



Data Capture and Extraction with C#

Succinctly[®]

by Ed Freitas

Data Capture and Extraction with C# Succinctly

By

Ed Freitas

Foreword by Daniel Jebaraj



Copyright © 2016 by Syncfusion Inc.

2501 Aerial Center Parkway

Suite 200

Morrisville, NC 27560

USA

All rights reserved.

I mportant licensing information. Please read.

This book is available for free download from www.syncfusion.com on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from www.syncfusion.com.

This book is licensed for reading only if obtained from www.syncfusion.com.

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.

Technical Reviewer: Zoran Maksimovic

Copy Editor: John Elderkin

Acquisitions Coordinator: Morgan Weston, marketing coordinator, Syncfusion, Inc.

Proofreader: Darren West, content producer, Syncfusion, Inc.

Table of Contents

About the Author	7
Acknowledgements	8
Introduction	9
Chapter 1 Extracting Data from Emails	10
Introduction.....	10
Understanding emails.....	11
MailKit basics	12
Parsing emails	17
Demo program	20
Using IMAP	27
Demo program source code.....	29
Chapter 2 Extracting Data from Screenshots	34
Introduction.....	34
Understanding formats	34
OpenCV basics.....	35
Parsing screenshots.....	37
Demo program.....	39
Summary	40
Complete demo program source code.....	41
Chapter 3 Extracting Data from the Web.....	45
Introduction.....	45
Understanding REST & HTTP requests	46
Parsing JSON responses	52
Demo program.....	55
Summary	57
Complete demo program source code.....	57
Chapter 4 Extracting Meaning from Text	62
Introduction.....	62
Understanding contextualization	63
Common data types & RegEx	77
Identifying entities	80
Summary	84

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet, and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just like everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!



About the Author

Ed Freitas works as consultant. He was recently involved in analyzing 1.6 billion rows of data using Redshift (Amazon Web Services) in order to gather valuable insights on client patterns. Ed holds a master's degree in computer science, and he enjoys soccer, running, travelling, and life hacking. You can reach him at Edfreitas.me.

Acknowledgements

My thanks to all the people who contributed to this book, especially Hillary Bowling, Tres Watkins, and Darren West, the Syncfusion team that helped make it a reality. Thanks also to Manuscript Manager Darren West and Technical Editor Zoran Maksimovic, who thoroughly reviewed the book's organization, code quality, and accuracy. My colleagues Simon, Neil, Josh, and John Robert acted as technical reviewers and provided many helpful suggestions regarding correctness, coding style, readability, and implementation alternatives. Thank you all.

Introduction

The world around us is filled with information. Valuable data is locked in silos such as emails, screenshots and the web. Capturing and extracting that information in order to process it, make sense of it, and use it to help us make better and informed decisions should be fun and stimulating.

This book will provide an overview on capturing and extracting data from various sources in an easy and comprehensible way, using open source technologies available to anyone. However, these technologies are not replacements for, nor intended to compete with, specialized commercial tools that provide a much broader range of possibilities and are case specific and fined-tuned for particular scenarios.

You will also gain an understanding of the methods, techniques, and libraries used in data extraction, which can lead to valuable insights and help you become a better manager, operate your business more effectively, and create a competitive advantage in the business world.

For readers with knowledge of C#, this book will offer exciting glimpses into what is technically possible without in-depth analyses of each topic. The techniques presented here, along with the clear, concise, and easy-to-follow examples provided, will provide a good head start on understanding what is feasible with data capture and extraction in C#. Have fun!

Chapter 1 Extracting Data from Emails

Introduction

Email has become a pillar of our modern and connected society, and it now serves as a primary means of communication. Because each email is filled with valuable information, data extraction has emerged as a worthwhile skill set for developers in today's business world. If you can parse an email and extract data from it, companies that automate processes, e.g., helpdesk systems, will value your expertise.

An email can be divided into several parts: subject, body, attachments, sender and receiver(s). We should also note that the headers section reveals important information about the mail servers involved in the process of sending and receiving an email.

Before addressing how we can extract information from each part of an email, we should understand that a mailbox can be viewed as a semistructured database that does not use a native querying language (e.g., SQL) to extract information.

Email
Headers
Contents
Sender (From)
Receiver (To)
CC (List of one or more Receivers visible to the main Receiver)
BCC (List of one or more Receivers not visible to the main Receiver)
Subject
Body
Content
Attachments (If any)
Attachment 1
..
Attachment N

Table 1: A Typical Email Structure

Table 1 depicts a typical email structure, which can be queried using C#. Note that the structure of an email is layered and some elements are contained within other elements. For example, attachments fall within content, which is part of the body. This internal structure might vary slightly, but this layered view makes the concepts easier to understand.

With this structure in mind, we can look for ways to extract data from each element and make meaning of it. We will address this later in Chapter 1.

Keep in mind that these elements will always contain data: Headers, Contents, To, Sender, and Receiver. These are essential—without them an email cannot be relayed. However, other email elements, such as CC, BCC, Subject, Body, Content and Attachments, might not contain data.

This chapter will address how to connect to a POP3 or IMAP mail server, how to retrieve, parse, extract data from emails, and how to send responses via SMTP using the MailKit library with C#.

Understanding emails

Now that we know an email's structure includes several elements, we need to understand which types of data exist within each email element.

Table 2 represents the data types of email elements.

Sender	contains	1 Email address (string)
Receiver	contains	1 or more Email addresses (string)
CC	contains	1 or more Email addresses (string)
BCC	contains	1 or more Email addresses (string)
Subject	contains	1 line of text (string)
Content	contains	1 or multiple lines of Text or HTML (string)
Attachments	contains	1 or multiple files

Table 2: Email Elements Data Types

Using our knowledge of the data types for email elements, we can determine how to treat each element and predict the type of data we can expect to extract. In order to connect to a mail server and extract data, we will be using a cross-platform C# library called MailKit.

MailKit basics

MailKit is cross-platform .NET library for IMAP, POP3, and SMTP built on top of MimeKit. Mailkit was developed by Jeffrey Stedfast from Xamarin, and more information about it can be found at <https://components.xamarin.com/view/mailkit> or on GitHub at <https://github.com/jstedfast/MailKit>.

You can install MailKit as a NuGet package with Visual Studio.

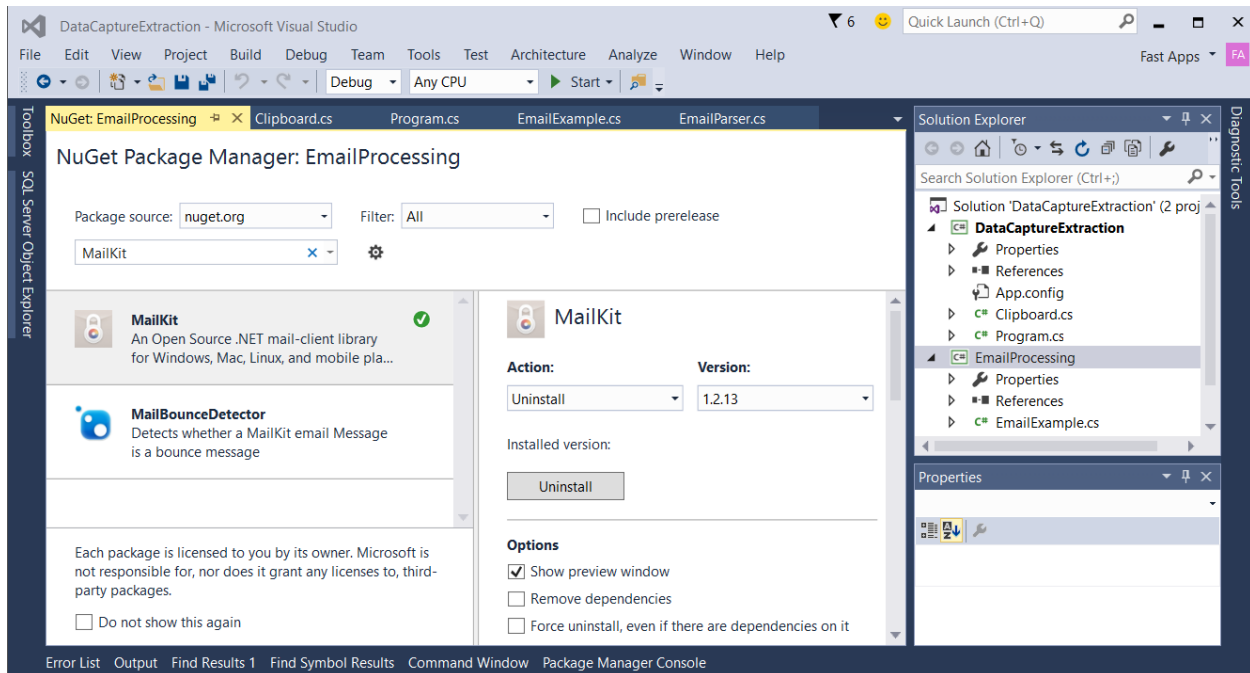


Figure 1: Installing MailKit as NuGet Package with Visual Studio 2015

MailKit supports IMAP4, POP3, SMTP clients, SASL authentication and client-side sorting and threading of messages. The full list of supported features can be found on MailKit website.

Because MailKit is a C# library, the following code examples have been written using Visual Studio 2015, which targets .NET 4.5.2 and might use some of the features of C# 6.0.

Connecting to a POP3 or IMAP server in order to later retrieve emails is a fundamental component of using MailKit. Let's have a look at how this can be done.

Code Listing 1: Reading Message Subjects from a POP3 Server

```
// EmailParser.cs: Using MailKit to Retrieve Email Data.
Using System;
using MailKit.Net.Pop3;
using MimeKit;

namespace EmailProcessing
{
```

```

public class EmailParser : IDisposable
{
    protected string User { get; set; }
    protected string Pwd { get; set; }
    protected string MailServer { get; set; }
    protected int Port { get; set; }

    public Pop3Client Pop3 { get; set; }

    public EmailParser(string user, string pwd, string mailserver,
        int port)
    {
        User = user;
        Pwd = pwd;
        MailServer = mailserver;
        Port = port;
        Pop3 = null;
    }

    public void OpenPop3()
    {
        if (Pop3 == null)
        {
            Pop3 = new Pop3Client();

            Pop3.Connect(this.MailServer, this.Port, false);
            Pop3.AuthenticationMechanisms.Remove("XOAUTH2");
            Pop3.Authenticate(this.User, this.Pwd);
        }
    }

    public void ClosePop3()
    {
        if (Pop3 != null)
        {
            Pop3.Disconnect(true);
            Pop3.Dispose();
            Pop3 = null;
        }
    }

    public void DisplayPop3Subjects()
    {
        for (int i = 0; i < Pop3?.Count; i++)
        {
            MimeMessage message = Pop3.GetMessage(i);
            Console.WriteLine("Subject: {0}", message.Subject);
        }
    }
}

```

```

        public void Dispose() { }
    }
}

// EmailExample.cs: Email Wrapper Class

using System;

namespace EmailProcessing
{
    public class EmailExample
    {
        private const string cPopUserName = "test@popserver.com";
        private const string cPopPwd = "testPwd123";
        private const string cPopMailServer = "mail.popserver.com";
        private const int cPopPort = 110;

        public static void ShowPop3Subjects()
        {
            using (EmailParser ep =
                new EmailParser(cPopUserName,
                    cPopPwd, cPopMailServer, cPopPort))
            {
                ep.OpenPop3();

                ep.DisplayPop3Subjects();

                ep.ClosePop3();
            }
        }
    }
}

// Program.cs: Show Subjects from POP3 Messages.

using EmailProcessing;

namespace DataCaptureExtraction
{
    class Program
    {
        static void Main(string[] args)
        {
            EmailExample.ShowPop3Subjects();
        }
    }
}

```

The main logic is contained within the **Main** method of **Program.cs**, which calls the **ShowPop3Subjects** method. **ShowPop3Subjects** then executes the **OpenPop3**, **DisplayPop3Subjects**, and **ClosePop3** methods of the **EmailParser** class.

In order to fully understand how MailKit is used, let's focus on the **EmailParser** class, which contains the calls to the MailKit methods. Three methods perform the connection to the Mail Server, retrieve the email messages, and close the connection. Let's look at each one.

As its name implies, **OpenPop3** opens a connection to a POP3 Server using the MailKit library.

Code Listing 2: Connecting to a POP3 Server

```
public void OpenPop3()
{
    if (Pop3 == null)
    {
        Pop3 = new Pop3Client();

        Pop3.Connect(this.MailServer, this.Port, false);
        Pop3.AuthenticationMechanisms.Remove("XOAUTH2");
        Pop3.Authenticate(this.User, this.Pwd);
    }
}
```

In Code Listing 2 a **Pop3Client** instance is created. Next, with the **Pop3** variable holding the instance, a call to the **Connect** method passes the name of the POP3 Mail Server, the POP3 port, and the third parameter indicating whether or not SSL will be used. In this case SSL is not used and is therefore set to **false**.

We next invoke **AuthenticationMechanisms**. Because we don't have an OAuth2 token we disable the **XOAUTH2** authentication mechanism.

Then, in order to establish the connection to the POP3 server, **Authenticate** is called, which passes the user name (usually the same as the name of the email address of the mailbox being queried) and its respective password.

With an open connection to the POP3 server, we can next loop through the email messages and retrieve information for each. In this case, we will be fetching the subject of each email. Code Listing 3 demonstrates how this can be achieved.

Code Listing 3: Fetching Email Subjects

```
public void DisplayPop3Subjects()
{
    for (int i = 0; i < Pop3?.Count; i++)
    {
        MimeMessage message = Pop3.GetMessage(i);
        Console.WriteLine("Subject: {0}", message.Subject);
    }
}
```

```
}
```

Each email message is represented by an instance of a **MimeMessage** class. The method **GetMessage** from the Pop3 object is responsible for retrieving the **MimeMessage** object. The **MimeMessage** object includes a property called **Subject** that returns the subject of the email.

Table 3 represents the properties available within a **MimeMessage** object.

.Attachments	IEnumerable<MimeEntity>
.Bcc	InternetAddressList
.Body	MimeEntity
.BodyParts	IEnumerable<MimeEntity>
.Cc	InternetAddressList
.Date	DateTimeOffset
.From	InternetAddressList
.Headers	HeaderList
.HtmlBody	String
.Importance	MessageImportance
.InReplyTo	String
.MessageId	String
.MimeVersion	Version
.Priority	MessagePriority
.References	MessageIdList
.ReplyTo	InternetAddressList
.ResentBcc	InternetAddressList
.ResentCc	InternetAddressList
.ResentDate	DateTimeOffset
.ResentFrom	InternetAddressList
.ResentMessageId	String

.ResentReplyTo	InternetAddressList
.ResentSender	MailboxAddress
.ResentTo	InternetAddressList
.Sender	MailboxAddress
.Subject	String
.TextBody	String
.To	InternetAddressList

Table 3: MimeMessage Properties

As you can see, the properties of **MimeMessage** objects are self-descriptive, and understanding their meanings is quite easy. Be sure to notice that MailKit uses the `InternetAddressList` object to represent a list of email addresses, and it uses `MailboxAddress` to represent a single email address.

Several of the properties, such as `Sender`, `To`, `ReplyTo`, `From`, `CC`, and `BCC`, use an equivalent prefix with the word `Resent`. This prefix is used when an email has been resent and allows MailKit to identify sender and receiver data from emails that have been resent and those that have been not.

Now that we have explored the basic of using MailKit, let's examine how to do extract more information from emails.

Parsing emails

MailKit is an awesome, easy-to-use library for handling emails. Let's look at other interesting email data manipulations it offers.

Code Listing 4 demonstrates how we can retrieve the header fields and values for any email.

Code Listing 4: Extracting Header Fields and Values

```
public void DisplayPop3HeaderInfo()
{
    for (int i = 0; i < Pop3?.Count; i++)
    {
        MimeMessage message = Pop3.GetMessage(i);
        Console.WriteLine("Subject: {0}", message?.Subject);

        foreach (Header h in message?.Headers)
        {

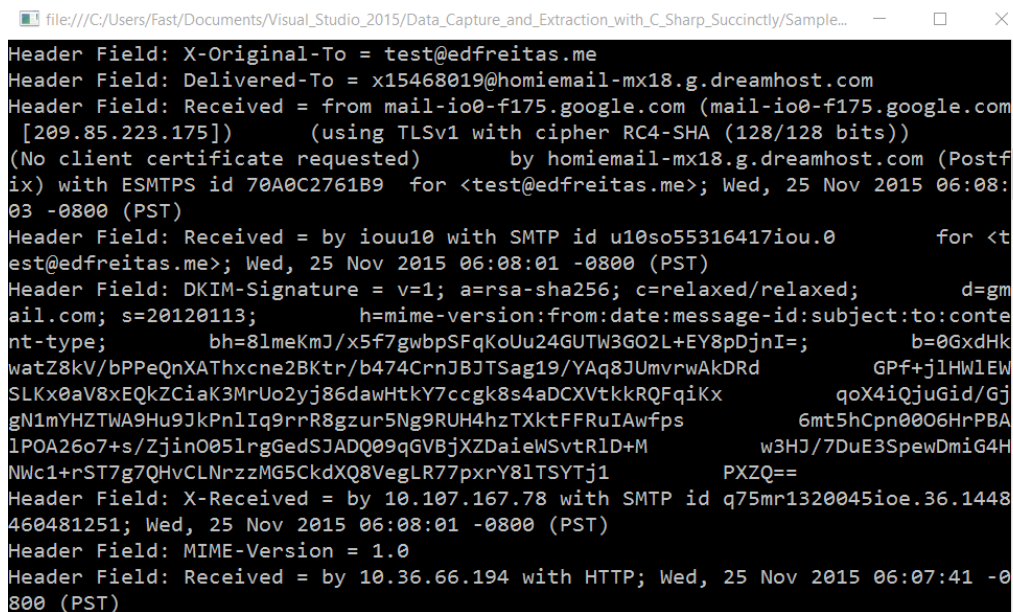
```

```

        Console.WriteLine("Header Field: {0} = {1}",
            h.Field, h.Value);
    }
}
}

```

When this code runs, the output produced looks similar to Figure 2.



```

file:///C:/Users/Fast/Documents/Visual_Studio_2015/Data_Capture_and_Extraction_with_C_Sharp_Succinctly/Sample...
Header Field: X-Original-To = test@edfreitas.me
Header Field: Delivered-To = x15468019@homiemail-mx18.g.dreamhost.com
Header Field: Received = from mail-io0-f175.google.com (mail-io0-f175.google.com
[209.85.223.175]) (using TLSv1 with cipher RC4-SHA (128/128 bits))
(No client certificate requested) by homiemail-mx18.g.dreamhost.com (Postf
ix) with ESMTPS id 70A0C2761B9 for <test@edfreitas.me>; Wed, 25 Nov 2015 06:08:
03 -0800 (PST)
Header Field: Received = by iouu10 with SMTP id u10so55316417iou.0 for <t
est@edfreitas.me>; Wed, 25 Nov 2015 06:08:01 -0800 (PST)
Header Field: DKIM-Signature = v=1; a=rsa-sha256; c=relaxed/relaxed; d=gm
ail.com; s=20120113; h=mime-version:from:date:message-id:subject:to:conte
nt-type; bh=8lmeKmJ/x5f7gwbpSFqKoUu24GUTW3GO2L+EY8pDjnI=; b=0GxdHk
watZ8kV/bPpEQnXAThxcne2BKtr/b474CrnJBjTSag19/YAq8JUmvrvwAkDRd GPf+jlHWlEW
SLKx0aV8xEQkZCiaK3MrUo2yj86dawHtkY7ccgk8s4aDCXVtkkRQFqikX qoX4iQjuGid/Gj
gN1mYHZTWA9Hu9JkPn1Iq9rrR8gzur5Ng9RUH4hzTXktFFRuIAwfps 6mt5hCpn0006HrPBA
lPOA26o7+s/Zjin005lrgGedSJADQ09qGVBJXZDaieWSvtR1D+M w3HJ/7DuE3SpewDmiG4H
NWc1+rST7g7QHvCLNrzzMG5CkdXQ8VegLR77pxrY81TSYTj1 PXZQ==
Header Field: X-Received = by 10.107.167.78 with SMTP id q75mr1320045ioe.36.1448
460481251; Wed, 25 Nov 2015 06:08:01 -0800 (PST)
Header Field: MIME-Version = 1.0
Header Field: Received = by 10.36.66.194 with HTTP; Wed, 25 Nov 2015 06:07:41 -0
800 (PST)

```

Figure 2: Output of Email Header Data

A wonderful feature of MailKit is that each of its email headers has a field property `that` indicates the name of the actual header along with a value.

In Code Listing 4, each header field's corresponding value is written to the Windows console. For example, the header `Delivered-To` has a value of x15468019@homiemail-mx18.g.dreamhost.com. This particular header allows us to easily determine that the destination email address is being hosted by a mail server at Dreamhost.

We can easily see that the `Received` header occurs several times and that that header generally describes the email going through several servers—e.g., `mail.google.com` (the origin server) and `dreamhost.com` (the destination server).

As Code Listing 4 demonstrates, all the email headers are stored inside a `HeaderList` within the `MimeMessage` object. This list can be iterated with a `foreach` loop through a `Header` object. Because headers contain valuable information about the processes involved in sending and receiving emails, they can be used to trace each email back to its original source. We will not address the details of that tracing here, but we can get a good idea of what is possible by checking email header data. MailKit makes retrieving this data very easy. Then it is up to you or your business logic to make sense of it.

Let's now process an email that has at least one attachment in order to extract the body and save each attachment on a local folder on the Windows file system.

Code Listing 5: Extracting and Saving Email Attachments

```
public void SavePop3BodyAndAttachments(string path)
{
    for (int i = 0; i < Pop3?.Count; i++)
    {
        MimeMessage msg = Pop3.GetMessage(i);

        if (msg != null)
        {
            string b = msg.GetTextBody(MimeKit.Text.TextFormat.Text);

            Console.WriteLine("Body: {0}", b);

            if (msg.Attachments != null)
            {
                foreach (MimeEntity att in msg.Attachments)
                {
                    if (att.IsAttachment)
                    {
                        if (!Directory.Exists(path))
                            Directory.CreateDirectory(path);

                        string fn = Path.Combine(path,
                            att.ContentType.Name);

                        if (File.Exists(fn))
                            File.Delete(fn);

                        using (var stream = File.Create(fn))
                        {
                            var mp = ((MimePart)att);
                            mp.ContentObject.DecodeTo(stream);
                        }
                    }
                }
            }
        }
    }
}
```

As with our previous examples, the preceding code demonstrates that the email (**MimeMessage** object) is retrieved using **GetMessage**. In order to retrieve the body of the email as plain text, we can invoke **GetTextBody** with the parameter **MimeKit.Text.TextFormat.Text**. We can achieve this by using the property **TextBody**. In order to retrieve the body in HTML format, we can use **GetTextBody** to bypass the parameter **MimeKit.Text.TextFormat.Html** or, alternatively, by using the property **HtmlBody**.

After we have retrieved the body of the email, we next check the **MimeMessage** object for any attachments. If attachments are present, we loop through each and retrieve a **MimeEntity** object. Before attempting to save the attachment (which is held by the **MimeEntity** object), we should first check that it is in fact a real attachment by inspecting its **IsAttachment** property to see if that evaluates to **true**. If so, the attachment is saved to disk to the location passed as a parameter when calling **WriteTo**.

Demo program

Let's now see how we can create an automatic response system using MailKit based on the contents of the email received. This demo program is based on the previous snippets of code provided. The response system will analyze the email's body, subject and contents of any plain text attachment, will search for particular keywords and, depending upon the type of keywords located, will send back a predefined reply. A similar process can be followed to automate almost any kind business processes that involve receiving and processing emails.

Code Listing 6: Sending Automatic Responses

```
// Program.cs: Send Automated SMTP Responses.

using EmailProcessing;

namespace DataCaptureExtraction
{
    class Program
    {
        static void Main(string[] args)
        {
            EmailExample.AutomatedSmtplibResponses();
        }
    }
}

// EmailExample.cs: Email Wrapper Class

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```

namespace EmailProcessing
{
    public class EmailExample
    {
        private const string cSmtpUserName = "hello@smtpserver.com";
        private const string cSmtpPwd = "1234";
        private const string cSmtpMailServer = "mail.smtpserver.com";
        private const int cSmtpPort = 465;

        public static void AutomatedSmtpResponses()
        {
            using (EmailParser ep = new
                EmailParser(cPopUserName, cPopPwd,
                    cPopMailServer, cPopPort))
            {
                ep.OpenPop3();

                ep.AutomatedSmtpResponses(cSmtpMailServer,
                    cSmtpUserName, cSmtpPwd, cSmtpPort,
                    "receiver@someserver.com", "Some Person");

                ep.ClosePop3();
            }
        }
    }
}

```

// EmailParser.cs: Use MailKit to Retrieve Email Data.

```

using System;
using System.Collections.Generic;
using System.Linq;
using MailKit.Net.Pop3;
using MimeKit;
using MailKit.Net.Smtp;

namespace EmailProcessing
{
    public class EmailParser : IDisposable
    {
        private const string cStrInvoice = "Invoice";
        private const string cStrMarketing = "Marketing";
        private const string cStrSupport = "Support";

        private const string cStrDftMsg = @"Hi,

We've received your message but we are unable to
classify it properly.

```

```

-- Cheers. Ed.";

private const string cStrMktMsg = @"Hi,

We've received your message and we've relayed
it to the Marketing department.

-- Cheers. Ed.";

private const string cStrAptMsg = @"Hi,

We've received your message and we've relayed
it to the Payment department.

-- Cheers. Ed.";

private const string cStrSupportMsg = @"Hi,

We've received your message and we've relayed
it to the Support department.

-- Cheers. Ed.";

protected string[] GetPop3EmailWords(ref MimeMessage m)
{
    List<string> w = new List<string>();

    string b = String.Empty, s = String.Empty, c = String.Empty;

    b = m.GetTextBody(MimeKit.Text.TextFormat.Text);
    s = m.Subject;

    if (m.Attachments != null)
    {
        foreach (MimeEntity att in m.Attachments)
        {
            if (att.IsAttachment)
            {
                if
                (att.ContentType.MediaType.
                 Contains("text"))
                {
                    c = ((MimeKit.TextPart)att).Text;
                }
            }
        }

        w = CleanMergeEmailWords(w, b, s, c);
    }
}

```

```

        return w.ToArray();
    }

    private static List<string> CleanMergeEmailWords(List<string> w,
        string b, string s, string c)
    {
        if (b != String.Empty || s != String.Empty || c !=
            String.Empty)
        {
            List<string> b1 = new List<string>();
            List<string> s1 = new List<string>();
            List<string> c1 = new List<string>();

            if (b != String.Empty)
                b1 = b.Split(new string[] { " ", "\r", "\n" },
                    StringSplitOptions.RemoveEmptyEntries).ToList();

            if (s != String.Empty)
                s1 = s.Split(new string[] { " ", "\r", "\n" },
                    StringSplitOptions.RemoveEmptyEntries).ToList();

            if (c != String.Empty)
                c1 = c.Split(new string[] { " ", "\r", "\n" },
                    StringSplitOptions.RemoveEmptyEntries).ToList();

            if (b1.Count > 0 || s1.Count > 0 || c1.Count > 0)
            {
                b1 = b1.Union(s1).ToList();
                w = b1.Union(c1).ToList();
            }
        }

        return w;
    }

    public void SendSmtpResponse(string smtp, string user, string
        pwd, int port, string toAddress, string toName, string msgTxt)
    {
        var message = new MimeMessage();

        message.From.Add(new MailboxAddress(
            "Ed Freitas (Automated Email Bot)",
            "hello@edfreitas.me"));

        message.To.Add(new MailboxAddress(toName, toAddress));
        message.Subject = "Thanks for reaching out";

        message.Body = new TextPart("plain")

```

```

    {
        Text = msgTxt
    };

    using (var client = new SmtpClient())
    {
        client.Connect(smtp, port, true);
        client.Authenticate(user, pwd);

        client.Send(message);

        client.Disconnect(true);
    }
}

protected int DetermineResponseType(string[] w)
{
    int res = -1;

    foreach (string ww in w)
    {
        if (ww.ToUpper().Contains(cStrInvoice.ToUpper()))
        {
            res = 1;
            break;
        }
        else if
            (ww.ToUpper().Contains(cStrMarketing.ToUpper()))
        {
            res = 0;
            break;
        }
        else if
            (ww.ToUpper().Contains(cStrSupport.ToUpper()))
        {
            res = 2;
            break;
        }
    }

    return res;
}

protected void SendResponses(string[] w, string smtp, string
user, string pwd, int port, string toAddress, string toName)
{
    switch (DetermineResponseType(w))
    {
        case 0: // Marketing

```



```

        SendSmtpResponse(smtp, user, pwd, port,
            toAddress, toName, cStrMktMsg);
        break;

    case 1: // Payment
        SendSmtpResponse(smtp, user, pwd, port,
            toAddress, toName, cStrAptMsg);
        break;

    case 2: // Support
        SendSmtpResponse(smtp, user, pwd, port,
            toAddress, toName, cStrSupportMsg);
        break;

    default: // Anything Else
        SendSmtpResponse(smtp, user, pwd, port,
            toAddress, toName, cStrDftMsg);
        break;
    }
}

public void Dispose() { }

public void AutomatedSmtpResponses(string smtp, string user,
    string pwd, int port, string toAddress, string toName)
{
    for (int i = 0; i < Pop3?.Count; i++)
    {
        MimeMessage message = Pop3.GetMessage(i);

        if (message != null)
        {
            string[] words = GetPop3EmailWords(ref message);

            if (words?.Length > 0)
                SendResponses(words, smtp, user, pwd, port,
                    toAddress, toName);
        }
    }
}
}

```

This demo program sends an automatic response email out based on the content of a previously received email.

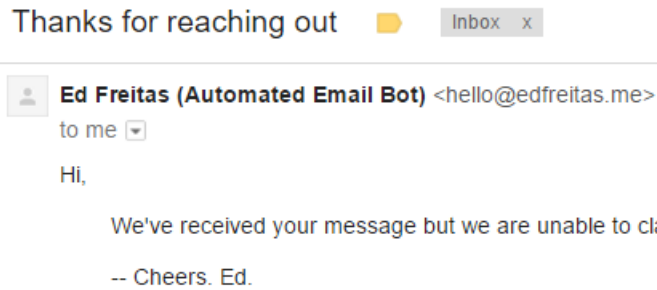


Figure 3: Automatic Email Response Based on the Contents of a Received Email

Figure 3 depicts a connection to a POP3 server and a particular email inbox and, based on the contents of the emails found, inspects each email subject, body and attachment contents, extracting all keywords and checking for any keywords that match a specific predefined set of words (e.g., support, marketing, or invoice). If so, the connection sends an automated response using SMTP. Let's dig into the details.

The **AutomatedSmtplibResponses** method is called from the Main Program. This method is simply a wrapper of the EmailExample class that invokes the **AutomatedSmtplibResponses** method from the EmailParser class.

AutomatedSmtplibResponses loops through all the emails on the POP3 inbox and, for each email, calls the **GetPop3EmailWords** method responsible for getting all the words located on the email's subject, body and attachments. If any words are repeated, only one instance of a particular word is left in the returned string array result from **GetPop3EmailWords**.

This string array result from **GetPop3EmailWords** is then passed to **SendResponses** which is also receives the SMTP server, user name, password and port in order to send back the automated response. Let's analyze a bit more of what **GetPop3EmailWords** does before moving on to **SendResponses**.

GetPop3EmailWords retrieves the email body using **GetTextBody**, and it also retrieves the email subject by calling the **Subject** property. **GetPop3EmailWords** next retrieves the contents of each attachment, which requires looping through the **Attachments** property of the **MimeMessage** object, because each individual attachment represents a **MimeEntity**.

The **MimeEntity** object includes a property called **IsAttachment** that, when set to **true**, indicates that the **MimeEntity** can be treated as an attachment. In order to extract the keywords from the attachment, we must know if the Attachment is indeed a text file. We can discover this by using the property **ContentType.MediaType**, which will indicate a text file with the word "text." If the attachment is a text file, the contents can be easily extracted by calling **((MimeKit.TextPart)att).Text**.

When all the words present on the email's subject, body and attachments are retrieved, **CleanMergeEmailWords** merges them and removes any repeated words from the final string array returned to the caller.

The extracted and filtered words get passed on to the **SendResponses** method, which loops through all the words and checks if any match the predefined set of words. When a match is produced, a predefined canned response is sent out by calling the **SendSmtpResponse** method.

SendSmtpResponse creates a **MimeMessage** object and assigns its corresponding sub-properties objects such as **MailboxAddress** (receiver details) and **TextPart** (body). Then a new **SmtpClient** object is created as the SMTP server and user credentials are assigned. Following a successful call to **Connect** and **Authenticate**, the message is finally sent using the **Send** method.

Using IMAP

MailKit is a simple but powerful library for managing and working with emails. We've learned how to connect to POP3 servers and retrieve email data from headers and contents such as body and attachments. We've also examined how responses can be sent out using SMTP. As a final matter, let's inspect how we can use IMAP instead of POP3.

Each of the POP3 methods we have addressed use a corresponding IMAP equivalent in the Demo Program Source Code.

You will find that working with IMAP in MailKit is similar to connecting and working with a POP3 server. However, in IMAP you must indicate which folder you want to open and what kind of operation you want to perform on the folder, e.g., a **Read** or **ReadWrite** operation.

In order to open the Inbox folder using IMAP, you need to call: (cut) **Imap.Inbox.Open**. With the folder open, you can loop through the **MimeMessage** objects as you would using POP3 until you reach **Imap.Inbox.Count**.

The following Code Listing shows the equivalent of the POP3 methods implemented using IMAP.

Code Listing 7: IMAP Equivalent of POP3 Methods

```
public ImapClient Imap { get; set; }

public void DisplayImapSubjects()
{
    var folder = Imap?.Inbox.Open(FolderAccess.ReadOnly);

    for (int i = 0; i < Imap?.Inbox.Count; i++)
    {
        MimeMessage message = Imap.Inbox.GetMessage(i);
        Console.WriteLine("Subject: {0}", message?.Subject);
    }
}

public void DisplayImapHeaderInfo()
{

```

```

var folder = Imap?.Inbox.Open(FolderAccess.ReadOnly);

for (int i = 0; i < Imap.Inbox?.Count; i++)
{
    MimeMessage message = Imap.Inbox.GetMessage(i);
    Console.WriteLine("Subject: {0}", message?.Subject);

    foreach (Header h in message?.Headers)
    {
        Console.WriteLine("Header Field: {0} = {1}",
            h.Field, h.Value);
    }
}

public void AutomatedSmtptResponsesImap(string smtp, string user,
string pwd, int port, string toAddress, string toName)
{
    var folder = Imap?.Inbox.Open(FolderAccess.ReadOnly);

    for (int i = 0; i < Imap?.Inbox.Count; i++)
    {
        MimeMessage message = Imap.Inbox.GetMessage(i);

        if (message != null)
        {
            string[] words = GetPop3EmailWords(ref message);

            if (words?.Length > 0)
                SendResponses(words, smtp, user, pwd, port,
                    toAddress, toName);
        }
    }
}

public void SaveImapBodyAndAttachments(string path)
{
    var folder = Imap?.Inbox.Open(FolderAccess.ReadOnly);

    for (int i = 0; i < Imap?.Inbox.Count; i++)
    {
        MimeMessage message = Imap.Inbox.GetMessage(i);

        if (message != null)
        {
            string b = message.GetTextBody(
                MimeKit.Text.TextFormat.Text);

            Console.WriteLine("Body: {0}", b);
        }
    }
}

```

```

        if (message.Attachments != null)
        {
            foreach (MimeEntity att in
                message.Attachments)
            {
                if (att.IsAttachment)
                {
                    if (!Directory.Exists(path))
                        Directory.CreateDirectory(path);

                    string fn = Path.Combine(path,
                        att.ContentType.Name);

                    if (File.Exists(fn))
                        File.Delete(fn);

                    using (var stream = File.Create(fn))
                    {
                        var mp = ((MimePart)att);
                        mp.ContentObject.DecodeTo(stream);
                    }
                }
            }
        }
    }
}

```

Code Listing 7 indicates two main differences with POP3-equivalent methods. First, with IMAP a call to **Imap.Inbox.Open** needs to specify which type of access to the inbox folder is required. And second, the looping takes into account the count on the inbox folder opened by calling **Imap.Inbox.Count**. The rest of the process is essentially the same, and each email is represented by a **MimeMessage** object.

To learn more about MailKit, to the project's GitHub website and check the code examples and further documentation.

Demo program source code

The following Code Listing contains complete source code for all the examples previously described using MailKit.

Code Listing 8: Demo Program Source Code Using MailKit

```
// Program.cs: Main Program
```

```

using EmailProcessing;

namespace DataCaptureExtraction
{
    class Program
    {
        static void Main(string[] args)
        {
            //EmailExample.ShowPop3Subjects();
            //EmailExample.DisplayPop3HeaderInfo();

            //EmailExample.SavePop3BodyAndAttachments(
            //    @"C:\Attach");

            //EmailExample.AutomatedSmtplibResponses();

            //EmailExample.ShowImapSubjects();
            //EmailExample.DisplayHeaderInfoImap();
            //EmailExample. SaveImapBodyAndAttachments(
            //    @"C:\Attach");

            EmailExample.AutomatedSmtplibResponsesImap();
        }
    }
}

// EmailExample.cs: Email Wrapper Class

using System;

namespace EmailProcessing
{
    public class EmailExample
    {
        private const string cPopUserName = "test@popserver.com";
        private const string cPopPwd = "1234";
        private const string cPopMailServer = "mail.popserver.com";
        private const int cPopPort = 110;

        private const string cImapUserName = "test@imapserver.com";
        private const string cImapPwd = "1234";
        private const string cImapMailServer = "mail.imapserver.com";
        private const int cImapPort = 993;

        private const string cSmtplibUserName = "test@smtpserver.com";
        private const string cSmtplibPwd = "1234";
        private const string cSmtplibMailServer = "mail.smtpserver.com";
        private const int cSmtplibPort = 465;
    }
}

```

```

public static void ShowPop3Subjects()
{
    using (EmailParser ep = new EmailParser(cPopUserName,
        cPopPwd, cPopMailServer, cPopPort))
    {
        ep.OpenPop3();

        ep.DisplayPop3Subjects();

        ep.ClosePop3();
    }
}

public static void ShowImapSubjects()
{
    using (EmailParser ep = new EmailParser(cImapUserName,
        cImapPwd, cImapMailServer, cImapPort))
    {
        ep.OpenImap();

        ep.DisplayImapSubjects();

        ep.CloseImap();
    }
}

public static void DisplayHeaderInfo()
{
    using (EmailParser ep = new EmailParser(cPopUserName,
        cPopPwd, cPopMailServer, cPopPort))
    {
        ep.OpenPop3();

        ep.DisplayPop3HeaderInfo();

        ep.ClosePop3();
    }
}

public static void DisplayHeaderInfoImap()
{
    using (EmailParser ep = new EmailParser(cImapUserName,
        cImapPwd, cImapMailServer, cImapPort))
    {
        ep.OpenImap();

        ep.DisplayImapHeaderInfo();
    }
}

```

```

        ep.CloseImap();
    }
}

public static void AutomatedSmtplibResponses()
{
    using (EmailParser ep = new EmailParser(cPopUserName,
        cPopPwd, cPopMailServer, cPopPort))
    {
        ep.OpenPop3();

        ep.AutomatedSmtplibResponses(cSmtplibMailServer,
            cSmtplibUserName, cSmtplibPwd,
            cSmtplibPort, "user@server.com", "Some Person");

        ep.ClosePop3();
    }
}

public static void AutomatedSmtplibResponsesImap()
{
    using (EmailParser ep = new EmailParser(cImapUserName,
        cImapPwd, cImapMailServer, cImapPort))
    {
        ep.OpenImap();

        ep.AutomatedSmtplibResponsesImap(cSmtplibMailServer,
            cSmtplibUserName, cSmtplibPwd,
            cSmtplibPort, "user@server.com", "Some Person");

        ep.CloseImap();
    }
}

public static void SavePop3BodyAndAttachments(string path)
{
    using (EmailParser ep = new EmailParser(cPopUserName,
        cPopPwd, cPopMailServer, cPopPort))
    {
        ep.OpenPop3();

        ep.SavePop3BodyAndAttachments(path);

        ep.ClosePop3();
    }
}

public static void SaveImapBodyandAttachments(string path)
{

```



```

        using (EmailParser ep = new EmailParser(cImapUserName,
            cImapPwd, cImapMailServer, cImapPort))
        {
            ep.OpenImap();

            ep.SaveImapBodyAndAttachments(path);

            ep.CloseImap();
        }
    }

    public static void OpenClosePop3()
    {
        using (EmailParser ep = new EmailParser(cPopUserName,
            cPopPwd, cPopMailServer, cPopPort))
        {
            ep.OpenPop3();
            ep.ClosePop3();
        }
    }

    public static void OpenCloseImap()
    {
        using (EmailParser ep = new EmailParser(cImapUserName,
            cImapPwd, cImapMailServer, cImapPort))
        {
            ep.OpenImap();
            ep.CloseImap();
        }
    }
}

```

The complete Visual Studio project source code can be downloaded from this URL:

<https://bitbucket.org/syncfusiontech/data-capture-and-extraction-with-c-succinctly>

Chapter 2 Extracting Data from Screenshots

Introduction

Like emails, screenshots that contain text are also filled with valuable information. For instance, some screenshots contain important material that can be extracted to automate processes such as typing and data entry. As companies and individuals increasingly automate their internal processes, extracting information from screenshots, which avoids manual data entry and typing, becomes ever more important in the business world.

The process of reading screenshots and extracting valuable information is often called Capture or Extraction. Extracting the words, numbers, or text contained within a screenshot is called Optical Character Recognition (OCR).

After reading this chapter, you should be able to install EmguCV for use within a C# program in order to perform OCR by extracting data as text from screenshots in either Portable Network Graphics (PNG) or Tagged Image Format (TIFF) formats.

Understanding formats

When a screenshot needs to be converted into a digital image, it can be saved using several different formats. The most commonly used formats for saving screenshots are TIFF, PNG, or Joint Photographic Experts Group (JPEG). The TIFF format is best suited for performing OCR, and most OCR Engines prefer working with TIFF as the predefined format for extracting text.

According to Wikipedia, the JPEG algorithm works best with pictures and drawings that use soft variations of tone and color, and this format is widely popular on the Internet. JPEG is also one of the most common formats used by digital cameras for saving pictures. However, JPEG may not be well suited for line drawings and other textual or iconic graphics (i.e. text), primarily due to sharp contrasts between adjacent pixels.

The TIFF format offers a great advantage over the others by providing special lossless compression algorithms like CCITT Group IV, which can compress bitonal images (e.g., faxes or black-and-white text) better than the JPEG or PNG compression algorithms.

When performing OCR, the preferred format is TIFF with CCITT Group IV Compression. PNG is the second most common choice.

If you want to know more about the differences between images formats, you can find valuable information in the Wikipedia reference for "[Comparison of graphics file formats](#)."

OpenCV basics

Open Source Computer Vision (OpenCV) is a C++ cross-platform library that was designed for use in implementing computer vision solutions (face detection, recognition of patterns in images, etc.). You can learn more about it on the [Open CV Wikipedia entry](#) and from the [OpenCV website](#).

Because OpenCV is a native (nonmanaged) C++ library, there is a .NET cross-platform wrapper called [EmguCV](#) that we will use to interact with Tesseract and to perform data extraction and OCR on TIFF with CCITT Group IV compression screenshots.

EmguCV allows OpenCV functions to be called from native .NET code written in C#, VB, VC++, or even IronPython. EmguCV is also compatible with [Mono](#) from [Xamarin](#) and can run on Windows, Mac OS X, Linux, iOS, and Android.

You can install EmguCV as a NuGet package using Visual Studio. Because there are several implementations available on NuGet, we'll use EmguCV.221.x64. If you are developing on a 32-bit OS, you can install EmguCV.221.x86.

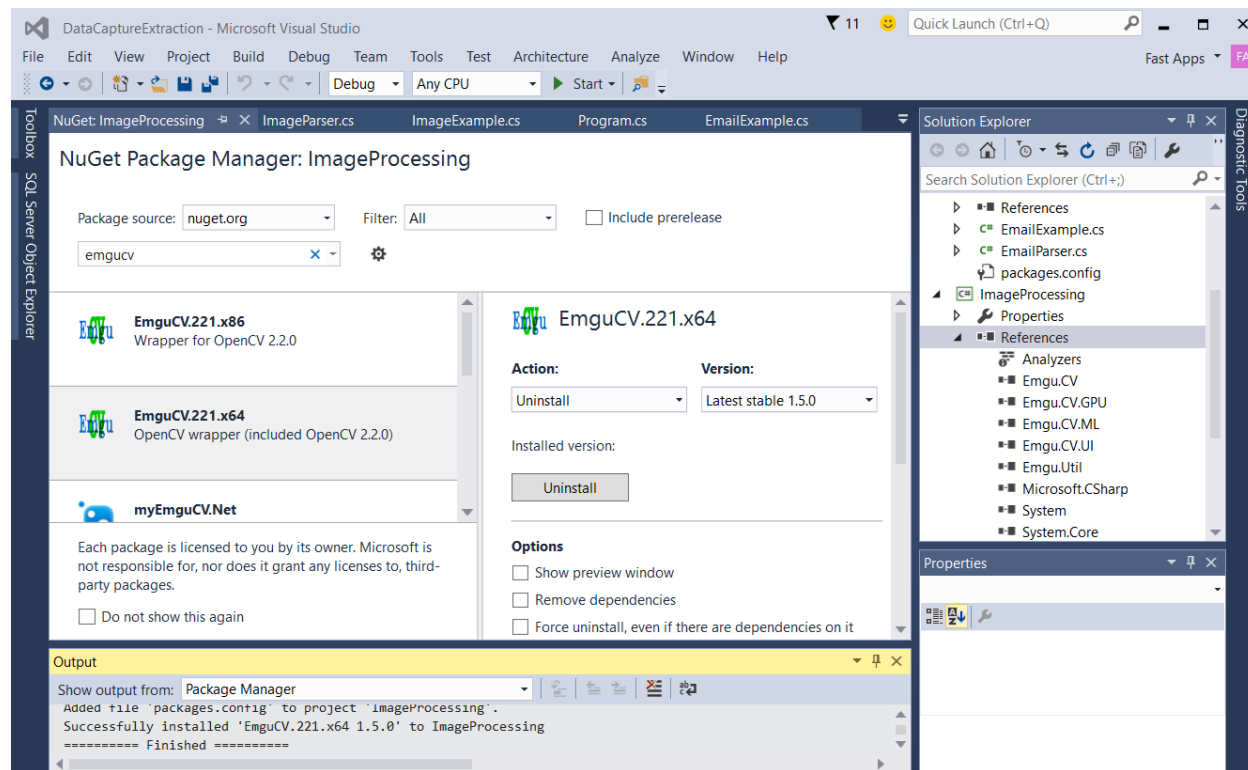


Figure 4: Installing EmguCV as a NuGet Package with Visual Studio 2015

With EmguCV installed, several dependencies are automatically added to the Visual Studio project. For us, the most important ones will be Emgu.CV, Emgu.CV.OCR, and Emgu.Util.

Be sure to notice that as you install the EmguCV NuGet package, you won't get the Emgu.CV.OCR namespace, which is essential for working with Tesseract. That means we need access to the Tesseract engine. In order to access the necessary file Emgu.CV.OCR.dll, we must download and install the full EmguCV setup, which can be found here:

https://sourceforge.net/projects/emguvcv/files/emguvcv/3.0.0/libemguvcv-windows-universal-3.0.0.2157.zip/download?use_mirror=freefr&r=&use_mirror=freefr

Therefore in order to get the file Emgu.CV.OCR.dll it is necessary to download and install the full EmguCV setup, which can be found here:

https://sourceforge.net/projects/emguvcv/files/emguvcv/3.0.0/libemguvcv-windows-universal-3.0.0.2157.zip/download?use_mirror=freefr&r=&use_mirror=freefr

If we did not need to perform OCR, we could simply use what has been installed with NuGet, and the following universal setup of Emgu.CV would be unnecessary.

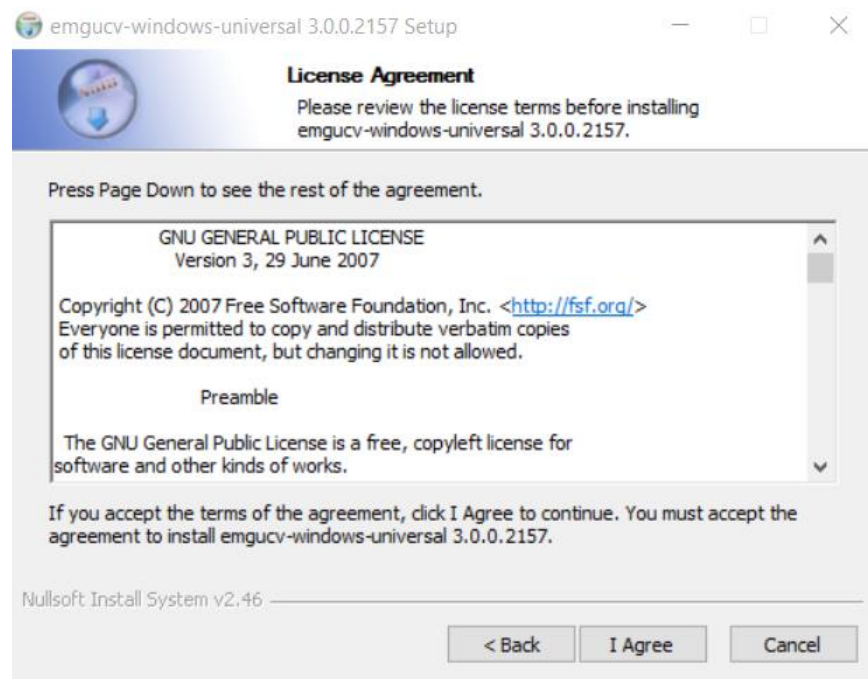


Figure 5: Universal Windows Setup of EmguCV

Once installed, the full EmguCV library can be found in Windows under "C:\Emgu\emguvcv-windows-universal 3.0.0.2157". Please bear in mind that the version number might change in the future, so the folder path might slightly vary (i.e. numbers at the end of the path).

Be sure to add these DLL tags as references: Emgu.CV.dll, Emgu.CV.OCR.dll, and Emgu.Util.dll.

Figure 6 depicts how the Visual Studio project references should look with the addition of these DLLs.

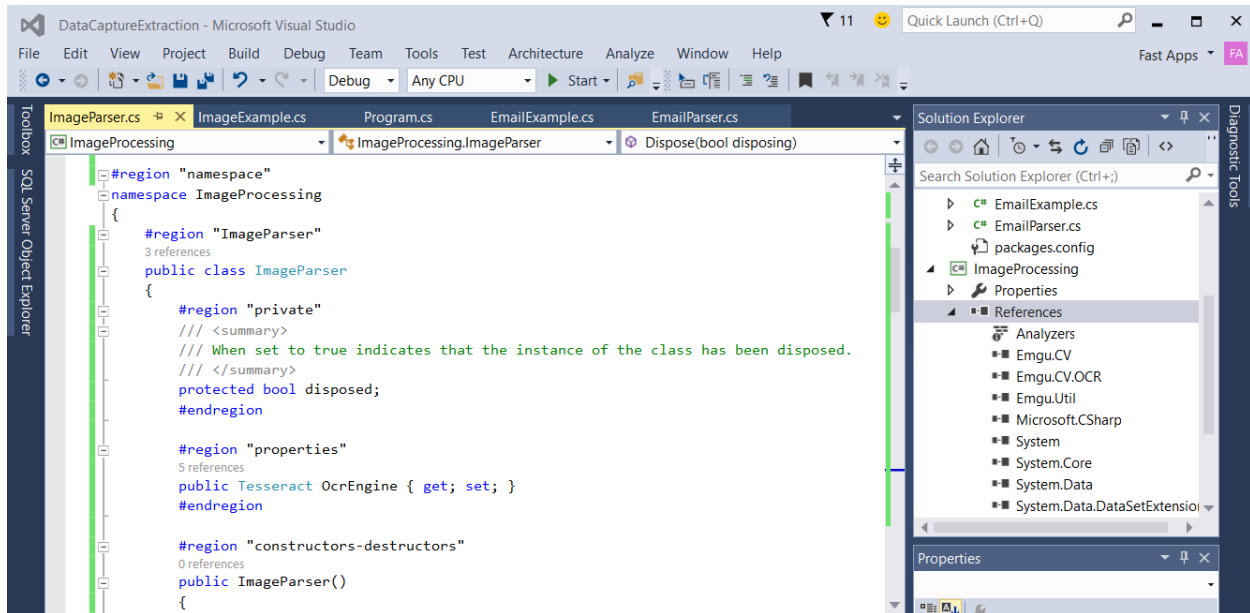


Figure 6: Visual Studio Project with EmguCV DLLs

Parsing screenshots

Now that we have EmguCV wired up, let's see how we can OCR a screenshot to extract text and words from it.

EmguCV is a wrapper around OpenCV. In order for it to run properly, the OpenCV runtimes need to be copied onto the Output folder of the Visual Studio project (either the Debug or Release folder). These OpenCV runtimes need to be copied: cvextern.dll, msvcp120.dll, msvcr120.dll, and opencv_ffmpeg300.dll.

If these files are not present in the Output folder, an exception will be produced at run time. With this in place, let's have a look how this can be done in code.

Code Listing 9: Performing OCR on a Screenshot with EmguCV

```
using Emgu.CV;
using Emgu.Util;
using Emgu.CV.Structure;
using Emgu.CV.OCR;

namespace ImageProcessing {

    public class ImageParser: IDisposable {

        public Tesseract OcrEngine { get; set; }

        public string[] Words { get; set; }

    }

}
```

```

public void Dispose() { }

public ImageParser(string dataPath, string lang)
{
    OcrEngine = new Tesseract(dataPath, lang,
        OcrEngineMode.TesseractCubeCombined);
}

public string OcrImage(string img)
{
    string res = String.Empty;
    List<string> wrds = new List<string>();

    if (File.Exists(img))
    {
        using (Image<Bgr, byte> i = new Image<Bgr,
            byte>(img)) {
            if (OcrEngine != null)
            {
                OcrEngine.Recognize(i);
                res = OcrEngine.GetText().TrimEnd();

                wrds.AddRange(res.Split(new string[] { " ",
                    "\r", "\n" },
                    StringSplitOptions.
                        RemoveEmptyEntries).ToList());

                this.Words = wrds?.ToArray();
            }
        }
    }

    return res;
}
}

```

The first step in Code Listing 9 depicts creating an instance of a **Tesseract** object within the constructor of the **ImageParser** class.

Two very important parameters get passed on to the **Tesseract** constructor. First, the parameter **dataPath** indicates the path of the Tessdata folder, which contains the **Tesseract** data definitions (located under “C:\Emgu\emgu-cv-windows-universal 3.0.0.2157\bin\tessdata”). And second, the parameter **lang** indicates the language that the OCR engine will attempt to recognize.

Once the **Tesseract** instance has been created, performing OCR on the screenshot is our next step. We do this inside the **OcrImage** method. An **Emgu.CV.Image<TColor, Depth>** instance must be created in order to load the actual screenshot file that OCR will perform on. Next **Emgu.CV.Image** instance is passed as a parameter to the **Recognize** method of the **Tesseract** instance. Once the **Recognize** has finished, the **GetText** method can be invoked, which returns a string of all words and characters found on the screenshot.

We can clean things up by converting the string result from **GetText** into a string array of words assigned to the **Words** property of the **ImageParser** class.

As you can see, performing OCR on a screenshot and extracting text from it can be a relatively simple operation. Setting up EmguCV and making sure the runtime files are in place, with the correct references used, is actually more time consuming than OCR.

Demo program

Standard development practice requires that we have a wrapper class call the **ImageParser** before requesting that the Main Program invoke the wrapper class. This same principal was applied in Chapter 1 with MailKit.

Let's look at how this can be done.

Code Listing 10: Using a Wrapper Class to Invoke ImageParser to OCR a Screenshot.

```
// ImageExample: A wrapper class around ImageParser.

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ImageProcessing
{
    public class ImageExample
    {
        public static string[] GetImageWords()
        {
            List<string> w = new List<string>();

            using (ImageParser ip = new ImageParser(
                @"C:\Emgu\emgucv-windows-universal 3.0.0.2157\bin\tessdata", "eng"))
            {
                if (ip.OcrImage(
                    @"C:\temp\screen_shot.tif")
                    != string.Empty)
                    w.AddRange(ip?.Words.ToList<string>());
            }
        }
    }
}
```

```

        }

        return w.ToArray();
    }
}

// Main Program that invokes ImageExample.

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using ImageProcessing;

namespace DataCaptureExtraction
{
    class Program
    {
        static void Main(string[] args)
        {
            ImageExample.GetImageWords();
        }
    }
}

```

The main program calls **GetImageWords**, which is a static method within the **ImageExample** class. Inside **GetImageWords**, an instance of the **ImageParser** class is created by passing both the location of the Tesseract data folder and the language to be used by the Tesseract engine. Next, the method **OcrImage** is called the result returned as a sting and the words found on the words string array property of the **ImageParser** instance.

Summary

From a programming perspective, using OpenCV and the .NET EmguCV wrapper to extract words from screenshots is relatively easy. We must only make sure that EmguCV is installed so that runtimes are in place and no runtime exceptions are produced.

As with other libraries, OpenCV and EmguCV can be used for much more than OCR screenshots and text and word extraction. We've only scratched the surface of what these libraries offer, so I invite you to further explore each of them and discover what they have to offer beyond OCR. I also recommend that you explore commercial OCR and image-processing platforms available in the market.

Complete demo program source code

Code Listing 11 contains the entire source code for each of the examples we have addressed using EmguCV.

Code Listing 10: Complete Demo Program Source Code with EmguCV

```
// ImageParser: OCR text extraction using EmguCV.

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using Emgu.CV;
using Emgu.Util;
using Emgu.CV.Structure;
using Emgu.CV.OCR;
using System.IO;

namespace ImageProcessing
{
    public class ImageParser: IDisposable
    {
        protected bool disposed;

        public Tesseract OcrEngine { get; set; }
        public string[] Words { get; set; }

        public ImageParser()
        {
            OcrEngine = null;
        }

        public ImageParser(string dataPath, string lang)
        {
            OcrEngine = new Tesseract(dataPath, lang,
                OcrEngineMode.TesseractCubeCombined);
        }

        ~ImageParser()
        {
            this.Dispose(false);
        }

        public string OcrImage(string img)
        {
            string res = String.Empty;
        }
    }
}
```

```

        List<string> wrds = new List<string>();

        if (File.Exists(img))
        {
            using (Image<Bgr, byte> i = new Image<Bgr,
                byte>(img))
            {
                if (OcrEngine != null)
                {
                    OcrEngine.Recognize(i);
                    res = OcrEngine.GetText().TrimEnd();

                    wrds.AddRange(res.Split(new string[] { " ",
                        "\r", "\n" }, StringSplitOptions.
                            RemoveEmptyEntries).ToList());

                    this.Words = wrds?.ToArray();
                }
            }
        }

        return res;
    }

    public virtual void Dispose(bool disposing)
    {
        if (!this.disposed)
        {
            if (disposing)
            {
                if (OcrEngine != null)
                {
                    OcrEngine.Dispose();
                    OcrEngine = null;
                }
            }

            this.disposed = true;
        }
    }

    public void Dispose()
    {
        this.Dispose(true);

        GC.SuppressFinalize(this);
    }
}

```

```

// ImageExample: A wrapper class around ImageParser.

using System;
using System.Collections.Generic;
using System.Linq;

namespace ImageProcessing
{
    public class ImageExample
    {
        public static string[] GetImageWords()
        {
            List<string> w = new List<string>();

            using (ImageParser ip = new
                ImageParser(@"C:\Emgu\emgucv-windows-universal
                    3.0.0.2157\bin\tessdata", "eng"))
            {
                if (ip.OcrImage(
                    @" C:\temp\screen_shot.tif ")
                    != string.Empty)
                    w.AddRange(ip?.Words.ToList<string>());
            }

            return w.ToArray();
        }
    }
}

// Main Program that invokes ImageExample.

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using ImageProcessing;

namespace DataCaptureExtraction
{
    class Program
    {
        static void Main(string[] args)
        {
            ImageExample.GetImageWords();

            Console.ReadLine();
        }
    }
}

```

```
}  
    }  
}
```

The complete Visual Studio project source code can be downloaded from this URL:

<https://bitbucket.org/syncfusiontech/data-capture-and-extraction-with-c-succinctly>

Chapter 3 Extracting Data from the Web

Introduction

The web has emerged as the world's online knowledge base—a seemingly unlimited pool of data and information. The rise of Google as the world's top search engine attests to the public's desire for access to the Internet's resources, and knowing how to extract data from the web is now a powerful asset for both individuals and businesses.

Whether you are a software developer looking for new skills or a business leader looking for a competitive advantage, you will find the data extraction instruction in this chapter helpful in creating new opportunities that you might have never thought possible.

We can extract data from a site using one of two methods: by using an API provided by the site or by scraping the HTML contents of the site. I recommend using a site's API, when it is provided, as the preferred method of data extraction. Scraping data can work if the site does not provide an API, but be aware that in some cases scraping is not allowed because the desired information is protected by copyright.

Furthermore, extracting data from the web using the scraping method might require creating code (WebBot) that acts like an automated browser. We should note here that, from a technical point of view, there is little difference between a “good” and a “bad” (harmful) WebBot. However, the intentions of developers can vary. It is your responsibility to use the code you write and the data you extract in ethical, lawful ways. You should never use the information here to violate copyright law, engage in illegal activities, or disrupt networks. If you do engage in unethical conduct, you alone are responsible for your actions.

Both the ethical and technical considerations of scraping go beyond the scope of this book, but some of those issues are worth noting here before we move forward—for example, using stealth to simulate human patterns, being kind to your information resource providers, avoiding the operation of your WebBot at the same time each day, stopping your WebBot on holidays and weekends, using random fetch delays, and taking no actions that lead to your IP address being banned.

This chapter will focus on working with RestSharp and understanding the concepts behind extracting data by using a site's API.

When a site provides its API, it is acknowledging that the data provided through that API can be consumed following its API terms. In other words, if you do not violate the API guidelines, you may use the data. By contrast, scraping means that you are parsing the HTML and extracting text from a site that does not provide an API, which in turn means you are going into uncharted territory, as the data may or may not be protected by copyright law. Typically, when a site doesn't provide an API, it is because it does not want to share its information.

Understanding REST & HTTP requests

Most sites that provide an API use [REST](#), which allows third party apps and developers to consume their data through HTTP(S).

When using C#, there are several ways to connect to an HTTP(S) end point and make a GET or POST request. We will be using a library called [RestSharp](#), which makes working with any REST API extremely easy.

Figure 7 shows how RestSharp can be installed as a NuGet package.

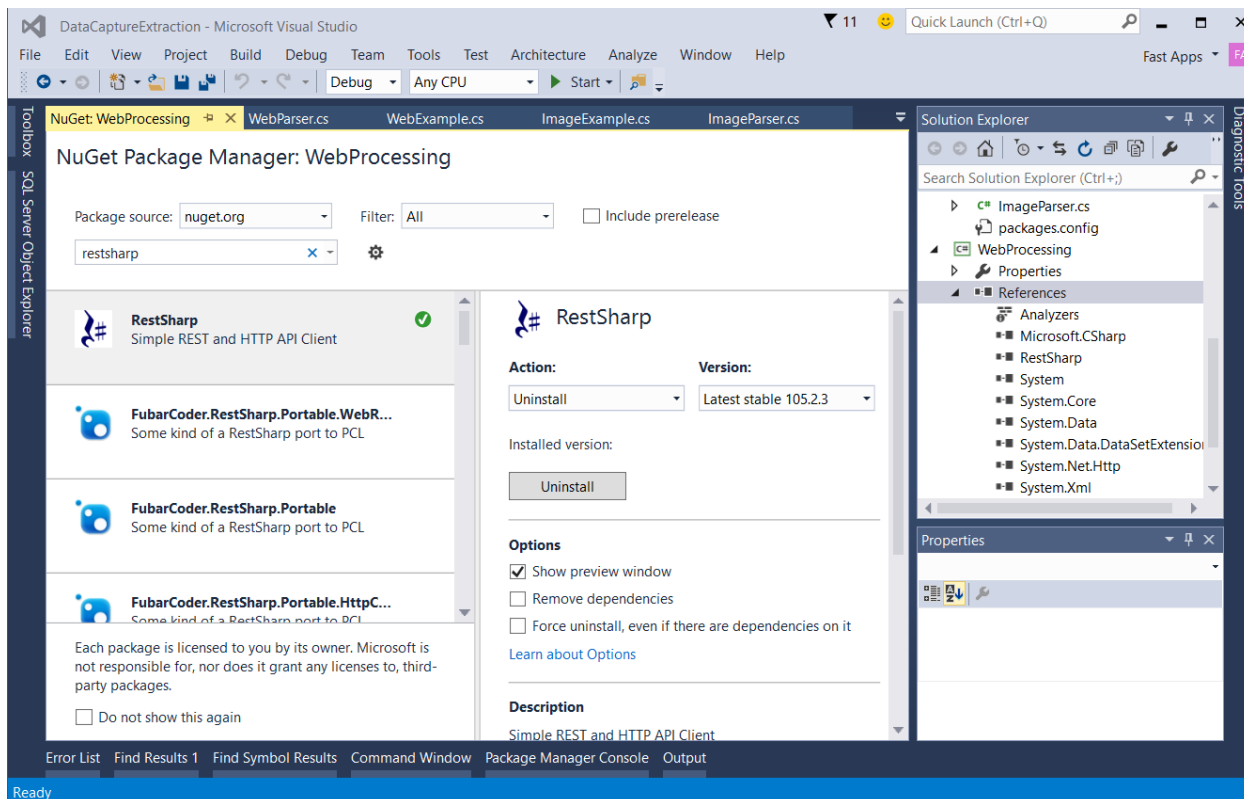


Figure 7: Installing RestSharp as a NuGet Package with Visual Studio 2015

Once RestSharp is installed, a reference is added to the Visual Studio project. Let's now see how to use RestSharp with C#.

Code Listing 12: Creating and Using a RestSharp WebClient

```
// WebParser: Web extraction using RestSharp.

using System;
using RestSharp;

namespace WebProcessing
{
```

```

public class WebParser : IDisposable
{
    protected RestClient WebClient = null;
    protected string response = String.Empty;

    public string Response
    {
        get { return response; }
    }

    public string Request(string resource, Method type,
        string param, string value)
    {
        string res = String.Empty;

        if (WebClient != null)
        {
            var request = new RestRequest(resource, type);
            request.AddParameter(param, value);

            IRestResponse htmlRes = WebClient.Execute(request);
            res = htmlRes.Content;

            if (res != String.Empty)
                response = res;
        }

        return res;
    }

    protected bool disposed;

    public WebParser()
    {
        WebClient = null;
        response = String.Empty;
    }

    public WebParser(string baseUrl)
    {
        WebClient = new RestClient(baseUrl);
    }

    ~WebParser() { this.Dispose(false); }

    public virtual void Dispose(bool disposing)
    {
        if (!this.disposed)
        {

```

```

        if (disposing)
            WebClient = null;
    }
    this.disposed = true;
}

public void Dispose()
{
    this.Dispose(true);
    GC.SuppressFinalize(this);
}
}
}

```

Let's examine the code using *The New York Times* (NYT) APIs as our examples. Inside the **WebParser** constructor an instance of a **RestClient** is created by passing the **baseUri** of the site's API address. By looking at the [Books Best Sellers API documentation](http://api.nytimes.com/svc/books), we can see that the **baseUri** of this API is <http://api.nytimes.com/svc/books>.



BOOKS API: BEST SELLERS

Requests
Responses
Examples
Errors
Feedback
Back to Top
All APIs
API Console

Books API: Best Sellers

With the Best Sellers request type, you can get data from all New York Times best-seller lists, including rank history for specific best sellers. To learn more about the Book Reviews request type, go [here](#).

Note: In this document, curly braces { } indicate required items. Square brackets [] indicate optional items or placeholders.

The Best Sellers Request Type at a Glance	
Base URI	<code>http://api.nytimes.com/svc/books/{version}/lists</code>
Scope	New York Times best-seller lists from June 2008 to present (one-week delay)
HTTP method	GET
Response formats	JSON (.json, default), XML (.xml), serialized PHP (.sphp), JSONP (.jsonp)
Quick links	Requests Responses Examples Errors Feedback

To retrieve best-seller lists, you must [sign up for an API key](#) for the Books API. Usage is limited to 5,000 requests per day (rate limits are subject to change). Please read and agree to the [API Terms of Use](#) and the [Attribution Guidelines](#) before you proceed.

Figure 8: The New York Times Books API baseUri for Best Sellers

This **RestClient** instance, called **WebClient**, will later be used inside the **Request** method in order to execute an HTTP(S) request to a specified API resource.

In order to understand how RestSharp works, we first need to understand how the **Request** method of the **WebParser** class works and what kinds of parameters it deals with. Inside the **Request** method, an instance of **RestRequest** is created. This instance receives two parameters: a **resource** and a **type**. In this case, the **resource** will be `{version}/lists/names.json` (i.e. `v3/lists/names.json`), where `{version}`, according to the documentation at the time of writing, is `v3`, and the type is `GET` (`Method.GET`). Notice that the **resource** `v3/lists/names.json` returns the names of the Books Best Seller lists in JavaScript Object Notation (JSON) format, which we will examine later in this chapter.

You will better understand APIs if you have a look around and experiment with the [API Console](#).

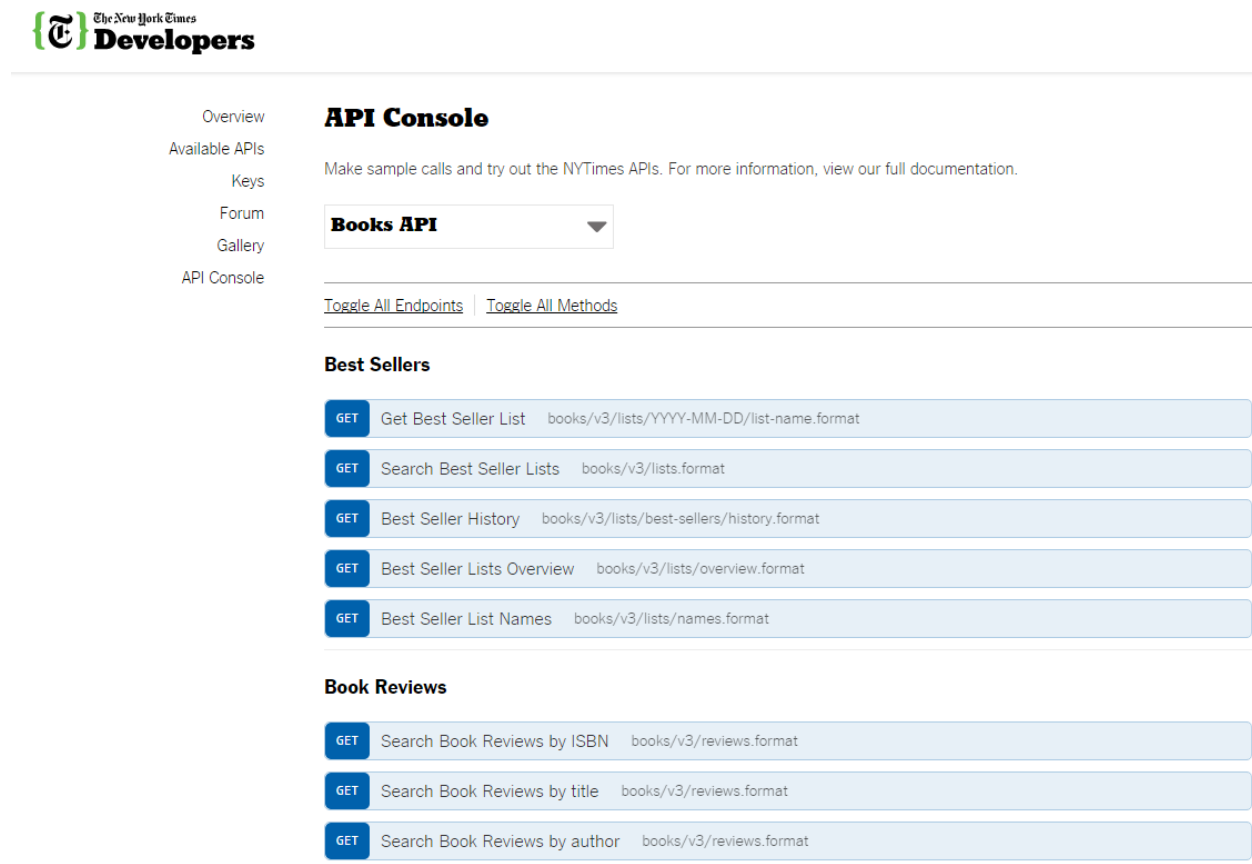


Figure 9: The New York Times Books API Console

With the **RestRequest** instance created, a call to the **AddParameter** method is necessary because the API requires an API key be passed. The API key corresponds to variables **param** and **value**, which are respectively “api-key” and the API key value you receive when registering on <http://developer.nytimes.com> for a specific NYT API.

Invoking the **Execute** method from the **RestClient** instance returns an **IRestResponse**, which is by default in JSON format (and will need to be parsed). For now, we will consider the JSON response a raw string result. Later we will learn how to parse JSON responses properly with another handy .NET library.

Registering for any of the NYT APIs (in order to get an API key) is a straightforward process—you simply need to provide an email address and a password.

Each of *The New York Times* APIs has its own API keys. This means that the Books API key is not the same as Top Stories or Congress API keys. Keep this in mind. *The New York Times* has done a great job making their developer site easy to follow, navigate, and understand.

Having examined how the **WebParser** class works, let's see how it can be invoked and used in order to obtain a raw JSON response using the Books Best Sellers API.

Code Listing 13: Invoking the WebParser Class for the NYT Best Seller Books List Names.

```
// WebExample: A wrapper around WebParser.

using RestSharp;
using System;

namespace WebProcessing
{
    public class WebExample
    {
        public const string cStrNYTBooksBaseUrl =
            "http://api.nytimes.com/svc/books/";
        public const string cStrNYTBooksResource = "v3/lists/names.json";
        public const string cStrNYTApiKeyStr = "api-key";
        public const string cStrNYTApiKeyVal =
            "<<Your NYT Books API key>>";

        public static string GetRawNYTBestSellersListNames()
        {
            string res = String.Empty;

            using (WebParser wp = new WebParser(cStrNYTBooksBaseUrl))
            {
                res = wp.Request(cStrNYTBooksResource, Method.GET,
                    cStrNYTApiKeyStr, cStrNYTApiKeyVal);

                Console.WriteLine(res);
            }

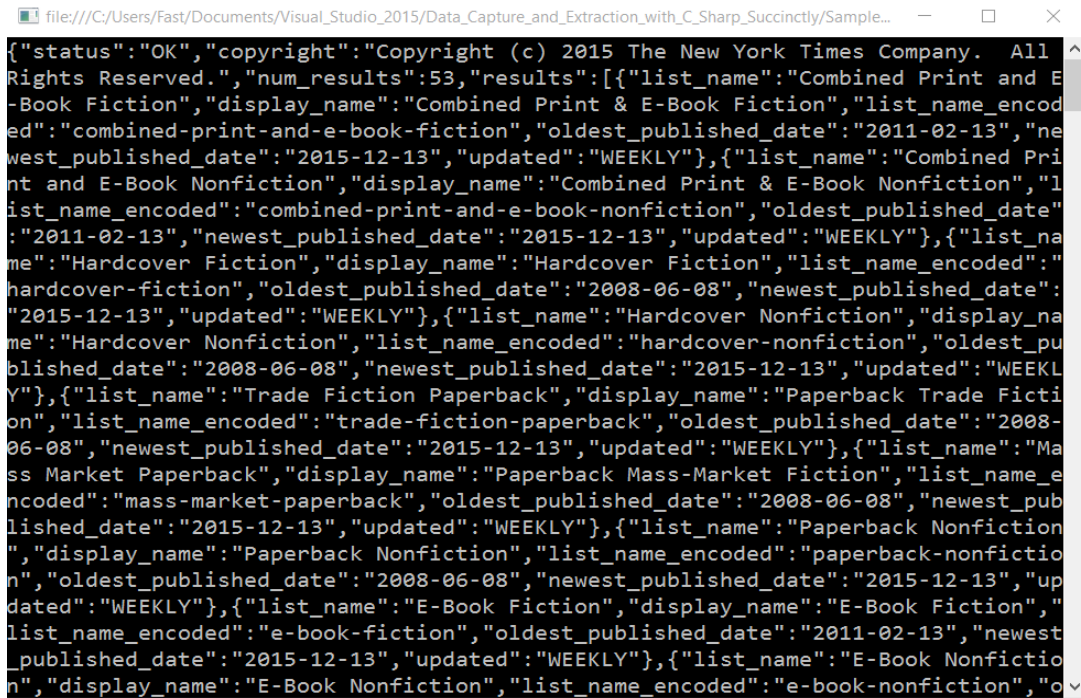
            return res;
        }
    }
}
```

Code Listing 14: Main Program Invoking NYT Best Seller Books List Names

```
// Main Program: Web extraction using the NYT API.

using System;
using WebProcessing;

namespace DataCaptureExtraction
{
    class Program
    {
        static void Main(string[] args)
        {
            WebExample.GetRawNYTBestSellersListNames();
            Console.ReadLine();
        }
    }
}
```



```
{
  "status": "OK",
  "copyright": "Copyright (c) 2015 The New York Times Company. All Rights Reserved.",
  "num_results": 53,
  "results": [
    {
      "list_name": "Combined Print and E-Book Fiction",
      "display_name": "Combined Print & E-Book Fiction",
      "list_name_encoded": "combined-print-and-e-book-fiction",
      "oldest_published_date": "2011-02-13",
      "newest_published_date": "2015-12-13",
      "updated": "WEEKLY"
    },
    {
      "list_name": "Combined Print and E-Book Nonfiction",
      "display_name": "Combined Print & E-Book Nonfiction",
      "list_name_encoded": "combined-print-and-e-book-nonfiction",
      "oldest_published_date": "2011-02-13",
      "newest_published_date": "2015-12-13",
      "updated": "WEEKLY"
    },
    {
      "list_name": "Hardcover Fiction",
      "display_name": "Hardcover Fiction",
      "list_name_encoded": "hardcover-fiction",
      "oldest_published_date": "2008-06-08",
      "newest_published_date": "2015-12-13",
      "updated": "WEEKLY"
    },
    {
      "list_name": "Hardcover Nonfiction",
      "display_name": "Hardcover Nonfiction",
      "list_name_encoded": "hardcover-nonfiction",
      "oldest_published_date": "2008-06-08",
      "newest_published_date": "2015-12-13",
      "updated": "WEEKLY"
    },
    {
      "list_name": "Trade Fiction Paperback",
      "display_name": "Paperback Trade Fiction",
      "list_name_encoded": "trade-fiction-paperback",
      "oldest_published_date": "2008-06-08",
      "newest_published_date": "2015-12-13",
      "updated": "WEEKLY"
    },
    {
      "list_name": "Mass Market Paperback",
      "display_name": "Paperback Mass-Market Fiction",
      "list_name_encoded": "mass-market-paperback",
      "oldest_published_date": "2008-06-08",
      "newest_published_date": "2015-12-13",
      "updated": "WEEKLY"
    },
    {
      "list_name": "Paperback Nonfiction",
      "display_name": "Paperback Nonfiction",
      "list_name_encoded": "paperback-nonfiction",
      "oldest_published_date": "2008-06-08",
      "newest_published_date": "2015-12-13",
      "updated": "WEEKLY"
    },
    {
      "list_name": "E-Book Fiction",
      "display_name": "E-Book Fiction",
      "list_name_encoded": "e-book-fiction",
      "oldest_published_date": "2011-02-13",
      "newest_published_date": "2015-12-13",
      "updated": "WEEKLY"
    },
    {
      "list_name": "E-Book Nonfiction",
      "display_name": "E-Book Nonfiction",
      "list_name_encoded": "e-book-nonfiction",
      "oldest_published_date": "2011-02-13",
      "newest_published_date": "2015-12-13",
      "updated": "WEEKLY"
    }
  ]
}
```

Figure 10: The NYT Best Seller Books List Names Result in Raw JSON.

Parsing JSON responses

Now that we have seen how to obtain raw JSON results, we can best make sense of the JSON data by serializing and structuring it into an object with specific fields. Keep in mind that JSON data is already an object representation of the response, but it is returned as a string, which means it can be serialized to a class with specific properties. An easy way to achieve this is by using the popular [JSON.NET](#) library. JSON.NET can be installed via NuGet.

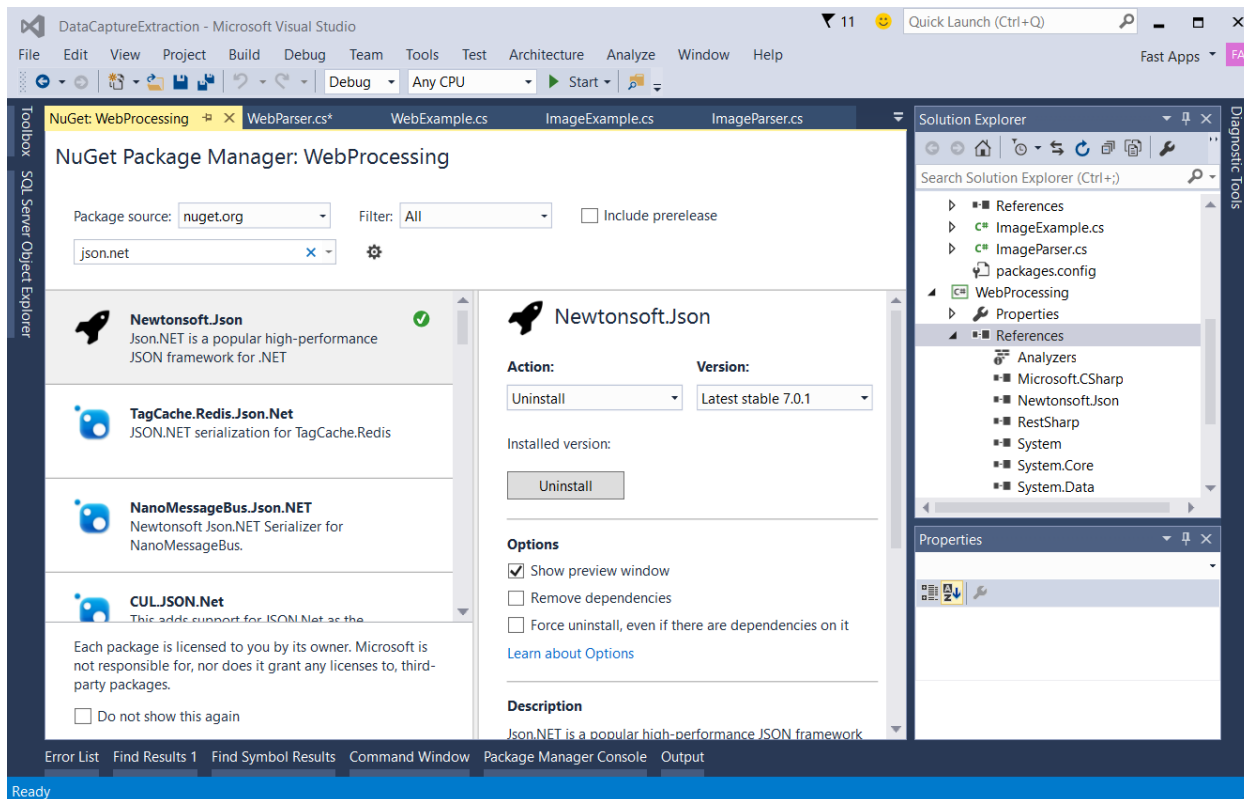


Figure 11: JSON.NET Installed via NuGet.

Instead of treating the returned JSON data as a raw string, let's add specific functionality to the previous code. But first, let's make sure we understand how the actual JSON object is structured.

The raw JSON result returned by **GetRawNYTBestSellersListNames** as a string is actually the JSON object in Figure 12. This can be also inspected by checking the [API Console](#).

```

{
  "status": "OK",
  "copyright": "Copyright (c) 2015 The New York Times Company. All Rights Reserved.",
  "num_results": 53,
  "results": [{
    "list_name": "Combined Print and E-Book Fiction",
    "display_name": "Combined Print & E-Book Fiction",
    "list_name_encoded": "combined-print-and-e-book-fiction",
    "oldest_published_date": "2011-02-13",
    "newest_published_date": "2015-12-13",
    "updated": "WEEKLY"
  }, {
    "list_name": "Combined Print and E-Book Nonfiction",
    "display_name": "Combined Print & E-Book Nonfiction",
    "list_name_encoded": "combined-print-and-e-book-nonfiction",
    "oldest_published_date": "2011-02-13",
    "newest_published_date": "2015-12-13",
    "updated": "WEEKLY"
  }, {
    "list_name": "Hardcover Fiction",
    "display_name": "Hardcover Fiction",
    "list_name_encoded": "hardcover-fiction",
    "oldest_published_date": "2008-06-08",
    "newest_published_date": "2015-12-13",
    "updated": "WEEKLY"
  }, {
    "list_name": "Hardcover Nonfiction",
    "display_name": "Hardcover Nonfiction",
    "list_name_encoded": "hardcover-nonfiction",
    "oldest_published_date": "2008-06-08",
    "newest_published_date": "2015-12-13",
    "updated": "WEEKLY"
  }, {
    "list_name": "Trade Fiction Paperback",
    "display_name": "Paperback Trade Fiction",

```

Figure 12: JSON Object Returned by NYT Best Seller Books List Names.

We can quickly see that **list_name**, **display_name**, **list_name_encoded**, **oldest_published_date**, **newest_published_date**, and **update** are actual properties we are interested in.

Let's deserialize this JSON response into a C# object with JSON.NET.

Code Listing 15: Using JSON.NET to Deserialize a JSON Object into a C# Class

```

public T DeserializeJson<T>(string res)
{
    return JsonConvert.DeserializeObject<T>(res);
}

```

Code Listing 15 shows that **JsonConvert** is a JSON.NET class that de-serializes a JSON object into a C# class. The C# class is represented by **T** in the preceding code.

T is a generic class type, but in order for JSON.NET to be able to de-serialize it properly, it needs to be defined with the same properties as contained in the JSON object response. Let's see how this can be done.

Code Listing 16: C# Classes Required for Deserializing the JSON Response

```
public class BestSellersListNames
{
    public string status;
    public string copyright;
    public int num_results;
    public BestsellerListNamesItem[] results;
}

public class BestsellerListNamesItem
{
    public string list_name;
    public string display_name;
    public string list_name_encoded;
    public DateTime oldest_published_date;
    public DateTime newest_published_date;
    public string updated;
}
```

Based on Code Listing 16, method **GetRawNYTBestSellersListNames** will next be adapted to parse the JSON response.

Code Listing 17: *GetRawNYTBestSellersListNames* Re-adapted to Parse JSON

```
public static BestSellersListNames GetNYTBestSellersListNames()
{
    BestSellersListNames bs = null;

    using (WebParser wp = new WebParser(cStrNYTBooksBaseUrl))
    {
        string res = wp.Request(cStrNYTBooksResource, Method.GET,
                                cStrNYTApiKeyStr, cStrNYTApiKeyVal);

        bs = wp.DeserializeJson<BestSellersListNames>(res);
    }

    return bs;
}
```

The key to being able to parse JSON responses successfully with JSON.NET is to have a C# equivalent (defined as one or more classes) with the same properties that are returned within the JSON response object. Be sure to use the same property names and the same data types, although you can name the C# class itself anything that suits. Notice that the names of the properties of the returned JSON object and the properties of the equivalent C# class are the same. Also notice that the data types are the same.

A JSON string property requires a string C# string property, while a JSON date property requires a C# datetime equivalent. A JSON array property with sub-properties (i.e. results) requires a C# array equivalent (i.e. **BestSellerListNamesItem**) with the same subproperties names and data types.

Demo program

Now that we have addressed how to parse JSON responses, let's create a quick program to clearly display the obtained results.

Listing 18: Displaying Deserialized JSON Response Properties

```
// WebExample - A wrapper around WebParser.

using RestSharp;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace WebProcessing
{
    public class BestSellersListNames
    {
        public string status;
        public string copyright;
        public int num_results;
        public BestSellerListNamesItem[] results;
    }

    public class BestSellerListNamesItem
    {
        public string list_name;
        public string display_name;
        public string list_name_encoded;
        public DateTime oldest_published_date;
        public DateTime newest_published_date;
        public string updated;
    }

    public class WebExample
    {
        public const string cStrNYTBooksBaseUrl =
            "http://api.nytimes.com/svc/books/";
        public const string cStrNYTBooksResource = "v3/lists/names.json";
        public const string cStrNYTApiKeyStr = "api-key";
        public const string cStrNYTApiKeyVal =
```

```

"<<Your NYT Books API key>>";

public static void DisplayNYTBestSellersListNames()
{
    BestSellersListNames b = GetNYTBestSellersListNames();

    if (b != null && b.results.Length > 0)
    {
        foreach (BestSellerListNamesItem i in b.results)
        {
            Console.WriteLine("Name: " + i.display_name);
            Console.WriteLine("Oldest Published Date: " +
                i.oldest_published_date.ToString());
            Console.WriteLine("Newest Published Date: " +
                i.newest_published_date.ToString());
        }
    }
}

public static BestSellersListNames GetNYTBestSellersListNames()
{
    BestSellersListNames bs = null;

    using (WebParser wp = new WebParser(cStrNYTBooksBaseUrl))
    {
        string res = wp.Request(cStrNYTBooksResource,
            Method.GET, cStrNYTApiKeyStr, cStrNYTApiKeyVal);

        bs = wp.DeserializeJson<BestSellersListNames>(res);
    }

    return bs;
}
}

// Main Program: Web extraction using the NYT API.

using System;
using WebProcessing;

namespace DataCaptureExtraction
{
    class Program
    {
        static void Main(string[] args)
        {
            WebExample.DisplayNYTBestSellersListNames();
            Console.ReadLine();
        }
    }
}

```



```
    }  
  }  
}
```

The method **DisplayNYTBestSellersListNames** calls **GetNYTBestSellersListNames**, which gets the deserialized JSON response as a **BestSellersListNames** object. It is through this **BestSellersListNames** object that the **results** property is iterated. For each object within **results**, the subproperties **display_name**, **oldest_published_date**, and **newest_published_date** are displayed.

Summary

APIs can be accessed on hundreds of sites, and they can perform a wide variety of functions that go far beyond retrieving information. A great resource for finding other sites that offer APIs is [Programmable Web](#).

We've seen how RestSharp can be used to invoke an online source of information such as the NYT Books API and also on any other platform that provides a REST API that can be queried using HTTP(S).

A final word on scraping: this subject deserves its own book, but if you are interested in learning more, I strongly advise you to first look into retrieving information via legitimate APIs rather than attempting to extract raw HTML contents from any site.

Complete demo program source code

Code Listing 19 contains the entire source code for all the examples we have addressed using RestSharp.

Code Listing 19: Complete Demo Program Source Code Using RestSharp

```
// Main Program: Web extraction using the NYT API.  
  
using System;  
using WebProcessing;  
  
namespace DataCaptureExtraction  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            WebExample.DisplayNYTBestSellersListNames();  
        }  
    }  
}
```

```

        Console.ReadLine();
    }
}

// WebExample - A wrapper around WebParser.

using RestSharp;
using System;

namespace WebProcessing
{
    public class BestSellersListNames
    {
        public string status;
        public string copyright;
        public int num_results;
        public BestsellerListNamesItem[] results;
    }

    public class BestsellerListNamesItem
    {
        public string list_name;
        public string display_name;
        public string list_name_encoded;
        public DateTime oldest_published_date;
        public DateTime newest_published_date;
        public string updated;
    }

    public class WebExample
    {
        public const string cStrNYTBooksBaseUrl =
            "http://api.nytimes.com/svc/books/";
        public const string cStrNYTBooksResource = "v3/lists/names.json";
        public const string cStrNYTApiKeyStr = "api-key";
        public const string cStrNYTApiKeyVal =
            "<<Your NYT Books API key>>";

        public static string GetRawNYTBestSellersListNames()
        {
            string res = String.Empty;

            using (WebParser wp = new WebParser(cStrNYTBooksBaseUrl))
            {
                res = wp.Request(cStrNYTBooksResource, Method.GET,
                    cStrNYTApiKeyStr, cStrNYTApiKeyVal);

                Console.WriteLine(res);
            }
        }
    }
}

```

```

    }

    return res;
}

public static void DisplayNYTBestSellersListNames()
{
    BestSellersListNames b = GetNYTBestSellersListNames();

    if (b != null && b.results.Length > 0)
    {
        foreach (BestSellerListNamesItem i in b.results)
        {
            Console.WriteLine("Name: " + i.display_name);
            Console.WriteLine("Oldest Published Date: " +
                i.oldest_published_date.ToString());
            Console.WriteLine("Newest Published Date: " +
                i.newest_published_date.ToString());
        }
    }
}

public static BestSellersListNames GetNYTBestSellersListNames()
{
    BestSellersListNames bs = null;

    using (WebParser wp = new WebParser(cStrNYTBooksBaseUrl))
    {
        string res = wp.Request(cStrNYTBooksResource,
            Method.GET, cStrNYTApiKeyStr, cStrNYTApiKeyVal);

        bs = wp.DeserializeJson<BestSellersListNames>(res);
    }

    return bs;
}
}

// WebParser: Web extraction using RestSharp.

using System;
using RestSharp;
using Newtonsoft.Json;

namespace WebProcessing
{
    public class WebParser : IDisposable
    {

```

```

protected RestClient WebClient = null;
protected string response = String.Empty;

public string Response
{
    get { return response; }
}

public string Request(string resource, Method type, string param,
string value)
{
    string res = String.Empty;

    if (WebClient != null)
    {
        var request = new RestRequest(resource, type);
        request.AddParameter(param, value);

        IRestResponse htmlRes = WebClient.Execute(request);
        res = htmlRes.Content;

        if (res != String.Empty)
            response = res;
    }

    return res;
}

public T DeserializeJson<T>(string res)
{
    return JsonConvert.DeserializeObject<T>(res);
}

protected bool disposed;

public WebParser()
{
    WebClient = null;
    response = String.Empty;
}

public WebParser(string baseUrl)
{
    WebClient = new RestClient(baseUrl);
}

~WebParser()
{
    this.Dispose(false);
}

```

```

    }

    public virtual void Dispose(bool disposing)
    {
        if (!this.disposed)
        {
            if (disposing)
            {
                WebClient = null;
            }
            this.disposed = true;
        }
    }

    public void Dispose()
    {
        this.Dispose(true);
        GC.SuppressFinalize(this);
    }
}

```

The complete Visual Studio project source code can be downloaded from this URL:

<https://bitbucket.org/syncfusiontech/data-capture-and-extraction-with-c-succinctly>

Chapter 4 Extracting Meaning from Text

Introduction

In previous chapters we've seen how text can be extracted from emails, screenshots, and the web. We have examined how emails with a particular set of words can trigger an automatic response in an attempt to simulate that the email context was somewhat understood (in a very basic and rudimentary way). Basically, given a particular set of words, emails are checked for a match and are sent back the response that would correlate to the match found.

It is one thing to extract text but quite another to understand what that text means in a given context. Extracting meaning from text is a broad field and falls mostly into [Natural Language Processing](#) (NLP), which includes subcategories such as Sentiment Analysis, Named Entity Recognition, and Relationship Extraction.

Sentiment Analysis consists of analyzing the set of words found within text and determining, based on particular rules, a polarity about a specific subject. For example, when analyzing the words in an email, Sentiment Analysis can be used to determine the tone of the email, e.g., polite, rude, adequate, offensive, negative, positive, etc. The extracted words are measured against standards that determine their sentiment.

Named Entity Recognition consists of determining how words map to proper names, i.e. places, people, organizations, etc. Capitalization of words also helps identify particular entities in some languages. It is also possible that two or more words might be used to identify a single entity in some languages, e.g., Las Ventas (Bull Arena in Madrid, Spain).

Relationship Extraction consists of determining the relationship between two or more entities, e.g., which person is the main subject of a news article or who is married to whom.

NLP is not limited to these three areas. It has a very broad scope. These three are just a few of the most common categories in which NLP is widely used and applied. In this chapter we will focus on the first two: Sentiment Analysis and Named Entity Recognition.

Being able to use NLP to make sense of text is a great asset for any developer, and, from a business perspective, this knowledge can help companies streamline and improve business processes such as automation of personnel recruitment (resume/ CV parsing and profile matching), classification and categorization, and email spam detection.

By the end of this chapter, you should be able to use some of these techniques in order to perform Naïves Bayesian Classification and Named Entity Recognition on any given text using C#.

Understanding contextualization

In order to categorize or classify items into particular categories, we must understand how particular words fit into a certain context. For our purposes, context is the part of a written or spoken statement that precedes or follows a specific word or passage, usually influencing its meaning or effect. Contextualization is the backbone of Sentiment Analysis, Text Classification and Categorization.

Text Classification and Categorization can be used to categorize a document or block of text. There are three commonly used techniques: Naïves Bayesian Classification, Vector Machines, and Semantic Indexing. In this chapter we will only focus on the first.

Naïves Bayesian is a machine learning technique that can predict to which category a particular data case belongs. It is accurate, robust, and relatively easy to implement. Naïves Bayesian was invented by British mathematician Thomas Bayes. In essence, we want to know $P(A|X)$, usually read as “the probability of A given independent variable values X,” where X is one or more attributes. The term “naïve” in Naïves Bayes indicates that all X attributes are assumed to be mathematically independent.

In order to visualize this more clearly, let’s look at it as a mathematical equation in which P represents probability.

$$P(A|X) = (P(X|A) * P(A)) / PP(X)$$

Imagine a scenario in which you have three groups of people (Groups 1, 2 and 3) involved in a project, and each group is tackling a subset of the same project. Group 1 is working on 50% of the modules (attribute 1) and is producing a 3% flaw rate (attribute F). Group 2 is spending 30% on project management (attribute 2) and runs a 4% chance of failure (attribute F). Group 3 is doing 20% of support (attribute 3) and, because the project is mostly live, its chance of failing to find a solution to a reported problem is roughly 5% (exposure to a flaw—attribute F). Our question: what would be the partial probability of running into a flaw (attribute F) in general for all the groups (attributes 1, 2 and 3)?

The partial probability (PP) of running into a flaw (attribute F) in strict mathematical terms for all the groups would be as follows:

$$PP(X) = P(1)*P(1|F) + P(2)*P(2|F) + P(3)*P(3|F)$$

Where $P(1|F)$ can be read as “the probability of Group 1 running into a flaw.”

$$PP(X) = (50\%)*(3\%) + (30\%)*(4\%) + (20\%)*(5\%) = 3.7\%$$

$PP(X)$ implies a 3.7% chance of running into a flaw throughout the whole group. This 3.7% represents PP in the Naïves Bayes equation.

However, the power of classification uncertainty requires us to calculate the probability that a randomly selected flaw might have come from any of the groups. Using the Naïves Bayes equation, we would get the following calculation for Group 1:

$$P(1|F) = ((3\%) * (50\%)) / 3.7\% = 40.5\%$$

For Group 2: $P(2|F) = ((4\%) * (30\%)) / 3.7\% = 32.5\%$

For Group 3: $P(3|F) = ((5\%) * (20\%)) / 3.7\% = 27\%$

This means that there is a 40.5% chance that a randomly selected flaw might come from Group 1, a 32.5% chance of such a random flaw coming from Group 2, and from Group 3 that chance is 27%.

Let's now work with the following data set in a database table.

Department	Gender	Age	Role
Finance	Female	32	Assistant Controller
Finance	Female	36	Senior Controller
Finance	Male	46	Finance Director
IT	Male	40	IT Manager
Finance	Male	30	Financial Lead

Table 4: Sample Data Table

If we want to determine the probability of finding a female (A) given attributes X (where X specifies Department = finance, Age = less than 40, Role = senior), the equation would look like this:

$$P(A|X) = [P(X|A) * P(A)] / PP(X)$$

In this case, our groups are male and female.

Breaking X into each attribute, we get the following:

$$P(A|X) = [P(\text{finance} | A) * P(<40 | A) * P(\text{senior} | A) * P(A)] / PP(X)$$

$$P(\text{female} | X) = [P(\text{finance} | \text{female}) * P(<40 | \text{female}) * P(\text{senior} | \text{female}) * P(\text{female})] / [PP(\text{female} | X) + PP(\text{male} | X)]$$

Because PP(X) considers all groups, the denominator becomes PP(female | X) + PP(male | X). In this example, because there are only two occurrences in the column Gender, the only two possible groups are female and male.

With our knowledge of the previous example, we should already know the probability of the X (attribute F). However, in this case we need to determine each one. Hence:

$$P(\text{finance} | \text{female}) = \text{count}(\text{finance} \& \text{female}) / \text{count}(\text{female})$$

$$P(<40 | \text{female}) = \text{count}(<40 \& \text{female}) / \text{count}(\text{female})$$

$$P(\text{senior} | \text{female}) = \text{count}(\text{senior} \& \text{female}) / \text{count}(\text{female})$$

$$P(\text{female}) = \text{count}(\text{female}) / \text{count}(\text{gender})$$

$$PP(\text{female} | X) = P(\text{finance} | \text{female}) * P(<40 | \text{female}) * P(\text{senior} | \text{female}) * P(\text{female})$$

Because we need to determine $P(\text{male} | X)$, the same calculations made for female must be made for male for each attribute X .

$$P(\text{finance} | \text{male}) = \text{count}(\text{finance} \& \text{male}) / \text{count}(\text{male})$$

$$P(<40 | \text{male}) = \text{count}(<40 \& \text{male}) / \text{count}(\text{male})$$

$$P(\text{senior} | \text{male}) = \text{count}(\text{senior} \& \text{male}) / \text{count}(\text{male})$$

$$P(\text{male}) = \text{count}(\text{male}) / \text{count}(\text{gender})$$

$$PP(\text{male} | X) = P(\text{finance} | \text{male}) * P(<40 | \text{male}) * P(\text{senior} | \text{male}) * P(\text{male})$$

There is always a possibility that one of these calculations would give zero as a result. For example:

$$P(>50 | \text{female}) = \text{count}(>50 \& \text{female}) / \text{count}(\text{female})$$

Because there are no females above the age of 50 in our dataset, the result of this equation would be zero. It is not good when a joint count is 0. In order to avoid this situation, you simply add 1 to all joint counts. Although this might look like a devious trick, it has a solid mathematical basis and is known as add-one smoothing, which is a specific type of [Laplacian](#) smoothing.

With smoothing, the previous equations would appear as follows:

$$P(\text{finance} | \text{female}) = \text{count}(\text{finance} \& \text{female}) + 1 / \text{count}(\text{female}) + 3$$

$$P(>50 | \text{female}) = \text{count}(>50 \& \text{female}) + 1 / \text{count}(\text{female}) + 3$$

$$P(\text{senior} | \text{female}) = \text{count}(\text{senior} \& \text{female}) + 1 / \text{count}(\text{female}) + 3$$

Smoothing can be resumed and applied as follows:

$$P(X|A) = \text{count}(X|A) + 1 / \text{count}(A) + \text{number of } X \text{ attributes}$$

Notice that smoothing is only applied to calculations that involve counts.

As we have seen, logically organizing the extracted text content (words) is important. The data set presented in Table 4 was extracted as text and organized into a logical structure (database table). We do this so that Naïves Bayes can be applied successfully.

Let's recap here. Naïves Bayes can be calculated as follows:

$$P(\text{female} | X) = [PP(\text{female} | X)] / [PP(\text{female} | X) + PP(\text{male} | X)]$$

$$PP(\text{female} | X) = P(\text{finance} | \text{female}) * P(<40 | \text{female}) * P(\text{senior} | \text{female}) * P(\text{female})$$

$$PP(\text{male} | X) = P(\text{finance} | \text{male}) * P(<40 | \text{male}) * P(\text{senior} | \text{male}) * P(\text{male})$$

Where female and male are the groups (described as A) and finance, <40, and senior are attributes (described as X).

With this theory behind us, let's write a simple Naïves Bayes engine in C# that includes add-one smoothing. The engine will require as input the same attributes (X) that the equation expects, and it will require that the groups be evaluated.

Code Listing 20: A Naïves Bayes Engine with Smoothing

```
// Bayes Engine with & without smoothing.

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace TextProcessing
{
    public class BayesEngine : IDisposable
    {
        protected bool disposed;

        public BayesEngine()
        {
            dataset = new List<KeyValuePair<string, string[]>>();
            StrictCounting = false;
        }

        ~BayesEngine()
        {
            this.Dispose(false);
        }

        protected bool ExistsAinXRow(int xRow, string A, string aCol =
            "")
        {
            bool res = false;

            if (dataset != null)
            {
                foreach (KeyValuePair<string, string[]> column in
                    dataset)
                {
                    if (aCol == String.Empty ||
                        column.Key.ToUpper().Contains(aCol.ToUpper()))
                    {
                        if (StrictCounting)
                            res = (column.Value[xRow].ToUpper() ==
```

```

        A.ToUpper()) ? true : false;
    else
    {
        if
        (column.Value[xRow].ToUpper().
        Contains(" "))
            res = (column.Value[xRow].ToUpper().
            Contains(A.ToUpper()))
            ? true : false;
        else
            res = (column.Value[xRow].ToUpper()
            == A.ToUpper()) ? true : false;
    }

    if (res) break;
}
}

return res;
}

public List<KeyValuePair<string, string[]>> dataset = null;
public bool StrictCounting { get; set; }

// Count(finance & male)
public double CountXA(string xAttribute, string A, string xCol =
"", string aCol = "")
{
    double res = 0;

    if (dataset != null)
    {
        foreach (KeyValuePair<string, string[]> xColumn in
            dataset)
        {
            if (xCol == String.Empty ||
                xColumn.Key.ToUpper().
                Contains(xCol.ToUpper()))
            {
                int xRow = 0;
                foreach (string x in xColumn.Value)
                {
                    if (StrictCounting)
                    {
                        if (x.ToUpper() ==
                            xAttribute.ToUpper() &&
                            ExistsAinXRow(xRow, A, aCol))
                            res++;
                    }
                }
            }
        }
    }
}

```

```

    }
    else
    {
        if (x.ToUpper().Contains(" "))
        {
            if (x.ToUpper().Contains
                (xAttribute.ToUpper()) &&
                ExistsAinXRow(xRow, A, aCol))
                res++;
        }
        else
            if (x.ToUpper() ==
                xAttribute.ToUpper() &&
                ExistsAinXRow(xRow, A, aCol))
                res++;
    }
    xRow++;
}
}
}
}

return res;
}

// Count(female, where female is a group)
public double CountA(string A, string col = "")
{
    double res = 0;

    if (dataset != null)
    {
        foreach (KeyValuePair<string, string[]> column in
            dataset)
        {
            if (col == String.Empty ||
                column.Key.ToUpper().
                Contains(col.ToUpper()))
            {
                foreach (string wrd in column.Value)
                {
                    if (StrictCounting)
                    {
                        if (wrd.ToUpper() == A.ToUpper())
                            res++;
                    }
                    else
                    {

```

```

        if (wrd.ToUpper().Contains(" "))
        {
            if
                (wrd.ToUpper().
                 Contains(A.ToUpper()))
                res++;
        }
        else
            if (wrd.ToUpper() == A.ToUpper())
                res++;
    }
}
}

return res;
}

// Count(gender, where gender is a column-i.e. all groups)
public double CountCol(string col)
{
    double res = 0;

    if (dataset != null)
    {
        foreach (KeyValuePair<string, string[]> column in
            dataset)
        {
            if (col != String.Empty && column.Key.ToUpper().
                Contains(col.ToUpper()))
            {
                res = column.Value.Length;
                break;
            }
        }
    }

    return res;
}

// P(male) = count(male) / count(gender)
public double ProbA(string A, string aCol)
{
    double res = 0;

    res = CountA(A, aCol) / CountCol(aCol);

    return res;
}

```

```

}

// P(finance | male) = count(finance & male) / count(male)
public double ProbXA(string xAttribute, string A, string xCol =
"", string aCol = "")
{
    double res = 0;

    res = CountXA(xAttribute, A, xCol, aCol) / CountA(A,
        aCol);

    return res;
}

// P(finance | male) = count(finance & male) + 1 / count(male) +
// 3 (Add-one smoothing)
public double SmoothingProbXA(string xAttribute, string A, int
numAttributes, string xCol = "", string aCol = "")
{
    double res = 0;

    res = (CountXA(xAttribute, A, xCol, aCol) + 1) /
        (CountA(A, aCol) + numAttributes);

    return res;
}

// Decides whether to use ProbXA or SmoothingProbXA.
public double CalcProbXA(bool smoothing, string xAttribute,
string A, int numAttributes = 0, string xCol = "", string aCol =
"")
{
    double res = 0;

    res = ProbXA(xAttribute, A, xCol, aCol);
    res = (res == 0 || smoothing) ?
        SmoothingProbXA(xAttribute, A,
            numAttributes, xCol, aCol) : res;

    return res;
}

// PP(male | X) = P(finance | male) * P(<40 | male) * P(senior |
// male) * P(male)
public double PProbAX(bool smoothing, string A, string[]
xAttributes, string[] xColls, string aCol = "")
{
    double res = 0;

```

```

    if (xAttributes != null && xAttributes.Length > 0)
    {
        int i = 0;

        List<double> rlts = new List<double>();

        foreach (string xAttrrib in xAttributes)
        {
            string xCol = (xColls != null && xColls.Length >
                0 && xColls.Length ==
                xAttributes.Length) ?
                xColls[i] : String.Empty;

            rlts.Add(CalcProbXA(smoothing, xAttrrib, A,
                xAttributes.Length, xCol, aCol));

            i++;
        }

        rlts.Add(ProbA(A, aCol));

        double tmp = 0;
        int cnt = 0;

        foreach (double r in rlts)
        {
            tmp = (cnt == 0) ? r : tmp * r;
            cnt++;
        }

        res = tmp;
    }

    return res;
}

// P(female | X) = [P(finance | female) * P(<40 | female) *
// P(senior | female) * P(female)] /
// [PP(female | X) + PP(male | X)]
public double BayesAX(string A, string[] G, string[] gColls,
    string[] xAttributes, string[] xColls, string aCol = "",
    bool smoothing = true)
{
    double res = 0;

    double nonimator = PProbAX(smoothing, A, xAttributes,
        xColls, aCol);

    double denominator = 0;

```

```

        if (G != null && G.Length > 0 && gColls != null &&
            gColls.Length > 0)
        {
            if (G.Length == gColls.Length)
            {
                int i = 0;
                foreach (string group in G)
                {
                    denominator += PProbAX(smoothing, group,
                                            xAttributes, xColls, gColls[i]);
                    i++;
                }
            }

            if (denominator > 0)
                res = nominator / denominator;

            return res;
        }

        public virtual void Dispose(bool disposing)
        {
            if (!this.disposed)
            {
                if (disposing)
                {
                    dataset = null;
                }
            }
            this.disposed = true;
        }

        public void Dispose()
        {
            this.Dispose(true);

            GC.SuppressFinalize(this);
        }
    }
}

```

In Code Listing 20, the Bayes engine's principal method is **BayesAX**, which calculates the nominator, i.e. $PP(\text{female} \mid X)$. Then the denominator is calculated, i.e. $[PP(\text{female} \mid X) + PP(\text{male} \mid X)]$, which returns the final result, i.e. $P(\text{female} \mid X) = [PP(\text{female} \mid X)] / [PP(\text{female} \mid X) + PP(\text{male} \mid X)]$.

Method **BayesAX** has the following parameters: **A**, **G**, **gCols**, **xAttributes**, **xCols**, **aCol**, and **smoothing**. **A** represents the group (A) for which the probability will be calculated (in the given example, A is female). **G** represents a string array of the of the possible groups (in the given example, these groups are female and male). The parameter **gCols** represents a string array of the name of the columns in which the female and male groups are found (in the given example, the column name for both female and male is Gender). The parameter **xAttributes** represents a string array of the X values for calculating probability (in the given example, finance, senior, and <40). **xCols** is also a string array that represents the name of the columns for the defined X attributes (in the given example, Department, Role, and Age respectively). And finally, **aCol** is the name of the column for the group Gender.

The Bayes engine counts the **xAttributes** per group, calculating the PP(X) for each group and then determining the overall probability. The class **BayesEngine** has a very important property: **StrictCounting**. When set to false (which is the default value), **StrictCounting** indicates that the count will be done on word matches using **Contains()**, whereas, when it is set to true, counting will be done on exact string matches (using the **==** operator). In both cases, string comparison is not case sensitive.

BayesEngine also caters for add-one **smoothing**, which by default is assumed and applied (value set to true). If **smoothing** is set to false, **smoothing** will only be automatically applied when a result joint count is zero. For top accuracy it is recommended to use **smoothing**.

In order to better understand these concepts, let's consider a scenario in which there are two groups: one group of 24 males and one group of 16 females. Job, Handed, and Height are X attributes. The possible values for Job are Admin, Const, Edu, and Tech. The possible values for Handed are Right and Left. The possible values for Height are Short, Tall, and Medium.

We can use this information to create the dataset in Table 5.

Males	Females
Admin = 2	Admin = 7
Const = 5	Const = 0
Edu = 2	Edu = 4
Tech = 15	Tech = 5
Left = 7	Left = 2
Right = 17	Right = 14
Short = 1	Short = 6
Medium = 19	Medium = 8
Tall = 4	Tall = 2

Table 5: Sample Dataset

Using Table 5's dataset, we can calculate $P(\text{male} | X)$ and $P(\text{female} | X)$ to obtain the results in Table 6.

Formula	Result with Smoothing	Result without Smoothing
$P(\text{Edu} \text{Male})$	0.1111	0.0833
$P(\text{Right} \text{Male})$	0.6667	0.7083
$P(\text{Tall} \text{Male})$	0.1852	0.1667
$P(\text{Male})$	0.6	0.6
$P(\text{Edu} \text{Female})$	0.2632	0.2500
$P(\text{Right} \text{Female})$	0.7895	0.8750
$P(\text{Tall} \text{Female})$	0.1579	0.1250
$P(\text{Female})$	0.4	0.4
$PP(\text{Male} X)$	0.008230	0.005903
$PP(\text{Female} X)$	0.013121	0.010938
$P(\text{Male} X)$	0.3855	0.3505
$P(\text{Female} X)$	0.6145	0.6495

Table 6: Results from Preceding Sample Dataset

In order to verify that our **BayesEngine** can produce the same results for this dataset, let's create a wrapper class around it.

Code Listing 21: A Bayes Engine Wrapper around the Sample Dataset

```
// BayesExample: A BayesEngine Wrapper

using System;
using System.Collections.Generic;

namespace TextProcessing
{
    public class BayesExample
    {
        public static void BayesEx()
        {
            using (BayesEngine b = new BayesEngine())
            {
```

```

        b.dataset.Add(new KeyValuePair<string,
string[]>("Gender",
new string[]
{ "male", "male", "male", "male", "male", "male", "male",
  "male", "male", "male", "male", "male",
  "male", "male", "male", "male", "male", "male", "male",
  "male", "male", "male", "male", "male",
  "female", "female", "female", "female", "female",
  "female", "female", "female", "female", "female",
  "female", "female", "female", "female", "female",
  "female" }));

        b.dataset.Add(new KeyValuePair<string, string[]>("Job",
new string[]
{ "tech", "tech", "tech", "tech", "tech", "tech", "tech",
  "tech", "tech", "tech", "tech", "tech",
  "tech", "tech", "tech", "const", "const", "const",
  "const", "const", "admin", "admin", "edu", "edu",
  "admin", "admin", "admin", "admin", "admin", "admin",
  "admin", "edu", "edu", "edu", "edu", "tech",
  "tech", "tech", "tech" }));

        b.dataset.Add(new KeyValuePair<string,
string[]>("Handed",
new string[]
{ "left", "left", "left", "left", "left", "left", "left",
  "right", "right", "right", "right", "right",
  "right", "right", "right", "right", "right", "right",
  "right", "right", "right", "right", "right", "right",
  "left", "left", "right", "right", "right", "right",
  "right", "right", "right", "right", "right", "right",
  "right", "right", "right", "right" }));

        b.dataset.Add(new KeyValuePair<string,
string[]>("Height", new string[]
{ "short", "tall", "tall", "tall", "tall", "medium",
  "medium", "medium", "medium", "medium", "medium",
  "medium", "medium", "medium", "medium", "medium",
  "medium", "medium", "medium", "medium", "medium",
  "medium", "medium", "medium", "short", "short",
  "short", "short", "short", "short", "tall", "tall",
  "medium", "medium", "medium", "medium", "medium",
  "medium", "medium", "medium" }));

// P(male|(edu | right | tall)) with smoothing
double r1 = b.BayesAX("male", new string[] { "male",
  "female" }, new string[] { "Gender", "Gender" },
  new string[] { "edu", "right", "tall" }, new string[]
  { "Prof", "Hand", "Height" },

```

```

        "Gender");

    // P(male|(edu | right | tall)) without smoothing
    double r2 = b.BayesAX("male", new string[] { "male",
        "female" }, new string[] { "Gender", "Gender" },
        new string[] { "edu", "right", "tall" }, new string[]
        { "Prof", "Hand", "Height" },
        "Gender", false);

    // P(female|(edu | right | tall)) with smoothing
    double r3 = b.BayesAX("female", new string[] { "male",
        "female" }, new string[] { "Gender", "Gender" },
        new string[] { "edu", "right", "tall" }, new string[]
        { "Prof", "Hand", "Height" },
        "Gender");

    // P(female|(edu | right | tall)) without smoothing
    double r4 = b.BayesAX("female", new string[] { "male",
        "female" }, new string[] { "Gender", "Gender" },
        new string[] { "edu", "right", "tall" }, new string[]
        { "Prof", "Hand", "Height" },
        "Gender", false);

    Console.WriteLine("P(male|(edu | right | tall)) with
    smoothing: " + r1);

    Console.WriteLine("P(male|(edu | right | tall))
    without smoothing: " + r2);

    Console.WriteLine("P(female|(edu | right | tall)) with
    smoothing: " + r3);

    Console.WriteLine("P(female|(edu | right | tall))
    without smoothing: " + r4);
    }
    }
}

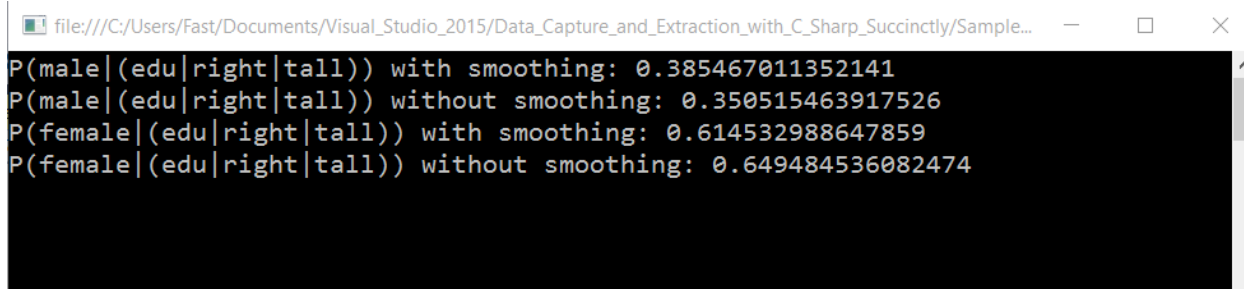
// Main Program that calls BayesExample.
using TextProcessing;

namespace DataCaptureExtraction
{
    class Program
    {
        static void Main(string[] args)
        {
            BayesExample.BayesEx();
        }
    }
}

```

```
}  
}  
}
```

Running this code produces the results shown in Figure 13.



```
file:///C:/Users/Fast/Documents/Visual_Studio_2015/Data_Capture_and_Extraction_with_C_Sharp_Succinctly/Sample...  
P(male|(edu|right|tall)) with smoothing: 0.385467011352141  
P(male|(edu|right|tall)) without smoothing: 0.350515463917526  
P(female|(edu|right|tall)) with smoothing: 0.614532988647859  
P(female|(edu|right|tall)) without smoothing: 0.649484536082474
```

Figure 13: Result from the BayesEngine Wrapper around the Sample Dataset

As you can see, **BayesEx** produces the same results (in fact, slightly more precise results) as were manually calculated and described in Table 5.

Naïves Bayes is fantastic method to determine the probability of running into a given group (A) that has one or more attributes (X). This method can be used a barometer for determining if one or more words (A) belong to certain categories (X), which allows for easier classification and categorization.

Common data types & RegEx

In order to give meaning to text, we must know exactly which string data types are going to be extracted, and we must know how they can be identified and extracted from a set of specific words.

[Regular Expressions](#) (RegEx) is nothing more than a defined sequence of characters that define a search pattern for a particular string data type, e.g., an email address, post code, or anything else that has a specific, formatted pattern.

The most basic RegEx consists of a single literal character, such as 'o.' RegEx will match the first occurrence of that character in the string. For example, if the string is 'John is a pilot,' RegEx will match the 'o' after the 'J.'

RegEx can also match the second 'o' as well. It will do this only when you tell the RegEx engine to search through the string after the first match.

There are 12 characters, called meta-characters, that have special meanings in RegEx, and it is important that you know these items:

- the backslash '\'
- the caret '^'

- the dollar sign '\$'
- the period or dot '.'
- the vertical bar or pipe symbol '|'
- the question mark '?'
- the asterisk or star '*'
- the plus sign '+'
- the opening parenthesis '('
- the closing parenthesis ')'
- the opening square bracket '['
- the opening curly brace '{'

If any of these characters are used as a literal in RegEx, they need to be escaped with a backslash '\' character. If we want to match 1+1=2, the correct RegEx is 1\\+1=2. Otherwise, the '+' sign would have a special meaning. A great tutorial to get you started with RegEx can be found at [RegExOne](#).

Let's quickly explore how we can implement RegEx with C#.

Code Listing 22: A RegEx C# Example

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Text.RegularExpressions;

namespace DataCaptureExtraction
{
    class Program
    {
        public static void RegExExample()
        {
            // First we see the input string.
            string input = "/server/thisUrl.html";

            // Here we call Regex.Match.
            Match match = Regex.Match(input, @"server/([A-Za-z0-9\-\
            ]+)\.html$",
            RegexOptions.IgnoreCase);
```

```

        // Here we check the Match instance.
        if (match.Success)
        {
            // Finally, we get the Group value and display it.
            string key = match.Groups[1].Value;
            Console.WriteLine(key);
        }
    }

    static void Main(string[] args)
    {
        RegExExample();
    }
}

```

Note that running the RegEx “server/([A-Za-z0-9\-\-]+\.)\.html\$” code on this string “/server/thisUrl.html” extracts the word “thisUrl” from the URL string.

Here is some common and useful RegEx code:

User Name: `^[a-z0-9_-]{3,16}$`

this-us3r_n4m3 would be a match. However, a string longer than 16 characters would not match.

Password: `^[a-z0-9_-]{6,18}$`

thisp4ssw0rd would be a match. However, a string shorter than 6 characters would not.

Hex Value: `^#?([a-f0-9]{6}|[a-f0-9]{3})$`

#a3c113 would be a match. However, #h3c113 would not because the letter ‘h’ is included.

Email: `^[a-z0-9_\.-]+\@([\da-z\.-]+\.[a-z\.-]{2,6})$`

vito@vito.me would match. However, vito@vito.someweirddomain would not because it is too long.

Full URL: `^(https?:\W)?([\da-z\.-]+\.[a-z\.-]{2,6})([\Ww \.-]*)*V?$`

<http://subdomain.vito.com/about> would match. However, <http://vito.com/some/page!.html> would not because it contains the ‘!’ character.

IP Address: `^(?:(?:25[0-5]|2[0-4][0-9]||01?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-9]||01?[0-9][0-9]?)$`

71.48.125.121 would match. However, 256.58.125.121 would not.

Let's slightly adjust the code, accommodating for 256.58.125.121 in order to display when a match is produced.

Code Listing 23: Adjusted RegEx C# Example

```
public static void RegExExample()
{
    // First we see the input string.
    string input = " this-us3r_n4m3";

    // Here we call Regex.Match.
    Match match = Regex.Match(input, @"^[a-z0-9_-]{3,16}$",
                                RegexOptions.IgnoreCase);

    // Here we check the Match instance.
    if (match.Success)
    {
        Console.WriteLine(match.Value);
    }
}
```

You may use any of the previous examples with the preceding code in order to verify that the matches provided work as expected.

Other common string data types that need to be extracted from text are emails, post codes, social security numbers, driver's licenses, fiscal identification numbers, bank account numbers, phone numbers, area codes, etc. The site [RegExLib](#) contains a wealth of popular RegEx that can be applied in C# projects. Another useful site is [RexEgg](#).

Identifying entities

[Named Entity Recognition](#) (NER) consists of locating and classifying subsets of text elements (Named Entities) into predefined categories such as the names of persons, organizations, locations, amounts, monetary values, percentages, etc.

Classification of words into categories using a probabilistic method like Naïves Bayes certainly allows us to place a set of words into specific categories based on a trained dataset, but this use of NER remains very basic. In order to have very accurate NER, we must use a well-trained dataset model.

A popular NER implementation for .NET and C# is the [Stanford Named Entity Recognizer](#) (NER) for .NET, which is also available via NuGet.

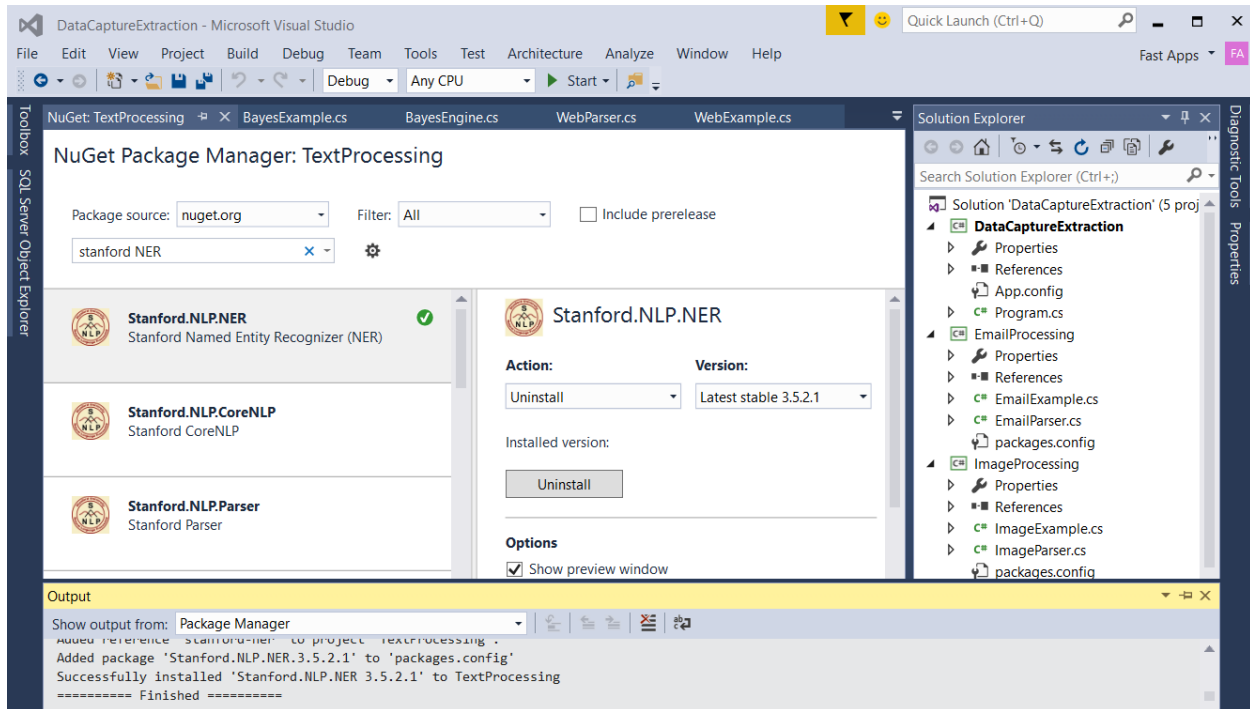


Figure 14: Stanford NER Installed as a NuGet Package

After you install the NuGet package, you will need to download the classifier library definitions, which can be found here: <http://nlp.stanford.edu/software/stanford-ner-2015-04-20.zip>. Further information can also be found here: <http://stanfordnlp.github.io/CoreNLP/download.html>.

When the library definitions have been downloaded, unzip and place the files into a folder on your local hard drive. You will need to reference this location in your code.

Let's examine how to quickly implement NER with the Stanford NER for .NET.

Code Listing 24: A C# Stanford NER Implementation Program

```

// Stanford NER C# Implementation

using System;
using System.Collections.Generic;
using System.Linq;

using edu.stanford.nlp.ie.crf;
using edu.stanford.nlp.pipeline;
using edu.stanford.nlp.util;

namespace TextProcessing
{
    public class NER : IDisposable
    {
        protected bool disposed;
    }
}

```

```

protected CRFClassifier Classifier = null;

protected string[] ParseResult(string txt)
{
    List<string> res = new List<string>();

    string[] tmp = txt.Split(' ');

    if (tmp != null && tmp.Length > 0)
    {
        foreach (string t in tmp)
        {
            if (t.Count(x => x == '/') == 2)
            {
                res.Add(t.Substring(
                    0, t.LastIndexOf("/") - 1));
            }
        }
    }

    return res.ToArray();
}

public NER()
{
    string root = @"D:\Temp\NER\classifiers";
    Classifier = CRFClassifier.getClassifierNoExceptions
        (root + @"\english.all.3class.distsim.crf.ser.gz");
}

~NER()
{
    this.Dispose(false);
}

public string[] Recognize(string txt)
{
    return ParseResult(Classifier.classifyToString(txt));
}

public virtual void Dispose(bool disposing)
{
    if (!this.disposed)
    {
        if (disposing)
        {
            Classifier = null;
        }
    }
}

```

```

        this.disposed = true;
    }

    public void Dispose()
    {
        this.Dispose(true);
        GC.SuppressFinalize(this);
    }
}

// Wrapper class around the Stanford NER Implementation.

using System;

namespace TextProcessing
{
    public class NerExample
    {
        public static void nerExample()
        {
            using (NER n = new NER())
            {
                string[] res =
                    n.Recognize("I went to Stanford,
                               which is located in California");

                if (res != null && res.Length > 0)
                {
                    foreach (string r in res)
                    {
                        Console.WriteLine(r);
                    }
                }
            }
        }
    }
}

// Main Program

using System;
using TextProcessing;

namespace DataCaptureExtraction
{
    class Program
    {
        static void Main(string[] args)
    }
}

```

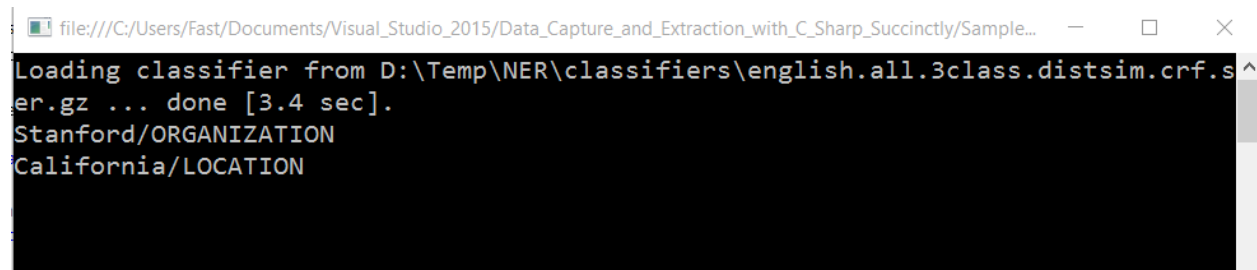
```

    {
        NerExample.nerExample();
    }
}

```

The most important part of the code is the call to **CRFClassifier.getClassifierNoExceptions**, which is where the location of the classifier definitions (english.all.3class.distsim.crf.ser.gz) are physically located on disk.

Within **Recognize**, the **Classifier.classifyToString** method of the Stanford NER is invoked, and the results parsed. This will produce the output we see in Figure 15.



```

file:///C:/Users/Fast/Documents/Visual_Studio_2015/Data_Capture_and_Extraction_with_C_Sharp_Succinctly/Sample...
Loading classifier from D:\Temp\NER\classifiers\english.all.3class.distsim.crf.ser.gz ... done [3.4 sec].
Stanford/ORGANIZATION
California/LOCATION

```

Figure 15: Stanford NER C# Implementation Output

Using the input string “I went to Stanford, which is located in California,” the Stanford NET C# program can recognize two named entities: Stanford (which is an organization) and California (which is a location).

Summary

Extracting meaning from text is a fascinating topic, whether we are examining how to extract specific data types, recognize entities, or classify words within text. When you are able to make sense of extracted data, you have access to a powerful tool that can help you improve, accelerate, and automate business processes. In fact, there is an unlimited potential of processes—from spam filters to text classification and beyond—that organizations can streamline and improve. We’ve only scratched the surface of what is possible with powerful C# code implementations.

Keep in mind that the techniques I have presented in this book are recommended for concept testing rather than production usage. We have focused on quick implementation of what might be achieved from a conceptual point of view, and these techniques do not compete with or undermine any commercial offerings. I encourage you to also consider the diverse range of commercial products that have powerful APIs and are professionally supported.

Thank you for reading. I hope this material helps had broaden your view on data capture and extraction with C#.

The complete Visual Studio project source code can be downloaded from this URL:

<https://bitbucket.org/syncfusiontech/data-capture-and-extraction-with-c-succinctly>