectorV2();
        }

    }
}
```

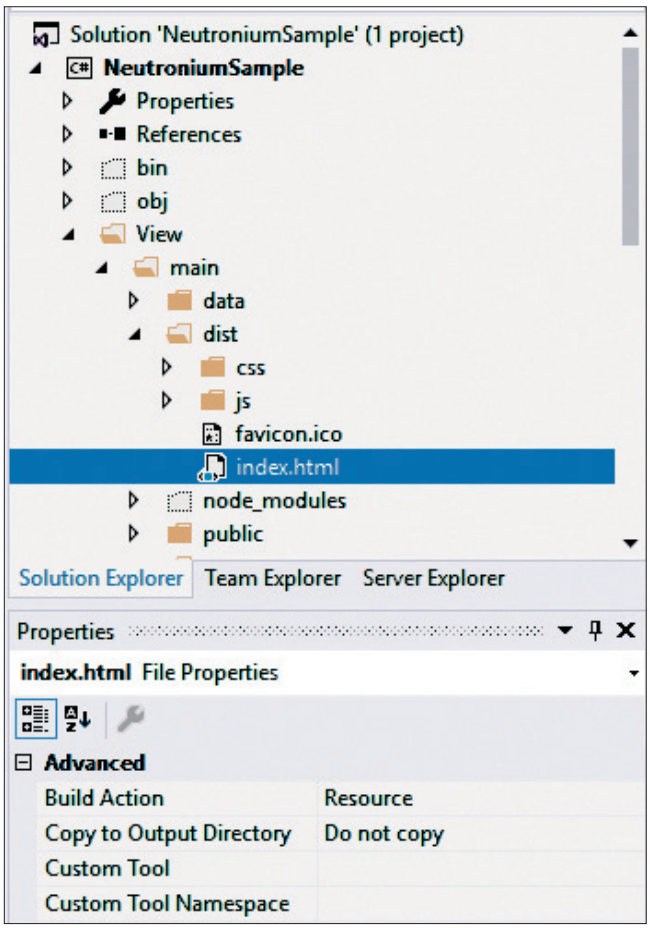
Listing 2: App.xaml

```
<neutroniumCfx:ChromiumFxWebBrowserApp
    x:Class="NeutronimApplication.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/
        xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/
        winfx/2006/xaml"
    xmlns:local="clr-namespace:NeutronimApplication"
    xmlns:neutroniumCfx="clr-namespace:Neutronium.
        WebBrowserEngine.ChromiumFx;assembly=
        Neutronium.WebBrowserEngine.ChromiumFx"
    StartupUri="MainWindow.xaml">
  <neutroniumCfx:ChromiumFxWebBrowserApp.Resources>
  </neutroniumCfx:ChromiumFxWebBrowserApp.Resources>
</neutroniumCfx:ChromiumFxWebBrowserApp>
```

In Listing 1 wird nun die Datei *App.cs* angepasst, um aus der Standard-WPF-App eine Neutronium-Anwendung zu machen. Hier wird zudem das Binding in Richtung Web instanziert – in diesem Fall über Vue.js, theoretisch wäre hier auch der Einstiegspunkt für andere UI-Bibliotheken. Die dazugehörige *App.xaml*-Datei muss ebenfalls angepasst werden, siehe Listing 2.

In der Code-behind-Datei von *MainWindow* kommen wir dem eigentlichen Neutronium-Einsatz schon sehr nah. Wie in normalen WPF-Projekten gilt es, hier dem *Window*-Objekt ein *DataContext*-Objekt zu übergeben. Das *DataContext*-Objekt ist wichtig; es muss vorhanden sein, ansonsten weigert sich Neutronium, irgendetwas darzustellen. Der folgende Beispielcode legt einfach ein Dummy-Objekt dieses Typs an:

```
public partial class MainWindow : Window
{
  public MainWindow()
  {
    InitializeComponent();

    DataContext = new HelloViewModel();
  }

  protected override void OnClosed(EventArgs e)
  {
    this.HtmlView.Dispose();
  }
}
```



Konfiguration der Dateien im dist-Ordner (Bild 1)

Listing 3: Definition des HtmlView-Objekts in XAML

```
<Window x:Class="NeutroniumSample.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/
      xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/
      2006/xaml"
    xmlns:d="http://schemas.microsoft.com/
      expression/blend/2008"
    xmlns:neutronium="clr-namespace:Neutronium.WPF;
      assembly=Neutronium.WPF"
    xmlns:mc="http://schemas.openxmlformats.org/
      markup-compatibility/2006"
    xmlns:local="clr-namespace:NeutroniumSample"
    mc:Ignorable="d"
    Title="MainWindow" Height="450" Width="800">
  <Grid>
    <neutronium:HTMLViewControl x:Name="HtmlView"
      IsDebug="True"
      Uri="pack://application:.../View/main/dist/
      index.html" HorizontalAlignment="Stretch"
      VerticalAlignment="Stretch"/>
  </Grid>
</Window>
```

```
public class HelloViewModel
{
  public string Message => "Hello Neutronium";
}
```

Dem aufmerksamen Leser wird das Objekt *HtmlView* aufgefallen sein. Aus WPF-Sicht ist es das Herzstück und wird in XAML definiert, siehe Listing 3. Das Objekt vom Typ *HtmlViewControl* zeigt auf die Datei *index.html* der Vue.js-Applikation, die als Ressource im Projekt mit verlinkt ist.

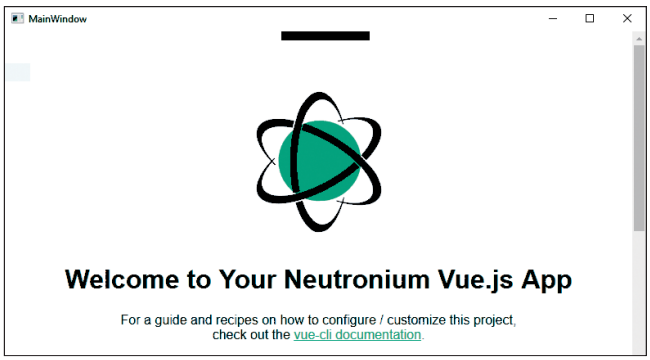
Hello Vue.js!

Szenenwechsel und Eintritt in die Webwelt. Bislang gibt es also eine WPF-Applikation, die im Grunde nur ein Webbrowser-Control enthält und auf eine HTML-Datei zeigt. Diese HTML-Datei ist der Einstiegspunkt für die Vue.js-Anwendung. Wie meist üblich in der Webwelt sind hierfür Node.js und npm notwendig, diese sind also einzurichten. Danach sollte man im Projekt einen *View*-Ordner anlegen und folgende Befehle ausführen:

```
npm install -g vue-cli
vue create main
cd main
vue add neutronium
npm install
```

```
npm run serve
... (develop)...
npm run build
```

Dieses Skript lädt alle Abhängigkeiten für das Entwickeln mit Vue.js auf den Rechner herunter und installiert das Vue.js-seitige Add-in für Neutronium. Bei diesem Schritt wird der Entwickler zudem nach der Version gefragt, ob er einen bestimmten „Router" anwenden möchte und welche Internationalisierungsstrategie. Die Fragen sind am besten mit den Standardwerten zu beantworten; *npm run serve* startet dann einen Webserver.



Das Neutronium/Vue.js-Gespann sagt Guten Tag (Bild 2)

Richtig einstellen

Ist das Ergebnis zufriedenstellend, kann der Befehl *npm run build* die entsprechenden Dateien für das WPF-Projekt kompilieren. Die Dateien im *dist*-Ordner sollten in das Projekt mit aufgenommen werden und bei den Dateieigenschaften der Datei *index.html* ist als *Build Action* die Angabe *Resource* auszuwählen, wie in Bild 1 zu sehen ist.

Auch eine andere Option sei hier ausdrücklich erwähnt: Unter den Eigenschaften des WPF-Projekts sollte auf der Registerkarte *Build* unter *Platform Target* unbedingt *Prefer 32bit* deaktiviert sein, ansonsten zeigt sich leider gar nichts.

Erste Erfolge

Ist all dies erledigt, sollte es möglich sein, in Visual Studio das WPF-Projekt zu starten, und es sollte sich ein WPF-Fenster mit der Willkommensmeldung von Neutronium/Vue.js zeigen (Bild 2).

Nachdem somit die Grundlagen geschaffen sind, steht die Datenbindung an. Hierfür wird die Klasse *HelloViewModel* in Listing 4 mit einer einfachen Implementierung der Schnittstelle *INotifyPropertyChanged* ausgestattet. In der Vue.js-App kommen nun eine weitere Überschrift und ein Input-Control mit einem Binding an das View-Modell hinzu:

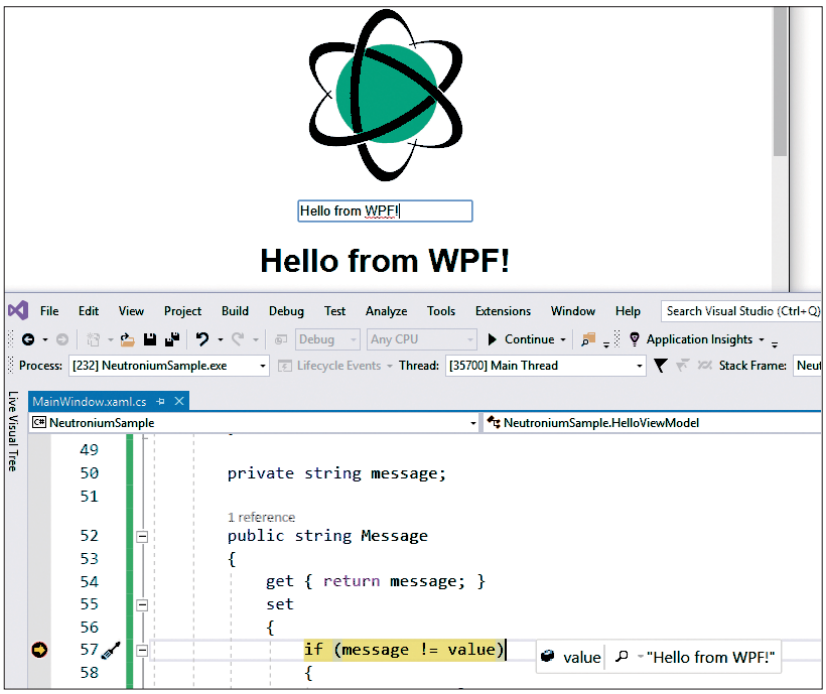
```
<template>
  <div id="app">
    <img class="logo" alt="Neutronium logo"
      src="./assets/logo.png">
    <br>
```

Listing 4: Die Klasse HelloViewModel mit INotifyPropertyChanged

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();

        DataContext = new HelloViewModel() {
            Message = "Hello from WPF" };
    }

    protected override void OnClosed(EventArgs e)
    {
        this.HtmlView.Dispose();
    }
}

public class HelloViewModel : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler
        PropertyChanged;

    protected void OnPropertyChanged(string name)
    {
        if (PropertyChanged != null)
        {
            PropertyChanged(this,
                new PropertyChangedEventArgs(name));
        }

    }

    private string message;

    public string Message
    {
        get { return message; }
        set
        {
            if (message != value)
            {
                message = value;
                OnPropertyChanged("Message");
            }
        }
    }
}
```

```
    <input v-model="Message"
      placeholder="edit me" />
    <br>
    <h1>{{ Message }}</h1>
    <HelloWorld msg="Welcome to Your
      Neutronium Vue.js App" />
  </div>
</template>
```

Bild 3 zeigt das Ergebnis: Es wird der Wert aus dem .NET-Objekt in der Überschrift und dem Input-Control dargestellt. Ändert sich der Wert im Input-Control, übernimmt das View-Modell den Wert und gibt ihn wieder zurück in die Vue.js-View.



Ändert sich der Wert im Eingabefeld, aktualisiert .NET die Oberfläche (Bild 3)

Mehr zu Bindungen, Commands und Konsorten

Neutronium bietet eine breite Palette an Optionen für die Datenbindung. Die „typischen" .NET-Objekte, wie zum Beispiel Zahlen, Strings, Datumsformate oder auch komplexere Objekte, lassen sich ebenfalls verknüpfen. Das GitHub-Repository von Neutronium [5] enthält ein umfangreicheres Beispiel, das etwa das Binding von Listen zeigt.

Über eine Erweiterung für Vue.js lässt sich auch das bekannte Command-Entwurfsmuster anwenden.

Wo Licht ist, ist auch Schatten

Wer Vue.js und .NET mag und es bis hier geschafft hat, der mag darin vielleicht schon den „Heilsbringer" der plattformübergreifenden UI-Entwicklung sehen. Leider ist Neutronium derzeit nur für Windows und als WPF-Control verfügbar. Bei GitHub ist ein Issue anhängig, das genau dieses Thema zusammenfasst, allerdings gibt es dazu wenig Fortschritte.

Die Neutronium-Architektur ist weitgehend modular aufgebaut und die eigentliche Bibliothek setzt auf dem .NET Standard 2.0 auf, allerdings ist ChromiumFX ziemlich spezifisch und im Moment auf Windows beschränkt – zumindest in dieser Konstellation.

Der ChromiumFX-Webbrowser selbst verhält sich im WPF-Kontext auch ähnlich stur wie das klassische Webbrowser-Control, das heißt, dass es nicht so einfach möglich ist, UI-Elemente „über" das Browser-Control selbst zu legen – dies sollte man bei entsprechenden Anwendungsfällen immer im Hinterkopf haben.

Einsatzgebiete und Fazit

Neutronium versucht den Spagat zwischen Desktop- und Webentwicklung. Technisch ist das Ergebnis durchaus interessant, jedoch kann man je nach Standpunkt zu dem Schluss „best of both worlds" oder „worst of both worlds" kommen.

Bis dato ist Neutronium auf die WPF-Shell beschränkt. In dem gezeigten Beispiel war die gesamte Applikation eine Neutronium-Anwendung, jedoch lässt sich die UI-Bibliothek auch in einer einzelnen WPF-View einsetzen; es ist also möglich, bestimmte Applikationsteile über Neutronium in Kombination mit Vue.js zu erweitern oder zu modernisieren, ohne die teure Geschäftslogik auswechseln zu müssen.

Entwickler, die ausschließlich mit JavaScript/Vue.js programmieren, werden sich von dem Modell vermutlich nicht recht überzeugen lassen – immerhin steht mit Electron eine weitverbreitete Plattform zur Verfügung.

Insgesamt ist Neutronium eine durchaus interessante Bibliothek und eignet sich vermutlich am ehesten für .NET-Entwickler, die sich auch gut in Vue.js auskennen und die bestehende WPF-Applikation mit ein paar modernen Elementen aus der Webwelt erweitern möchten. ■

[1] Robert Müsig, Einmal nach Rom, bitte! Eto.Forms und OmniGUI, dotnetpro 9/2918, Seite 12 ff., www.dotnetpro.de/A1809Eto
[2] Neutronium, www.dotnetpro.de/SL1908Neutronium1
[3] ChromiumFX, www.dotnetpro.de/SL1908Neutronium2
[4] Wikipedia, Chromium Embedded Framework, www.dotnetpro.de/SL1908Neutronium3
[5] Neutronium-Beispiel, www.dotnetpro.de/SL1908Neutronium4

Robert Mühsig
ist Softwareentwickler, Microsoft MVP und arbeitet in der Schweiz bei der Sevitec Informatik AG. Seine Schwerpunkte sind die Desktop- und Webentwicklung und generell alles, was im Microsoft-Umfeld passiert.

dnpCode A1908Neutronium