netgenium

Console applications

Framework NET Genium

Content

1 Basic information

- Console applications are used to call their own C# program code, either on the server side or on the client side.
- Console applications are used in cases where
  - you need to run one-time service activities over the database or
  - these activities need to be performed at regular intervals.
- You can set console applications to run regularly in the Windows Task Scheduler, or you can customize console applications to run as a Windows service.
- Console applications are also used to prepare and debug external functions. A detailed description of external functions is given in the separate manual “External functions”.
- Console applications connect to the NET Genium database
  - directly from an application server that is located on the same network as the database server (the application server and the database server are often the same computer), or
  - using the web services that are part of every NET Genium.
- Direct connection to the NET Genium database is the recommended method because it is an order of magnitude faster than connection via web services, and it supports database transactions that cannot be used at all in the case of web services.
- Debugging or editing source codes that use a direct connection to the NET Genium database requires a database ideally located on the local computer or on the same network. This is often not possible either for security reasons or because the database is too large, which is not realistic to back up on the database server and transfer to the local computer. In this case, it is recommended that you debug the source code using APIs in conjunction with web services.
- Console applications use objects and methods from the “NETGeniumConnection.dll” library, which is stored in the “NETGenium\bin” or “N:\NetGenium\Projekty\NetGenium\References” directory. “NETGeniumConnection.dll” is a library with basic functions for working with databases and file attachments.
- Console applications are developed through a separate project in the application “Visual Studio 2015” and higher, respectively. programming the source code of this project, and then compiling the project into an “exe” file.

2 Creating a console application

- In the first step, you need to create a new console application project in Visual Studio using the following steps, and add a reference to the "NETGeniumConnection.dll" library:
  - Start Visual Studio
  - From the menu on the main bar, select "File / New / Project…" (Ctrl+Shift+N)
    - Project type: Console App (.NET Framework)
    - Project name: ConsoleApp1
    - Location: optional project location
    - Solution: Create new solution
    - Place solution and project in the same directory: Yes
    - Create directory for solution: No (Visual Studio 2015)
    - Framework: .NET Framework 4.5.2
  - Right-click on "References", select "Add Reference…", and select the path to the file "References\NETGeniumConnection.dll" on the computer disk
  - Choose "Debug" compilation mode
    - "Debug" mode generates "exe" and "pdb" files by default
    - Thanks to the "pdb" file, errors and interrupts in console applications are easily detected, because the "Stack Trace" of each error also includes the file name and the line number on which the interrupt occurred.
    - The "Release" mode is only recommended for the final version of the tuned source code in the console application. By default, "Release" mode only generates an "exe" file.
  - Start project – select "Debug" / "Start Debugging" (F5) from the menu on the main bar

2.1 Direct connection to the NET Genium database

- The direct connection to the database uses the “DbConnection” class located in the “NETGenium” namespace. This class behaves analogously to the usual class “DbConnection” from the namespace “System.Data” or “SqlConnection” from the namespace “System.Data.SqlClient”.
- The “DbConnection” class is universal, and is used for both Firebird and MSSQL database servers.

```
using NETGenium;
using System;

namespace ConsoleApplication1
{
  class Program
  {
    static void Main(string[] args)
    {
      NETGeniumConsole console = new NETGeniumConsole();

      using (DbConnection conn = new
DbConnection(@"driver=firebird;datasource=localhost;user=SYSDBA;password=masterkey;database=C:
\Firebird\netgenium.fdb;charset=WIN1250;collation=WIN_CZ"))
      // using (DbConnection conn = new
DbConnection("server=(local);Trusted_Connection=true;database=netgenium"))
      using (DbCommand cmd = new DbCommand(conn))
      {
        conn.Open();

        conn.RootPath = "C:\\inetpub\\wwwroot\\netgenium";
        DateTime now = DateTime.Now;

        Console.WriteLine("Connected to database");
        Console.WriteLine();
        Console.WriteLine(conn.User.FormatTimeSpan(DateTime.Now - now));
      }

      console.Exit();
    }
  }
}
```

2.2 Connect to the NET Genium database using web services

- An alternative to connecting directly to the database using the "DbConnection" object is to use the "NETGeniumConnection" object. It is a class located in the "System" namespace of the "NETGeniumConnection.dll" library, which allows you to connect to the NET Genium database using web services.
- Web services as well as the "NETGeniumConnection.dll" library form a single API, and are part of every NET Genium.
- Most of the methods used to work with data in the "NETGeniumConnection.dll" library have implemented both the method variant specified for the "DbConnection" object and for "NETGeniumConnection".
- After creating an instance of the "NETGeniumConnection" object, it is necessary to log in with the name and password of a user with administrator rights. This login is subject to the IP address restrictions defined in the NET Genium settings. Each such login draws licenses to log in to NET Genium, so it is important to log out at the end of the work.
- The "NETGeniumConnection" class is universal, and is used for both Firebird and MSSQL database servers.
- Communication with the remote NET Genium should always be secured with an SSL certificate, and therefore it is necessary to set the correct and server-supported way of communication using "ServicePointManager.SecurityProtocol".
    - By default, console applications use the outdated HTTPS protocol "SecurityProtocolType.Tls", which is disabled on properly secured servers.
        - Enabling/disabling HTTPS protocols takes place on the server by writing to the registers.
        - Each NET Genium has a program stored in the "Config/Tools" directory
            - "SSL.reg" for setting the recommended security configuration of HTTPS protocols a
            - "SSL-ie6.reg" for configuration settings that allow access to NET Genium even obsolete devices such as Internet Explorer version 6, old tablets, mobile phones, etc.
            - A detailed description of the "SSL.reg" and "SSL-ie6.reg" programs is given in the separate "Utilities" manual.
    - The current recommended method of communication in 2020 is via HTTPS protocol "SecurityProtocolType.Tls12".
    - The only exception when we can use insecure communication is the connection to the local NET Genium via the address "http://localhost/...".

```
using NETGenium;
using System;
using System.Net;

namespace ConsoleApplication1
{
  class Program
  {
    static void Main(string[] args)
    {
      ServicePointManager.SecurityProtocol = SecurityProtocolType.Tls12;

      NETGeniumConsole console = new NETGeniumConsole();

      NETGeniumConnection conn = new NETGeniumConnection("http://localhost/netgenium");
      conn.Login("NETGeniumConnection", "heslo");

      using (DbCommand cmd = new DbCommand(conn))
      {
        DateTime now = DateTime.Now;

        Console.WriteLine("Connected to database");
        Console.WriteLine();
        Console.WriteLine(conn.User.FormatTimeSpan(DateTime.Now - now));
      }

      conn.Logout();

      console.Exit();
    }
  }
}
```

3 Reading data from the database

3.1 Retrieve records from an SQL query into a DataTable object

```
// using NETGenium;
// using System;
// using System.Data;

DataTable data = Data.Get("SELECT * FROM sholiday WHERE date_ > " + conn.Format(new
DateTime(DateTime.Today.Year, 1, 1)), conn);
Console.WriteLine(conn.User.FormatDataTableText(data));

foreach (DataRow row in data.Rows)
{
  Console.WriteLine(row["id"]);
}
```

3.2 Saves the DataTable object to disk and reloads it from disk

```
// using NETGenium;
// using System;
// using System.Data;

DataTable data = Data.Get("SELECT * FROM sholiday WHERE date_ > " + conn.Format(new
DateTime(DateTime.Today.Year, 1, 1)), conn);

// HTML
Files.Write(Config.RootPath + "sholiday.html", conn.User.FormatDataTable(data));

// TXT
Files.Write(Config.RootPath + "sholiday.txt", conn.User.FormatDataTableText(data));

// XML
DataSet ds = new DataSet();
ds.Tables.Add(data);
ds.WriteXml(Config.RootPath + "sholiday.xml", XmlWriteMode.WriteSchema);

ds = new DataSet();
ds.ReadXml(Config.RootPath + "sholiday.xml", XmlReadMode.ReadSchema);
data = ds.Tables[0];

// JSON
Files.Write(Config.RootPath + "sholiday.json", ParserJSon.ToString(data));
data = ParserJSon.ToDataTable(Files.Read(Config.RootPath + "sholiday.json"));
```

3.3 Retrieve records from an SQL query into a DbRow object

```
// using NETGenium;
// using System;

DbRow row = new DbRow("SELECT * FROM sholiday WHERE date_ > " + conn.Format(new
DateTime(DateTime.Today.Year, 1, 1)), conn);
if (row.Read())
{
  Console.WriteLine(row["id"]);
  Console.WriteLine(row.Report());
}
```

3.4 Value parsing

```
// using NETGenium;
// using System;

DbRow row = new DbRow("SELECT * FROM sholiday WHERE date_ > " + conn.Format(new
DateTime(DateTime.Today.Year, 1, 1)), conn);
if (row.Read())
{
  int id = (int)row["id"];
  Console.WriteLine("id: " + id);

  int pid = Parser.ToInt32(row["pid"]);
  Console.WriteLine("pid: " + pid);

  double _pid = Parser.ToDouble(row["pid"]);
  Console.WriteLine("pid: " + _pid);

  string name = row["name"].ToString();
  Console.WriteLine("name: " + name);

  DateTime date = Parser.ToDateTime(row["date_"]);
  Console.WriteLine("date: " + conn.User.FormatDateTime(date));
}
```

3.5 Indexing of retrieved records by primary key

```
// using NETGenium;
// using System;
// using System.Collections.Generic;
// using System.Data;

DataTable data = Data.Get("SELECT * FROM sholiday", conn);
Dictionary<int, DataRow> dictionary = Data.DictionaryInt32(data);

int id = 1;
if (dictionary.ContainsKey(id))
{
  DataRow row = dictionary[id];
  Console.WriteLine(row["id"]);
}
```

3.6 Indexing of retrieved nested records according to the foreign key “pid”

```
// using NETGenium;
// using System;
// using System.Collections.Generic;
// using System.Data;

DataTable data = Data.Get("SELECT * FROM sholiday", conn);
Dictionary<int, List<DataRow>> dictionary = Data.DictionaryInt32(data, "pid", true);

int pid = 0;
if (dictionary.ContainsKey(pid))
{
  List<DataRow> rows = dictionary[pid];
  Console.WriteLine(rows.Count + " rows");
}
```

3.7 Sequential reading of records

```
// using NETGenium;
// using System;

using (DbCommand cmd = new DbCommand("SELECT id, date_ FROM sholiday WHERE date_ > " +
conn.Format(new DateTime(DateTime.Today.Year, 1, 1)), conn))
using (DbDataReader dr = cmd.ExecuteReader())
  while (dr.Read())
  {
    Console.WriteLine(dr["id"] + ": " + dr["date_"]);
  }
```

3.8 Sequential reading of records into the dictionary

```
// using NETGenium;
// using System;
// using System.Collections.Generic;

Dictionary<int, DateTime> dictionary = new Dictionary<int, DateTime>();

using (DbCommand cmd = new DbCommand("SELECT id, date_ FROM sholiday WHERE date_ > " +
conn.Format(new DateTime(DateTime.Today.Year, 1, 1)), conn))
using (DbDataReader dr = cmd.ExecuteReader())
  while (dr.Read())
  {
    dictionary.Add((int)dr["id"], (DateTime)dr["date_"]);
  }
```

3.9 Sequential reading of records into the sheet

```
// using NETGenium;
// using System;
// using System.Collections.Generic;

private class Item
{
  public int ID;
  public DateTime Date;

  public Item(DbDataReader dr)
  {
    ID = (int)dr["id"];
    Date = (DateTime)dr["date_"];
  }
}

List<Item> items = new List<Item>();

using (DbCommand cmd = new DbCommand("SELECT id, date_ FROM sholiday WHERE date_ > " +
conn.Format(new DateTime(DateTime.Today.Year, 1, 1)), conn))
using (DbDataReader dr = cmd.ExecuteReader())
  while (dr.Read())
  {
    items.Add(new Item(dr));
  }
```

4 Writing data to the database

4.1 INSERT INTO – create a new record in the database

4.1.1 DataSaver – create one record

```
// using NETGenium;
// using System;

DataSaver ds = new DataSaver("sholiday", 0, cmd);
ds.Add("name", "New year");
ds.Add("date_", new DateTime(DateTime.Today.Year, 1, 1));
ds.Execute();

Console.WriteLine(ds.Report());
```

4.1.2 DataSaver – creation of two records in a separate transaction

```
// using NETGenium;
// using System;

cmd.Transaction = conn.BeginTransaction();

try
{
  DataSaver ds = new DataSaver("sholiday", 0, cmd);
  ds.Add("name", "New year");
  ds.Add("date_", new DateTime(DateTime.Today.Year, 1, 1));
  ds.Execute();

  Console.WriteLine(ds.Report());

  ds = new DataSaver("sholiday", 0, cmd);
  ds.Add("name", "New year");
  ds.Add("date_", new DateTime(DateTime.Today.Year + 1, 1, 1));
  ds.Execute();

  Console.WriteLine(ds.Report());

  cmd.Transaction.Commit();
}
catch (Exception ex)
{
  cmd.Transaction.Rollback();
  throw ex;
}
```

4.1.3 DataSaver – create one record by copying another record

```
// using NETGenium;
// using System;

int id = 1;

DbRow row = new DbRow("SELECT * FROM sholiday WHERE id = " + id, conn);
if (row.Read())
{
  // row["ng_enteredby"] = conn.User.LoginName;
  // row["ng_enteredwhen"] = DateTime.Now;
  // row["ng_changedby"] = DBNull.Value;
  // row["ng_changedwhen"] = DBNull.Value;

  DataSaver ds = new DataSaver("sholiday", 0, cmd);
  ds.Add(row);
  ds.Execute();

  Console.WriteLine(ds.Report());
}
```

4.1.4 DataSaverSynchro – creation of one record, including creation of history record and ensuring record synchronization

```
// using NETGenium;
// using System;

DataSaverSynchro ds = new DataSaverSynchro("sholiday", 0, conn);
ds.Add("name", "New year");
ds.Add("date_", new DateTime(DateTime.Today.Year, 1, 1));
ds.Save(cmd);

Console.WriteLine(ds.Report());
```

4.2 UPDATE – editing an existing record in the database

4.2.1 DataSaver – editing a record whose ID is retrieved from the database

```
// using NETGenium;
// using System;

int id = Data.ExecuteScalar2("SELECT id FROM sholiday WHERE name = " + conn.Format("New Year")
+ " AND date_ = " + conn.Format(new DateTime(DateTime.Today.Year, 1, 1)), conn);
if (id != 0)
{
  DataSaver ds = new DataSaver("sholiday", id, cmd);
  ds.Add("name", "New year - TEST");
  ds.Execute();

  Console.WriteLine(ds.Report());
}
```

4.2.2 DataSaver – editing a record whose ID is stored in a variable; if the record does not exist, it will be created

```
// using NETGenium;
// using System;

int id = 1;

DataSaver ds = new DataSaver("sholiday", id, true, cmd);
ds.Add("name", "New year - TEST");
ds.Execute();

Console.WriteLine(ds.Report());
```

4.2.3 DataSaverSynchro – editing a record whose ID is retrieved from the database, including creating a history record and ensuring record synchronization

```
// using NETGenium;
// using System;

int id = Data.ExecuteScalar2("SELECT id FROM sholiday WHERE name = " + conn.Format("New year")
+ " AND date_ = " + conn.Format(new DateTime(DateTime.Today.Year, 1, 1)), conn);
if (id != 0)
{
  DataSaverSynchro ds = new DataSaverSynchro("sholiday", id, conn);
  ds.Add("name", "New year - TEST");
  ds.Save(cmd);

  Console.WriteLine(ds.Report());
}
```

4.2.4 DataSaverSynchro – editing a record whose ID is stored in a variable, including creating a history record and ensuring record synchronization; if the record does not exist, it will be created

```
// using NETGenium;
// using System;

int id = 1;

DataSaverSynchro ds = new DataSaverSynchro("sholiday", id, true, conn);
ds.Add("name", "New year - TEST");
ds.Save(cmd);

Console.WriteLine(ds.Report());
```

4.3 Data synchronization of two database tables according to a single key

```
// using NETGenium;
// using System;
// using System.Collections.Generic;
// using System.Data;

string table1 = "ng_book1", table2 = "ng_book2", key = "id";

DataTable data1 = Data.Get("SELECT * FROM " + table1, conn), data2 = Data.Get("SELECT * FROM "
+ table2, conn);
Dictionary<object, DataRow> dictionary = Data.Dictionary(data2);

foreach (DataRow row1 in data1.Rows)
{
  DataRow row2 = Data.DataRow(dictionary, row1[key]);
  if (row2 == null)
  {
    DataSaver ds = new DataSaver(table2, 0, cmd);

    foreach (DataColumn c in data2.Columns)
      if (c != "id")
      {
        ds.Add(c, row1[c]);
      }

    ds.Execute();
  }
  else
  {
    DataSaver ds = new DataSaver(table2, (int)row2["id"], cmd);

    foreach (DataColumn c in data2.Columns)
      if (c != "id" && row2[c].ToString() != row1[c].ToString())
      {
        ds.Add(c, row1[c]);
      }

    if (!ds.Empty)
    {
      ds.Execute();
    }
  }
}
```

5 Deleting data from the database

5.1 DELETE FROM – delete a record from the database

```
// using NETGenium;
// using System;

int id = Data.ExecuteScalar2("SELECT id FROM sholiday WHERE name = " + conn.Format("New year -
TEST") + " AND date_ = " + conn.Format(new DateTime(DateTime.Today.Year, 1, 1)), conn);
if (id != 0)
{
  cmd.CommandText = "DELETE FROM sholiday WHERE id = " + id;
  cmd.ExecuteNonQuery();
}
```

5.2 DataSaverSynchro – deleting a record from the database, including creating a history record and ensuring record synchronization

```
// using NETGenium;
// using System;

int id = Data.ExecuteScalar2("SELECT id FROM sholiday WHERE name = " + conn.Format("New year -
TEST") + " AND date_ = " + conn.Format(new DateTime(DateTime.Today.Year, 1, 1)), conn);
if (id != 0)
{
  DataSaverSynchro ds = new DataSaverSynchro("sholiday", id, conn);
  ds.Delete(cmd);
}
```

6 File attachments

6.1 Creating a file attachment

```
// using NETGenium;
// using System;
// using System.IO;

string temp = Path.GetTempFileName();
Files.Write(temp, "abc");

int file = Attachment.Add("test.txt", temp, conn);
File.Delete(temp);
Console.WriteLine(file);
```

6.2 Retrieve the contents of a file attachment using a DbConnection object

```
// using NETGenium;
// using System;
// using System.IO;

string temp = Path.GetTempFileName();
Files.Write(temp, "abc");

int file = Attachment.Add("test.txt", temp, conn);
File.Delete(temp);
Console.WriteLine(file);

string path = Attachment.FilePath(file, conn);
if (File.Exists(path))
{
  string s = Files.Read(path);
  Console.WriteLine(s);
}
```

6.3 Retrieve the contents of a file attachment using the NETGeniumConnection object

```
// using NETGenium;
// using System;
// using System.IO;

string temp = Path.GetTempFileName();
Files.Write(temp, "abc");

int file = Attachment.Add("test.txt", temp, conn);
File.Delete(temp);
Console.WriteLine(file);

// Attachment
Attachment = conn.GetAttachment(id);
if (attachment != null)
{
  Console.WriteLine(attachment.Name + ": " + attachment.ContentBytes.Length + " bytes");
}

// byte[]
byte[] buffer = conn.GetAttachmentAsBytes(id);
if (buffer != null)
{
  Console.WriteLine(buffer.Length + " bytes");
}

// string
string s = conn.GetAttachmentAsString(id, Encoding.UTF8);
if (s != null)
{
  Console.WriteLine(s);
}
```

7 E-mails

7.1 Sending an e-mail message

```
// using NETGenium;
// using System;
// using System.Net.Mail;

string html = NETGenium.Email.Message.Container("<b>Hello</b>");

MailMessage message = new MailMessage();
message.From = new MailAddress("@");
message.To.Add(new MailAddress("@"));
message.Subject = "";
message.AlternateViews.Add(Config.CreateTextAlternateView(Html.ToText(html)));
message.AlternateViews.Add(Config.CreateHtmlAlternateView(html, conn));

Config.SendMessage(message, conn);
```

8 Logging

8.1 Logging to disk in the “Logs” directory

```
// using NETGenium;
// using System;
// using System.IO;

try
{
  throw new NullReferenceException("args");
}
catch (Exception ex)
{
  L.E("MyFirstFunction", ex);
  // L.LogError("MyFirstFunction", ex);
  // L.Error("MyFirstFunction", ex);

  L.W("MyFirstFunction", ex);
  // L.LogWarning("MyFirstFunction", ex);
  // L.Warning("MyFirstFunction", ex);

  L.N("MyFirstFunction", ex);
  // L.LogNotice("MyFirstFunction", ex);
  // L.Notice("MyFirstFunction", ex);
}
```

8.2 General disk logging

```
// using NETGenium;
// using System;

Files.Write(Config.RootPath + "test.log", "TEST");

try
{
  Files.WriteLine(Config.RootPath + "test.log", DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss") +
Convert.ToChar(160) + "TEST");
}
catch { }

Files.WriteLineTryCatch(Config.RootPath + "test.log", DateTime.Now.ToString("yyyy-MM-dd
HH:mm:ss") + Convert.ToChar(160) + "TEST");
```

9 Services

9.1 Basic information

- Console applications can be installed as a Windows service, so they then run in the background of the operating system and run at regular intervals.
- Console applications in service mode ("Services") must have a defined service name and a unique GUID of the console application project.
    - The service name is used when installing the service, and under this name the service can be found (stopped, started, restarted) in the list of Windows operating system services or in the Task Manager.
    - The project GUID is used to safely start only one instance of the service at a time so that multiple instances of the service do not start in parallel (when copying an existing console application project, you must set a new project GUID so that the two services do not conflict with each other).
- The services use configuration files from which they read the connection settings to the database, the address of the outgoing mail server, the start and end time of the technological pause during which the service is not working, etc.
- The services do not connect to the database themselves, and use a unified system to call the main source code that occurs in a common console application within the "static void Main (string[] args)" method.
- Services do not have access to attached disk drives, only hard or network drives. Disks mounted using the "mount" command must be accessed using a network path, such as "\\192.168.0.1\D\NETGenium".
- Services require administrator privileges to be assigned using the manifest, otherwise basic operations such as backup, read or write to disk, etc. will fail.

9.2 Create a service envelope and edit the “Program.cs” file

- In the first step, you need to move the console application logic located in the “static void Main (string[] args)” method to the new “Service.cs” file.
- You must remove the instance of the “DbConnection”, “DbCommand”, and NET Genium directory settings from this source code.
- The minimum source code of the service shows the following example, the namespace and the name of the service are set to “CRMService” for clarity:

```
using NETGenium;
using System;

namespace CRMService
{
  public class Service : ServiceTemplate, IServiceTemplate
  {
    public override string ServiceName { get { return "CRMService"; } }

    public Service()
    {
    }

    protected override void HandleProcess(string[] args, Config config, DbCommand cmd)
    {
      DbConnection conn = cmd.Connection;
      DateTime now = DateTime.Now;
    }
  }
}
```

- In the second step, it is necessary to change the content of the "Program.cs" file and set the service startup interval.
- From now on, the console application will not be able to start normally using the "Debug" / "Start Debugging" command (F5), so it is important to schedule this step until the console application is tuned and stable.

```
using NETGenium;

namespace CRMService
{
    class Program
    {
        static void Main(string[] args)
        {
            if (false)
            {
                new Service().Run(args);
                return;
            }

            new Service().Process(args,
            new ServiceTemplateInstaller.InstallOptions()
            {
                Interval = 5
            });
        }
    }
}
```

- At any time in the future, the console application can be retroactively modified so that it can be started in the usual way using the "Debug" / "Start Debugging" command (F5), and further debugging or testing of the console application is possible.

```
using NETGenium;

namespace CRMService
{
    class Program
    {
        static void Main(string[] args)
        {
            if (true)
            {
                new Service().Run(args);
                return;
            }

            new Service().Process(args,
            new ServiceTemplateInstaller.InstallOptions()
            {
                Interval = 5
            });
        }
```

```
        }
}
```

9.3 Manifesto

- In the third step, you must grant the console application administrator privileges using a manifest file:
    - Start Visual Studio and open the console application project
    - Right-click on the project name in the Solution Explorer, select “Add / New Item…”, type “manifest” in the search field, find the item “Application Manifest File”, and select “Add”
    - In the contents of the manifest file, find “<requestedExecutionLevel level="asInvoker" uiAccess="false" />” and change to “<requestedExecutionLevel level="requireAdministrator" uiAccess="false" />”

9.4 CRMService example

- The following example demonstrates a simple “CRMService” service that downloads technology page content, backs up incoming e-mails to the “POP3Downloads” directory, and prints e-mail details to the console application window and the “Logs” directory to disk.
- Contents of the “Service.cs” file:

```
using NETGenium;
using System;
using System.Collections.Generic;
using System.Data;
using System.IO;

namespace CRMService
{
  public class Service : ServiceTemplate, IServiceTemplate
  {
    public override string ServiceName { get { return "CRMService"; } }

    private string dir, backupdir;

    public Service()
    {
      AddRequiredConfigKey("pop3Email");
      AddRequiredConfigKey("pop3Url");
      AddRequiredConfigKey("pop3Name");
      AddRequiredConfigKey("pop3Password");

      dir = Config.RootPath + "Temp\\";
      if (!Directory.Exists(dir))
      {
          Directory.CreateDirectory(dir);
      }

      backupdir = Config.RootPath + "POP3Downloads\\";
      if (!Directory.Exists(backupdir))
      {
        Directory.CreateDirectory(backupdir);
      }
    }

    protected override void HandleProcess(string[] args, Config config, DbCommand cmd)
    {
      Files.DeleteDirectoriesYYYYMMDD(backupdir, DateTime.Today.AddMonths(-6));
      Files.DeleteLogs(Config.RootPath, DateTime.Today.AddYears(-1));

      DbConnection conn = cmd.Connection;
      DateTime now = DateTime.Now;
```

```
      NETGenium.Email.Pop3 pop3 = new NETGenium.Email.Pop3(0, config["pop3Url"],
config["pop3Name"], config["pop3Password"]);
      try
      {
        pop3.Open();

        for (int index = 0; index < pop3.Count; index++)
        {
          foreach (string file in Directory.GetFiles(dir))
          {
            File.Delete(file);
          }

          string backupdir2 = backupdir + now.ToString("yyyy-MM-dd") + "\\";
          if (!Directory.Exists(backupdir2))
          {
            Directory.CreateDirectory(backupdir2);
          }

          string eml = backupdir2 + Guid.NewGuid().ToString().ToUpper() + ".eml";

          NETGenium.Email.Message message = pop3.ReadMessage_SaveEmlAndAttachments(index, eml,
dir);
          Email.Eval(config, message, eml, true, "ID " + index, dir, ServiceName, now, cmd,
conn);

          // pop3.DeleteMessage(index); // Delete messages from the POP3 server only after
debugging the service
          if (index == 99) break;
        }

        pop3.Close();
      }
      catch (Exception ex)
      {
        try { pop3.Close(); }
        catch { }

        if (ServiceLogger.ImportantError(ex))
        {
          L.E("Download e-mails from POP3", ex);
        }
        else
        {
          L.W("Download e-mails from POP3", ex);
        }

        return;
      }
    }
  }
}
```

- The logic of processing the incoming e-mail message is solved in a separate file “Email.cs”.
- Contents of the “Email.cs” file:

```
using NETGenium;
using NETGenium.Email;
using System;

namespace CRMService
{
  internal class Email
  {
    public static bool Eval(Config config, Message message, string eml, bool pop3message,
string name, string dir, string serviceName, DateTime now, DbCommand cmd, DbConnection conn)
    {
      L.N(name);

      foreach (string file in Directory.GetFiles(dir))
      {
        L.N(Path.GetFileName(file));
      }

      L.N(message.DateOriginal.ToString());
      L.N(message.From.EmailAddress);
      L.N(message.To);
      L.N(message.Subject);
      L.N(message.Text);

      return true;
    }
  }
}
```

9.5 Configuration filey services

- Services can contain two types of configuration files:
  - The main configuration file (for example, "CRMService.exe.config"), the name of which must be based on the application name (in this case "CRMService.exe").
  - XML configuration files that are used when the service serves multiple databases at once (for example, "netgenium1.xml", "netgenium2.xml", etc.).
- The service source code runs for each configuration file separately. Acceptable combinations include:
  - CRMService.exe.config
  - netgenium1.xml, netgenium2.xml, …
  - CRMService.exe.config, netgenium1.xml, netgenium2.xml, …
- The configuration files must be located in the same directory as "CRMService.exe". If we are in the debugging phase of the service in Visual Studio, it is not necessary to create the configuration file manually, because each console application project automatically contains the default configuration file "App.config". This file is copied to the "Debug" or "Release" directory (depending on the selected compilation mode) to a file named "abc.exe.config" each time the project is compiled. In our case, the contents of the configuration file "App.config" are copied to the file "CRMService.exe.config" when the project is compiled.
- Example configuration file:

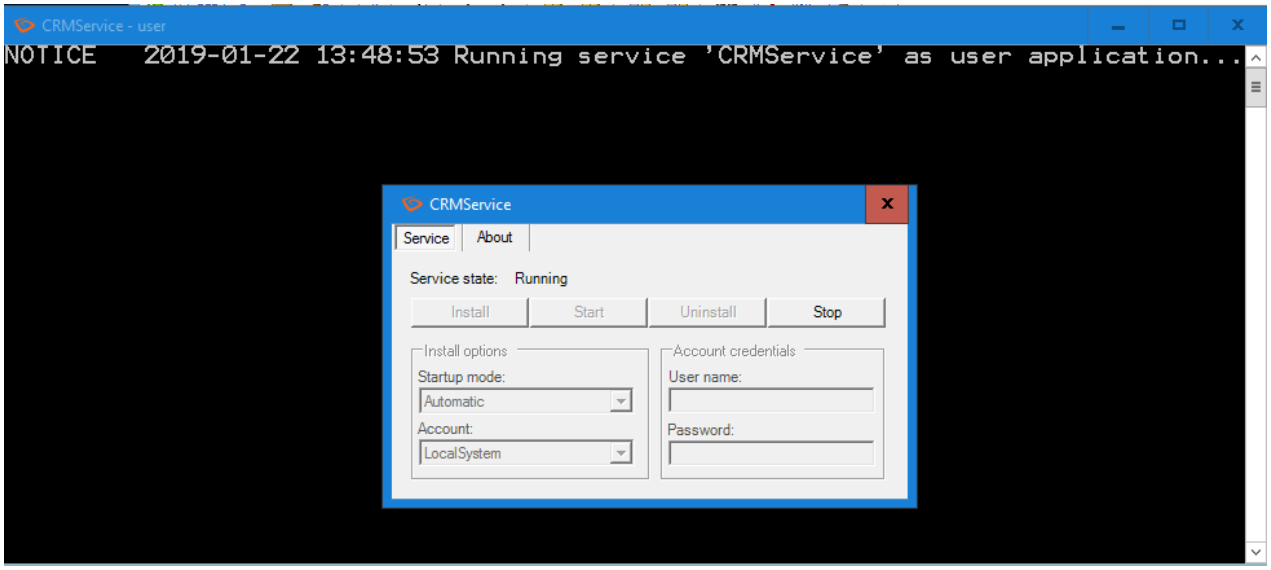
```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>

    <add key="rootPath" value="C:\inetpub\wwwroot\netgenium" />

    <add key="stopFrom" value="00:00" />
    <add key="stopTo" value="06:00" />

    <add key="smtpServer" value="localhost" />

    <add key="errorFrom" value="crmservice@firma.cz" />
    <add key="errorTo" value="support@firma.cz" />

    <add key="pop3Url" value="pop3.firma.cz" />
    <add key="pop3Email" value="crm@firma.cz" />
    <add key="pop3Name" value="crm@firma.cz" />
    <add key="pop3Password" value="pop3heslo" />

  </appSettings>
</configuration>
```

- **rootPath** – absolute disk path to the NET Genium directory
- **stopFrom** – the start time of the technological pause in the format "HH: mm", during which the service is not working
- **stopTo** – end time of the technological pause in the format "HH: mm", during which the service is not working
- **smtpServer** – the address of the outgoing mail server

- **errorFrom** – e-mail address from which notifications of possible errors in the service are sent
- **errorTo** – e-mail address to which notifications of possible errors in the service are sent
- **pop3Url** – incoming mail server address (for SSL security it is possible to use the syntax “server name: port number”)
- **pop3Email** – e-mail address of the technology box
- **pop3Name** – login name to the technology box
- **pop3Password** – password for the technology box

9.6 Service installation

- The file “service-name.exe – user.lnk” is used to install the service, which is created automatically by starting the console application “service-name.exe”.
- After running the “service-name.exe – user.lnk” file, it is important to check the settings of the account under which the service is to run. The default setting is “LocalSystem”, which ensures safe service startup without possible future complications with service authorization.
- The service is installed by clicking on the “Install” button.



- The service is uninstalled by clicking on the “Uninstall” button.

9.7 Exchange of service program files

- Once the service is installed, it is not possible to change the service's program files because they are constantly protected by the operating system.
- Before exchanging program files, it is necessary to stop the service using the “Stop” button, and close the service settings dialog using the cross icon.
- Subsequently, the service program files can be overwritten and the service restarted using the “Start” button.