DEV

Ook wel eens een vraag gehad voor het maken over het inlezen van een flat file tekstbestand. Even snel iets in elkaar zetten? Onlangs was ik in de gelegenheid zulke functionaliteit te bouwen. Net als iedere ontwikkelaar wilde ik de functionaliteit zo generiek mogelijk houden, zodat het nog enige mate van herbruikbaarheid geniet. De valkuil daarbij is uiteraard dat je zo druk bent met generieke functionaliteit dat je je doel voorbij schiet en software bouwt die veel features bevat, waarvan er maar een paar gebruikt worden. You Aren't Gonna Need It.

Bouw een basic file importer

Implementatie met behulp van generics

Wouter Goedvriend
is Senior Consultant
bij Capgemini.
Weblog:
http://blog.goedvriend.com.

De implementatie in dit artikel toont een manier om een relatief eenvoudige, generieke oplossing te maken. Het bestand wordt geladen van een ASP.NET webform. Dit artikel behandelt verschillende onderwerpen, waaronder het lezen van data uit een CSV-tekstbestand, het maken van een generieke metode, het gebruiken van het generieke .NET list object, bewaren van uit het tekstbestand gelezen data door gebruik van .NET reflection principes. De belangrijkste stap is om de data uit het bestand te lezen en deze te gebruiken in een object binnen de applicatie. Daarna zijn er nog allerlei stappen te verzinnen om iets met de date te doen, maar dat valt buiten de scope van dit artikel. We beginnen met het inlezen van een file stream vanaf een willekeurige plaats binnen de applicatie.

Inlezen van een stream

Het lezen van een stream is eenvoudig te implementeren. ASP.NET heeft een standaard FileUpload control om die klus te klaren. Door middel van een browse button selecteer je een tekstbestand en krijg je de inhoud van het bestand als stream of byte array beschikbaar binnen de applicatie. Creëer een event handler op de click van de browse button en implementeer de code om de stream te verwerken.

```
protected void btnStartImport_Click(object sender,
EventArgs e)
{
  if (FileUploadControl1.HasFile)
  {
        Stream selectedStream = FileUploadControl1.
FileContent;
        ...
  }
}
```

Dat is alles wat je nodig hebt om de file stream beschikbaar te krijgen vanuit de FileUpload control.

Manieren om een stream te lezen

De FileUpload control is ook in staat om het geuploade bestand als byte array aan te leveren. Dat is op zichzelf een valide manier om zaken af te handelen, maar het wordt gecompliceerder wanneer je met de inhoud van het bestand werkt. Je krijgt de complete inhoud als 1 byte array terug. Je moet dan zelf bepalen van welke byte tot welke byte de regel is die je nodig hebt. Zo zijn er nog meer dingen die je moet regelen en weten over de inhoud van de byte array om er op de juiste manier de data uit te halen. Dus je moet meer doen om om hetzelfde te bereiken dan in het geval van een stream. Het .NET framework beschikt over componenten die goed met streams kunnen omgaan. De .NET Stream class is een goede optie wanneer je geen *die hard* byte array surfing wilt doen.

De import methode

Nu komen we bij het echte werk. Importeer de stream en stel een lijst samen met bruikbare rijen uit het bestand. Om dit voor elkaar te krijgen moet je een aantal stappen doorlopen. Verderop staat een implementatiepad uitgewerkt met de te nemen stappen. Maar eerst geef je de methode

een naam en moet hij netjes in een klasse worden ingebed. De klasse waar ik de importfunctionaliteit in gestopt hebt, heet FileImporter. De importmethode die ik heb gemaakt heet Import.

```
public static List<T> Import<T>(Stream importStream){}
```

De aanroep van de methode ziet er zo uit:

```
List<Customer> customers = fileImporter.Import<Customer>(selectedStream);
```

Implementatiepad

Nu de klasse is opgezet moeten de volgende stappen worden geïmplementeerd:

- Maak een lijst van het type T om instanties van dit type in te bewaren.
- Verwerk de stream rij voor rij en bewaar de rijen in een nieuwe lijst.
- Gebruik de nieuwe lijst met rijen om instanties van het type aan te maken die doorgegeven wordt via de aanroep van de methode
- Voeg de instanties toe aan de lijst voor objecten van het type T.
- Retourneer de lijst met objecten van het type T

Deze stappen vormen een ruw implementatiepad. Ik gebruik een dergelijk pad als richtlijn bij het maken van een implementatie.

Maak een lijst voor objecten met type T

De eenvoudigste component is een generieke lijst. Gebruik het beschikbare type T (vanuit aanroep van de methode).

```
...
List<T> instanceList = new List<T>();
...
```

De *instanceList* kan nu worden gebruikt om objecten van het type T in op te slaan. Wanneer de aanroep van de methode er bijvoorbeeld zo uitziet: `FileImporter.Import<Customer>()`, dan is het type van T gelijk aan Customer. De `container.ToArray()` produceert een array van Customer-objecten. Wanneer je een generieke lijst maakt met T, betekent dit dat de generieke lijst alleen objecten van het type Customer kan bevatten.

Verwerken van de stream

De volgende stap is het aanmaken van een lijst met rijen. Mijn basic file importer gebruikt een string array voor eenvoudige toegang tot de rijdata. Om een dergelijke lijst met rijen te maken, heb ik een private method gemaakt op de FileImporter, met de naam `ProcessFile()`.

```
private static string[] ProcessFile(Stream file)
{
                        List<string> rows = new List<string>();
                    StreamReader reader = new StreamReader(file);

            String line;
               // Remove the header line from the file.
            reader = RemoveHeader(reader);

              while ((line = reader.ReadLine()) != null)
            {
                    rows.Add(line);
            }

            return rows.ToArray();
}
```

Maak een tweede generieke lijst van het type string. Gebruik een StreamReader om het bestand te kunnen benaderen. Itereer over alle regels in de stream. Ik heb ervoor gekozen om de eerste regel over te slaan (via een private method RemoveHeader()), omdat het gebruikelijk is de eerste regel van een tekstbestand te gebruiken voor de definitie van de velden en hun volgorde. Dit wordt ook wel de metadata van het tekstbestand genoemd. De volgende stap is het toevoegen van iedere rij aan de lijst met strings. Retourneer ten slotte de opgebouwde lijst.

Itereer over de bruikbare elementen

Nu er een leesbare lijst met strings is gemaakt, kun je beginnen met de data daaruit te lezen door het construeren van een eenvoudige loop.

```
foreach (string row in rows)
{
          string[] fields = row.Split(separator.ToCharArray());
     object instance = ((T)Activator.CreateInstance(typeof(T)));

   ...
}
```

Je kunt de rijen nu nog splitsen zoals je dat wilt. In mijn geval heb ik gekozen om een private field te maken. Deze seperator lees ik vervolgens uit mijn applicatie configuratie file. Met deze techniek gebruik je een flexibele manier om het type seperator te configureren dat gebruikt wordt in de te importeren bestanden. De regel in het configuratiebestand voor de definitie van de seperator ziet er zo uit:

```
<add key="CustomerSeparator" value=";"/>
```

Gebruik de Split() methode om een string array te maken, met daarin alle velden. Gebruik de sepe-

rator uit de configuratie file om dit te doen. Zie onderstaande code.

```
string separator = States.Config[typeof(T).Name + "Separator"].ToString();
...
string[] fields = row.Split(separator.ToCharArray());
```

Naast het aanmaken van deze string array heb je nog een ander object nodig. Dit object is van het type T. Voordat we er data in kunnen opslaan, moet dit object geïnstantieerd worden. Regel 4 van het voorgaande codevoorbeeld toont hoe je een instantie creëert van het beschikbare type. Dit kan met behulp van de Activator klasse uit het .NET framework.

Data bewaren

Het belangrijkste doel van deze bestandsimport functionaliteit is om de data uit het bestand te krijgen en te bewaren in een object binnen de applicatie. In het voorbeeld gebruik ik een Customer object. Hiervoor wordt een mapping uitgevoerd, waarbij iedere regel in het bestand een Customer voorstelt. We lopen nu door de properties van het object met behulp van reflectie en vullen dan de waarde uit de fields lijst. Het type dat gebruikt wordt om informatie over een property te verkrijgen heet *PropertyInfo*. Onderstaand voorbeeld geeft een overzicht van het gebruik van reflectie om gegevens in een object te zetten. Deze code wordt in het vervolg onder de loop genomen.

```
foreach (PropertyInfo pi in typeof(T).GetProperties())
{
    string name = typeof (T).Name + pi.Name;
         string value = ConfigurationManager.AppSettings[name].ToString();

    int index;
    if (!int.TryParse(value, out index)) continue;

    if (index < fields.Length && index >= 0)
    {
        pi.SetValue(instance, pi, fields[index]);
    }
}
```

Opnieuw gebruik ik het configuratiebestand om de positie van velden in de fields array te achterhalen voor ieder van deze properties. Om de waarde voor de positie van het veld uit het configuratie bestand op te halen moet je de bijbehorende key kennen. Het configuratie bestand bevat informatie in het key/value formaat. Zie onderstaand voorbeeld.

```
<add key="CustomerAchternaam" value="8"/>
<add key="CustomerHuisnummer" value="1"/>
<add key="CustomerPostcode" value="3"/>
```

Door de naam van het type T te gebruiken samen met de naam van de *PropertyInfo* kun je een key samenstellen. Met behulp van deze key kan de bijbehorende value achterhaald worden.

```
string name = typeof (T).Name + pi.Name;
string value = ConfigurationManager.AppSettings[name].ToString();
```

In dit voorbeeld zou de property Customer.Achternaam resulteren in CustomerAchternaam. De value vertegenwoordigt, in mijn geval, de index van het veld met de waarde voor de desbetreffende property. De index kan vervolgens geparst worden met behulp van *int.TryParse*, omwille van de consistentie. Als de *int.TryParse* mislukt, betekent dit dat de key niet is gevonden in het configuratie bestand of dat er iets mis is met de waarde daarvan. Mocht er enig probleem optreden gaat de loop door naar de volgende *PropertyInfo* in de lijst met properties. Als de *int.TryParse* slaagt, is er een valide waarde gevonden in het configuratie bestand en kan het proces verder gaan. Zet nu de waarde van het veld uit de fields array in het object, door de *PropertyInfo.SetValue()* method te gebruiken. Zie onderstaand codevoorbeeld.

```
pi.SetValue(instance, pi, fields[index]);
```

Voeg het nieuwe object toe aan de lijst

De laatste klus is om het gepersisteerde object toe te voegen aan de lijst, die eerder werd aangemaakt in de Import method.

```
instanceList.Add(instance);
```

De laatste stap van de Import methode is het retourneren van de lijst.

```
return instanceList;
```

Conclusie

Met dit artikel heb ik laten zien hoe generics kunnen helpen bij het bouwen van eenvoudige file importing functionaliteit. Omdat het een implementatie op generiek niveau betreft, kan deze gemakkelijk in verschillende scenario's gebruikt worden.