Robust seeding

using

Seido Seed Generator

Git repositories

https://github.com/SEIDOAB-DACC/L04_seedGenerator.git

Exploring the csSeedGenerator

A topic that I have seen repeatedly in software development, is the underestimation of the value of robust seeding of a model for testing. I have seen developers put hours and hours of development time into a model and services, only to hand seed with 10 or in best case 100 items. Even in EFC literatures, I see hand seeding with only few items, mostly as part of a migration, being the norm.

The whole point about seeding is to test and stress-test the program and model under as realistic circumstances as possible using test data. Now, stress-testing your algorithms with 10 items or 100 000 items will give very different results as you will see challenges with performance, security, unforeseen data combinations, etc., as your model item population grows.

To overcome this, I developed the class csSeedGenerator, in order to easily be able to generate a large set of random seeds, be it names, addresses, emails, dates, enum types, lists, poems, or simple lorum-ipsum texts. Using a json file, you can also create your seeding source.

With csSeedGenerator it is as easy to generate 10 addresses or customer orders as it is to generate 10 000 000.

Using a json file, you can also create your seeding source, for your own seeding needs.

Let's explore csSeedGenerator.

In the project seedGenerator, you find the file csSeedGenerator.cs, open it and explore region exported types.

```
csSeedGenerator.cs M ×
L04_seedGenerator > seedGenerator > csSeedGenerator.cs > ISeed > Seed
9   {
10      #region exported types
        4 references | You, 21 seconds ago | 1 author (You)
11      public interface ISeed<T>
12      {
13          //In order to separate from real and seeded instances
            5 references
14          public bool Seeded { get; set; }
15
16          //Seeded The instance, you should always set Seeded = true in this method
            5 references
17          public T Seed(csSeedGenerator seedGenerator);      You, 3 days ago • initial
18      }
```

First you will see an interface declaration ISeed<T>. Any class you declare that implements ISeed will have to implement two members:

- public bool Seeded { get; set; } : The is used to determine an instance of you class contains real or seeded data.
- public T Seed(csSeedGenerator seedGenerator): This method is used to seed an instance of the class, which is then returned. This methos should also set Seeded = true.

```
csSeedGenerator.cs M ×
L04_seedGenerator > seedGenerator > csSeedGenerator.cs > ISeed > Seed
        3 references | You, 3 days ago | 1 author (You)
20      public class csSeededLatin
21      {
            2 references
22          public string Paragraph { get; init; }
            3 references
23          public List<string> Sentences { get; init; }
            3 references
24          public List<string> Words { get; init; }
25      }
26
        5 references | You, 3 days ago | 1 author (You)
27      public class csSeededQuote
28      {
            2 references
29          public string Quote { get; init; }
            2 references
30          public string Author { get; init; }
31      }
32      #endregion
```

Next, you will see two used to get more complex seeds from csSeedGenerator.

- csSeededLatin is used to get complex random latin data
- csSeededQuote is used to get random quotes and its author.

Further down, you see the class csSeedGenerator. The class is derived from .NET Random class which gives me basic random functionality. Let's first look at the overall structure.

```
csSeedGenerator.cs M ×
L04_seedGenerator > seedGenerator > csSeedGenerator.cs > csSeedGenerator
 8   namespace SeedGenerator
 9   {
10 >     #region exported types ···
33
         7 references | You, 30 seconds ago | 1 author (You)
34       public class csSeedGenerator : Random
35       {
             32 references
36           csSeedJsonContent _seeds = null;
37
38 >         #region get simple seeds for Names ···
44
45 >         #region get simple seeds for Addresses ···
77
78 >         #region get simple seeds for Emails and PhoneNr ···
89
90 >         #region get complex seeds for Quotes ···
103
104 >        #region get complex and simple seeds for bogus Latin ···
143
144 >        #region get simple seeds for names of Music Groups and Albums ···
153
154 >        #region get simple seeds for DateTime, bool and decimal numbers ···
187
188 >        #region pick seeds from your provided String, Enum type and List<TItem>
211
212 >        #region pick unique seeds from your provided List<TItem> ···
291
292 >        #region Generate seeded Lists of TItem, TItem must implement ISeed<TIte
341
342 >        #region internal classes to initialize master seed source content ···
601
602 >        #region create seed source json file called master-seed.json        You,
608
609 >        #region contructors ···
623
624 >        #region internal classes to read seed source from json file ···
910      }
911  }}}
```

Each region is self-explanatory and used to generate simple and complex seeds, create a seed source json file and read a seed source from a json file.

Let's look at the constructor and how master seeds are generated.

```
csSeedGenerator.cs M ×
L04_seedGenerator > seedGenerator > csSeedGenerator.cs > csSeedGenerator
34      public class csSeedGenerator : Random
342         #region internal classes to initialize master seed source content
            2 references
343         csSeedJsonContent CreateMasterSeeds()
344         {
345             return new csSeedJsonContent()
346             {
347 >               _quotes = new List<csSeedQuote>…
534 >               _latin = new List<csSeedLatin> {…
556 >               _addresses = new List<csSeedAddress>…
579                 _names = new csSeedNames
580                 {
581                     jsonFirstNames = "Harry, Lord, Hermione, Albus, Severus, Ron, Draco, Frodo,
582                     jsonLastNames = "Potter, Voldemort, Granger, Dumbledore, Snape, Malfoy, Bag
583                     jsonPetNames = "Max, Charlie, Cooper, Milo, Rocky, Wanda, Teddy, Duke, Leo,
584                 },
585 >               _domains = new csSeedDomains…
589 >               _music = new csSeedMusic…
598             };
599         }
600         #endregion
601
602 >       #region create seed source json file called master-seed.json      You, 22 seconds ago •
608
609         #region contructors
            1 reference
610         public csSeedGenerator()
611         {
612             _seeds = CreateMasterSeeds();
613         }
```

You can see that the first call is to CreateMasterSeeds(), which initializes a set of internal fields. In the field _names you see the typical pattern: strings with items separated by a delimiters “, ”.

The class csSeedNames converts the strings by breaking the string into a string[] based on the delimiters.

```
csSeedGenerator.cs M ×
L04_seedGenerator > seedGenerator > csSeedGenerator.cs > csSeedGenerator
            3 references | You, 3 days ago | 1 author (You)
725         class csSeedNames
726         {
727             #region Names towards json file
                3 references
728             string _jsonFirstNames;
                1 reference
729             public string jsonFirstNames
730             {
731                 get => _jsonFirstNames;
732                 set
733                 {
734                     _jsonFirstNames = value;
735                     _firstNames = _jsonFirstNames.Split(", ").ToList();
736                 }
737             }
738
                3 references
739             string _jsonLastNames;
                1 reference
740 >           public string jsonLastNames…
749
                3 references
750             string _jsonPetNames;
                1 reference
751 >           public string jsonPetNames…
760             #endregion
761
                2 references
762             List<string> _firstNames;
763             [JsonIgnore]
                2 references
764             public List<string> FirstNames => _firstNames;
```

When asking csSeedGenerator to provide a random name, this is done by simply picking randomly from the string[].

```
csSeedGenerator.cs M
L04_seedGenerator > seedGenerator > csSeedGenerator.cs > csSeedGenerator
33
    7 references | You, 7 minutes ago | 1 author (You)
34  public class csSeedGenerator : Random
35  {
        32 references
36      csSeedJsonContent _seeds = null;
37
38      #region get simple seeds for Names
        3 references
39      public string PetName => _seeds._names.PetNames[this.Next(0, _seeds._names.PetNames.Count)];
        4 references
40      public string FirstName => _seeds._names.FirstNames[this.Next(0, _seeds._names.FirstNames.Count)];
        4 references
41      public string LastName => _seeds._names.LastNames[this.Next(0, _seeds._names.LastNames.Count)];
        3 references
42      public string FullName => $"{FirstName} {LastName}";
43      #endregion
```

Simple, isn't it? I use the same pattern for many of the simple seeds.

Before moving on, open the file master-seeds.json. Here you will find an alternative seed source, with the same structure as when I generated the master seeds previously.

```
csSeedGenerator.cs M    {} master-seeds.json
L04_seedGenerator > seedGenerator > {} master-seeds.json > ...
    You, 3 days ago | 1 author (You)
1   {
2 >   "_quotes": [ ...
147   ],
148 >   "_latin": [ ...
179   ],
180 >   "_addresses": [ ...
201   ],
202   "_names": {
203     "jsonFirstNames": "Harry, Lord, Hermione, Albus, Severus, Ron, Draco, Frodo, Gandalf, Sam, Pere
204     "jsonLastNames": "Potter, Voldemort, Granger, Dumbledore, Snape, Malfoy, Baggins, the Gray, Gam
205     "jsonPetNames": "Asterix, Obelix, Dogmatix, Getafix, Vitalstatistix, Cacofonix, Geriatrix, Unhy
206   },
207 >   "_domains": { ...
209   },
210 >   "_music": { ...
215   }
216 }     You, 3 days ago • initial commit
```

You can simply change the names in this file to give your own flavors to the seeds as I will show later.

Further, I build more advanced methods, for example a random date. Here the key point is to create valid dates, possibly withing a year span.

```
csSeedGenerator.cs M ×   Program.cs
L04_seedGenerator > seedGenerator > csSeedGenerator.cs > csSeedGenerator
34      public class csSeedGenerator : Random
153
154         #region get simple seeds for DateTime, bool and decimal numbers
            3 references
155         public DateTime DateAndTime(int? fromYear = null, int? toYear = null)
156         {
157             bool dateOK = false;
158             DateTime _date = default;
159             while (!dateOK)
160             {
161                 fromYear ??= DateTime.Today.Year;
162                 toYear ??= DateTime.Today.Year + 1;
163
164                 try
165                 {
166                     int year = this.Next(Math.Min(fromYear.Value, toYear.Value),
167                         Math.Max(fromYear.Value, toYear.Value));
168                     int month = this.Next(1, 13);
169                     int day = this.Next(1, 32);
170
171                     _date = new DateTime(year, month, day);
172                     dateOK = true;
173                 }
174                 catch
175                 {
176                     dateOK = false;
177                 }
178             }
179
180             return DateTime.SpecifyKind(_date, DateTimeKind.Utc);
181         }
```

Or a simple 50/50 chance of true/false and random decimal number:

```
csSeedGenerator.cs M ×   Program.cs
L04_seedGenerator > seedGenerator > csSeedGenerator.cs > csSeedGenerator
34      public class csSeedGenerator : Random
154         #region get simple seeds for DateTime, bool and decimal numbers
            3 references
155         public DateTime DateAndTime(int? fromYear = null, int? toYear = null)…
182
            1 reference
183         public bool Bool => (this.Next(0, 10) < 5) ? true : false;
184
            3 references
185         public decimal NextDecimal(int _from, int _to) => this.Next(_from * 1000, _to * 1000) / 1000M;
186         #endregion      You, 3 days ago • initial commit
```

Or randomly pick from string, your own enum type, or list of complex types:

csSeedGenerator.cs M × Program.cs

L04_seedGenerator > seedGenerator > csSeedGenerator.cs > csSeedGenerator

```
34      public class csSeedGenerator : Random
187
188         #region pick seeds from your provided String, Enum type and List<TItem>
            4 references
189         public string FromString(string _inputString, string _splitDelimiter = ", ")
190         {
191             var _sarray = _inputString.Split(_splitDelimiter);
192             return _sarray[this.Next(0, _sarray.Length)];
193         }
            1 reference
194         public TEnum FromEnum<TEnum>() where TEnum : struct
195         {
196             if (typeof(TEnum).IsEnum)
197             {
198
199                 var _names = typeof(TEnum).GetEnumNames();
200                 var _name = _names[this.Next(0, _names.Length)];
201
202                 return Enum.Parse<TEnum>(_name);
203             }
204             throw new ArgumentException("Not an enum type");
205         }
            1 reference
206         public TItem FromList<TItem>(List<TItem> items)
207         {
208             return items[this.Next(0, items.Count)];
209         }
210         #endregion
```

Or to generate a List of your own type, TItem, each item randomly seeded. To be able to do this, I need to rely on that TItem implements a method called Seed(). An interface, ISeed<Item> will do the trick.

csSeedGenerator.cs M × Program.cs

L04_seedGenerator > seedGenerator > csSeedGenerator.cs > csSeedGenerator

```
34      public class csSeedGenerator : Random
291
292         #region Generate seeded Lists of TItem, TItem must implement ISeed<TItem>
293
294         //ISeed<TItem> and, in some cases, IEquatable<TItem> has to be implemented to use this method
            3 references
295         public List<TItem> ItemsToList<TItem>(int NrOfItems)
296             where TItem : ISeed<TItem>, new()
297         {
298             //Create a list of seeded items
299             var _list = new List<TItem>();
300             for (int c = 0; c < NrOfItems; c++)
301             {
302                 _list.Add(new TItem() { Seeded = true }.Seed(this));
303             }
304             return _list;
305         }
```

Another challenge is to generate a list of TItem, but each item should be unique. Well, if TItem promises to implement IEquatable<TItem>, then I can compare two items.

HashSet<> is using IEquatable<> to ensure all items with the same Hashcode are overwritten in the set, because they have the same key. Object.GetHashCode() is used by HashSet<> and should be overridden as part of a recommended implementation of IEquatable<>, so we are home.

csSeedGenerator.cs M Program.cs

L04_seedGenerator > seedGenerator > csSeedGenerator.cs > csSeedGenerator

```
34      public class csSeedGenerator : Random
306
307         //Create a list of unique randomly seeded items
            1 reference
308         public List<TItem> UniqueItemsToList<TItem>(int tryNrOfItems, List<TItem> appendToUnique = null)
309             where TItem : ISeed<TItem>, IEquatable<TItem>, new()
310         {
311             //Create a list of uniquely seeded items
312             HashSet<TItem> _set = (appendToUnique == null) ? new HashSet<TItem>() : new HashSet<TItem>(appendTo
313
314             while (_set.Count < tryNrOfItems)
315             {
316                 var _item = new TItem() { Seeded = true }.Seed(this);
317
318                 int _preCount = _set.Count();
319                 int tries = 0;
320                 do
321                 {
322                     _set.Add(_item);
323
324                     if (_set.Count == _preCount)
325                     {
326                         //Item was already in the _set. Generate a new one
327                         _item = new TItem() { Seeded = true }.Seed(this);
328                         ++tries;
329
330                         //Does not seem to be able to generate new unique item
331                         if (tries > 5)
332                             return _set.ToList();
333                     }
334
335                 } while (_set.Count <= _preCount);
336             }
337
338             return _set.ToList();
339         }
340         #endregion
```

As a final challenge let's generate a set of unique samples from a list of your own type. The list does not have to carry unique items, but your result set should. Again, HashSet<> will do the trick.

csSeedGenerator.cs M Program.cs

L04_seedGenerator > seedGenerator > csSeedGenerator.cs > csSeedGenerator

```
34      public class csSeedGenerator : Random
212         #region pick unique seeds from your provided List<TItem>
213
214         //Pick a number of unique items from a list of TItem (the List does not have to be unique)
215         //ISeed<TItem> and IEquatable<TItem> has to be implemented to use this method
            2 references
216         public List<TItem> UniqueItemsPickedFromList<TItem>(int tryNrOfItems, List<TItem> list)
217         where TItem : IEquatable<TItem>
218         {
219             //Create a list of uniquely seeded items
220             HashSet<TItem> _set = new HashSet<TItem>();
221
222             while (_set.Count < tryNrOfItems)
223             {
224                 var _item = list[this.Next(0, list.Count)];
225
226                 int _preCount = _set.Count();
227                 int tries = 0;
228                 do
229                 {
230                     _set.Add(_item);
231
232                     if (_set.Count == _preCount)
233                     {
234                         //Item was already in the _set. Pick a new one
235                         _item = list[this.Next(0, list.Count)];
236                         ++tries;
237
238                         //Does not seem to be able to pick new unique item
239                         if (tries > 5)
240                             return _set.ToList();
241                     }
242
243                 } while (_set.Count <= _preCount);
244             }
245
246             return _set.ToList();
247         }
```

Seeding the C# model with 10 to 10 000 000 random items

So let's put csSeedGenerator() to use. In project AppUsage file Program.cs you can see examples of how to use all the methods from csSeedGenerator. Here I generate both simple and complex seeds from 10 to 10 000 000.

Notice that I have to use the namespace Seido.Utilities.SeedGenerator.

```
Program.cs ×
L04_seedGenerator > AppUsage > Program.cs > Program > Main
8
9   using Seido.Utilities.SeedGenerator;
10
11  using Models;
12
13  namespace AppSeeding
14  {
        2 references | 0 references | 0 references | 0 references | 0 references
15      public enum enGreetings { Hello, Goodbye, GoodMorning, GoodEvening }
16
        0 references | You, 3 days ago | 1 author (You)
17      class Program
18      {
            0 references
19          static void Main(string[] args)
20          {
21              #region csSeedGenerator Usage Examples
22              Console.WriteLine("csSeedGenerator Usage Examples");
23
24              //Create a generator, inherited from .NET Random
25              var rnd = new csSeedGenerator();
26
27              Console.WriteLine("Random Names");
28              Console.WriteLine($"Firstname: {rnd.FirstName}");
29              Console.WriteLine($"Lastname: {rnd.LastName}");
30              Console.WriteLine($"Fullname: {rnd.FullName}");
31              Console.WriteLine($"Petname: {rnd.PetName}");
32
```

Here you can also see how to use your own seed source json file.

```
Program.cs ×
L04_seedGenerator > AppUsage > Program.cs > Program > Main
17      class Program
19          static void Main(string[] args)
123
124             Console.WriteLine("\nRead seeds from master-seeds.json file");
125             try
126             {
127                 fn = "./master-seeds.json";
128                 System.Console.WriteLine(Path.GetFullPath(fn));
129                 var rndMySeeds = new csSeedGenerator(fn);
130
131
132                 Console.WriteLine("Random Names using master-seeds.json file");
133                 Console.WriteLine(fn);
134
135                 Console.WriteLine($"Firstname: {rndMySeeds.FirstName}");
136                 Console.WriteLine($"Lastname: {rndMySeeds.LastName}");
137                 Console.WriteLine($"Fullname: {rndMySeeds.FullName}");
138                 Console.WriteLine($"Petname: {rndMySeeds.PetName}");
139             }
140             catch (Exception ex)
141             {
142                 Console.WriteLine("Could not read seeds from master-seed.json file");
143                 Console.WriteLine($"Error {ex.GetType()} {ex.Message}");
144             }
145
```