

Project Code: **BS1**

Source Code Samples for:

S3ONTHEGO

A secure offsite file backup & accessibility solution

Prepared by: Cormac Redmond
CASE4
53478637

Date: 02/05/2007

Supervisor: Dr. Brian Stone

OVERVIEW

This document contains sample code from the S3OnTheGo system. Rather than summarising “best code”, it tries to take core ideas from each component. If two key operations are somewhat similar, only one will be displayed. Also, most helper methods have been omitted.

Note: In developing the S3OnTheGo system, there was some generated code, such as the AmazonS3 proxy, and the UserService proxy, which were both generated (and then slightly edited) from WSDL, as described in the “Technical Manual”.

None of the code presented here is generated. Also omitted are most of tedious procedures involved in initialising forms/dialogs, informing user of operations, setting up Timers, displaying pop-ups, window management, etc.

Apologies for the format of the samples. No line number denotes a continuation from the previous line.

CODE SAMPLES

Shows how the entry class deals with unhandled errors:

```
9  /// <summary>
10  /// Startup class
11  /// </summary>
12  internal static class Program
13  {
14      /// <summary>
15      /// Handles unhandled exceptions
16      /// </summary>
17      /// <param name="o"></param>
18      private static void HandleUnhandledException(object o)
19      {
20
21          try
22          {
23              StringBuilder errorMsg = new StringBuilder();
24
25              //Use 'as' as it does not throw an exception
26              Exception someUnknownException = o as Exception;
27              //Get details of unhandled error
28              if (someUnknownException != null)
29              {
30                  errorMsg.AppendLine("Error in S3OnTheGo v" +
Global.ProductVersion + "\n");
31                  errorMsg.AppendLine("Error type\n" +
someUnknownException.GetType());
32                  errorMsg.AppendLine("\nError message\n" +
someUnknownException.Message);
33                  errorMsg.AppendLine("\nFull error\n" +
someUnknownException.ToString());
34                  if (someUnknownException.InnerException != null)
35                  {
36                      errorMsg.AppendLine("InnerExceptionString - " +
someUnknownException.InnerException.ToString());
37                  }
38              }
39              else
40              {
41                  errorMsg.AppendLine("Error in S3OnTheGo v" +
Global.ProductVersion + "\n");
```

```

42             errorMsg.AppendLine("\nError type\n" + o.GetType());
43             errorMsg.AppendLine("\nFull error\n" + o.ToString());
44         }
45         //Inform user
46         MessageBox.Show(errorMsg.ToString(), Global.ProductName,
MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
47     }
48     finally
49     {
50         MessageBox.Show("An unexpected has error occurred in " +
Global.ProductName + ". The application will now restart.", Global.ProductName,
MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
51         WindowManager.CurrentForm.AllowClose = true;
52         Application.Restart();
53     }
54 }
55
56 /// <summary>
57 /// Entry main method
58 /// </summary>
59 /// <param name="argv"></param>
60 [STAThread]
61 private static void Main(String[] argv)
62 {
63     try
64     {
65         Program.SubMain(argv);
66     }
67     catch (Exception exception1)
68     {
69         Program.HandleUnhandledException(exception1);
70     }
71 }
72
73 private static void OnGuiUnhandedException(object sender,
ThreadExceptionEventArgs e)
74 {
75     Program.HandleUnhandledException(e.Exception);
76 }
77
78 private static void OnUnhandledException(object sender,
UnhandledExceptionEventArgs e)
79 {
80     Program.HandleUnhandledException(e.ExceptionObject);
81 }
82
83
84 public static void SubMain(String[] argv)
85 {
86     //Register for any exceptions which are not handled down the
hierarchy
87     AppDomain.CurrentDomain.UnhandledException += new
UnhandledExceptionEventHandler(Program.OnUnhandledException);
88
89     //Register for an untrapped thread error
90     Application.ThreadException += new
ThreadExceptionHandler(Program.OnGuiUnhandedException);
91
92     //Necessary before creating any GUI components
93     Application.EnableVisualStyles();
94     Application.SetCompatibleTextRenderingDefault(false);
95
96     bool minimised = false;
97
98     //Check for the start minimised argument
99     foreach (String args in argv)
100     {
101         if (args.ToUpper() == "/M")
102         {
103             minimised = true;
104         }
105     }
106     //Create the main form
107     Form MainForm = new MainForm(minimised);
108
109     //Run the form, making sure that there is only one instance of the
program

```

```

110         SingleInstance.Run(mainForm);
111     }
112 }

```

Shows how SingleInstance deals with preventing two instances of the S3OnTheGo application from running:

```

11     public class SingleInstance
12     {
13         private static Mutex mutex;
14         private const int SW_RESTORE = 9;
15
16         private static IntPtr GetCurrentInstanceWindowHandle()
17         {
18             IntPtr pointer = IntPtr.Zero;
19             Process thisProcess = Process.GetCurrentProcess();
20             Process[] allProcesses =
21             Process.GetProcessesByName(thisProcess.ProcessName);
22             foreach (Process process in allProcesses)
23             {
24                 if ((process.Id != thisProcess.Id) &&
25                     (process.MainModule.FileName == thisProcess.MainModule.FileName) &&
26                     (process.MainWindowHandle != IntPtr.Zero))
27                 {
28                     return process.MainWindowHandle;
29                 }
30             }
31             return pointer;
32         }
33
34         private static bool IsAlreadyRunning()
35         {
36             bool createdNew;
37             FileSystemInfo currentPath = new
38             FileInfo(Assembly.GetExecutingAssembly().Location);
39             String fileName = currentPath.Name;
40             SingleInstance.mutex = new Mutex(true, @"Global\" + fileName, out
41             createdNew);
42             if (createdNew)
43             {
44                 SingleInstance.mutex.ReleaseMutex();
45             }
46             return !createdNew;
47         }
48
49         [DllImport("user32.dll")]
50         private static extern int IsIconic(IntPtr hWnd);
51         public static bool Run()
52         {
53             if (SingleInstance.IsAlreadyRunning())
54             {
55                 return false;
56             }
57             return true;
58         }
59
60         public static bool Run(Form frmMain)
61         {
62             if (SingleInstance.IsAlreadyRunning())
63             {
64                 MessageBox.Show("S3OnTheGo is already running.", "S3OnTheGo",
65                 MessageBoxButtons.OK, MessageBoxIcon.Asterisk);
66                 return false;
67             }
68             Application.Run(frmMain);
69             return true;
70         }
71
72         [DllImport("user32.dll")]
73         private static extern int SetForegroundWindow(IntPtr hWnd);
74         [DllImport("user32.dll")]
75         private static extern int ShowWindow(IntPtr hWnd, int nCmdShow);

```

```

71     /// <summary>
72     /// Switch to current instance - not currently used
73     /// </summary>
74     private static void SwitchToCurrentInstance()
75     {
76         IntPtr ptr1 = SingleInstance.GetCurrentInstanceWindowHandle();
77         if (ptr1 != IntPtr.Zero)
78         {
79             if (SingleInstance.IsIconic(ptr1) != 0)
80             {
81                 SingleInstance.ShowWindow(ptr1, 9);
82             }
83             SingleInstance.SetForegroundWindow(ptr1);
84         }
85     }
86 }

```

A sample from MainForm, to show what happens when a user chooses to backup:

```

1116     private void PerformBackup(BackupItem aBackupItem)
1117     { //Cannot 'Exit' from this, must click "Cancel"
1118         this.quitToolStripMenuItem.Enabled = false;
1119         //Inform user
1120         this.notifyIcon.ShowBalloonTip(0, Global.ProductName, "Currently
backing up '" + aBackupItem.Name + "'", ToolTipIcon.Info);
1121         using (PerformBackupDialog backingUpDialog = new
PerformBackupDialog())
1122         {
1123             backingUpDialog.BackupItem = aBackupItem;
1124             WindowManager.CurrentForm = backingUpDialog;
1125             DialogResult dialogResult = backingUpDialog.ShowDialog();
1126             if (dialogResult == DialogResult.Cancel)
1127             {
1128                 WindowManager.CurrentForm = this;
1129                 if (backingUpDialog.Error == "")
1130                 {
1131                     this.notifyIcon.ShowBalloonTip(0, Global.ProductName,
"Cancelled backup of [" + aBackupItem.Name + "]", ToolTipIcon.Warning);
1132                 }
1133                 else
1134                 {
1135                     this.notifyIcon.ShowBalloonTip(0, Global.ProductName,
backingUpDialog.Error, ToolTipIcon.Error);
1136                 }
1137             }
1138             else
1139             {
1140                 aBackupItem.LastBackup = DateTime.Now.ToUniversalTime();
1141                 Global.Engine.PutBackupItems(Global.User.BackupItems);
1142                 this.RefreshListView();
1143                 WindowManager.CurrentForm = this;
1144                 this.notifyIcon.ShowBalloonTip(0, Global.ProductName,
"Finished backing up", ToolTipIcon.Info);
1145             }
1146         }
1147         lock (this)
1148         {
1149             this.myBackupQueue.Dequeue(); // Remove from queue
1150             this.myBackingUp = false; //Application is no longer Backing Up
1151         }
1152         this.RefreshListView();
1153         this.quitToolStripMenuItem.Enabled = true;
1154     }

```

Shows what MainForm's scheduleTimer checks every 60 seconds:

```

1334     /// <summary>
1335     /// Handler called every 60 seconds to add scheduled Backup Items to
1336     /// the backup queue if it is due.
1337     /// </summary>
1338     /// <param name="sender"></param>
1339     /// <param name="e"></param>

```

```

1340     private void scheduleTimer_Tick(object sender, EventArgs e)
1341     {
1342         DateTime currentTime = DateTime.Now;
1343
1344         //Check all of the user's Backup Items
1345         foreach (BackupItem backupItem in Global.User.BackupItems)
1346         {
1347             DateTime backupItemScheduleTime;
1348             //If the Backup Item is locally created and is not currently in
the queue
1349             if (backupItem.IsLocal &&
!this.myBackupQueue.Contains(backupItem))
1350             {
1351                 backupItemScheduleTime = backupItem.Time;
1352                 switch (backupItem.Frequency)
1353                 {
1354                     case ScheduleFrequency.Daily:
1355                         if ((currentTime.TimeOfDay.Hours ==
backupItemScheduleTime.TimeOfDay.Hours) && (currentTime.TimeOfDay.Minutes ==
backupItemScheduleTime.TimeOfDay.Minutes))
1356                         {
1357                             lock (this)
1358                             {
1359                                 this.myBackupQueue.Enqueue(backupItem);
1360                             }
1361                         }
1362                         break;
1363
1364                     case ScheduleFrequency.Weekly:
1365                         if (((currentTime.TimeOfDay.Hours ==
backupItemScheduleTime.TimeOfDay.Hours) && (currentTime.TimeOfDay.Minutes ==
backupItemScheduleTime.TimeOfDay.Minutes)) && (currentTime.DayOfWeek ==
backupItem.DayOfWeek))
1366                         {
1367                             lock (this)
1368                             {
1369                                 this.myBackupQueue.Enqueue(backupItem);
1370                             }
1371                         }
1372                         break;
1373
1374                     case ScheduleFrequency.Monthly:
1375                         if (((currentTime.TimeOfDay.Hours ==
backupItemScheduleTime.TimeOfDay.Hours) && (currentTime.TimeOfDay.Minutes ==
backupItemScheduleTime.TimeOfDay.Minutes)) && ((currentTime.DayOfWeek ==
backupItem.DayOfWeek) && this.IsWeekOfMonthSame(currentTime, backupItem.WeekOfMonth)))
1376                         {
1377                             lock (this)
1378                             {
1379                                 this.myBackupQueue.Enqueue(backupItem);
1380                             }
1381                         }
1382                         break;
1383                 }
1384             }
1385         }
1386     }

```

Shows Engine retrieving lists of BackupItems for a User:

```

394     public List<BackupItem> GetBackupItems ()
395     {
396         try
397         {
398             List<BackupItem> backupItemList = new List<BackupItem> ();
399             List<S3OnTheGo.Backup.Engine.ListEntry> list =
this.myStorageService.ListBucket(this.MakeAllBackupItemsPrefix());
400             foreach (S3OnTheGo.Backup.Engine.ListEntry listEntry in list)
401             {
402                 GetObjectResult backupItemObject =
this.myStorageService.GetObject(listEntry.Key, false, true);

```

```

403             List<BackupItem> backupItemList = (List<BackupItem>)
Utility.DeserializeFromByteArray(backupItemObject.Data);
404             backupItemList.AddRange(backupItemList);
405         }
406         return backupItemList;
407     }
408     catch (SoapException soapException)
409     {
410         if (soapException.Code.Name != "Client.NoSuchKey")
411         {
412             throw;
413         }
414         return new List<BackupItem>();

```

Shows Engine retrieving a list of snapshots for a given BackupItem:

```

436     public List<Snapshot> GetSnapshots(BackupItem backupItem, bool getData)
437     {
438         List<Snapshot> allSnapshots = new List<Snapshot>();
439
440         //Get list of this Backup Item's snapshots
441         List<S3OnTheGo.Backup.Engine.ListEntry> snapshotList =
this.myStorageService.ListBucket(String.Format("{0}.BackupItem{1}.snapshot",
Global.User.ID, backupItem.ID));
442
443         //For each entry in the retrieved list
444         foreach (S3OnTheGo.Backup.Engine.ListEntry snapshotListEntry in
snapshotList)
445         {
446             //No need for this really - might be useful in future though
447             String text1 =
snapshotListEntry.Key.Substring(snapshotListEntry.Key.IndexOf("Backup") + 6,
snapshotListEntry.Key.Length - (snapshotListEntry.Key.IndexOf("Backup") + 6));
448
449             //Get snapshot information
450             GetObjectResult snapshotInfo =
this.myStorageService.GetObject(snapshotListEntry.Key, true, getData);
451
452             //Get time of backup
453             DateTime backupTime =
DateTime.Parse(this.GetMetadataValue("datetime",
snapshotInfo.Metadata)).ToLocalTime();
454
455             Snapshot snapshot = null;
456
457             if (getData)
458             { //Re-create like original snapshot
459                 snapshot = new Snapshot(text1, backupItem.ID, backupTime,
(List<BackupFile>) Utility.DeserializeFromByteArray(snapshotInfo.Data));
460             }
461             else
462             {
463                 snapshot = snapshot = new Snapshot(text1, backupItem.ID,
backupTime, null);
464             }
465             //Get size of transfer to aid progress reporting
466             snapshot.SizeOfTransfer =
long.Parse(this.GetMetadataValue("sizeoftransfer", snapshotInfo.Metadata));
467             allSnapshots.Add(snapshot);
468         }
469         return allSnapshots;
470     }
471 }

```

A sample from Engine, to create a list of files necessary for backup:

```

43     public List<BackupFile> analyseFilesForBackup(Snapshot aSnapshot,
BackgroundWorker performBackupBackgroundWorker, DoWorkEventArgs aDoWorkEventArgs)
44     {
45         int fileCount = 0;

```

```

46         //Get a list of all files from this Backup Item which is already
uploaded (if any)
47         List<S3OnTheGo.Backup.Engine.ListEntry> uploadedFiles =
this.myStorageService.ListBucket(this.MakeBackupItemFilesPrefix(aSnapshot.BackupItemID
));
48         List<BackupFile> filesNeedToBackup = new List<BackupFile>();
49         foreach (BackupFile currentSnapshotFile in aSnapshot.FileList)
50         {
51             //Report progress
52             performBackupBackgroundWorker.ReportProgress(-1, new
ProgressInfo(0, (long) aSnapshot.FileList.Count, (long) fileCount,
Path.GetFileName(currentSnapshotFile.Path), (long) 1, (long) 0));
53
54             bool found = false;
55
56             if (performBackupBackgroundWorker.CancellationPending)
57             {
58                 aDoWorkEventArgs.Cancel = true;
59                 return null;
60             }
61             //Get the key for the file
62             String filePartKey = this.MakeFilePartKey(currentSnapshotFile);
63
64             //Go through the files already uploaded
65             List<S3OnTheGo.Backup.Engine.ListEntry>.Enumerator enumerator =
uploadedFiles.GetEnumerator();
66             try
67             {
68                 while (enumerator.MoveNext() && found == false)
69                 {
70                     //See if the stored and local file have the same keys.
71                     if (enumerator.Current.Key == filePartKey)
72                     {
73                         found = true;
74                     }
75                 }
76             }
77             finally
78             {
79                 enumerator.Dispose();
80             }
81             if (!found)
82             {
83                 currentSnapshotFile.NumParts =
this.GetNumPartsForFile(currentSnapshotFile);
84                 filesNeedToBackup.Add(currentSnapshotFile);
85                 aSnapshot.SizeOfTransfer += currentSnapshotFile.Size;
86             }
87             fileCount++;
88             performBackupBackgroundWorker.ReportProgress(-1, new
ProgressInfo(0, (long) aSnapshot.FileList.Count, (long) fileCount,
Path.GetFileName(currentSnapshotFile.Path), (long) 1, (long) 1));
89             Thread.Sleep(1); //Give chance to update progress bars
90         }
91         return filesNeedToBackup;
92     }

```

A sample from Engine, showing how it handles encryption:

```

125     private void InitRijndael()
126     {
127         if (this.myRijndael == null)
128         {
129             //Get users crypto key and create the initial digest
130             byte[] hashedDigest = new SHA384Managed().ComputeHash(new
UTF8Encoding().GetBytes(Global.UserService.GetCryptoKey(Global.User.ID,
Global.User.HashedPassword)));
131             //Use digest to create a real key and initialisation vector
132             byte[] encryptionKey = new byte[32];
133             byte[] initVector = new byte[16];
134             for (int x = 0; x < 32; x++)

```

```

135         {
136             encryptionKey[x] = hashedDigest[x];
137         }
138         for (int x = 0; x < 16; x++)
139         {
140             initVector[x] = hashedDigest[x+32];
141         }
142         this.myRijndael = Rijndael.Create();
143         this.myRijndael.Key = encryptionKey;
144         this.myRijndael.IV = initVector;
145     }
146 }
147
148 private byte[] Encrypt(byte[] aData)
149 {
150     this.InitRijndael();
151     MemoryStream toEncrypt = new MemoryStream();
152     CryptoStream cryptoStream = new CryptoStream(toEncrypt,
this.myRijndael.CreateEncryptor(), CryptoStreamMode.Write);
153     cryptoStream.Write(aData, 0, aData.Length);
154     cryptoStream.Flush();
155     cryptoStream.FlushFinalBlock();
156     return toEncrypt.ToArray();
157 }
158
159 private byte[] Decrypt(byte[] aData, String decryptionSalt)
160 {
161     this.InitRijndael();
162     MemoryStream toDecrypt = new MemoryStream();
163     CryptoStream cryptoStream = new CryptoStream(toDecrypt,
this.myRijndael.CreateDecryptor(), CryptoStreamMode.Write);
164     cryptoStream.Write(aData, 0, aData.Length);
165     cryptoStream.Flush();
166     cryptoStream.FlushFinalBlock();
167     return toDecrypt.ToArray();
168 }

```

Shows how Engine deals with backing from a BackupFile file which is handed in from PerformBackupDialog:

```

228     public void DoBackupFile(BackupFile aFile, BackgroundWorker
aBackgroundWorker, DoWorkEventArgs aDoWorkEventArgs, List<String> aPartialBackupInfo,
long aTotalSize, ref long aCompleted)
229     {
230
231         long readFrom = 0;
232         long readTo = 0;
233         int filePart = 0;
234         bool retry = false;
235
236         //Initialise progress bar
237         aBackgroundWorker.ReportProgress(-1, new ProgressInfo(1, aTotalSize,
aCompleted, Path.GetFileName(aFile.Path), aFile.Size, (long) 0));
238
239         //Open file for reading
240         using (FileStream fileStream = new FileStream(aFile.Path,
FileStream.Open, FileAccess.Read))
241         {
242             while (readFrom < aFile.Size)
243             {
244                 byte[] fileBuffer;
245                 int nonWebExceptionRetries = MAX_NON_WEB_EXCEPTION_RETRIES;
246
247                 if ((readFrom + FILE_PART_LENGTH) > aFile.Size)
248                 {
249                     readTo = aFile.Size - readFrom;
250                 }
251                 else
252                 {
253                     readTo = FILE_PART_LENGTH;
254                 }
255                 fileBuffer = fileBuffer = new byte[readTo];
256                 fileStream.Seek(readFrom, SeekOrigin.Begin);

```

```

257         fileStream.Read(fileBuffer, 0, (int) readTo);
258
259         do
260         {
261             if (aBackgroundWorker.CancellationPending)
262             {
263                 aDoWorkEventArgs.Cancel = true;
264                 return;
265             }
266             try
267             {
268                 //Create file part key, metadata, and encrypt
buffer, and hand to StorageService to upload
269
270             this.myStorageService.PutObjectInline(this.MakeFilePartKey(aFile, filePart),
271             this.MakeFilePartMetadata(aFile), this.Encrypt(fileBuffer));
272                 retry = false;
273             }
274             catch (Exception storageServiceException)
275             {
276                 if (storageServiceException is WebException)
277                 {
278                     if
279                     ((WebException)storageServiceException).Status != WebExceptionStatus.Timeout)
280                     {
281                         Thread.Sleep(WEB_EXCEPTION_WAIT_TIME);
282                     }
283                     retry = true; //Retry to upload this buffer
284                 }
285                 else
286                 {
287                     Thread.Sleep(NON_WEB_EXCEPTION_WAIT_TIME);
288                     nonWebExceptionRetries--; //Reduce retries
289                     if (nonWebExceptionRetries == 0) //Too many
290                     exceptions, abort
291                     {
292                         throw;
293                     }
294                     retry = true; // Retry to upload this buffer
295                 }
296             }
297         }
298         while (retry); // Keeping trying if necessary
299         //Maintain this list in case user cancels and we need to
remove clutter from Amazon S3
300         aPartialBackupInfo.Add(this.MakeFilePartKey(aFile,
301         filePart));
302         filePart++;
303         readFrom += readTo;
304         aCompleted += readTo;
305         //Report progress
306         aBackgroundWorker.ReportProgress(-1, new ProgressInfo(1,
307         aTotalSize, aCompleted, Path.GetFileName(aFile.Path), aFile.Size, readFrom));
308     }
309 }

```

This shows how StorageService sends an object to Amazon S3:

```

191     public void PutObjectInline(String aKey, MetadataEntry[] aMetadata,
192     byte[] aData)
193     {
194         int soapExceptionRetryCount = 0;
195         bool getSignatureFromCache = true;
196         bool retry = false;
197         do
198         {
199             try

```

```

200             S3OnTheGo.Backup.Engine.StorageService.Signature
putObjectInlineSignature = this.GetSignature("PutObjectInline",
getSignatureFromCache);
201             this.myAmazonS3.PutObjectInline("S3OnTheGo.Backup", aKey,
aMetadata, aData, aData.LongLength, null, StorageClass.STANDARD, true,
Global.AwsAccessKeyId, putObjectInlineSignature.TimeStamp, true,
putObjectInlineSignature.SigString, null);
202             retry = false;
203             return;
204         }
205         catch (SoapException soapException)
206         {
207             if (soapExceptionRetryCount == 2)
208             {
209                 throw;
210             }
211             if ((soapException.Code.Name !=
"Client.RequestTimeTooSkewed") && (soapException.Code.Name !=
"Client.SignatureDoesNotMatch"))
212             {
213                 throw;
214             }
215             soapExceptionRetryCount++;
216             getSignatureFromCache = false;
217             retry = true;
218         }
219     }
220 } while (retry);
221 }

```

Shows how StorageService can get a list of certain objects from Amazon S3:

```

111     public List<ListEntry> ListBucket(String aPrefix)
112     {
113         ListBucketResult result;
114         int soapExceptionRetryCount;
115         List<ListEntry> resultsList = new List<ListEntry>();
116         String marker = ""; //Where to start off (used if results are
truncated)
117
118         soapExceptionRetryCount = 0;
119         bool getSignatureFromCache = true;
120         bool retry;
121         do
122         {
123             try
124             {
125                 S3OnTheGo.Backup.Engine.StorageService.Signature
listBucketSignature = this.GetSignature("ListBucket", getSignatureFromCache);
126                 result = this.myAmazonS3.ListBucket("S3OnTheGo.Backup",
aPrefix, marker, 1000, true, null, Global.AwsAccessKeyId,
listBucketSignature.TimeStamp, true, listBucketSignature.SigString, null);
127                 retry = false;
128                 //If null then there's nothing to deal with
129                 if (result.Contents != null)
130                 {
131                     //Add the results to the list
132                     foreach (ListEntry entry1 in result.Contents)
133                     {
134                         resultsList.Add(entry1);
135                     }
136                     //If the results we're not complete (which is at
Amazon's discretion)
137                     if (result.IsTruncated)
138                     {
139                         //Redo the procedure and only start from where it
left off
140                         marker = result.Contents[result.Contents.Length -
1].Key;
141                         soapExceptionRetryCount = 0;
142                         getSignatureFromCache = true;
143                         retry = true;

```

```

144         }
145     }
146 }
147 catch (SoapException soapException)
148 {
149     if (soapExceptionRetryCount == 2)
150     {
151         throw; //Can't handle, throw to Engine
152     }
153     if ((soapException.Code.Name !=
"Client.RequestTimeTooSkewed") && (soapException.Code.Name !=
"Client.SignatureDoesNotMatch"))
154     {
155         throw; //Can't handle, throw to Engine
156     }
157     soapExceptionRetryCount++;
158     getSignatureFromCache = false;
159     retry = true;
160 }
161 }
162 while (retry);
163 return resultsList;
164 }

```

Shows how StorageService requests a signature:

```

166     private S3OnTheGo.Backup.Engine.StorageService.Signature
MakeSignature(String aMethodName)
167     {
168         DateTime time = this.MakeTimeStamp();
169         return new
S3OnTheGo.Backup.Engine.StorageService.Signature(Global.UserService.MakeSignature(Global
al.User.ID, Global.User.HashedPassword, aMethodName + this.FormatAsIso8601(time)),
time);
170     }

```

This shows the bones of the S3OnTheGo web service, which talks to the S3OnTheGo database:

```

13 [WebService(Namespace = "http://www.S3OnTheGo.com/")]
14 [WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
15 public class S3OnTheGoService : System.Web.Services.WebService
16 {
17     public S3OnTheGoService()
18     {
19     }
20
21     [WebMethod]
22     public string GetCryptoKey(String suppliedUserName, String suppliedPassword)
23     {
24         //Get connection string
25         String connectionString =
System.Configuration.ConfigurationManager.ConnectionStrings["S3OnTheGoServiceDB"].Conn
ectionString;
26         SqlConnection dbConnection = new SqlConnection(connectionString);
27         SqlCommand sqlCommand = new SqlCommand("GetCryptoKey", dbConnection);
28         sqlCommand.CommandType = CommandType.StoredProcedure;
29         //Usage of Sql parameters also helps avoid SQL Injection attacks.
30         SqlParameter sqlParam = sqlCommand.Parameters.Add("@userName",
SqlDbType.VarChar, 20);
31         sqlParam.Value = suppliedUserName;
32         String cryptoSalt;
33         try
34         {
35             dbConnection.Open();
36             SqlDataReader reader = sqlCommand.ExecuteReader();
37             reader.Read(); // Advance to the one and only row
38             cryptoSalt = reader.GetString(0);
39             reader.Close();
40

```

```

41     }
42     catch (Exception ex)
43     {
44         throw new Exception("Error getting crypto key. " +
45             ex.Message);
46     }
47     finally
48     {
49         dbConnection.Close();
50     }
51     //Return key
52     return cryptoSalt;
53 }
54
55 [WebMethod]
56 public string GetSaltValue(String suppliedUserName)
57 {
58     String connectionString =
System.Configuration.ConfigurationManager.ConnectionStrings["S3OnTheGoServiceDB"].Conn
ectionString;
59     SqlConnection conn = new SqlConnection(connectionString);
60     SqlCommand cmd = new SqlCommand("LookupUser", conn);
61     cmd.CommandType = CommandType.StoredProcedure;
62     //Usage of Sql parameters also helps avoid SQL Injection attacks.
63     SqlParameter sqlParam = cmd.Parameters.Add("@userName",
SqlDbType.VarChar, 20);
64     sqlParam.Value = suppliedUserName;
65     String userSalt;
66     try
67     {
68         conn.Open();
69         SqlDataReader reader = cmd.ExecuteReader();
70         reader.Read();
71         if (reader.HasRows)
72         {
73             userSalt = reader.GetString(1);
74         }
75         else userSalt = null;
76         reader.Close();
77     }
78     catch (Exception ex)
79     {
80         throw new Exception("Exception getting salt value. " +
81             ex.Message);
82     }
83     finally
84     {
85         conn.Close();
86     }
87     return userSalt;
88 }
89 }
90
91 [WebMethod]
92 public String MakeSignature(string aUsername, string aPassword, string
aStringToSign)
93 {
94     //Authenticate user again, for clarity
95     if (VerifyPassword(aUsername, aPassword) == 0)
96     {
97         string CanonicalString = "AmazonS3" + aStringToSign;
98         Encoding ae = new UTF8Encoding();
99         HMACSHA1 Signature = new
HMACSHA1(ae.GetBytes((String)ConfigurationSettings.AppSettings["awsSecretAccessKey"]))
;
100         return
Convert.ToBase64String(Signature.ComputeHash(ae.GetBytes(CanonicalString.ToCharArray()
)));
101     }
102     else
103     {
104         return null;
105     }
106 }
107
108 //Authenticates a user
109 [WebMethod]

```

```

110     public int AuthenticateUser(String userName, String hashedPassword)
111     {
112         if (VerifyPassword(userName, hashedPassword) == 0)
113         {
114             return 0;
115         }
116         else
117         {
118             return 1;
119         }
120     }
121
122     [WebMethod]
123     public string Ping()
124     {
125         return "Pong";
126     }
127
128     private int VerifyPassword(string suppliedUserName,
129                               string suppliedPassword)
130     {
131         int validUser = -1;
132
133         String connectionString =
System.Configuration.ConfigurationManager.ConnectionStrings["S3OnTheGoServiceDB"].Conn
connectionString;
134         SqlConnection conn = new SqlConnection(connectionString);
135         SqlCommand cmd = new SqlCommand("LookupUser", conn);
136         cmd.CommandType = CommandType.StoredProcedure;
137         //Usage of Sql parameters also helps avoid SQL Injection attacks.
138         SqlParameter sqlParam = cmd.Parameters.Add("@userName",
SqlDbType.VarChar, 20);
139         sqlParam.Value = suppliedUserName;
140         try
141         {
142             conn.Open();
143             SqlDataReader reader = cmd.ExecuteReader();
144
145             reader.Read();
146             if (!reader.HasRows)
147             {
148                 validUser = 1;//Not such user
149             }
150             else
151             {
152                 String userPasswordHash = reader.GetString(0);
153                 reader.Close();
154                 if (userPasswordHash.Equals(suppliedPassword))
155                 {
156                     validUser = 0;//User authenticated
157                 }
158             }
159         }
160         catch (Exception ex)
161         {
162             throw new Exception("Exception verifying password. " +
ex.Message);
163         }
164     }
165     finally
166     {
167         conn.Close();
168     }
169     return validUser;
170 }
171 }

```

A sample from the SelectFilesControl custom control, which is used by a user to select a group of files to restore:

```

11 public class SelectFilesControl : UserControl
12 {
13     private IContainer components;
14     private ImageList imageList;
15     private List<BackupFile> myFileList;
16     private Dictionary<int, BackupFile> myFiles;
17     private ImageList imageList1;
18     private ImageList stateImages;
19     private TreeView filesView;
20
21     public SelectFilesControl()
22     {
23         this.myFiles = new Dictionary<int, BackupFile>();
24         this.InitializeComponent();
25     }
26
27     /// <summary>
28     /// Populates the TreeView
29     /// </summary>
30     private void PopulateTree()
31     {
32         this.filesView.Nodes.Clear();
33         this.filesView.BeginUpdate();
34
35         foreach (BackupFile backupFile in this.myFileList)
36         {
37             String[] arrayOfDirsAndFiles = backupFile.Path.Split(new
String[] { @"\" }, StringSplitOptions.RemoveEmptyEntries);
38             TreeNodeCollection nodes = this.filesView.Nodes;
39             //Deals with adding all the directorys for a file
40             for (int x = 0; x < (arrayOfDirsAndFiles.Length - 1); x++)
41             {
42                 String currentName = arrayOfDirsAndFiles[x];
43                 int index = this.GetIndexOfNode(nodes, currentName);
44                 if (index != -1)
45                 {
46                     nodes = nodes[index].Nodes;
47                 }
48                 else
49                 {
50                     TreeNode tempNode = nodes.Add(currentName.ToString());
51                     tempNode.ImageIndex = 0;
52                     tempNode.SelectedImageIndex = 0;
53                     tempNode.Tag = CheckState.Unchecked;
54                     tempNode.Checked = TagToBool(tempNode);
55                     nodes = tempNode.Nodes;
56                 }
57             }
58             //Actual name of file
59             String fileName = arrayOfDirsAndFiles[arrayOfDirsAndFiles.Length
- 1];
60
61             //Then add filename to nodes
62             TreeNode fileNode = nodes.Add(fileName.ToString());
63             fileNode.ImageIndex = 1;
64             fileNode.SelectedImageIndex = 1;
65             fileNode.Tag = CheckState.Unchecked;
66             fileNode.Checked = TagToBool(fileNode);
67             //Add the file the the myFiles Dictionary
68             this.myFiles.Add(fileNode.GetHashCode(), backupFile);
69         }
70         this.filesView.EndUpdate(); //Redraw tree
71         this.filesView.ExpandAll(); //Expand all
72     }
73
74     private void AddCheckedNodes(TreeNodeCollection nodes, List<BackupFile>
fileList)
75     {
76         foreach (TreeNode node in nodes)
77         {
78             TreeNodeCollection tempNodes = node.Nodes;
79             if (tempNodes.Count == 0)
80             {
81                 if (((CheckState)node.Tag) == CheckState.Checked)
82                 {
83                     fileList.Add(this.myFiles[node.GetHashCode()]);
84                 }
85             }
86         }
87     }

```

```

85         else
86         {
87             this.AddCheckedNodes(tempNodes, fileList);
88         }
89     }
90 }
91
92 private int GetIndexOfNode(TreeNodeCollection nodes, String aName)
93 { //See if the node we're dealing with has already been added (i.e., a
directory)
94     for (int x = 0; x < nodes.Count; x++)
95     {
96         if ((String)nodes[x].Text) == aName)
97         {
98             return x;
99         }
100     }
101     return -1;
102 }
103
104 /// <summary>
105 /// Gets all the selected files
106 /// </summary>
107 /// <returns>Select files</returns>
108 public List<BackupFile> GetSelectedFileList()
109 {
110     List<BackupFile> fileList = new List<BackupFile>();
111     this.AddCheckedNodes(this.filesView.Nodes, fileList);
112     return fileList;
113 }
114
115 /// <summary>
116 /// Recursively checks/unchecks all of a nodes child nodes as necessary
117 /// </summary>
118 /// <param name="node"></param>
119 /// <param name="state"></param>
120 private void SetCheckedChildNodes(TreeNode node, CheckState newState)
121 {
122     for (int x = 0; x < node.Nodes.Count; x++)
123     {
124         node.Nodes[x].Tag = newState;
125         node.Nodes[x].Checked = TagToBool(node.Nodes[x]);
126         this.SetCheckedChildNodes(node.Nodes[x], newState);
127     }
128 }
129
130
131 private void SetCheckedNode(TreeNode node)
132 {
133     CheckState newState = (CheckState)node.Tag;
134     switch (newState)
135     {
136         case CheckState.Unchecked:
137             newState = CheckState.Checked;
138             break;
139
140         default:
141             newState = CheckState.Unchecked;
142             break;
143     }
144     this.filesView.BeginUpdate(); //Disable redrawing
145     node.Tag = newState;
146     //Check all child nodes
147     this.SetCheckedChildNodes(node, newState);
148     //Check all parent nodes
149     this.SetCheckedParentNodes(node, newState);
150     this.filesView.EndUpdate(); //Allow it to be redrawn
151 }
152
153 /// <summary>
154 /// Recursively checks/unchecks all of a nodes parent nodes as
necessary
155 /// </summary>
156 /// <param name="aNode"></param>
157 /// <param name="aCheck"></param>
158 private void SetCheckedParentNodes(TreeNode node, CheckState newState)
159 {

```

```

160         //If a node has a parent
161         if (node.Parent != null)
162         {
163             CheckState state = CheckState.Unchecked;
164
165             for (int x = 0; x < node.Parent.Nodes.Count; x++)
166             {
167                 //If parent nodes state is different
168                 if (!newState.Equals(node.Parent.Nodes[x].Tag))
169                 { //Swap states
170                     if (state == CheckState.Unchecked)
171                     {
172                         state = CheckState.Checked;
173                         break;
174                     }
175                     state = CheckState.Unchecked;
176                     break;
177                 }
178             }
179             //Apply states
180             node.Parent.Tag = (state == CheckState.Checked) ?
CheckState.Checked : newState;
181             node.Parent.Checked = TagToBool(node.Parent);
182             //Recurse
183             this.SetCheckedParentNodes(node.Parent, newState);
184         }
185     }
186     public List<BackupFile> FileList
187     {
188         get
189         {
190             return this.myFileList;
191         }
192         //When handed a file list, populate tree
193         set
194         {
195             this.myFileList = value;
196             if (this.myFileList != null)
197             {
198                 this.PopulateTree();
199             }
200         }
201     }
202
203     private bool TagToBool(TreeNode temp)
204     {
205         if ((CheckState)temp.Tag == (CheckState.Checked))
206             return true;
207         else
208             return false;
209     }
210
211
212
213     //Deals with a click, and changes the state of the tree as needed
214     private void treeView1_MouseDown(object sender, MouseEventArgs e)
215     {
216         if (e.Button == MouseButtons.Left && e.Clicks == 1)
217         {
218             System.Windows.Forms.TreeViewHitTestInfo info =
filesView.HitTest(e.X, e.Y);
219             if (info.Node != null && info.Location.ToString() ==
"StateImage")
220             {
221                 this.SetCheckedNode(info.Node);
222             }
223         }
224     }
225
226     private void InitializeComponent()
227     {
228         //Removed for code sample
229     }
230 }
231 }

```

The ProgressInfo class, which is used to define the progress of a current operation:

```
5     internal class ProgressInfo
6     {
7         public long CurrentItemCompletedOperations;
8         public String CurrentItemName;
9         public long CurrentItemOperations;
10        public int Phase;
11        public long TotalCompletedOperations;
12        public long TotalOperations;
13
14        private ProgressInfo()
15        {
16        }
17
18        public ProgressInfo(int phase, long totalOperations, long
totalCompletedOperations, String currentItemName, long currentItemOperations, long
currentItemCompletedOperations)
19        {
20            this.Phase = phase;
21            this.TotalOperations = totalOperations;
22            this.TotalCompletedOperations = totalCompletedOperations;
23            this.CurrentItemName = currentItemName;
24            this.CurrentItemOperations = currentItemOperations;
25            this.CurrentItemCompletedOperations =
currentItemCompletedOperations;
26        }
27    }
```

Shows most of the PerformBackupDialog which deals with backing up files, and reporting progress information, etc:

```
12 public class PerformBackupDialog : CloseToSystemTrayForm
13 {
14     private Label analysingLabel;
15     private ProgressBar analysingProgressBar;
16     private BackgroundWorker performBackupBackgroundWorker;
17     private Button cancelButton;
18     private ColumnHeader completedColumn;
19     private IContainer components;
20     private Timer countdownTimer;
21     private BackgroundWorker deleteBackgroundWorker;
22     private GroupBox detailsGroupBox;
23     private ListView detailsListView;
24     private Label downloadingLeftLabel;
25     private ColumnHeader fileHeader;
26     private Button hideButton;
27     private GroupBox mainGroupBox;
28     private S3OnTheGo.Backup.Model.BackupItem myBackupItem;
29     private ProgressPopup myCancelProgressPopup;
30     private DateTime myCurrentPhaseStarted;
31     private String myError;
32     private int myLastPhase;
33     private ProgressInfo myLastProgressInfo;
34     private List<String> myPartialBackupInfo;
35     private ColumnHeader operationHeader;
36     private Label uploadingLabel;
37     private Label uploadingLeftLabel;
38
39     private ProgressBar uploadingProgressBar;
40
41     public PerformBackupDialog()
42     {
43         this.myPartialBackupInfo = new List<String>();
44         this.myLastPhase = -1;
45         this.myError = "";
46         this.InitializeComponent();
47         this.Text = Global.ProductName;
48     }
49
50     private void backgroundWorker_DoWork(object sender, DoWorkEventArgs e)
```

```

51     {
52         Snapshot currentSnapshot;
53         try
54         {
55             //Create new snapshot
56             currentSnapshot = new Snapshot(this.myBackupItem);
57         }
58         catch (DirectoryNotFoundException)
59         {
60             this.myError = String.Format("Cannot backup the backup item
'{}' because the folder '{}' could not be found.", this.myBackupItem.Name,
this.myBackupItem.FolderPath);
61             e.Cancel = true;
62             return;
63         }
64         //Get a list of files that must be uploaded from Engine
65         List<BackupFile> filesNeedToBackup =
Global.Engine.analyseFilesForBackup(currentSnapshot,
this.performBackupBackgroundWorker, e);
66         if (this.performBackupBackgroundWorker.CancellationPending)
67         {
68             e.Cancel = true;
69         }
70         else
71         { //Report progress
72             this.performBackupBackgroundWorker.ReportProgress(-1, new
ProgressInfo(0, (long) currentSnapshot.FileList.Count, (long)
currentSnapshot.FileList.Count, "", (long) (-1), (long) (-1)));
73             bool canContinue = false;
74             do
75             {
76                 //Get a list of files that may be currently locked
77                 List<BackupFile> lockedFileList =
Global.Engine.DiscoverLockedFiles(filesNeedToBackup);
78
79                 //If there's any locked files, inform user
80                 if (lockedFileList.Count > 0)
81                 {
82                     canContinue = false;
83                     String lockedFiles = "";
84                     foreach (BackupFile lockedFile in lockedFileList)
85                     {
86                         lockedFiles = lockedFiles + lockedFile.Path + "\n";
87                     }
88                     DialogResult informDialog = MessageBox.Show("The backup
cannot continue because the following files are being used by another user or
program:\n\n" + lockedFiles + "\nPlease close the programs that are using these files
and click 'Retry' to continue the backup, or click 'Cancel' to cancel the backup.",
Global.ProductName, MessageBoxButtons.RetryCancel, MessageBoxIcon.Exclamation);
89                     if (informDialog == DialogResult.Cancel)
90                     {
91                         e.Cancel = true;
92                         return;
93                     }
94                 }
95                 else
96                 {
97                     canContinue = true;
98                 }
99             }
100             while (!canContinue);
101             long num = 0;
102             //Request that Engine backs up all the files in the new list
103             foreach (BackupFile backupFile in filesNeedToBackup)
104             {
105                 Global.Engine.DoBackupFile(backupFile,
this.performBackupBackgroundWorker, e, this.myPartialBackupInfo,
currentSnapshot.SizeOfTransfer, ref num);
106                 if (this.performBackupBackgroundWorker.CancellationPending)
107                 {
108                     e.Cancel = true;
109                     return;
110                 }
111             }
112             this.performBackupBackgroundWorker.ReportProgress(-1, new
ProgressInfo(1, currentSnapshot.SizeOfTransfer, currentSnapshot.SizeOfTransfer, "",
(long) (-1), (long) (-1)));

```

```

113             Global.Engine.PutSnapshot(currentSnapshot); //Save snapshot
information on S3
114         }
115     }
116
117
118     /// <summary>
119     /// Handler (called by a BackgroundWorkers ProgressChanged delegate) to
update progress bars & counters
120     /// </summary>
121     /// <param name="sender"></param>
122     /// <param name="e"></param>
123     private void performBackupbackgroundWorker_ProgressChanged(object
sender, ProgressChangedEventArgs e)
124     {
125         int percentDone;
126
127         //Get the ProgressInfo passed back from Engine
128         ProgressInfo currentBackupProgress = (ProgressInfo) e.UserState;
129
130         if (currentBackupProgress.Phase != this.myLastPhase)
131         {
132             this.myCurrentPhaseStarted = DateTime.Now;
133             this.myLastPhase = currentBackupProgress.Phase;
134         }
135         if (currentBackupProgress.TotalCompletedOperations ==
currentBackupProgress.TotalOperations)
136         {
137             percentDone = 100;
138         }
139         else
140         {
141             percentDone = Convert.ToInt32((float) (100f * ((float)
currentBackupProgress.TotalCompletedOperations) / ((float)
currentBackupProgress.TotalOperations)));
142         }
143         ProgressBar progressBar = null;
144         String phaseTextToDisplay = "";
145         Label phaseLabelToUse = null;
146         Label phaseWhatsLeftLabelToUse = null;
147
148         int imageIndexToUse = -1;
149         //Deal with analysing progress bar, or upload progress bar
150         switch (currentBackupProgress.Phase)
151         {
152             case 0:
153                 progressBar = this.analysingProgressBar;
154                 phaseTextToDisplay = "Analysing";
155                 phaseLabelToUse = this.analysingLabel;
156                 phaseWhatsLeftLabelToUse = this.downloadingLeftLabel;
157                 imageIndexToUse = 0; //Use the analysing icon
158                 break;
159
160             case 1:
161                 progressBar = this.uploadingProgressBar;
162                 phaseTextToDisplay = "Uploading";
163                 phaseLabelToUse = this.uploadingLabel;
164                 phaseWhatsLeftLabelToUse = this.uploadingLeftLabel;
165                 imageIndexToUse = 1; //Use the file icon
166                 break;
167         }
168         //Set percentage
169         progressBar.Value = percentDone;
170         phaseLabelToUse.Text = String.Format("{0} ({1}%)",
phaseTextToDisplay, percentDone);
171         //If progress has been made
172         if (currentBackupProgress.TotalCompletedOperations > 0)
173         {
174             TimeSpan timeTakenSoFar = (TimeSpan) (DateTime.Now -
this.myCurrentPhaseStarted);
175             TimeSpan estimatedTimeLeft = new TimeSpan((timeTakenSoFar.Ticks
/ currentBackupProgress.TotalCompletedOperations) *
(currentBackupProgress.TotalOperations -
currentBackupProgress.TotalCompletedOperations));
176             //Set label to estimated time left
177             phaseWhatsLeftLabelToUse.Text = String.Format("{0}h:{1}m:{2}s
left", this.FormatAsDoubleDigits(estimatedTimeLeft.Hours),

```

```

this.FormatAsDoubleDigits(estimatedTimeLeft.Minutes),
this.FormatAsDoubleDigits(estimatedTimeLeft.Seconds));
178         CountdownInfo countDownInfo = new CountdownInfo();
179         //Label To decrement
180         countDownInfo.Label = phaseWhatsLeftLabelToUse;
181         //Time to decrement from
182         countDownInfo.Left = estimatedTimeLeft;
183         //The CountdownInfo to use
184         this.countdownTimer.Tag = countDownInfo;
185         this.countdownTimer.Enabled = true;
186     }
187     //Otherwise
188     else
189     {
190         phaseWhatsLeftLabelToUse.Text = "<pending> left";
191         //Don't decrement
192         this.countdownTimer.Enabled = false;
193     }
194     if (currentBackupProgress.CurrentItemName != "")
195     {
196         if (currentBackupProgress.CurrentItemOperations == 0)
197         {
198             currentBackupProgress.CurrentItemOperations = 1;
199             currentBackupProgress.CurrentItemCompletedOperations = 1;
200         }
201         if ((this.myLastProgressInfo == null) ||
((this.myLastProgressInfo != null) && (currentBackupProgress.CurrentItemName !=
this.myLastProgressInfo.CurrentItemName)))
202         {
203             ListViewItem currentFileItem = new ListViewItem();
204             //Set name
205             currentFileItem.Name =
currentBackupProgress.CurrentItemName;
206             //What to display (i.e., Analysing or Uploading)
207             currentFileItem.Text = phaseTextToDisplay;
208             //The image index to use
209             currentFileItem.ImageIndex = imageIndexToUse;
210             //Add filename
211             currentFileItem.SubItems.Add(new
ListViewItem.ListViewSubItem(currentFileItem, currentBackupProgress.CurrentItemName));
212             int percentage = Convert.ToInt32((float) (100f * ((float)
currentBackupProgress.CurrentItemCompletedOperations) / ((float)
currentBackupProgress.CurrentItemOperations)));
213             //Add percentage
214             currentFileItem.SubItems.Add(new
ListViewItem.ListViewSubItem(currentFileItem, percentage.ToString() + "%"));
215             this.detailsListView.Items.Add(currentFileItem);
216             //Scroll with the ListView to make sure current file is
always being displayed
217             this.detailsListView.EnsureVisible(currentFileItem.Index);
218         }
219         else
220         {
221             //Else set percentage of last ListView item
222             this.detailsListView.Items[this.detailsListView.Items.Count
- 1].SubItems[2].Text = Convert.ToInt32((float) (100f * ((float)
currentBackupProgress.CurrentItemCompletedOperations) / ((float)
currentBackupProgress.CurrentItemOperations))).ToString() + "%";
223         }
224     }
225     this.myLastProgressInfo = currentBackupProgress;
226 }
227
228     private void backgroundWorker_RunWorkerCompleted(object sender,
RunWorkerCompletedEventArgs e)
229     {
230         if (e.Error != null)
231         {
232             throw e.Error;
233         }
234         if (e.Cancelled)
235         {
236             //If "Cancelled", deal with deleting clutter on Amazon S3
237             this.deleteBackgroundWorker.RunWorkerAsync();
238         }
239         else
240         {

```

```

241         base.DialogResult = DialogResult.OK;
242         base.AllowClose = true;
243         base.Close();
244     }
245 }
246
247 private void cancelButton_Click(object sender, EventArgs e)
248 {
249     this.myCancelProgressPopup = new ProgressPopup("Cancelling...",
this);
250     this.myCancelProgressPopup.Show();
251     this.performBackupBackgroundWorker.CancelAsync();
252     this.cancelButton.Enabled = false;
253 }
254
255 //Decrements the count down timers every second
256 private void countdownTimer_Tick(object sender, EventArgs e)
257 {
258     CountdownInfo countDownInfo = (CountdownInfo)
this.countdownTimer.Tag;
259     countDownInfo.Left = countDownInfo.Left.Subtract(new TimeSpan(0, 0,
1));
260     if (countDownInfo.Left.Ticks < 0)
261     {
262         countDownInfo.Left = new TimeSpan((long) 0);
263     }
264     countDownInfo.Label.Text = String.Format("{0}h:{1}m:{2}s left",
this.FormatAsDoubleDigits(countDownInfo.Left.Hours),
this.FormatAsDoubleDigits(countDownInfo.Left.Minutes),
this.FormatAsDoubleDigits(countDownInfo.Left.Seconds));
265 }
266
267 private void deleteBackgroundWorker DoWork(object sender,
DoWorkEventArgs e)
268 {
269     Global.Engine.DeletePartialBackup(this.myPartialBackupInfo);
270 }
271
272 private void deleteBackgroundWorker_RunWorkerCompleted(object sender,
RunWorkerCompletedEventArgs e)
273 {
274     if (e.Error != null)
275     {
276         throw e.Error;
277     }
278     if (this.myCancelProgressPopup != null)
279     {
280         this.myCancelProgressPopup.Close();
281     }
282     base.DialogResult = DialogResult.Cancel;
283     base.AllowClose = true;
284     base.Close();
285 }
286
287 private void detailsListView_SelectedIndexChanged(object sender,
EventArgs e)
288 {
289     //Disallow selecting
290     foreach (ListViewItem item in this.detailsListView.SelectedItems)
291     {
292         item.Selected = false;
293     }
294 }
295
296 protected override void Dispose(bool disposing)
297 {
298     if (disposing && (this.components != null))
299     {
300         this.components.Dispose();
301     }
302     base.Dispose(disposing);
303 }
304
305 private String FormatAsDoubleDigits(int aDigit)
306 {
307     if (aDigit < 10)
308     {

```

```

309         return ("0" + aDigit.ToString());
310     }
311     return aDigit.ToString();
312 }
313
314 private void hideButton_Click(object sender, EventArgs e)
315 {
316     base.Close();
317 }
318
319 private void InitializeComponent()
320 {
321     //OMITTED FOR CODE SAMPLES
322 }
323
324 private void PerformBackupDialog_Load(object sender, EventArgs e)
325 {
326     this.mainGroupBox.Text = String.Format(this.mainGroupBox.Text,
327     this.myBackupItem.Name);
328 }
329
330 private void PerformBackupDialog_Shown(object sender, EventArgs e)
331 {
332     this.performBackupBackgroundWorker.RunWorkerAsync();
333 }
334
335 public S3OnTheGo.Backup.Model.BackupItem BackupItem
336 {
337     get
338     {
339         return this.myBackupItem;
340     }
341     set
342     {
343         this.myBackupItem = value;
344     }
345 }
346
347 public String Error
348 {
349     get
350     {
351         return this.myError;
352     }
353 }
354 }

```

The heart of a BackupItem:

```

6     [Serializable]
7     public class BackupItem : ModelObject
8     {
9         private const String MACHINE_SEPARATOR = ":";
10        private System.DayOfWeek myDayOfWeek;
11        private String myFolderPath;
12        private ScheduleFrequency myFrequency;
13        private DateTime myLastBackup;
14        private String myName;
15        private DateTime myTime;
16        private S3OnTheGo.Backup.Model.WeekOfMonth myWeekOfMonth;
17
18        public BackupItem(String anID, String aName) : base(anID)
19        {
20            this.myName = ":";
21            this.Name = aName;
22        }
23
24        private String GetDisplayName()
25        {
26            int index = this.myName.IndexOf(":") + ":".Length;
27            return this.myName.Substring(index, this.myName.Length - index);
28        }

```

```

29
30     public String GetLastBackupUpString()
31     {
32         if (this.myLastBackup == DateTime.MinValue)
33         {
34             return "Never";
35         }
36         TimeSpan timeSpan =
DateTime.Now.Subtract(this.myLastBackup.ToLocalTime());
37         if (timeSpan.Days > 30)
38         {
39             return ((timeSpan.Days / 30) + " months ago");
40         }
41         if (timeSpan.Days > 7)
42         {
43             return ((timeSpan.Days / 7) + " weeks ago");
44         }
45         if (timeSpan.Days > 1)
46         {
47             return (timeSpan.Days + " days ago");
48         }
49         DateTime time3 = this.myLastBackup.ToLocalTime();
50         if (DateTime.Now.DayOfYear > time3.DayOfYear)
51         {
52             return "Yesterday";
53         }
54         return "Today";
55     }
56
57     public String GetMachineName()
58     {
59         int index = this.myName.IndexOf(":");
60         return this.myName.Substring(0, index);
61     }
62
63     public String GetScheduleString()
64     {
65         switch (this.myFrequency)
66         {
67             case ScheduleFrequency.None:
68                 return "None";
69
70             case ScheduleFrequency.Daily:
71                 return String.Format("Daily ({0})",
this.myTime.ToShortTimeString());
72
73             case ScheduleFrequency.Weekly:
74                 return String.Format("Weekly ({0} at {1})",
this.GetShortDayString(this.myDayOfWeek), this.myTime.ToShortTimeString());
75
76             case ScheduleFrequency.Monthly:
77                 return String.Format("Monthly ({0} {1} at {2})",
this.GetShortWeekOfMonthString(this.myWeekOfMonth),
this.GetShortDayString(this.myDayOfWeek), this.myTime.ToShortTimeString());
78         }
79         return null;
80     }
81
82     private String GetShortDayString(System.DayOfWeek aDayOfWeek)
83     {
84         switch (aDayOfWeek)
85         {
86             case System.DayOfWeek.Sunday:
87                 return "Sunday";
88
89             case System.DayOfWeek.Monday:
90                 return "Monday";
91
92             case System.DayOfWeek.Tuesday:
93                 return "Tuesday";
94
95             case System.DayOfWeek.Wednesday:
96                 return "Wednesday";
97
98             case System.DayOfWeek.Thursday:
99                 return "Thursday";
100

```

```

101         case System.DayOfWeek.Friday:
102             return "Friday";
103
104         case System.DayOfWeek.Saturday:
105             return "Saturday";
106     }
107     return null;
108 }
109
110 private String
175 GetShortWeekOfMonthString (S3OnTheGo.Backup.Model.WeekOfMonth aWeekOfMonth)
111 {
112     switch (aWeekOfMonth)
113     {
114         case S3OnTheGo.Backup.Model.WeekOfMonth.First:
115             return "1st";
116
117         case S3OnTheGo.Backup.Model.WeekOfMonth.Second:
118             return "2nd";
119
120         case S3OnTheGo.Backup.Model.WeekOfMonth.Third:
121             return "3rd";
122
123         case S3OnTheGo.Backup.Model.WeekOfMonth.Last:
124             return "last";
125     }
126     return null;
127 }
128
129 public System.DayOfWeek DayOfWeek
130 {
131     get
132     {
133         return this.myDayOfWeek;
134     }
135     set
136     {
137         this.myDayOfWeek = value;
138     }
139 }
140
141 public String FolderPath
142 {
143     get
144     {
145         return this.myFolderPath;
146     }
147     set
148     {
149         this.myFolderPath = value;
150     }
151 }
152
153 public ScheduleFrequency Frequency
154 {
155     get
156     {
157         return this.myFrequency;
158     }
159     set
160     {
161         this.myFrequency = value;
162     }
163 }
164
165 public bool IsLocal
166 {
167     get
168     {
169         return (this.GetMachineName() != Global.LocalMachineName);
170     }
171 }
172
173 public DateTime LastBackup
174 {
175     get
176     {

```

```

177         return this.myLastBackup;
178     }
179     set
180     {
181         this.myLastBackup = value;
182     }
183 }
184
185 public String Name
186 {
187     get
188     {
189         return this.GetDisplayName();
190     }
191     set
192     {
193         this.myName = Global.LocalMachineName + ":" + value;
194     }
195 }
196
197 public DateTime Time
198 {
199     get
200     {
201         return this.myTime;
202     }
203     set
204     {
205         this.myTime = value;
206     }
207 }
208
209 public S3OnTheGo.Backup.Model.WeekOfMonth WeekOfMonth
210 {
211     get
212     {
213         return this.myWeekOfMonth;
214     }
215     set
216     {
217         this.myWeekOfMonth = value;
218     }
219 }
220 }

```

How's how keys and metadata are made:

```

482     private String MakeAllBackupItemsPrefix()
483     {
484         return String.Format("{0}.BackupItems.", Global.User.ID);
485     }
486
487     private String MakeBackupItemFilesPrefix(String aBackupItemID)
488     {
489         return String.Format("{0}.BackupItem{1}.file", Global.User.ID,
aBackupItemID);
490     }
491
492     private String MakeFilePartKey(BackupFile aFile)
493     {
494         return this.MakeFilePartKey(aFile, 0);
495     }
496
497     private String MakeFilePartKey(BackupFile aFile, int aPart)
498     {
499         return String.Format("{0}.BackupItem{1}.file{2}.{3}", new object[] {
Global.User.ID, aFile.BackupItemID, aFile.ID, aPart });
500     }
501
502     private MetadataEntry[] MakeFilePartMetadata(BackupFile aFile)
503     {
504         {
505             MetadataEntry numOfPartsEntry = new MetadataEntry();
506             numOfPartsEntry.Name = "numparts";

```

```

507         numOfPartsEntry.Value = aFile.NumParts.ToString();
508         MetadataEntry totalSizeEntry = new MetadataEntry();
509         totalSizeEntry.Name = "totalsize";
510         totalSizeEntry.Value = aFile.Size.ToString();
511         return new MetadataEntry[] { numOfPartsEntry, totalSizeEntry };
512     }
513
514     private String MakeLocalBackupItemsKey()
515     {
516         return String.Format("{0}.BackupItems.{1}", Global.User.ID,
Global.LocalMachineName);
517     }
518
519     private String MakeSnapshotKey(Snapshot aSnapshot)
520     {
521         return String.Format("{0}.BackupItem{1}.snapshot{2}",
Global.User.ID, aSnapshot.BackupItemID, aSnapshot.ID);
522     }

```