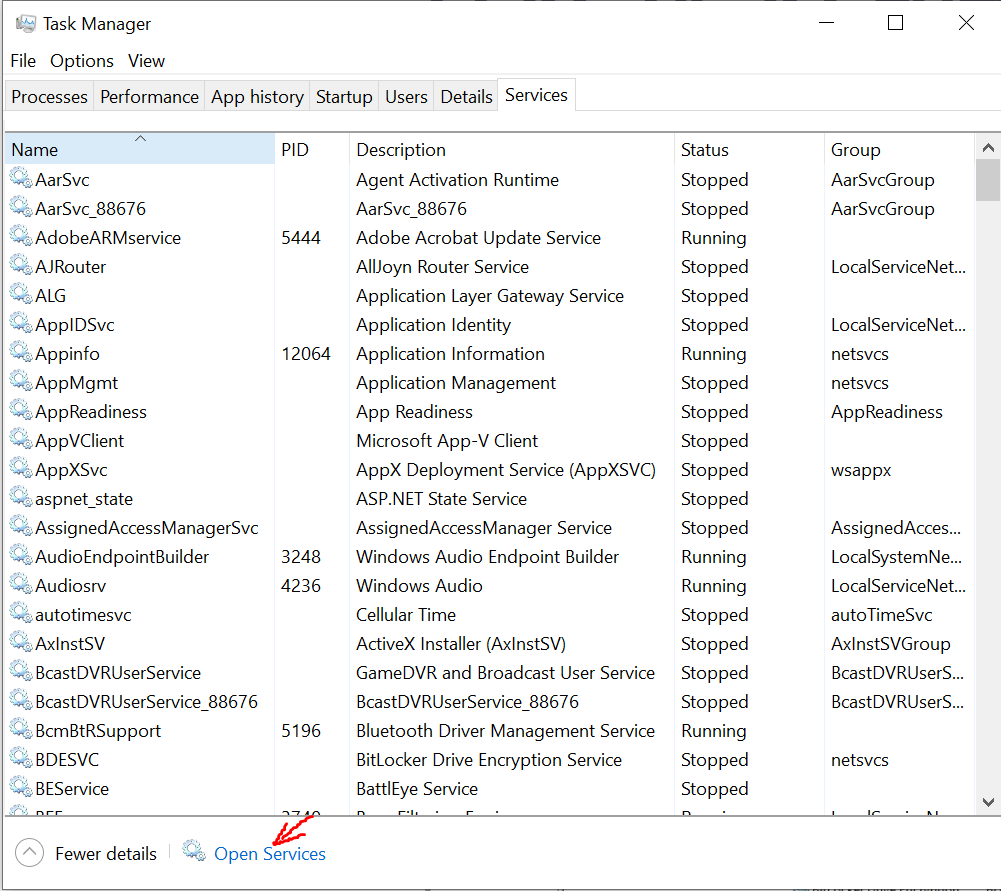
**Usługi Windows**

Usługa Windows, to proces, który jest wykonywany w obrębie systemu operacyjnego Windows. Nie posiada on własnego interfejsu użytkownika,  
nie podejmuje żadnej interakcji z użytkownikiem, działa w tle i wykonuje różne zadania.  
Działa od razu po uruchomieniu systemu operacyjnego.

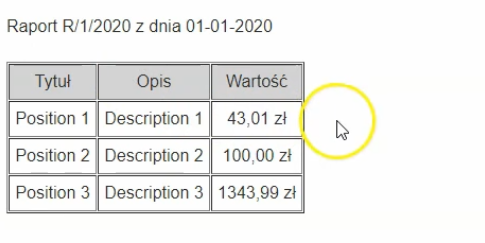
Ponieważ usługa nie ma żadnego interfejsu, to jedynym, w którym możemy ją  
znaleźć jest okno Usług Windows.

Aby przejść do usług wpisujemy: **services.msc**,  
lub z poziomu menedżera zadań w zakładce Services



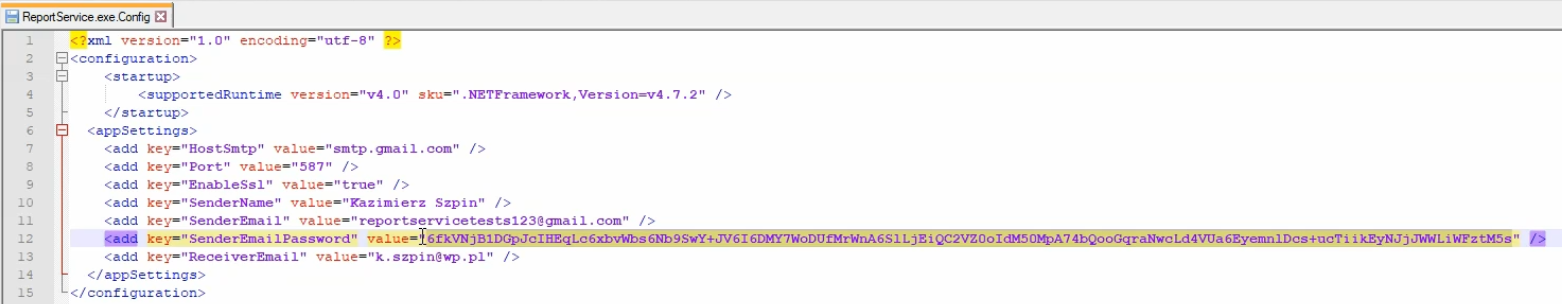
**Co będzie robiła usługa**

Usługa, którą utworzymy będzie wysyłała codziennie o godzinie 8.00 raport sprzedaży z poprzedniego dnia w postaci maila.  
Mail zawiera dane przedstawione w postaci tabelki w html’u.

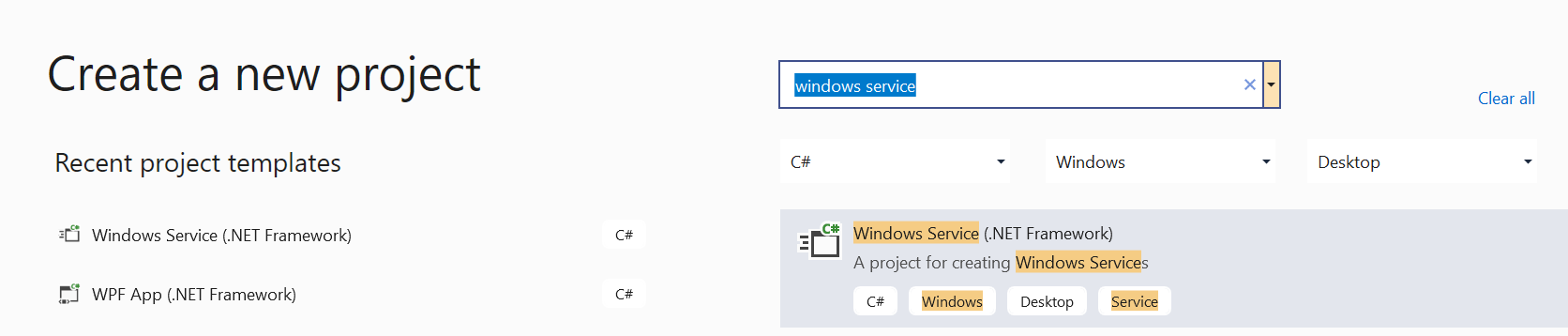


Oprócz tego, jeżeli w bazie danych pojawi się błąd to również wyślemy podobnego maila.  
Aplikacja będzie również logowała do pliku różne informacje związana z działaniem aplikacji, takie jak start usługi, zatrzymanie usługi, błędy itd.

Aplikacja będzie również szyfrowała dane wrażliwe. Takie jak hasło do maila.



**Tworzenie Usługi Windows w Visual Studio**

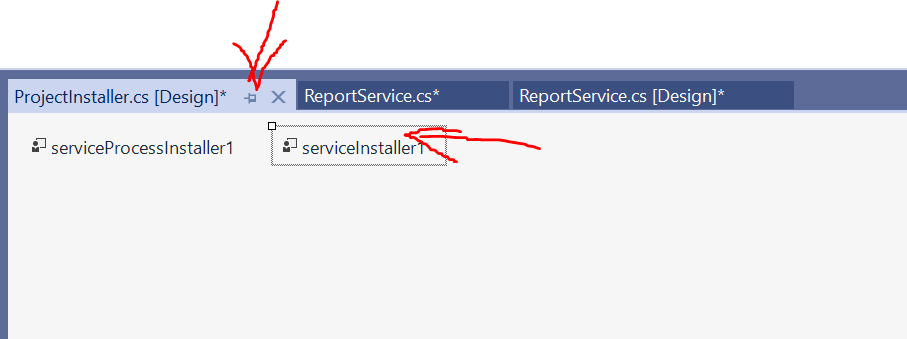


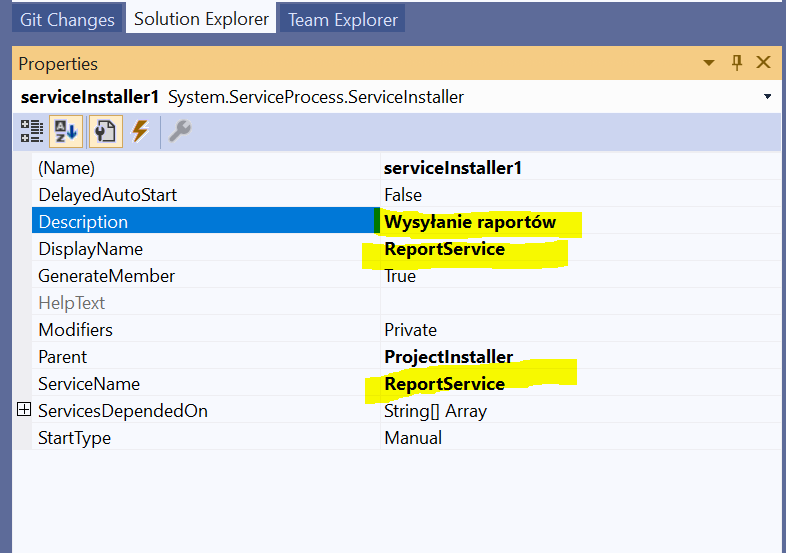
Zostaje utworzony szkielet aplikacji  
Klasę Program.cs a w niej metodę Main(), która startuje naszą usługę.

Następnie klasa Service.cs. Jeżeli chcemy zobaczyć jej kod to  
wciskamy **F7** lub z menu kontekstowego wybieramy: **View Code**.   
Metoda ma konstruktor, w nim inicjalizację komponentów i dwie   
metody: OnStart() wywoływana przy starcie usługi i OnStop() wywoływana przy zatrzymaniu usługi. Zmieniamy nazwę tej klasy z Service.cs na ReportService.cs

**Dodajemy Installera**

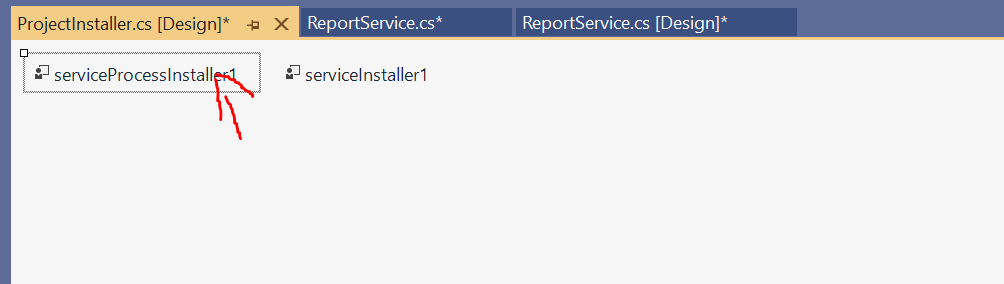
Po dwukrotnym kliknięci na Service.cs pojawia się okno designera.  
Na oknie Designer’a wybieramy z menu Add Installer, zostaje dodany plik   
ProjectInstaller.cs. Zmieniamy nazwy ServiceName, DisplayName i Description.

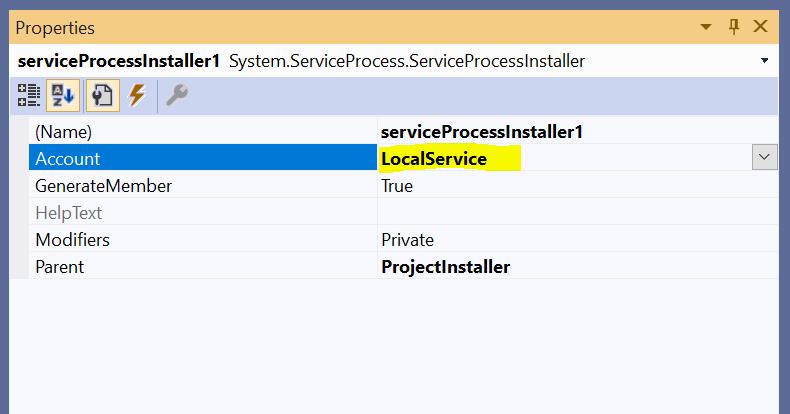




Dane te będą wyświetlane w oknie gdzie możesz zarządzać usługami.

Klikamy jeszcze na ServiceProcessInstaller i zmieniamy przy opcji Account  
na LocalService aby usługa uruchamiała się dla wszystkich użytkowników.





Dodajemy trochę kodu ułatwiającego uruchomienie usługi

Modyfikujemy metodę Main w klasie program

|  |
| --- |
| static void Main(string[] args)  {  ServiceBase[] ServicesToRun;  ServicesToRun = new ServiceBase[]  {  new ReportService()  };  ServiceBase.Run(ServicesToRun);  } |

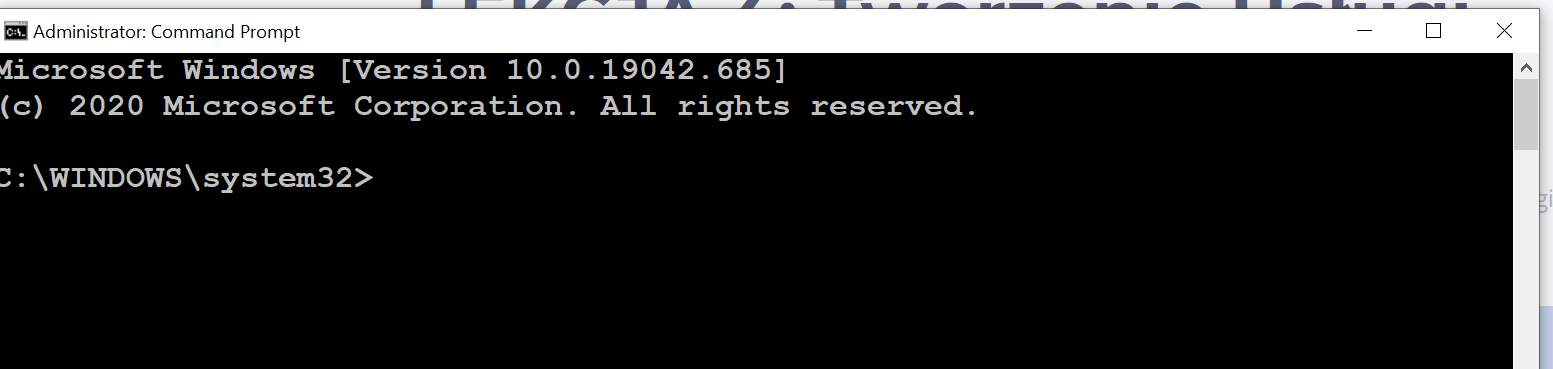
|  |
| --- |
| ServiceBase[] ServicesToRun;  ServicesToRun = new ServiceBase[]  {  new ReportService()  };  ServiceBase.Run(ServicesToRun); |

|  |
| --- |
| if (Environment.UserInteractive)  {  var parameter = string.Concat(args);  switch (parameter)  {  case "--install":  ManagedInstallerClass.InstallHelper(new[]  {Assembly.GetExecutingAssembly().Location});  break;  case "--uninstall":  ManagedInstallerClass.InstallHelper(new[]  {"/u",Assembly.GetExecutingAssembly().Location});  break;  default:  break;  }  }  else  {  ServiceBase[] ServicesToRun;  ServicesToRun = new ServiceBase[]  {  new ReportService()  };  ServiceBase.Run(ServicesToRun);  } |

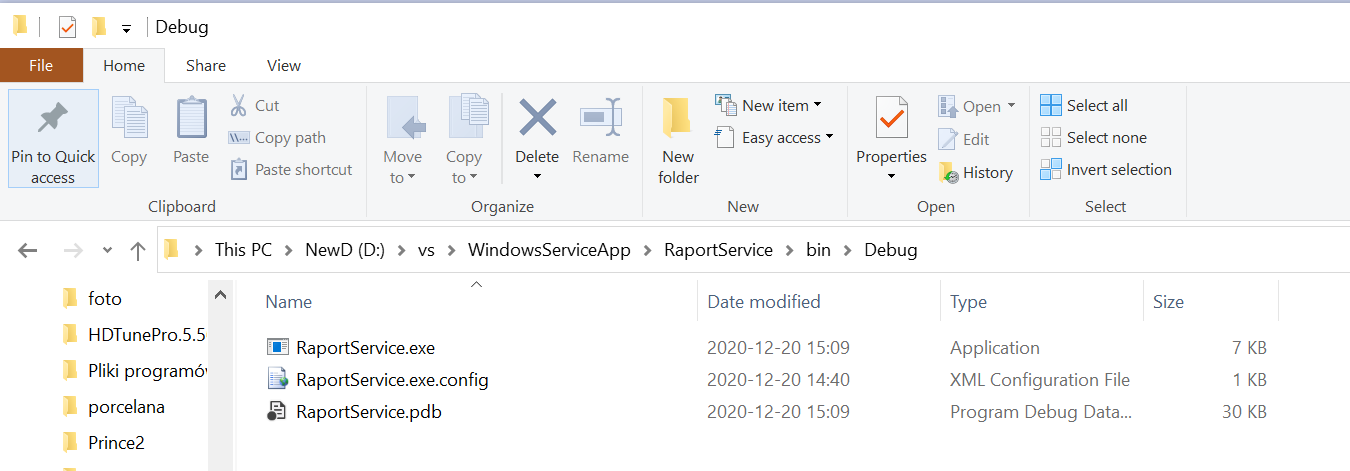
Po powyższych zmianach kompilujemy program.

**Instalowanie usługi w Windows**

Uruchamiamy cmd z uprawnieniami administratora

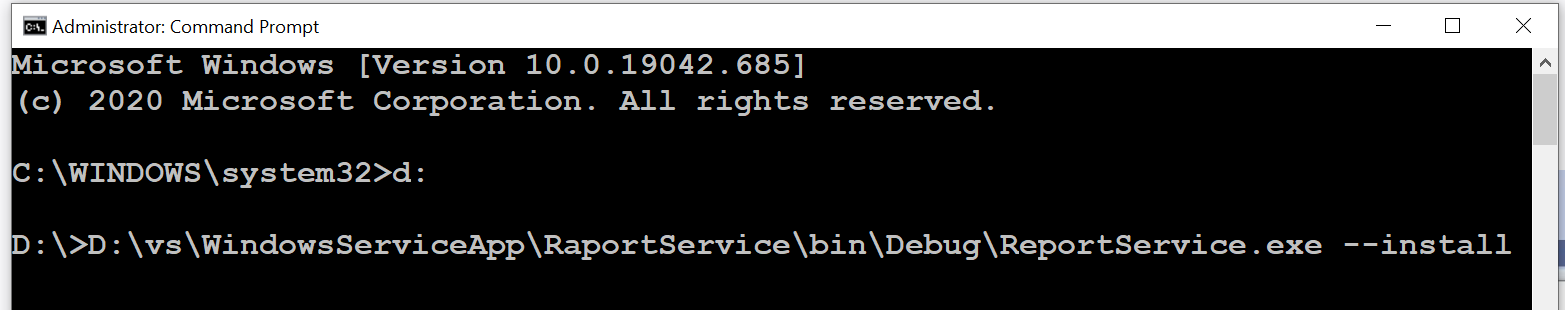


Na projekcie open in file explorer,  
przechodzimy do bin, następnie do debug



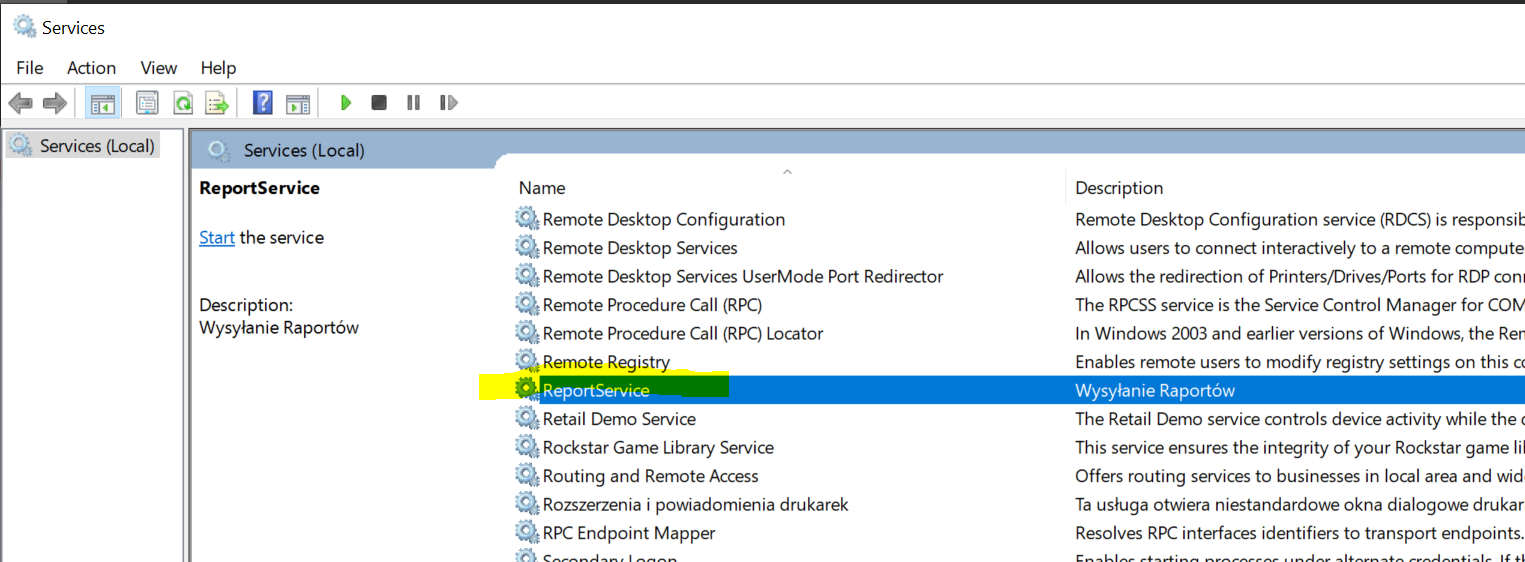
Wklejamy ścieżkę, dopisujemy nazwę naszego exeka i dopisujemy –install

D:\vs\WindowsServiceApp\ReportService\bin\Debug\ReportService.exe --install



Powstaje plik : ReportService.InstallLog

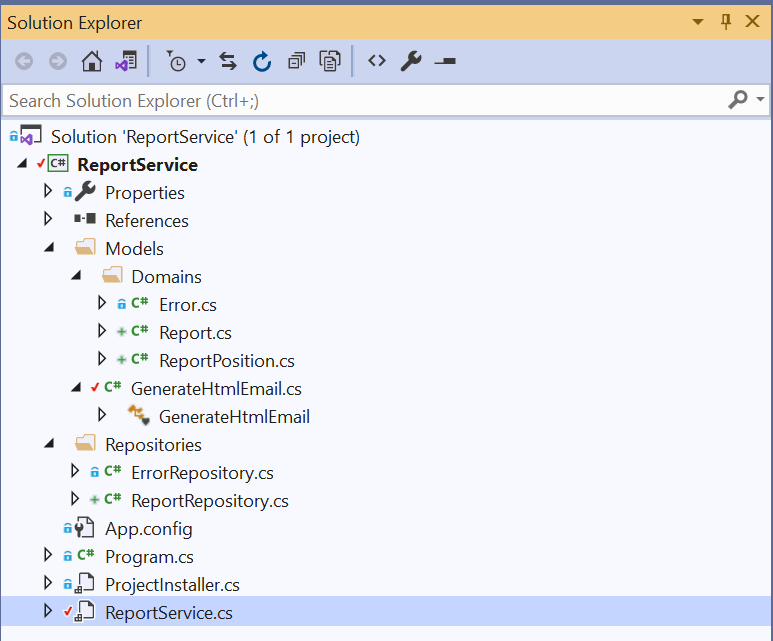
Uruchamiamy services.msc



Aby odinstalować usługę wpisujemy w konsoli:

D:\vs\WindowsServiceApp\ReportService\bin\Debug\ReportService.exe –uninstall

**Implementacja Logiki Biznesowej**



Modyfikujemy naszą klasę ReportService

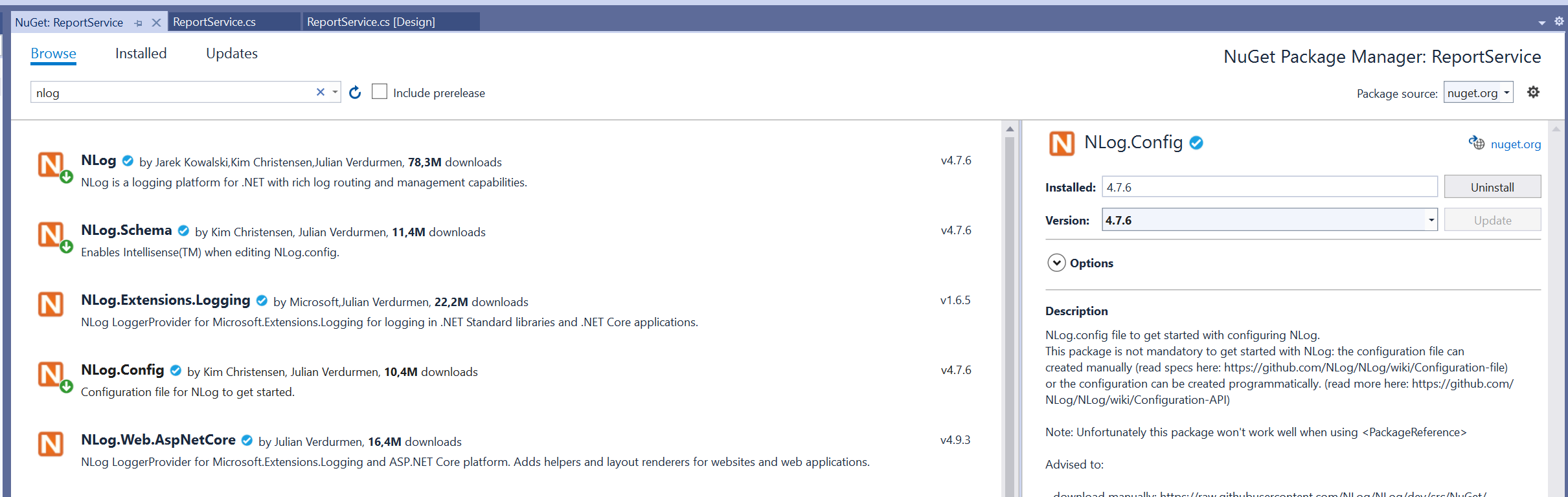
|  |
| --- |
| public partial class ReportService : ServiceBase  {  private const int \_sendHour = 8;    private const int \_intervalInMinutes = 30;  private Timer \_timer = new Timer(\_intervalInMinutes \* 1000);//ważne namespace: System.Timers    private ErrorRepository \_errorRepository = new ErrorRepository();  private ReportRepository \_reportRepository = new ReportRepository();  /// <summary>  /// Metoda, która będzie wywoływana co określony interval  /// </summary>  public ReportService()  {  InitializeComponent();  }  protected override void OnStart(string[] args)  {  \_timer.Elapsed += DoWork;  \_timer.Start();  }  private void DoWork(object sender, ElapsedEventArgs e)  {  SendError();  SendReport();  } |

Dodajemy klasę **Timer**.   
W metodzie OnStart() definiujemy co jaki czas będzie wykonywana metoda przypisana do tego Timera, oraz uruchamiamy ten timer.

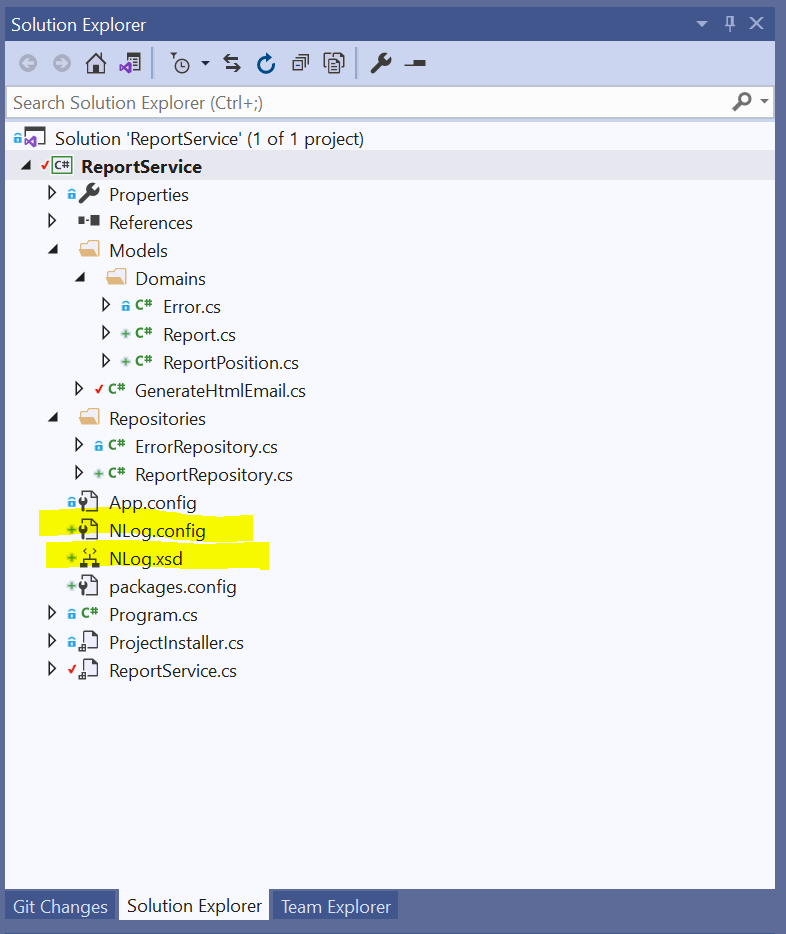
Logger – korzystamy z gotowych rozwiązań, jednego z wielu dostępnych frameworków np. z NLog’a.

<https://nlog-project.org/>  
https://github.com/NLog/NLog/wiki/Tutorial

Przez managera nuget instalujemy NLog NLog.Schema i NLog.Config.  
Jeżeli zaczniemy od NLog.Config wszystkie pozostałe zostaną zainstalowane.

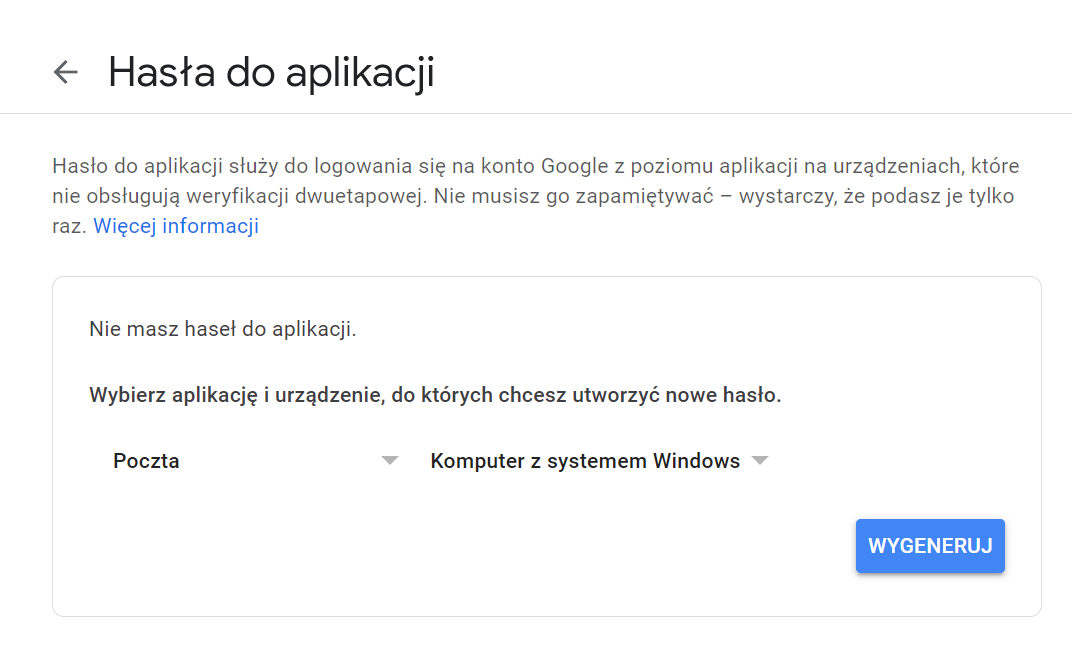


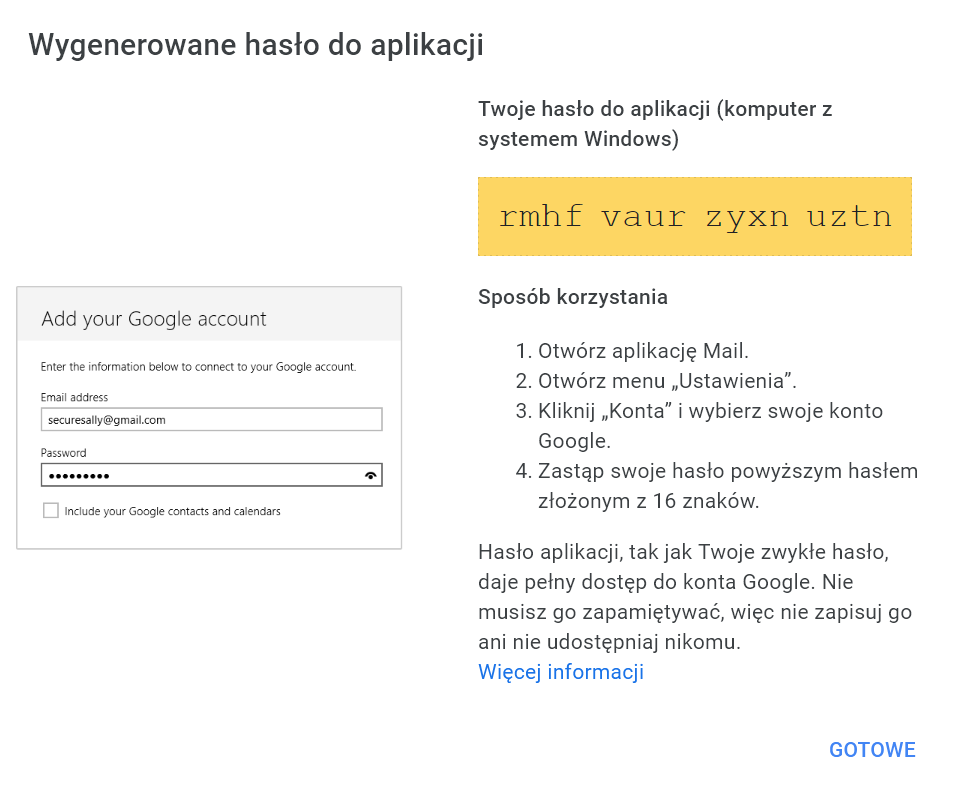
Do projektu zostały dodane referencje, w packages.config odwołania do NLog’a oraz dodatkowe pliki konfiguracyjne.



Konfiguracja poczty

Poczta Gmail, włączenie podwójnej weryfikacji





**Implementacja wysyłania maili**

Dodajemy klasę Email z wszystkimi niezbędnymi polami

|  |
| --- |
| public class Email  {  private SmtpClient \_smtp;  private MailMessage \_mail;  private string \_hostSmtp;  private bool \_enableSsl;  private int \_port;  private string \_senderEmail;  private string \_senderEmailPassword;  private string \_senderName;  public Email(EmailParams emailParams)  {  \_hostSmtp = emailParams.HostSmtp;  \_enableSsl = emailParams.EnableSsl;  \_port = emailParams.Port;  \_senderEmail = emailParams.SenderEmail;  \_senderEmailPassword = emailParams.SenderEmailPassword;  \_senderName = emailParams.SenderName;  }  } |

inicjujemy wszystkie pola po przez konstruktor  
aby nie przekazywać aż 6 parametrów  
definiujemy klasę EmailParams

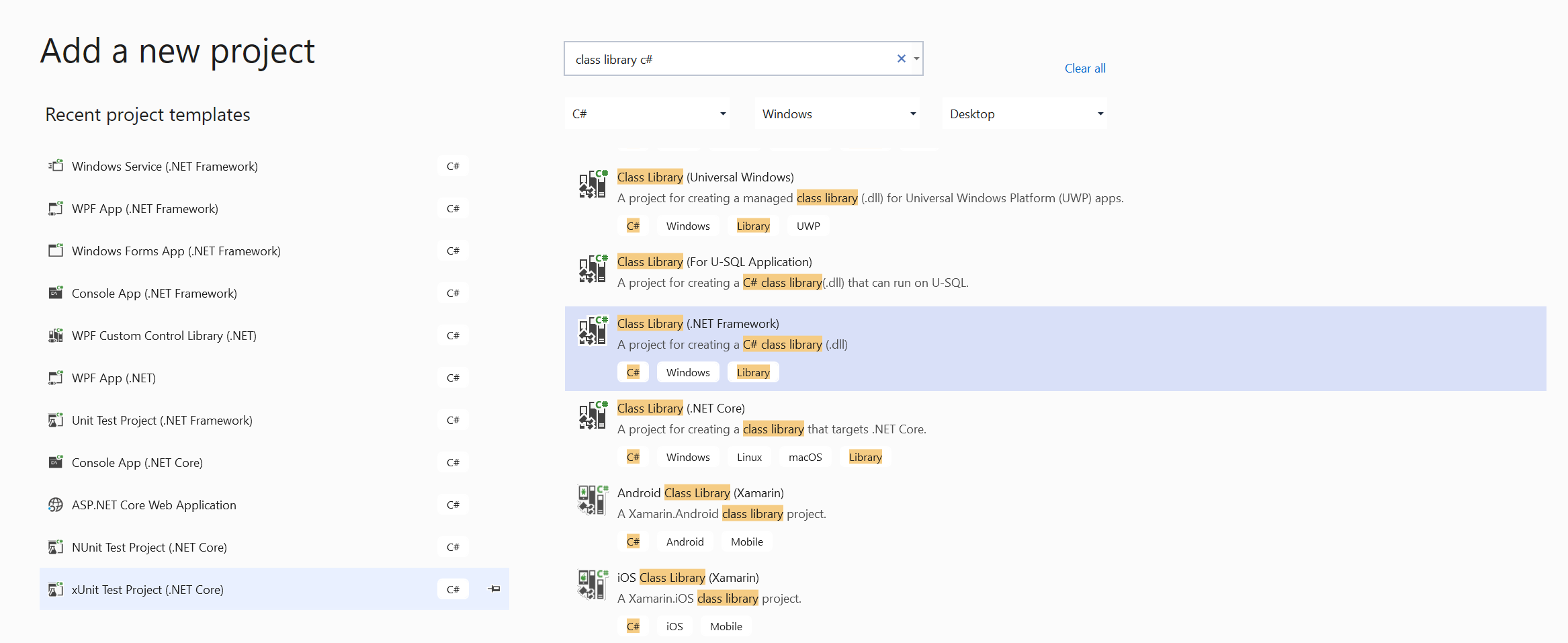
|  |
| --- |
| public class EmailParams  {  public string HostSmtp { get; set; }  public bool EnableSsl { get; set; }  public int Port { get; set; }  public string SenderEmail { get; set; }  public string SenderEmailPassword { get; set; }  public string SenderName { get; set; }  } |

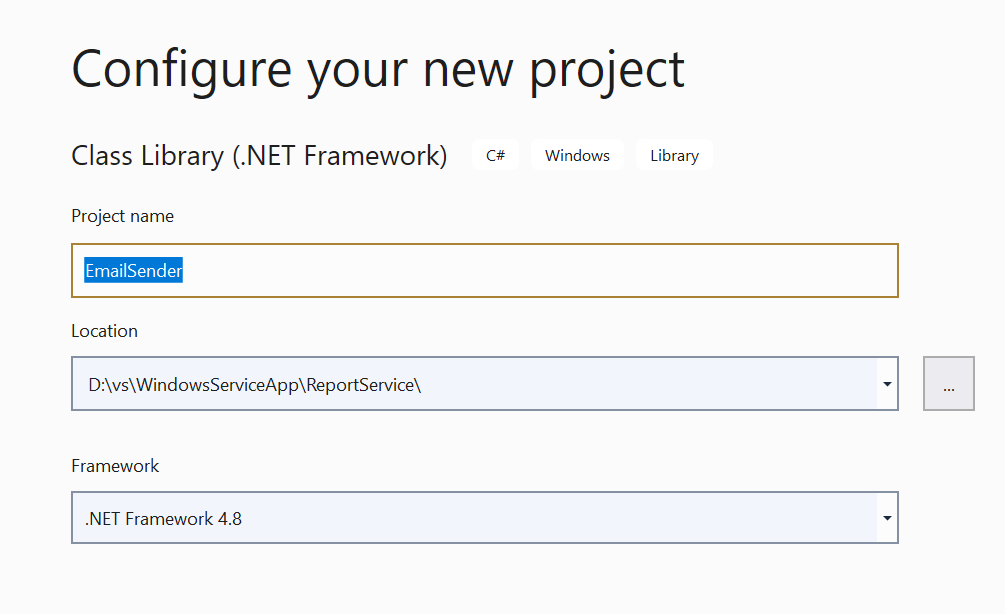
i obiekt tej klasy będziemy przekazywać do konstruktora

c# remove tags from string

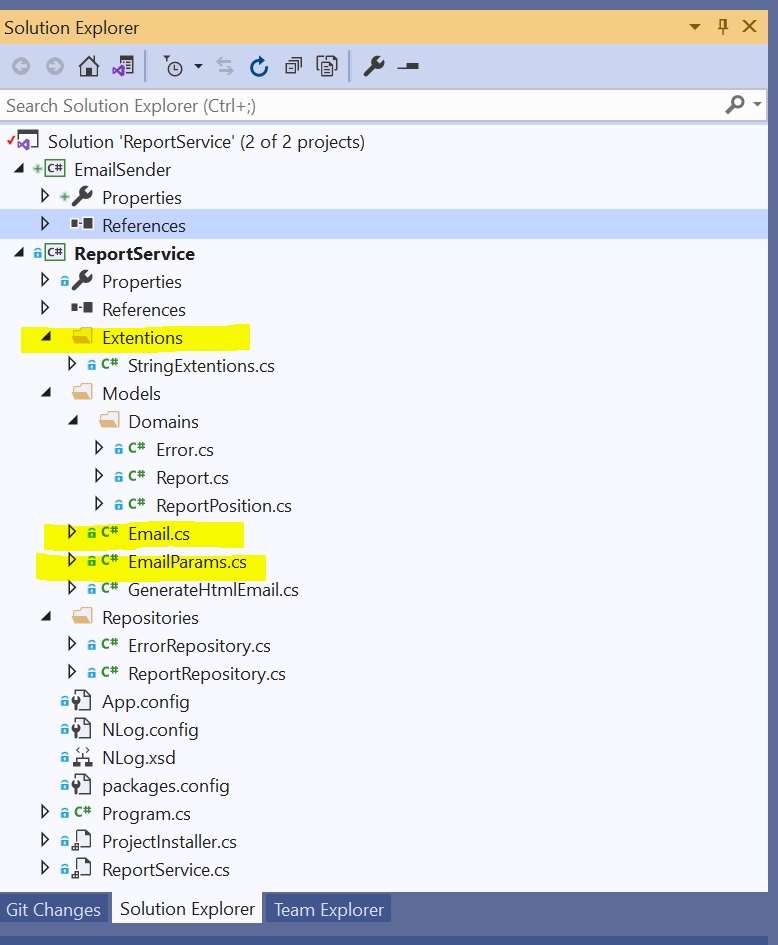
**Reużywanie kodu**

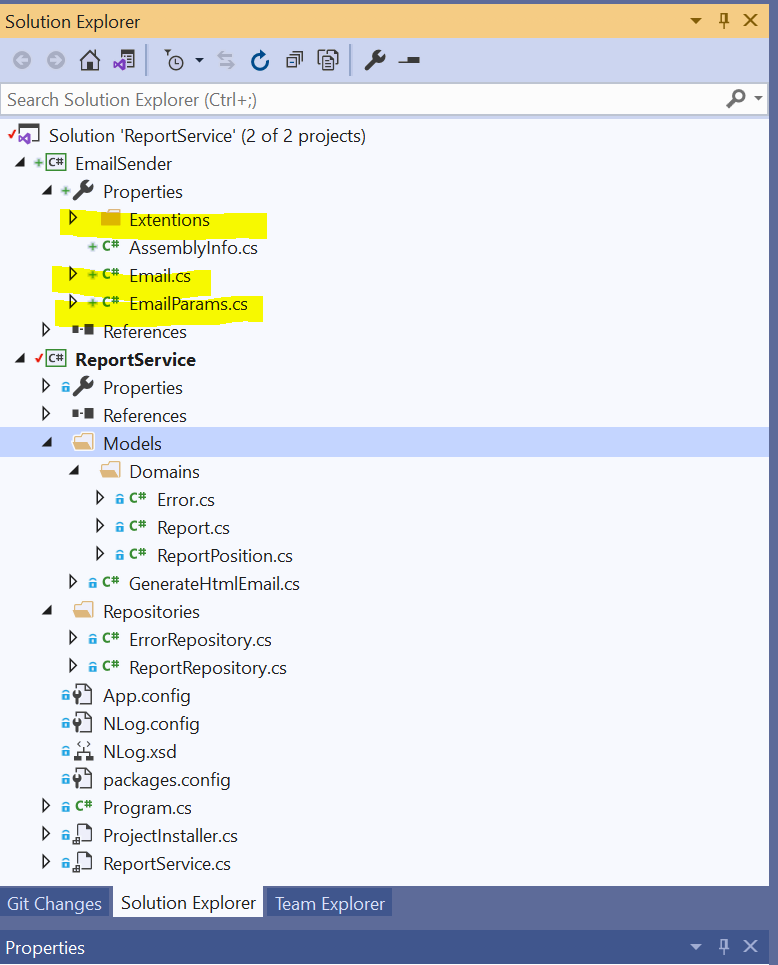
Tworzymy nowy projekt





Usuwamy domyślną klasę Class1.cs   
Przenosimy Email.cs, EmailParams.cs i Extentions.cs do nowego projektu





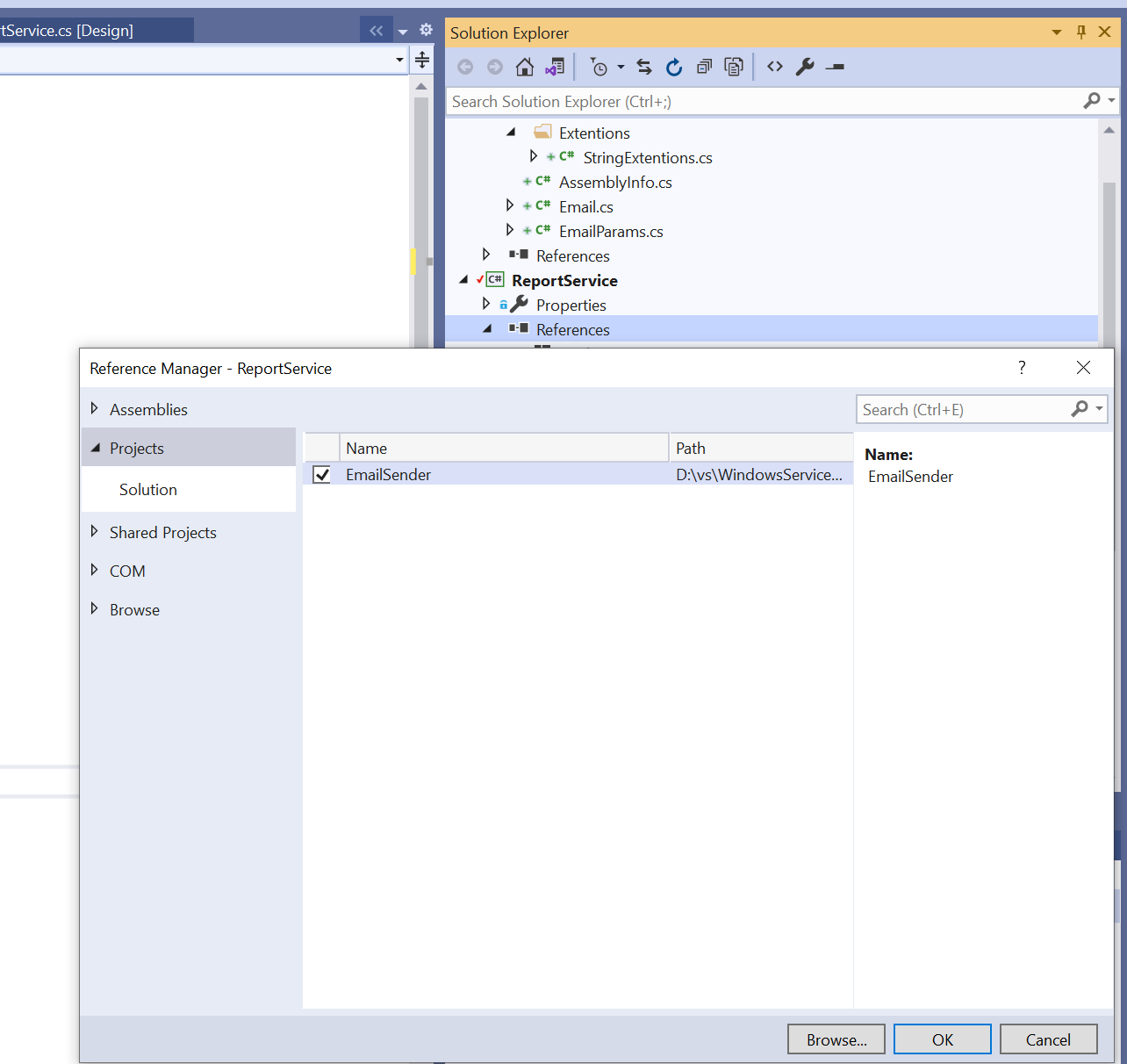
Poprawiamy namespace’y w przeniesionych za pomocą drag&drop klasach

|  |
| --- |
| namespace ReportService.Models  {  public class Email  {  private SmtpClient \_smtp;  …  namespace EmailSender  {  public class Email  {  private SmtpClient \_smtp;  … |

Aktualizujemy odwołania do klasy EmailSender w klasie ReportService  
Możemy to zrobić na kilka sposobów

Po przez refakotryzację kodu

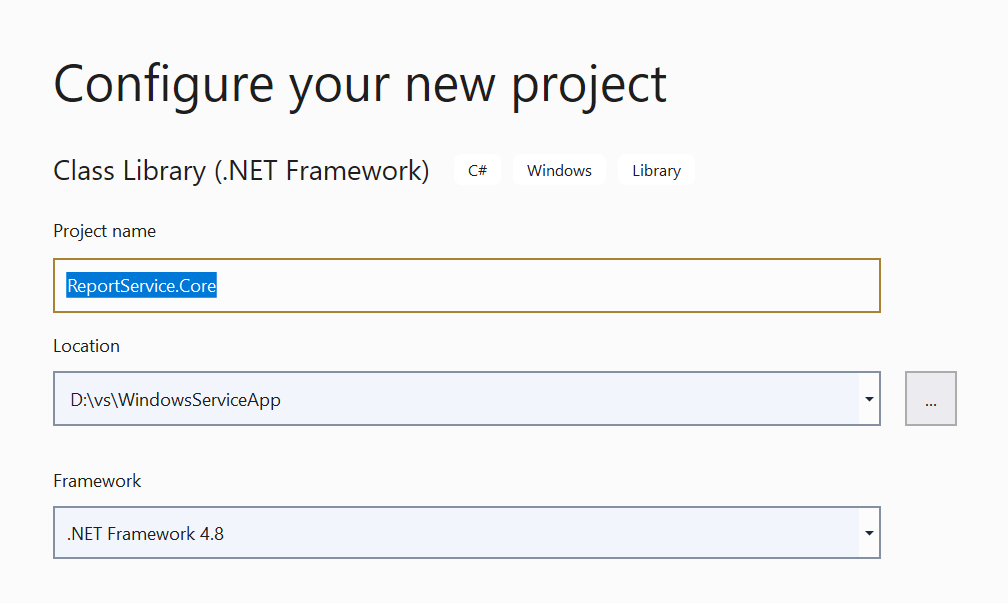
Albo ręczenie na referencjach klikamy AddReferences

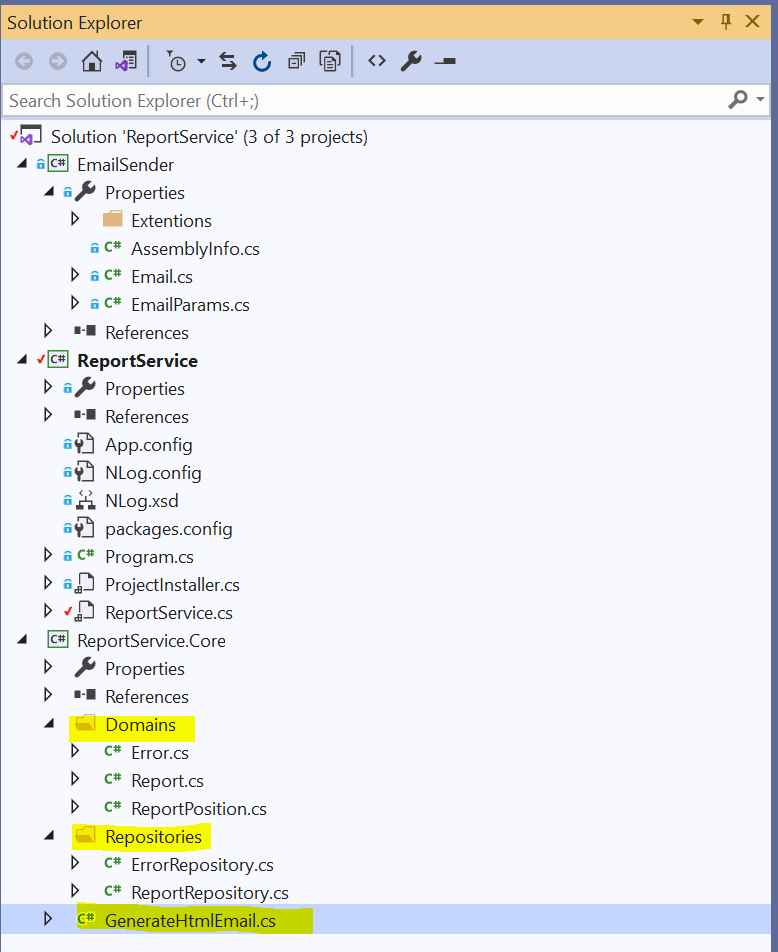


Potem dodajemy tylko odpowiednie namespace’y

Jeżeli będziemy Modele używać w innej aplikacji możemy je również wyodrębnić do nowego projektu

Możemy zostawić taką samą nazwę jak w głównym projekcie i dodać   
do nazwy przyrostek .Core



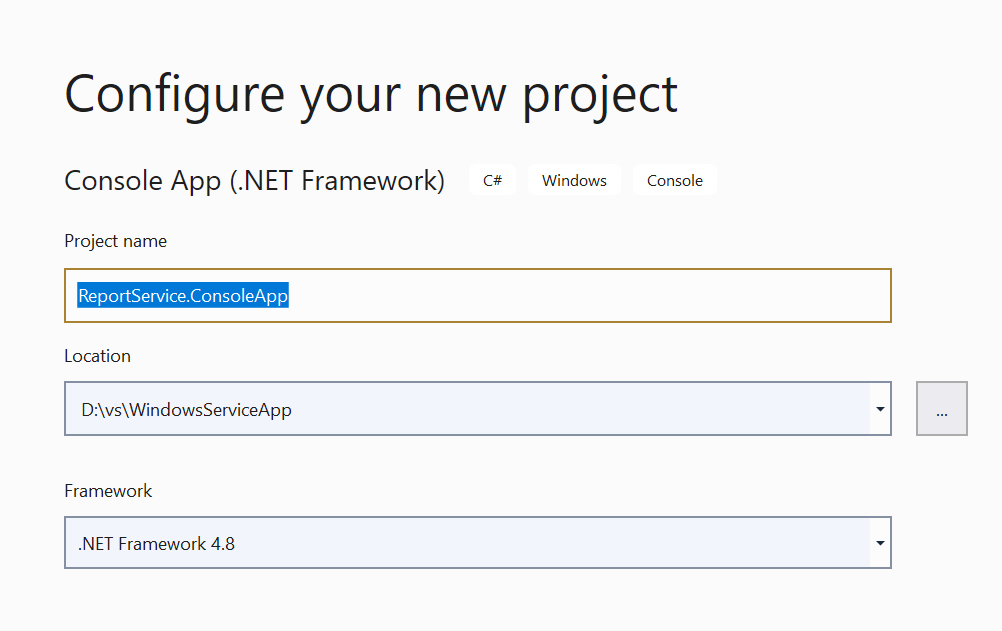


Przenosimy katalogi Domains, Repositories oraz GenerateHtmlEmails.cs  
Dodajemy referencje w ReportService do ReportService.Core i modyfikujemy using’i w obu projektach.

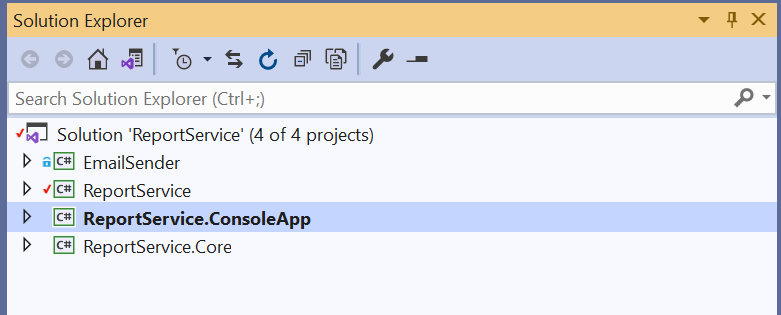
**Testujemy usługę**

Dodajemy nowy projekt – aplikację konsolową

Uwaga metoda testowania, gdy nie posiadamy wiedzy o testach jednostkowych



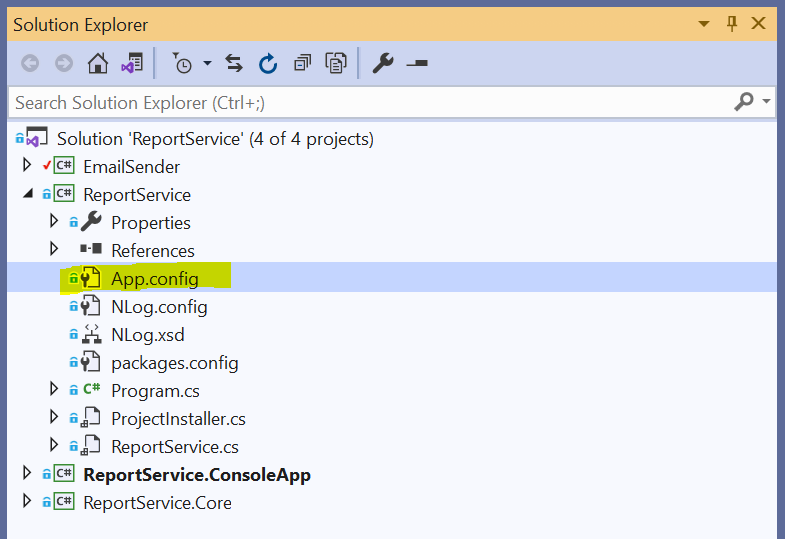
Uruchomienie



Ustawiamy na projekcie **Set as startup project**.

**Dane Konfiguracyjne**

Dane przy pliku exe naszej aplikacji, nie w katalogu **AppData**, jak w przypadku ustawień użytkownika.



|  |
| --- |
| <?xml version="1.0" encoding="utf-8" ?>  <configuration>  <startup>  <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.8" />  </startup> |
| <appSettings>  <add key="HostSmtp" value="smtp.gmail.com"/>  <add key="Port" value="587"/>  <add key="EnableSsl" value="true"/>  <add key="SenderName" value="Mariusz Wieczorek"/>  <add key="SenderEmail" value="mariusz.wieczorek.testy@gmail.com"/>  <add key="SenderEmailPassword" value="rmhfvaurzyxnuztn"/>  <add key="ReceiverEmail" value="m.wieczorek1972@gmail.com"/>  </appSettings> |
| </configuration> |

Dostęp do tych danych za pomocą ConfigurationManager’a

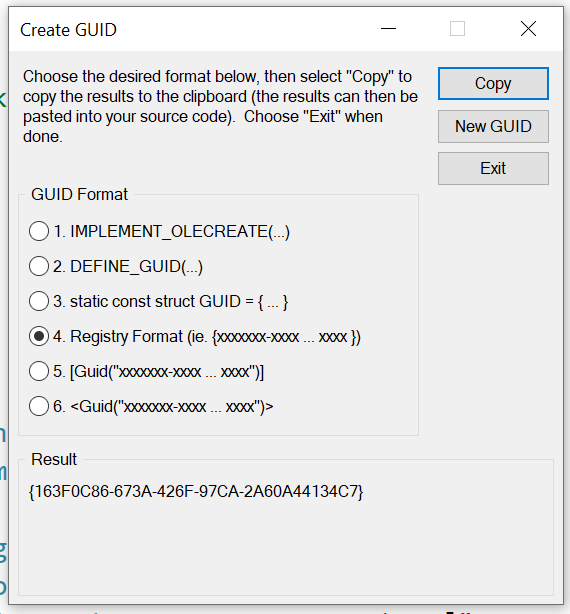
|  |
| --- |
| try  {  \_receiverEmail = ConfigurationManager.AppSettings["ReceiverEmail"];  \_email = new Email(new EmailParams  {  HostSmtp = ConfigurationManager.AppSettings["HostSmtp"],  Port = int.Parse(ConfigurationManager.AppSettings["Port"]),  EnableSsl = bool.Parse(ConfigurationManager.AppSettings["HostSmtp"]),  SenderName = ConfigurationManager.AppSettings["SenderName"],  SenderEmail = ConfigurationManager.AppSettings["SenderEmail"],  SenderEmailPassword = ConfigurationManager.AppSettings["SenderEmailPassword"],  });  }  catch (Exception ex)  {  Logger.Error(ex, ex.Message);  throw new Exception(ex.Message);  }  } |

**Szyfrowanie Danych Wrażliwych**

<https://stackoverflow.com/questions/10168240/encrypting-decrypting-a-string-in-c-sharp>

Kopiujemy klasę, tworzymy nowy projekt, class library.

Do wygenerowania klucza używamy narzędzia typu GUID, można wbudowane w Visual Studio Tools -> Create GUID.



Ustawiamy aby aplikacja sama zaszyfrowała hasło, jeżeli nie jest ono jeszcze zaszyfrowane.

|  |
| --- |
| <appSettings>  <add key="HostSmtp" value="smtp.gmail.com"/>  <add key="Port" value="587"/>  <add key="EnableSsl" value="true"/>  <add key="SenderName" value="Mariusz Wieczorek"/>  <add key="SenderEmail" value="mariusz.wieczorek.testy@gmail.com"/>  <add key="SenderEmailPassword" value="encrypt:rmhfvaurzyxnuztn"/>  <add key="ReceiverEmail" value="m.wieczorek1972@gmail.com"/>  </appSettings> |

Dodajemy jakieś słowo do hasła, które będzie mówiło aplikacji, że jest to jeszcze nie zaszyfrowane hasło – np.: encrypted:.

Aplikacja podczas pierwszego uruchomienia sprawdzi czy jest to słowo na początku hasła, jeżeli jest to zastąpi je zaszyfrowaną wersją.

|  |
| --- |
| var encryptedPassword = ConfigurationManager.AppSettings["SenderEmailPassword"];  if (encryptedPassword.StartsWith("encrypt:"))  {  var passwordToEncrypt = encryptedPassword.Replace("encrypt:",string.Empty);  encryptedPassword = \_stringCipher.Encrypt(passwordToEncrypt);  var configFile = ConfigurationManager.OpenExeConfiguration(ConfigurationUserLevel.None);  configFile.AppSettings.Settings["SenderEmailPassword"].Value = encryptedPassword;  } |

|  |
| --- |
| \_email = new Email(new EmailParams  {  HostSmtp = ConfigurationManager.AppSettings["HostSmtp"],  Port = int.Parse(ConfigurationManager.AppSettings["Port"]),  EnableSsl = bool.Parse(ConfigurationManager.AppSettings["HostSmtp"]),  SenderName = ConfigurationManager.AppSettings["SenderName"],  SenderEmail = ConfigurationManager.AppSettings["SenderEmail"],  SenderEmailPassword = \_stringCipher.Decrypt(ConfigurationManager.AppSettings["SenderEmailPassword"]),  }); |

Uwaga:

W projekcie App.config pozostaje niezmieniony. Przy pliku exe w katalogu Debug powstaje kopia tego pliku o zmienionej nazwie zamiast App nazwa aplikacji ReportService.ConsoleApp.exe.Config.  
W tym pliku zostaje zmieniona wartość SenderEmailPassword.

Wdrażanie Usługi Windows

Przebudowujemy aplikację w trybie release  
Kopiujemy pliki:

