



Customer Success for C# Developers

Succinctly[®]

by Ed Freitas

Customer Success for C# Developers 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.

Iimportant 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: James McCaffrey

Copy Editor: John Elderkin

Acquisitions Coordinator: Hillary Bowling, online marketing manager, Syncfusion, Inc.

Proofreader: Graham High, senior content producer, Syncfusion, Inc.

Table of Contents

The Story behind the <i>Succinctly</i> Series of Books	6
About the Author.....	8
Acknowledgments	9
Introduction	10
Chapter 1 Why Customer Success Matters.....	11
Introduction	11
Isn't customer service for helpdesks only?	11
Blurring the line between dev and support	13
Why you need to wear two hats.....	14
Summary.....	15
Chapter 2 Incident Management.....	16
Introduction	16
Simple, awesome CRM tools.....	17
Setting up DocumentDB	19
Simple CRM code	26
Understanding incidents	43
Find methods	50
Comments and resources.....	59
Changing properties.....	66
Summary.....	68
Chapter 3 Helpdesk Tactics	69
Introduction	69
Steps toward evangelism.....	69
Defining and finding evangelists	70
How to create evangelists.....	71
The art of building trust	72
The power of asking.....	72
Highly empowered language	73
Keep the communication flowing	74
Show appreciation	75
Make a bond	75

Converting incidents into sales	75
Company initiative and culture.....	77
Summary.....	77
Chapter 4 Reflection to the Rescue	79
Introduction	79
What is Reflection?	79
Speed and consistency.....	82
Reflection strategy	88
Pluggable agents	90
Best practices	93
SOA.....	94
Secrets	95
Obfuscation	95
Summary.....	96

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 as 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 a 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, traveling, and life hacking. You can reach him at Edfreitas.me.

Acknowledgments

My thanks to all the people who contributed to this book, especially Hillary Bowling, Tres Watkins, and Graham High, the Syncfusion team that helped make this a reality. Thanks also to manuscript manager Graham High and technical editor James McCaffrey, who thoroughly reviewed the book's organization, code quality, and accuracy. I am also grateful to the close friends who acted as technical reviewers and provided many helpful suggestions for improving the book in areas such as overall correctness, coding style, readability, and implementation alternatives. Thank you all.

Introduction

Happy customers are repeat customers.

If customers feel they have been taken care of and appreciated, they can bring in new business and become loyal advocates of your brand and products. But the high stakes of acquiring customers and keeping them happy should not be underestimated. Whether you are a new employee or you run your own business, the effects of neglecting a customer can turn into a problem from which your organization never recovers. Think of it this way—your success is directly proportional to the success your customers achieve, and they will judge their success not only by your products or services, but also through their perceptions of how you treated them.

Most recurring business opportunities originate from existing customers, the same people who will refer you to others if you provide a good service and address their concerns. In reality, we are all customers of other businesses, and we all like to be treated with the utmost attention and in a timely manner. Why would software developers be any different?

Some software developers believe their role is only to produce high-quality software, with anything that gets in the way merely an obstacle to achieving software-development nirvana. But although this approach might appear logical, it defies the very essence of what businesses are ultimately trying to achieve: increasing their number of happy customers, the ones who will help create more revenue and more recurring business.

By the end of this e-book, you should be able to understand and use the techniques described here to become a customer-success advocate for your employer or your own company while improving your code quality in the process.

If you know C#, this should be a fun e-book to follow along with. You will get a glimpse of what is technically possible and how this relates to customer success. Even though the techniques presented here are not in-depth analyses of each topic, the information will be enough to give you a good head start while helping you acknowledge and become aware of what is feasible through clear, concise, enjoyable, and easy-to-follow examples.

The code samples presented in this e-book were created with Visual Studio 2015 and .NET 4.5.2 and can be found here:

<https://bitbucket.org/syncfusiontech/customer-success-for-c-developers-succinctly>

Have fun!

Chapter 1 Why Customer Success Matters

Introduction

Customers are the lifeblood of any organization. Although many of us tend to believe that customer-service advocacy is a role reserved for talented salespeople, marketers, growth specialists, community managers, and of course the humble helpdesk worker, in fact everyone employed by an organization, from the CEO to the cleaning crew, is a potential customer-service advocate. Even software developers.

Developers must be effective in producing high-quality code and creating systems that make organizations more efficient and leaner, which will improve business processes. As a result, developers are often placed inside a protected bubble and are not exposed to end-users or customers directly.

However, there are organizations out there (often startups) that require every new hire to begin working in the support helpdesk, regardless of the role they were hired to fill. As radical as this might seem compared to a more traditional corporate and hierarchical organization, the fact is that getting people to feel the customers' needs is a great way to get everyone aligned and in tune with the company's vision, mission, and direction.

Customer success matters because it pays the salaries and keeps the business moving. Your boss is ultimately the customer.

For a software developer, customer success should mean fewer bugs going out the door, improved sprints and releases, and a better overall customer experience while using the application and software products.

By the end of this chapter, you—as a developer and part of a larger dev team—should understand why having a customer-oriented mindset will benefit the quality of the code you produce and decrease any future troubleshooting efforts. As a dev team member, you should be able to blur the line between what is considered support and what is considered development (coding). This will allow your team to become more agile and to respond better to customer problems.

Isn't customer service for helpdesks only?

For customers, a helpdesk is the gateway into an organization. It's the customer's entry point for raising concerns and having them dealt with—it is not a shield that tries to deflect issues from other departments (especially the software-development team) as quickly as possible.

The helpdesk's role should be to properly quantify and translate into technical language the information required for the development team to make adjustments or to deliver the correct fixes and solutions to specific problems.

In a perfect world, both the helpdesk and the development team should be able to “talk” using the same technical terms, and both should have a clear understanding of the product and its roadmap. However, this is not always the case.

Table 1 represents a typical flow between the helpdesk and the dev team when an issue is reported.

Table 1: A Typical Helpdesk to Dev Team Flow

1	2	3	4	5	6
Customer reports the issue to the helpdesk.	Helpdesk relays the issue to the dev team.	Dev team assigns a priority to the issue reported.	Helpdesk informs the customer. Dev team works on a fix.	Dev team delivers the fix to the helpdesk.	Helpdesk checks the fix and sends it to the customer.

Items in green usually happen quickly. The item in orange normally takes a bit longer, given that assigning a priority to a reported issue might involve part of the dev team deviating from a current sprint or from the overall roadmap for a period of time. Most dev teams tend to prioritize on the product roadmap over fixing issues.

Typically, item 4 (the dev team working on the fix) will consume the most time within that flow. The dev team might also require additional information and need to request that the helpdesk ask the customer for extra details, delaying the entire process.

This approach is not only lengthy, but it leaves the helpdesk to manage customer expectations before it knows how much time or effort will be required by the dev team to produce a fix and provide a definite solution.

During this “fixing” time, the helpdesk can only inform the customer that they are “following up with the dev team.” In the meantime, the customer sometimes becomes tense and agitated.

Going forward, this juncture can negatively impact how the customer will view the company. If the “fixing” process takes longer than initially anticipated, the customer can feel neglected.

So the question is, why do so many software companies keep using this flow?

The answer is that the product roadmap drives the dev team forward. The entire company has bought into the same vision: releasing new software and/or existing software with more features is what drives the business forward.

We should acknowledge here that in today’s competitive landscape, customers are well informed and able to make quick decisions, which leads to increased corporate competition and pressure to stay in a perpetual hamster-wheel race to release the next big thing or the latest feature in their products.

Of course companies must stay innovative, but this process can often lead to customer success being ignored, and if helpdesk issues are not addressed quickly, companies risk losing even longtime customers. One solution is to blur the line between support and development.

Blurring the line between dev and support

In order to respond more efficiently to customer issues and provide faster fixes, the currently rigid views about how to support the helpdesk and dev team need to be reconsidered.

One simple way to blur the lines is to employ software developers at the helpdesk who are not solely interested in working on R&D projects or under a product management structure. Instead, employ people who prefer to work with internal projects or their own projects. Internal projects can be anything software related that helps the organization operate more efficiently or improves internal business processes.

Many prospective employees who know how to program and write code will enjoy this approach and will want to keep doing it.

Companies often make the mistake of assuming that anyone capable of writing code wants to sit behind an R&D structure and under product management. But there are a lot of software engineers out there who excel when working on their own projects, whether they're internal projects or opportunities to help and mentor others.

Why not tap into this existing talent pool and create a support helpdesk that allows software engineers who are not R&D or product-management oriented to excel in what they love—coding—while also working on internal and individual projects that let them become customer advocates?

If you are a one-person show running a small, independent software vendor (ISV) shop, you will surely be acquainted with the notion of solving critical customer issues and will see it as essential to keeping your reputation and business afloat. You have already blurred the thin line that separates your support and development operations.

However, if you work for an organization that has one or more dev teams within an R&D and product-management structure, it is likely that the helpdesk will be more traditional in that it includes support specialists (who typically do not create code fixes) rather than software developers.

In practical terms, blurring the line between support and development means having the capability within your helpdesk team to provide product fixes or at least workarounds (code solutions) that can keep customers happy while at the same time keeping the dev teams under the R&D and product-management structure focused on the product roadmap.

Table 2 represents what the typical flow between the helpdesk and the dev team looks like when the helpdesk also employs software engineers who might not be interested in working within an R&D and product-management structure.

Table 2: A Bug-Fixing and Solution-Oriented Helpdesk-to-Dev Team Flow

1	2	3	4	5	6
Customer reports the issue to the helpdesk.	Helpdesk reproduces and attempts to fix the issue (code solution).	At least a workaround is provided and given to the customer by the helpdesk.	If the customer requires a full fix (not a workaround), this is given to the dev team.	Because the customer already has a workaround, the issue is not urgent anymore.	When a definite fix is available from the dev team, the helpdesk will deliver it to the customer.

The flow from Table 2 has the same number of steps as Table 1, however there are two subtle differences.

First note that the premise here is that the helpdesk will do whatever it can within its capabilities (it must employ developers) to provide the customer with at least a workaround (even if that means the workaround or temporary fix is a code fix, e.g., a DLL).

This is done in order to minimize the impact on the customer (less anxiety and waiting time) and to allow the dev team under an R&D and product-management structure to focus on the product roadmap.

Also note that the helpdesk will only resort to the dev team if the customer requires a permanent fix or if they explicitly require the fix to come from the dev team (certified by R&D). However, if the customer is happy with the code solution provided by the helpdesk, there will be no need to get the dev team involved (other than to validate and certify the solution provided by the developers employed by the helpdesk).

The helpdesk also wins, as support specialists who are not coders gain know-how, and developers employed by the helpdesk get to work on an internal project (the customer's issue), allowing them to put their creativity and imagination to work.

Remember that the goal is to solve the customer's issue and avoid using the dev team unless strictly necessary.

Why you need to wear two hats

At the end of the day, what customers want are a fast, proactive response and a solution to their problems. They will value that solution no matter where it comes from.

Winning your customers' hearts requires agility. When you engineer your support helpdesk to act as a Customer-Oriented R&D (CORD) by employing software developers as part of the helpdesk, you will not only close issues more quickly, but you keep your core dev teams on schedule.

Table 3 represents the advantages of employing this mentality within your organization's helpdesk.

Table 3: Comparison of Traditional and Customer-Oriented R&D Helpdesks

Traditional Helpdesk	Customer-Oriented R&D Helpdesk
Acts as a middle man.	Acts as a buffer to R&D.
Doesn't get involved with code fixes.	Provides core code fixes or workarounds.
Depends on R&D to fix a core problem.	Doesn't depend on R&D.
Incapable of reproducing a problem.	Capable of reproducing a problem.
Could deviate R&D from the roadmap.	Doesn't deviate R&D from the roadmap.

It's important to understand the huge impact that both approaches can have on your business. When you have a helpdesk that enjoys the challenges of reverse engineering an issue and providing a custom fix (that is not dependent on the dev team under an R&D or product-management structure), your organization suddenly becomes incredibly agile and able to respond more quickly to customer issues while still accelerating development under the product roadmap.

The key differentiator of both approaches is that when operating in the traditional model, apart from configuration problems, your helpdesk will require input from the dev team (for anything code related, which means any time an urgent customer issue involves a code fix, you risk breaking the momentum of the dev team working on a sprint).

With the CORD approach, your helpdesk is empowered by developers who do not necessarily like the hustle of sticking to a rigorous roadmap—instead they love hacking code and coming up with innovative solutions. This allows the dev team to focus exclusively on their roadmap.

Summary

We've seen how making a small change in how we view a traditional software helpdesk could revolutionize your organization, allowing you to be more agile and to respond faster to customer issues while sticking to your roadmap without major deviations.

We've introduced this mindset so that we can next focus on how this approach can be semiautomated with some customer relationship management (CRM) code classes and some reverse engineering.

Later we'll explore tactics and opportunities for customer service that can help grow your business and company value.

Chapter 2 Incident Management

Introduction

Incident management, which can be defined as a series of steps taken to resolve a customer's problem, lies at the heart of customer service. And clear communication is the key to success when managing customer incidents.

Managing expectations—both the customer's and your team's—is the essential element of good communication. In short, your customers should know what to expect after they report a problem (a clear roadmap with an ongoing feedback loop is essential), and your manager should know how much time and effort you will spend on a particular incident.

Managing the expectations of a customer's incident begins with asking if what the user and your boss expect from you is achievable. Before answering, you must also ask yourself what you can reasonably expect from yourself as far as time constraints go.

If what is being asked is unreasonable, you must take a step back and make this clear to all parties (primarily your boss) and get back to the customer with a coordinated explanation of why responding to the current request is not feasible. If, on the other hand, the request is reasonable, you should respond as quickly as possible with a clear time estimation and an indication of the steps involved in solving the customer's problem.

Managing expectations for a customer's incident means understanding the following:

- When a user expects a quick solution, your boss might want you to spend less time on that task and instead ask for a quick fix.
- You must understand your limits and let all parties involved know what is reasonable.
- If you are the main point of contact, you must let people know when you are out.
- If you don't have a full response or resolution immediately at your fingertips, a simple “no worries, I've got this” goes a long way.

Incident management also involves determining how much communication is required, the tone of communication, and, most importantly, the frequency of communication.

In this chapter, we will automate incident management by creating a C# solution that includes expectation management and the communication loop required to successfully close any reported incident. Most helpdesk and CRM tools do not focus on expectation management or the tone and frequency of communication, but these tools should be useful when included in your projects or products directly, and they will allow you to integrate these capabilities out of the box in your company's line of products.

Simple, awesome CRM tools

CRM is the approach taken in order to manage a company's interaction with current and future customers.

Most current CRM software, such as Salesforce and Microsoft Dynamics, focuses more on closing a sales cycle than on incident management itself. Although most CRM software includes a helpdesk subset, none of the CRM tools I've examined (even those built specifically for customer support, e.g., Zendesk) give much importance to expectation management and frequency or tone of communication.

My suggestion is that you offer a C# class responsible for managing expectations around communication, and the frequency and tone of communication related to any reported customer incident. This class can serve as an embeddable, small incident-management-oriented CRM system within your projects or products, allowing customers to report a problem to your CORD helpdesk within the product itself.

Table 4 represents the specs that our Simple Awesome CRM Tool will emphasize.

Table 4: Specs for the Simple Awesome CRM Tool

Allow the user within the product to report an issue to the CORD helpdesk.
Clearly define and redefine the customer's expectations of the issue.
Manage and set the frequency of the communication.
Report back to the user the state and stage of the incident until resolution.
Indicate to the helpdesk worker the need to follow up or provide an update.
Indicate to the user the resource allocated for the task and its availability.

With these thoughts in writing, let's consider how we might code this. Ideally, the Simple Awesome CRM Tool should be able to store its data in the cloud, so that it can be easily stored and retrieved from any location.

When it comes to storing data in the cloud, there are myriad options available using both traditional relational databases (e.g., MySQL, SQL Server, Postgres SQL, Oracle, etc.) and NoSQL (e.g., MongoDB, DynamoDB, RavenDB, etc.) that can be hosted privately or, alternatively, hosted on services like Amazon Web Services (AWS) or Microsoft Azure.

Although setting up any of those relational or nonrelational (NoSQL) databases on AWS or Azure has been greatly simplified in recent years, you must still provision instances and throughput, and you must consider factors such as requests and pricing in order to properly deploy an online data store. That's quite a process and, frankly, a very demanding one.

Our objective is to build a simple CRM tool that can be easily embedded into existing projects and products without causing us to break our heads on deploying an online data store.

Table 5 represents what our data store should allow our Simple Awesome CRM Tool to achieve.

Table 5: Simple Awesome CRM Tool Data Store Specs

Zero maintenance and run out of the box.
Scalable without restrictions on throughput and requests.
Inexpensive, schema-less, excellent security features.
NoSQL (nonrelational) DB but with SQL language capabilities.

Now, wait a second. At first sight it seems we are asking a bit too much. An almost infinitely scalable, schema-less NoSQL DB that allows SQL-compatible querying capabilities? Does this actually exist?

Well, yes, such a marvel does indeed exist. The folks at Microsoft have developed an amazingly flexible and impressive, fully hosted NoSQL document (JSON) database that has SQL querying language capabilities. It is called DocumentDB and can be found at [Microsoft Azure](#).

In order to write the Simple Awesome CRM Tool, we'll use the .NET SDK for DocumentDB, which can be found on [GitHub](#).

Although the code examples shown are quite intuitive for following along, I recommend that you have a look at the excellent DocumentDB online [documentation](#).

According to the specs in [Table 4](#), we will need to create a base C# class that should be able to:

- Allow the customer to report an incident along with their expectations (severity and importance).
- Allow the customer to have one or multiple communication interactions.
- Allow the customer to set a desired frequency level of communication interaction (bidirectional). This should alert the helpdesk to follow up or act.

- Allow the helpdesk to report the state of the incident (as referred to in [Table 2](#)) until resolution, clearly indicating a probable delivery date for the fix.
- Inform the customer clearly of the resources allocated (persons responsible) during each step and status.

Setting up DocumentDB

We'll need to create a DocumentDB instance on Microsoft Azure, and you'll need to sign in or sign up for Azure with a Microsoft account. You can do that by visiting azure.microsoft.com.

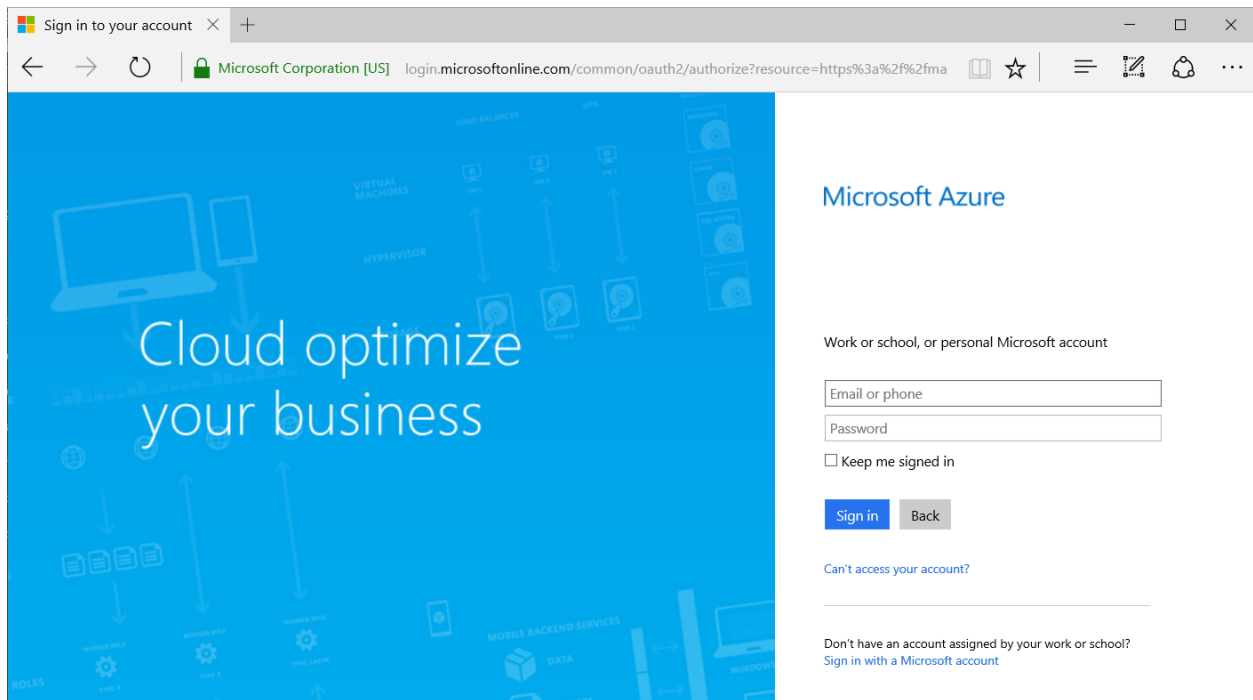


Figure 1: Microsoft Azure Sign-In Screen

Once you've signed up for or signed in to the Azure portal, you can browse through the list of Azure services and select the **DocumentDB Accounts** option.

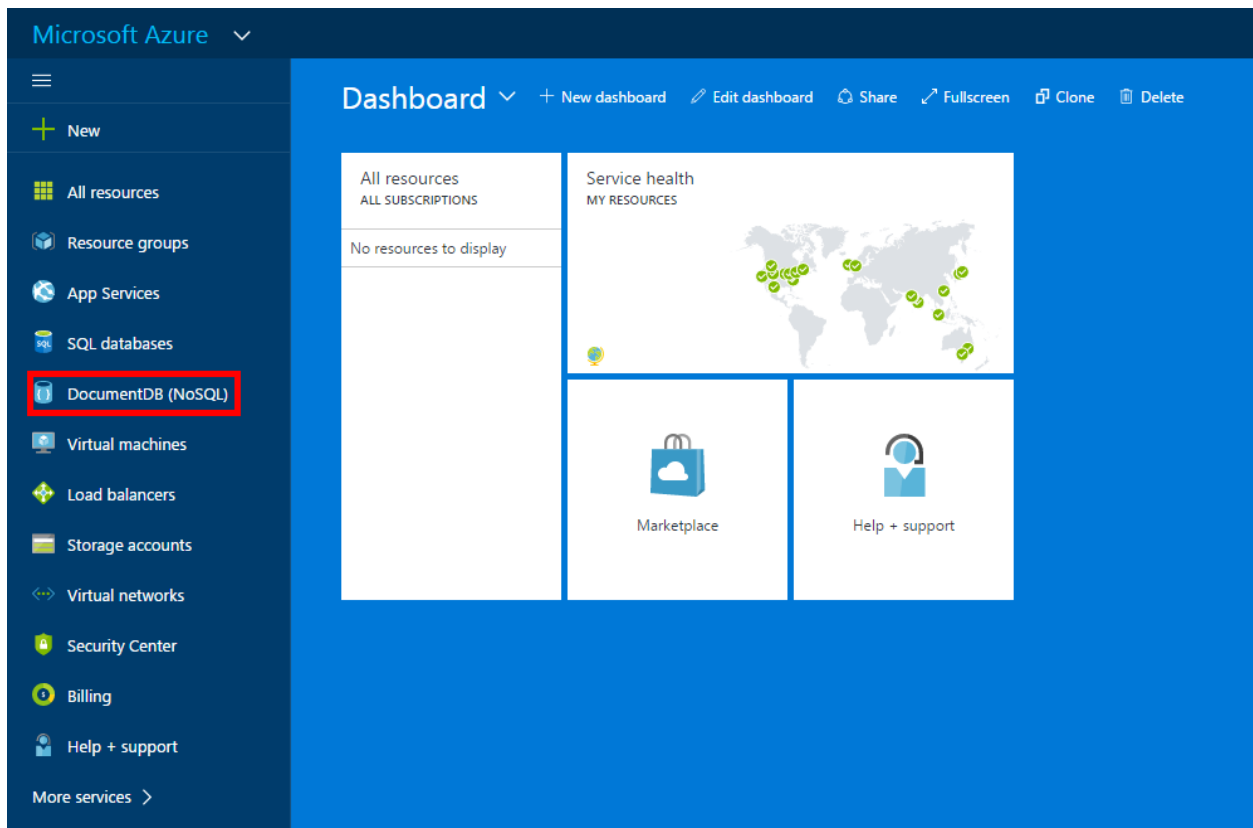


Figure 2: DocumentDB within the List of Azure Services

After you select DocumentDB, you must create a new DocumentDB account by clicking **Add**.

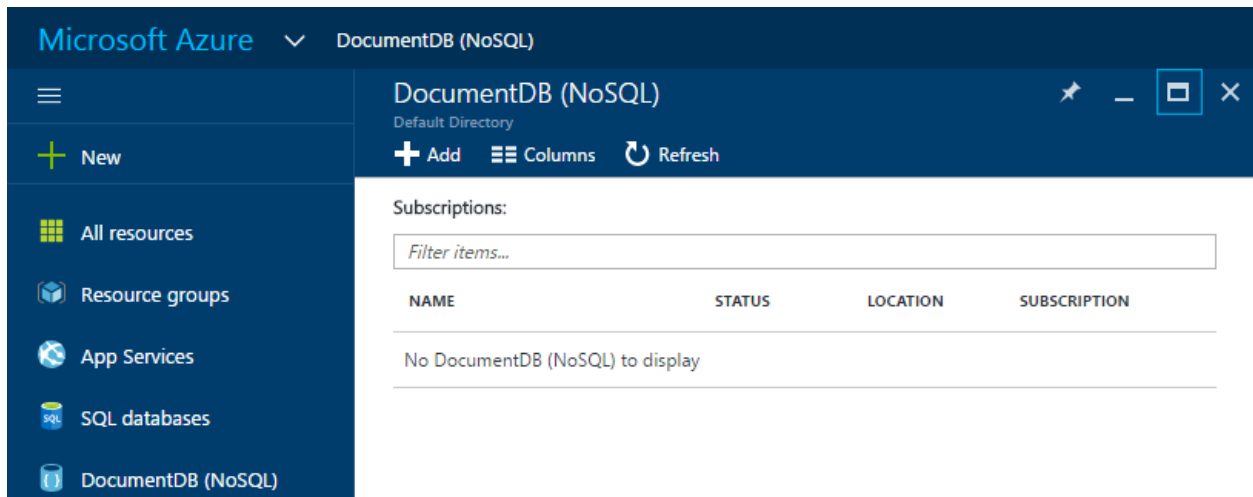


Figure 3: Screen to Add a DocumentDB Account

This will lead to a screen that allows you to enter the details of the DocumentDB account: ID, NoSQL API, Subscription, Resource Group, and Location. You can select the Azure region that will host your account.

The ID is a unique global identifier within Microsoft Azure for the DocumentDB account. To finalize the creation of the account, click **Create**. The DocumentDB account creation process can take a few minutes.

The screenshot displays the Microsoft Azure portal interface for creating a new DocumentDB (NoSQL) account. The left-hand navigation pane lists various Azure services, including All resources, Resource groups, App Services, SQL databases, DocumentDB (NoSQL), Virtual machines, Load balancers, Storage accounts, Virtual networks, Security Center, Billing, and Help + support. The top header shows the Microsoft Azure logo and the breadcrumb navigation: DocumentDB (NoSQL) > DocumentDB (NoSQL). The main content area is titled "DocumentDB (NoSQL) New account" and contains the following configuration fields:

- ID**: A text input field containing "fastapps" with a green checkmark icon on the right. Below the field, the text "documents.azure.com" is visible.
- NoSQL API**: A section with two tabs: "DocumentDB" (selected) and "MongoDB".
- Subscription**: A dropdown menu showing "BizSpark".
- Resource Group**: A section with two radio buttons: "Create new" (selected) and "Use existing". Below the radio buttons is a dropdown menu showing "Default-Storage-EastUS".
- Location**: A dropdown menu showing "East US".

At the bottom of the form, there is a checkbox labeled "Pin to dashboard" which is checked. Below this checkbox is a blue "Create" button and a link labeled "Automation options".

Figure 4: Final DocumentDB Account Creation Screen

Figure 5 depicts how the DocumentDB account will appear after it is created.

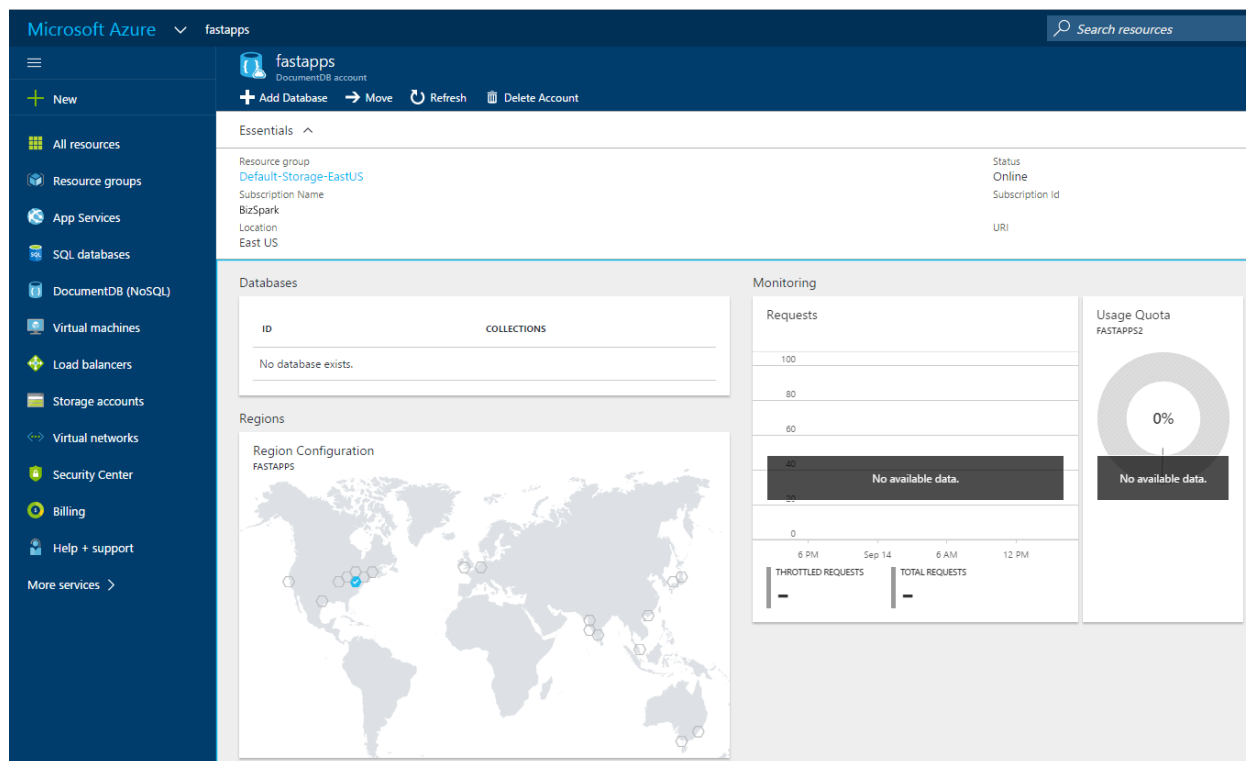


Figure 5: DocumentDB Account Dashboard

A DocumentDB account can host multiple DocumentDB databases, but simply having a DocumentDB account doesn't mean you'll have a DocumentDB database ready for use. However, such a database can be created by clicking **Add Database**.

The only requirement is that you provide an ID for the new DocumentDB database.

The screenshot shows a dark blue 'Add Database' dialog box. It has a title bar with a minus sign, a maximize button, and a close button (X). Below the title bar, there is a red asterisk followed by 'ID' and an information icon. A text input field is present with the placeholder text 'Enter database id'.

Figure 6: New DocumentDB Database Creation Screen

When the DocumentDB has been created, it will be configurable through an intuitive dashboard.

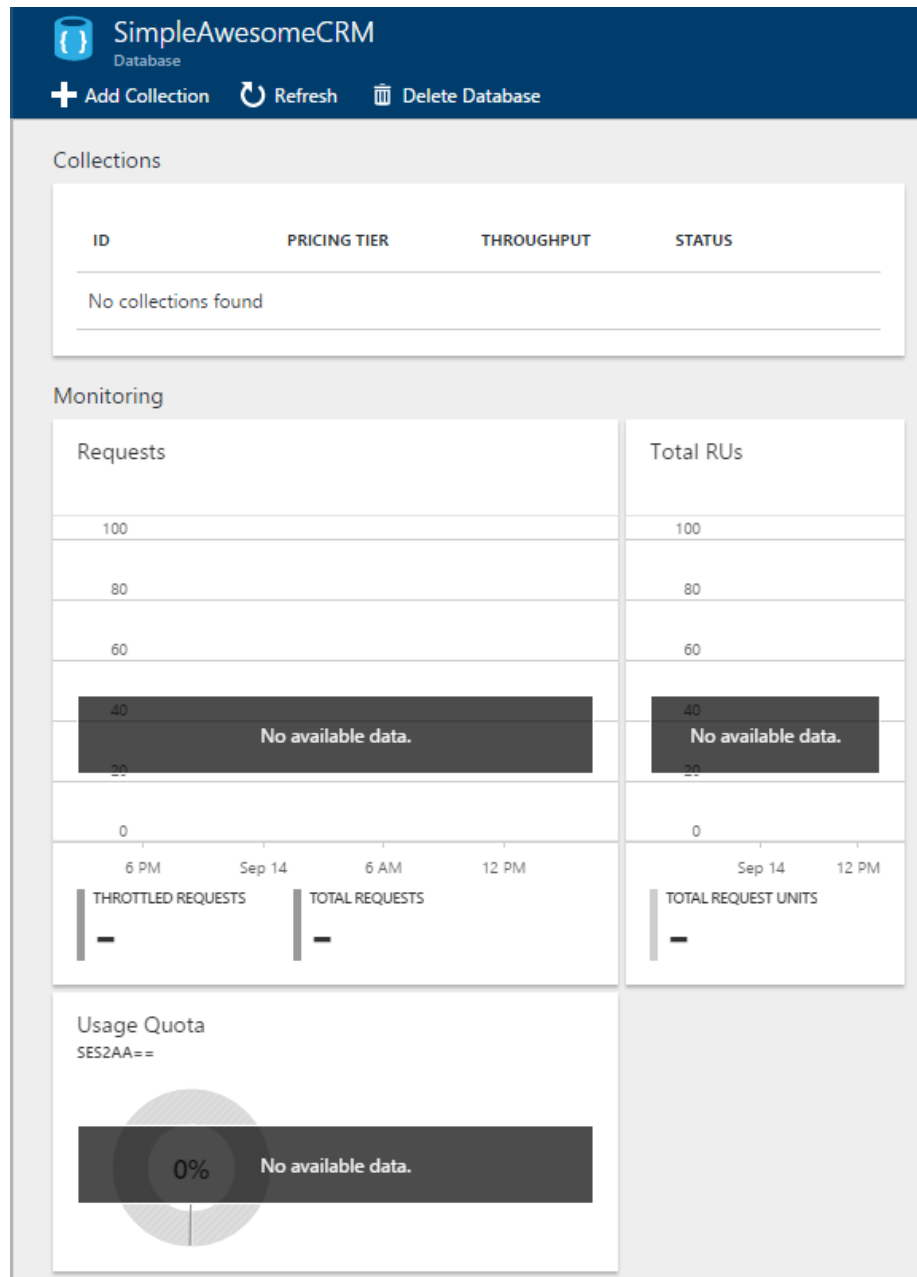


Figure 7: DocumentDB Database Dashboard

In DocumentDB, JSON documents are stored under collections. A collection is a container of JSON documents and the associated JavaScript application logic (user functions, stored procedures, and triggers).

Figure 8 depicts how DocumentDB is structured.

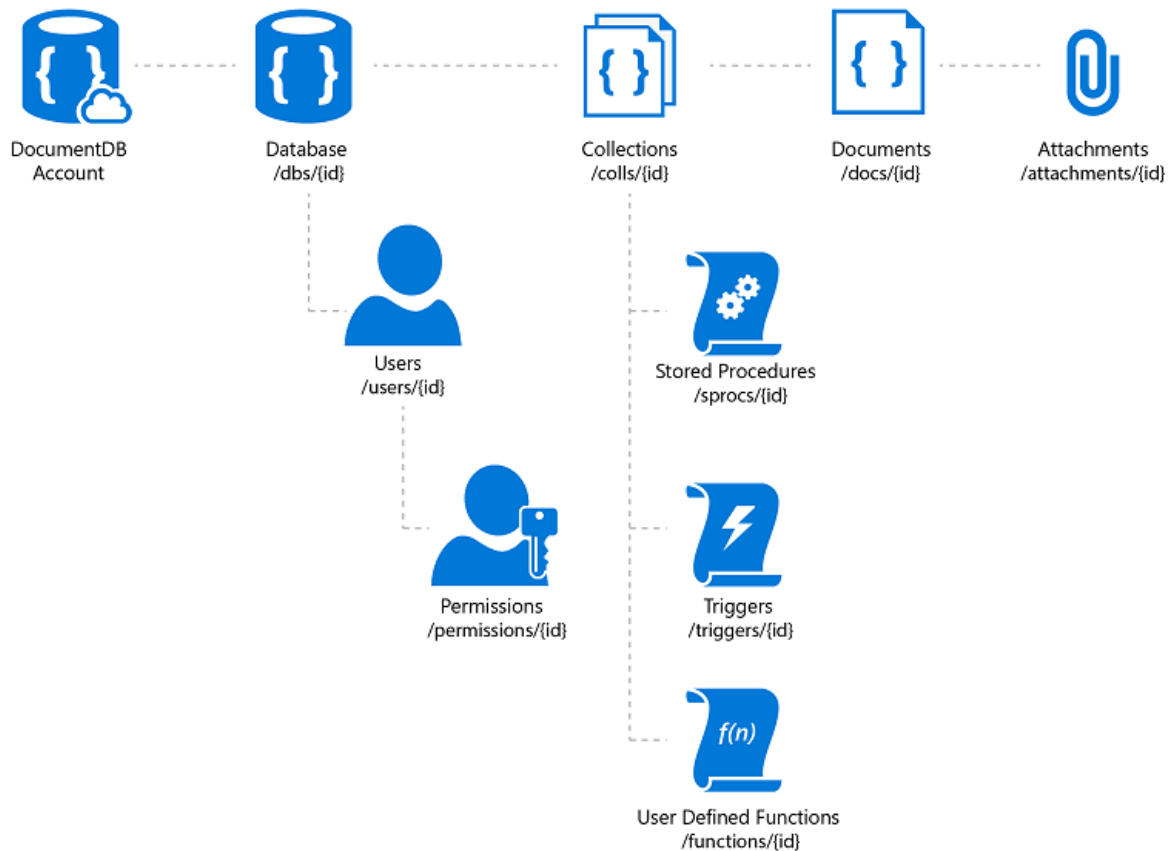


Figure 8: DocumentDB Internal Structure

A collection is a billable entity in which the cost is determined by the performance level associated with the collection. The performance levels (S1, S2, and S3) provide 10GB of storage and a fixed amount of throughput.

S1 Standard	S2 Standard ★	S3 Standard ★
250 RUs	1K RUs	2.5K RUs
10 GB Storage	10 GB Storage	10 GB Storage
99.95% SLA	99.95% SLA	99.95% SLA
Production ready	Production ready	Production ready
25.00 USD/MONTH (ESTIMATED)	50.00 USD/MONTH (ESTIMATED)	100.00 USD/MONTH (ESTIMATED)

Figure 9: DocumentDB Collection's Performance Levels and Pricing Tier

A Request Unit (RU) is measurable in seconds. For example, on the S1 level, 250 RUs (API calls) can be executed per second. More information about performance levels in DocumentDB can be found [here](#).

When creating a collection, the default pricing tier is S2. However, for demos and Proof of Concepts (POCs), an S1 tier is enough.

A collection can be created by clicking **Add Collection** on the dashboard. You will need to specify the pricing tier (by selecting S1) and the indexing policy. The indexing policy's default setting is in fact Default. If you want to take advantage of full string queries, simply change the indexing policy from Default to Hash in the **Scale & Settings** options after creating the collection. Open the **Indexing Policy** options and change the **indexingMode** property from **consistent** to **hash**. For the Simple Awesome CRM, we will be using the Hash indexing policy.

The figure consists of three side-by-side screenshots from the Azure portal interface:

- Add Collection:** Shows the initial configuration for a new collection. The ID is 'CrmObjects'. The pricing tier is 'Standard'. The partitioning mode is 'Single Partition'. The throughput is set to '1000' RU/s. The default storage capacity is '10 GB'. A pricing summary at the bottom shows a throughput cost of 60.00 USD and storage based on usage.
- Scale & Settings:** Shows the configuration for the collection's scale and settings. The pricing tier remains 'Standard'. The throughput is '1000' RU/s. The default storage capacity is '10 GB'. The 'INDEXING POLICY' is set to 'Custom'.
- Indexing Policy:** Shows the configuration for the indexing policy. The 'indexingMode' is set to 'hash', which is highlighted with a red box. The 'automatic' property is set to 'true'. The 'includedPaths' array contains a single path '/*' with a range index of kind 'Range' and dataType 'Number'.

Figure 10: Creating a DocumentDB Collection and Changing the Indexing Policy

Now that the DocumentDB account, database, and collection are ready on Microsoft Azure, you must next install the .NET DocumentDB Client library from NuGet called **Microsoft.Azure.DocumentDB** in order to start coding.

First, launch Visual Studio 2015 and create a C# Console Application. After the template code has loaded, go to **Tools > NuGet Package Manager > Manage NuGet Packages** and select **nuget.org** in the **Package Source** drop-down control. In the search box, type **DocumentDB** and the package will be displayed and available for installation.

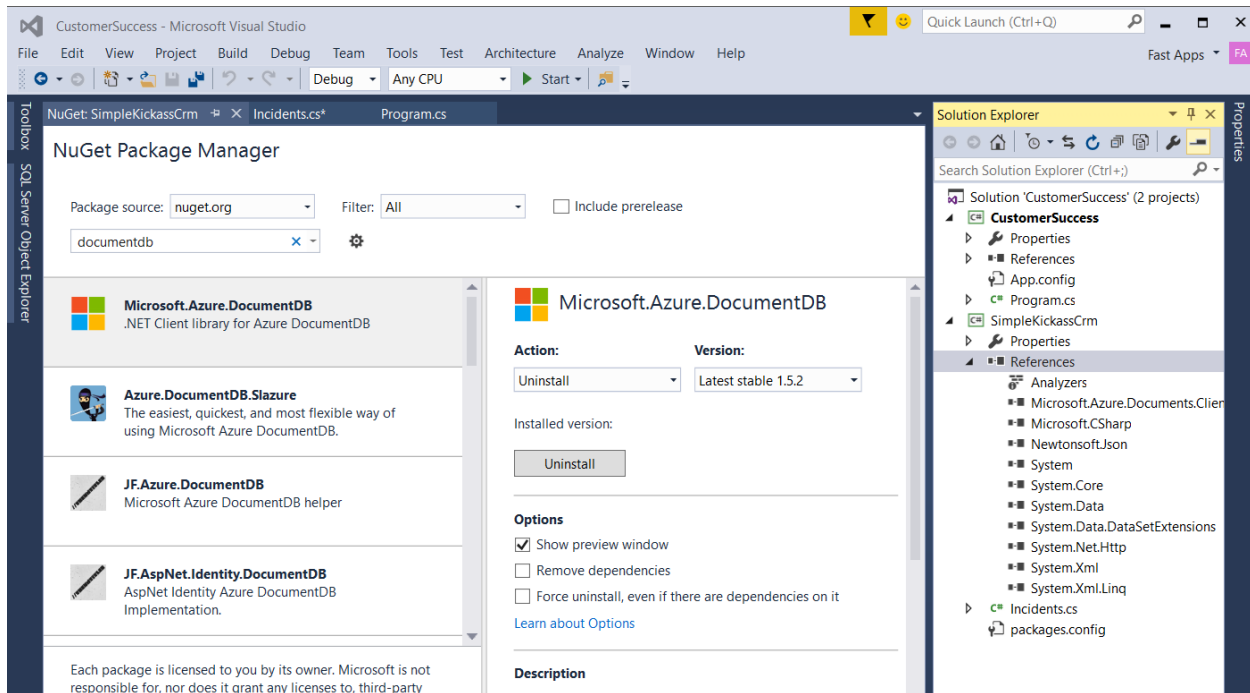


Figure 11: Installing the .NET DocumentDB Client as a NuGet Package with Visual Studio 2015

Once the .NET DocumentDB Client has been installed, you will have the **Microsoft.Azure.Documents.Client** and **Newtonsoft.Json** assemblies referenced in your Visual Studio solution.

Simple CRM code

With DocumentDB wired up and ready, we can start coding.

Based on the guidelines set forth, our Simple Awesome CRM software should be able to allow a user to open incidents, find incidents by attributes, change incident attributes, add comments, and allow the helpdesk worker to indicate any resources allocated for addressing an incident.

DocumentDB has support for range-based indexes on numeric fields that allow you to do range queries (e.g., where field > 10 and field < 20). In order to avoid doing costly scans when making range queries on dates (e.g., records older than yesterday or records placed last week), we need to convert the string representation of a date to a number. This will allow us to use range indexes on these fields.

We will be treating **DateTime** values as epoch values (the number of seconds since a particular date). In this class we are using 1 January 1970 00:00 as a starting point; however, you are free to use a different value.

Let's now examine how we can implement this with Visual Studio 2015 and .NET 4.5.2.

First, we will create a namespace called **CrmCore.DateEpoch** that will be responsible for implementing epoch values. This will include the **DateEpoch** and **Extension** classes.

Later, we will need to implement a **CrmCore.Enum** namespace where the enum definitions will be defined, and we'll need to create a utility namespace called **CrmCore.EnumUtils**.

Later we will also need to implement a **CrmCore.Enum** namespace where the enum definitions will be defined, and we'll need a utility namespace called **CrmCore.EnumUtils**.

Code Listing 1: Implementing Epoch

```
using System;

namespace CrmCore.DateEpoch
{
    public static class Extensions
    {
        public static int ToEpoch(this DateTime date)
        {
            if (date == null) return int.MinValue;

            DateTime epoch = new DateTime(1970, 1, 1);
            TimeSpan epochTimeSpan = date - epoch;

            return (int)epochTimeSpan.TotalSeconds;
        }
    }

    public class DateEpoch
    {
        public DateTime Date { get; set; }
        public int Epoch
        {
            get
            {
                return (Date.Equals(null) ||
                    Date.Equals(DateTime.MinValue))
                    ? DateTime.UtcNow.ToEpoch()
                    : Date.ToEpoch();
            }
        }

        public DateEpoch(DateTime dt)
        {
            Date = dt;
        }
    }
}
```

Now that we have seen how dates are going to be handled by DocumentDB, we also need to define several **Enum** types that will be used to indicate the status, severity, feedback frequency, and type of communication of an incident.

Code Listing 2: Enums Representing States of an Incident

```
using System;

namespace CrmCore.Enums
{
    public enum IncidentSeverity {
        [Description("Urgent")]
        Urgent,
        [Description("High")]
        High,
        [Description("Normal")]
        Normal,
        [Description("Low")]
        Low
    };

    public enum IncidentFeedbackFrequency {
        [Description("Hourly")]
        Hourly,
        [Description("Every4Hours")]
        Every4Hours,
        [Description("Every8Hours")]
        Every8Hours,
        [Description("Daily")]
        Daily,
        [Description("Every2Days")]
        Every2Days,
        [Description("Weekly")]
        Weekly,
        [Description("Every2Weeks")]
        Every2Weeks,
        [Description("Monthly")]
        Monthly
    };

    public enum IncidentCommunicationType {
        [Description("ReceiveUpdatesOnly")]
        ReceiveUpdatesOnly,
        [Description("Bidirectional")]
        Bidirectional
    };

    public enum IncidentStatus {
        [Description("NotReported")]
        NotReported,
        [Description("Reported")]
        Reported,
        [Description("FeedbackRequested")]
    }
}
```

```

        FeedbackRequested,
        [Description("UnderAnalysis")]
        UnderAnalysis,
        [Description("IssueFound")]
        IssueFound,
        [Description("WorkingOnFix")]
        WorkingOnFix,
        [Description("FixDelivered")]
        FixDelivered,
        [Description("FixAccepted")]
        FixAccepted,
        [Description("Solved")]
        Solved,
        [Description("Closed")]
        Closed,
        [Description("Reopen")]
        Reopen
    };
}

```

With the **IncidentSeverity**, **IncidentFeedbackFrequency**, **IncidentCommunicationType**, and **IncidentStatus** in place, we have the necessary categories to assign to any incident submitted using the system.

However, given that DocumentDB works with JSON, it is important to understand that a C# **enum** will be stored as an integer when converted to JSON, which can make it difficult to read when browsing through the list of documents on DocumentDB.

Therefore, in order for any **Enum** value to be readable as a **string**, we must use **System.ComponentModel** and **System.Reflection** before storing the value on DocumentDB. Let's implement an **EnumUtils** class to take care of this process.

Code Listing 3: EnumUtils Class

```

using System;
using System.ComponentModel;
using System.Reflection;

namespace CrmCore.EnumUtils
{
    public class EnumUtils
    {
        protected const string cStrExcep =
            "The string is not a description or value of the enum.";

        public static string stringValueOf(Enum value)
        {
            FieldInfo fi = value.GetType().GetField(value.ToString());

```

```

        DescriptionAttribute[] attributes = (DescriptionAttribute[])
            fi.GetCustomAttributes(typeof(DescriptionAttribute),
                false);

        return (attributes.Length > 0) ? attributes[0].Description :
            value.ToString();
    }

    public static object enumValueOf(string value, Type enumType)
    {
        string[] names = Enum.GetNames(enumType);
        foreach (string name in names)
        {
            if (stringValueOf((Enum)Enum.Parse(enumType,
                name)).Equals(value))
            {
                return Enum.Parse(enumType, name);
            }
        }

        throw new ArgumentException(cStrExcep);
    }
}

```

The **stringValueOf** method converts an integer **Enum** value to its **string** representation, and **enumValueOf** takes a **string** value and converts it to its corresponding integer **Enum** value (given its type).

EnumUtils is an incredibly handy and essential part of the Simple Awesome CRM system, as it will be used across many of the methods that follow.

Now that we know how to deal with dates in DocumentDB and categorize incidents by using **Enum** types, let's create the classes that will store incident details.

Code Listing 4: Incident Details Classes

```

using CrmCore.EnumUtils;
using CrmCore.Enums;
using CrmCore.DateEpoch;

using System;
using System.ComponentModel;
using System.Reflection;

namespace CrmCore.IncidentInfo
{
    public sealed class AllocatedResource

```

```

{
    private IncidentStatus stage;
    public string Engineer { get; set; }

    public string Stage {
        get
        {
            return EnumUtils.stringValueOf(stage);
        }
        set
        {
            stage = (IncidentStatus)EnumUtils.
                enumValueOf(value, typeof(IncidentStatus));
        }
    }

    public DateEpoch Start { get; set; }
    public DateEpoch End { get; set; }
}

public sealed class Comment
{
    public string Description { get; set; }
    public string UserId { get; set; }
    public string AttachmentUrl { get; set; }
    public DateEpoch When { get; set; }
}

public sealed class IncidentInfo
{
    private IncidentSeverity severity;
    private IncidentStatus status;
    private IncidentFeedbackFrequency feedbackFrequency;
    private IncidentCommunicationType communicationType;

    public string Description { get; set; }

    public string Severity
    {
        get
        {
            return EnumUtils.stringValueOf(severity);
        }
        set
        {
            severity = (IncidentSeverity)EnumUtils.
                enumValueOf(value, typeof(IncidentSeverity));
        }
    }
}

```

```

    }

    public string Status
    {
        get
        {
            return EnumUtils.stringValueOf(status);
        }
        set
        {
            status = (IncidentStatus)EnumUtils.
                enumValueOf(value, typeof(IncidentStatus));
        }
    }

    public string FeedbackFrequency
    {
        get
        {
            return EnumUtils.stringValueOf(feedbackFrequency);
        }
        set
        {
            feedbackFrequency = (IncidentFeedbackFrequency)EnumUtils.
                enumValueOf(value, typeof(IncidentFeedbackFrequency));
        }
    }

    public string CommunicationType
    {
        get
        {
            return EnumUtils.stringValueOf(communicationType);
        }
        set
        {
            communicationType = (IncidentCommunicationType)EnumUtils.
                enumValueOf(value, typeof(IncidentCommunicationType));
        }
    }

    public AllocatedResource[] Resources { get; set; }
    public Comment[] Comments { get; set; }
    public DateEpoch Opened { get; set; }
    public DateEpoch Closed { get; set; }
}

```


Each specific incident's details are stored as a single JSON document within DocumentDB, which, in C#, is represented by the **IncidentInfo** class. The **IncidentInfo** class is sealed, which indicates that it is simply a container for data. It might contain no comments or resources (as object arrays), or it might contain multiple items.

A **Comment** object represents an instance of a comment that is related to a particular incident. An **AllocatedResource** object represents an instance of a resource that is assigned to a particular incident.

This is all good, but we are not yet able to do anything meaningful with **IncidentInfo** or its related classes, because so far they are mere data definitions.

In order to interact with DocumentDB and use **IncidentInfo**, we must create an **Incident** class that will serve as the class responsible for posting and retrieving any incident data (**IncidentInfo**) to DocumentDB.

Code Listing 5 includes the complete code for the **Incident** class. Next, we will examine each method individually.

Code Listing 5: Incident Class Implementation

```
using Microsoft.Azure.Documents;
using Microsoft.Azure.Documents.Client;
using Microsoft.Azure.Documents.Linq;
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Linq;
using System.Reflection;
using System.Threading.Tasks;

using CrmCore.EnumUtils;
using CrmCore.Enums;
using CrmCore.DateEpoch;
using CrmCore.IncidentInfo;

namespace CrmCore.Incident
{
    public class Incident : IDisposable
    {
        private bool disposed = false;

        private const string docDbUrl =
            "dbs/SimpleAwesomeCrm/colls/CrmObjects";
        private const string docDbEndpointUrl =
            "https://fastapps.documents.azure.com:443/";
        private const string docDbAuthorizationKey = "<<DocDb Key>>";

        private const string cStrSeverityProp = "Severity";
    }
}
```

```

private const string cStrStatusProp = "Status";
private const string cStrFrequencyProp = "FeedbackFrequency";
private const string cStrCTProp = "CommunicationType";
private const string cStrClosedProp = "Closed";
private const string cStrComments = "Comments";
private const string cStrResources = "Resources";
private const string cStrOpened = "Opened";

public IncidentInfo info = null;

// Class defined in Microsoft.Azure.Documents.Client
protected DocumentClient client = null;

~Incident()
{
    Dispose(false);
}

public Incident()
{
    info = new IncidentInfo();

    info.Status =
        EnumUtils.stringValueOf(IncidentStatus.Reported);
    info.Severity =
        EnumUtils.stringValueOf(IncidentSeverity.Normal);
    info.FeedbackFrequency =
        EnumUtils.stringValueOf(IncidentFeedbackFrequency.Daily);
    info.CommunicationType =
        EnumUtils.stringValueOf(IncidentCommunicationType.
            ReceiveUpdatesOnly);
    info.Opened = new DateTimeOffset(DateTime.UtcNow);
    info.Resources = null;
    info.Comments = null;
}

public Incident(IncidentStatus status, IncidentSeverity severity,
    IncidentFeedbackFrequency freq,
    IncidentCommunicationType comType)
{
    info = new IncidentInfo();

    info.Status = EnumUtils.stringValueOf(status);
    info.Severity = EnumUtils.stringValueOf(severity);
    info.FeedbackFrequency = EnumUtils.stringValueOf(freq);
    info.CommunicationType = EnumUtils.stringValueOf(comType);
    info.Opened = new DateTimeOffset(DateTime.UtcNow);
    info.Resources = null;
    info.Comments = null;
}

```

```

    }

    private void Connect2DocDb()
    {
        client = new DocumentClient(new Uri(docDbEndpointUrl),
            docDbAuthorizationKey);
    }

    public IEnumerable<Document> FindById(string id)
    {
        if (client == null)
            Connect2DocDb();

        if (info != null && client != null)
        {
            var cases =
                from c in client.CreateDocumentQuery(docDbUrl)
                where c.Id == id
                select c;

            return cases;
        }
        else
            return null;
    }

    public IEnumerable<IncidentInfo> FindByDescription(string
description)
    {
        if (client == null)
            Connect2DocDb();

        if (info != null && client != null)
        {
            var cases =
                from c in client.CreateDocumentQuery
                <IncidentInfo>(docDbUrl)
                where c.Description.ToUpper().
                    Contains(description.ToUpper())
                select c;

            return cases;
        }
        else
            return null;
    }

    public IEnumerable<IncidentInfo> FindByDateOpenedAfter(DateTime

```

```

opened)
{
    if (client == null)
        Connect2DocDb();

    if (info != null && client != null)
    {
        var cases =
            from c in client.CreateDocumentQuery
            <IncidentInfo>(docDbUrl)
            where c.Opened.Epoch >= opened.ToEpoch()
            select c;

        return cases;
    }
    else
        return null;
}

public IEnumerable<IncidentInfo> FindByDateOpenedBefore(DateTime
opened)
{
    if (client == null)
        Connect2DocDb();

    if (info != null && client != null)
    {
        var cases =
            from c in client.CreateDocumentQuery
            <IncidentInfo>(docDbUrl)
            where c.Opened.Epoch < opened.ToEpoch()
            select c;

        return cases;
    }
    else
        return null;
}

public IEnumerable<IncidentInfo> FindByDateOpenedBetween(DateTime
start, DateTime end)
{
    if (client == null)
        Connect2DocDb();

    if (info != null && client != null)
    {
        var cases =
            from c in client.CreateDocumentQuery

```

```

        <IncidentInfo>(docDbUrl)
        where c.Opened.Epoch >= start.ToEpoch()
        where c.Opened.Epoch < end.ToEpoch()
        select c;

        return cases;
    }
    else
        return null;
}

public IEnumerable<IncidentInfo> FindByStatus(IncidentStatus
status)
{
    if (client == null)
        Connect2DocDb();

    if (info != null && client != null)
    {
        var cases =
            from c in client.CreateDocumentQuery
            <IncidentInfo>(docDbUrl)
            where c.Status == status.ToString()
            select c;

        return cases;
    }
    else
        return null;
}

public IEnumerable<IncidentInfo> FindBySeverity(IncidentSeverity
severity)
{
    if (client == null)
        Connect2DocDb();

    if (info != null && client != null)
    {
        var cases =
            from c in client.CreateDocumentQuery
            <IncidentInfo>(docDbUrl)
            where c.Severity == severity.ToString()
            select c;

        return cases;
    }
    else

```

```

        return null;
    }

    public IEnumerable<IncidentInfo>
    FindByFeedbackFrequency(IncidentFeedbackFrequency ff)
    {
        if (client == null)
            Connect2DocDb();

        if (info != null && client != null)
        {
            var cases =
                from c in client.CreateDocumentQuery
                <IncidentInfo>(docDbUrl)
                where c.FeedbackFrequency == ff.ToString()
                select c;

            return cases;
        }
        else
            return null;
    }

    public IEnumerable<IncidentInfo>
    FindByCommunicationType(IncidentCommunicationType ct)
    {
        if (client == null)
            Connect2DocDb();

        if (info != null && client != null)
        {
            var cases =
                from c in client.CreateDocumentQuery
                <IncidentInfo>(docDbUrl)
                where c.CommunicationType == ct.ToString()
                select c;

            return cases;
        }
        else
            return null;
    }

    public async Task<Document> Open(string description, bool check =
    false)
    {
        if (client == null)
            Connect2DocDb();
    }

```

```

        if (info != null && client != null)
        {
            info.Description = description;

            Document id = await client.CreateDocumentAsync
                (docDbUrl, info);

            if (check)
                return
                    (id != null) ? client.CreateDocumentQuery(docDbUrl).
                        Where(d => d.Id ==
                            id.Id).AsEnumerable().FirstOrDefault() :
                            null;
            else
                return id;
        }
        else
            return null;
    }

    public async Task<IncidentInfo> AddResource(string id,
        IncidentStatus stage, string engineer, DateTime st, DateTime end)
    {
        if (client == null)
            Connect2DocDb();

        if (info != null && client != null)
        {
            var cases =
                from c in client.CreateDocumentQuery(docDbUrl)
                where c.Id.ToUpper().Contains(id.ToUpper())
                select c;

            IncidentInfo issue = null;
            Document oDoc = null;

            foreach (var cs in cases)
            {
                var it = await client.ReadDocumentAsync(cs.AltLink);

                oDoc = it;
                issue = (IncidentInfo)(dynamic)it.Resource;

                break;
            }

            if (oDoc != null)
            {

```

```

        AllocatedResource rc = new AllocatedResource();

        rc.End = new DateEpoch((end != null) ?
            end.ToUniversalTime() : DateTime.UtcNow);
        rc.Engineer = engineer;
        rc.Stage = EnumUtils.stringValueOf(stage);
        rc.Start = new DateEpoch((st != null) ?
            st.ToUniversalTime() : DateTime.UtcNow);

        List<AllocatedResource> rRsc = new
            List<AllocatedResource>();

        if (issue?.Resources?.Length > 0)
        {
            rRsc.AddRange(issue.Resources);
            rRsc.Add(rc);

            oDoc.SetPropertyValue(cStrResources,
                rRsc.ToArray());
        }
        else
        {
            rRsc.Add(rc);
            oDoc.SetPropertyValue(cStrResources,
                rRsc.ToArray());
        }

        var updated = await
            client.ReplaceDocumentAsync(oDoc);
        issue = (IncidentInfo)(dynamic)updated.Resource;
    }

    return issue;
}
else
    return null;
}

public async Task<IncidentInfo> AddComment(string id, string
userId, string comment, string attachUrl)
{
    if (client == null)
        Connect2DocDb();

    if (info != null && client != null)
    {
        var cases =
            from c in client.CreateDocumentQuery(docDbUrl)
            where c.Id.ToUpper().Contains(id.ToUpper())

```



```

        select c;

        IncidentInfo issue = null;
        Document oDoc = null;

        foreach (var cs in cases)
        {
            var it = await client.ReadDocumentAsync(cs.AltLink);

            oDoc = it;
            issue = (IncidentInfo)(dynamic)it.Resource;

            break;
        }

        if (oDoc != null)
        {
            Comment c = new Comment();

            c.AttachmentUrl = attachUrl;
            c.Description = comment;
            c.UserId = userId;
            c.When = new DateEpoch(DateTime.UtcNow);

            List<Comment> cMts = new List<Comment>();

            if (issue?.Comments?.Length > 0) {
                cMts.AddRange(issue.Comments);
                cMts.Add(c);

                oDoc.SetPropertyValue(c.StrComments,
                    cMts.ToArray());
            }
            else {
                cMts.Add(c);
                oDoc.SetPropertyValue(c.StrComments,
                    cMts.ToArray());
            }

            var updated = await
            client.ReplaceDocumentAsync(oDoc);
            issue = (IncidentInfo)(dynamic)updated.Resource;
        }

        return issue;
    }
    else
        return null;

```

```

}

public async Task<IncidentInfo> ChangePropertyValue(string id,
string propValue, string propName)
{
    if (client == null)
        Connect2DocDb();

    if (info != null && client != null)
    {
        var cases =
            from c in client.CreateDocumentQuery(docDbUrl)
            where c.Id.ToUpper().Contains(id.ToUpper())
            select c;

        IncidentInfo issue = null;
        Document oDoc = null;

        foreach (var cs in cases)
        {
            var it = await client.ReadDocumentAsync(cs.AltLink);

            oDoc = it;
            issue = (IncidentInfo)(dynamic)it.Resource;

            break;
        }

        if (oDoc != null)
        {
            switch (propName)
            {
                case cStrSeverityProp:
                    oDoc.SetPropertyValue(cStrSeverityProp,
                        propValue);
                    break;

                case cStrStatusProp:
                    oDoc.SetPropertyValue(cStrStatusProp,
                        propValue);
                    break;

                case cStrFrequencyProp:
                    oDoc.SetPropertyValue(cStrFrequencyProp,
                        propValue);
                    break;

                case cStrCTProp:
                    oDoc.SetPropertyValue(cStrCTProp, propValue);
            }
        }
    }
}

```

```

        break;

        case cStrClosedProp:
            oDoc.SetPropertyValue(cStrClosedProp,
                issue.Closed);
            break;
    }

    var updated = await
        client.ReplaceDocumentAsync(oDoc);
    issue = (IncidentInfo)(dynamic)updated.Resource;
}

return issue;
}
else
    return null;
}

protected virtual void Dispose(bool disposing)
{
    if (!disposed)
    {
        if (disposing)
            client.Dispose();
    }
    disposed = true;
}

public void Dispose()
{
    Dispose(true);

    GC.SuppressFinalize(this);
}
}
}

```

Understanding incidents

In order to understand how our Simple Awesome CRM works with DocumentDB, let's take a detailed look at each method within the **Incident** class, and let's make sure we understand what each of the following items do.

Code Listing 6: Important Initialization Details for DocumentDB

```
private const string docDbUrl = "dbs/SimpleAwesomeCrm/colls/CrmObjects";

private const string docDbEndpointUrl =
    "https://fastapps.documents.azure.com:443/";

private const string docDbAuthorizationKey = "<<DocDb Key>>";

protected DocumentClient client = null;
```

The **docDbUrl** string represents the relative URL to the **CrmObjects** collection that was created within DocumentDB using the Azure portal. The URL is formed as follows:

dbs/<DocumentDB database name>/colls/<DocumentDB collection name>

In our case, **SimpleAwesomeCrm** is the DocumentDB database name, and **CrmObjects** is the DocumentDB collection name (which is where our CRM JSON documents will be stored).

The **docDbEndpointUrl** string represents the actual URL of the DocumentDB instance running on Azure. We've chosen the subdomain "fastapps," but you may use anything else (so long as it corresponds to what you defined when creating the DocumentDB account on the Azure Portal).

The **docDbAuthorizationKey** string is the Azure access key to the DocumentDB instance specified by **docDbEndpointUrl**.

The **client** object is the DocumentDB Azure SDK for .NET instance class that is responsible for all the communication with DocumentDB.

Now we can establish a connection to DocumentDB.

Code Listing 7: Connection to DocumentDB

```
private void Connect2DocDb()
{
    client = new DocumentClient(new Uri(docDbEndpointUrl),
                                docDbAuthorizationKey);
}
```

The connection to DocumentDB is made by creating an instance of **DocumentClient** that will be properly disposed later.

Code Listing 8: Disposal of the DocumentClient Instance

```
protected virtual void Dispose(bool disposing)
{
    if (!disposed)
    {
        if (disposing)
```

```

        client.Dispose();
    }
    disposed = true;
}

```

With a connection established, we can now open an incident and submit it to DocumentDB. We can do this as shown in Code Listing 9.

Code Listing 9: Opening an Incident

```

public async Task<Document> Open(string description, bool check = false)
{
    if (client == null)
        Connect2DocDb();

    if (info != null && client != null)
    {
        info.Description = description;
        Document id = await client.CreateDocumentAsync(docDbUrl, info);

        if (check)
            return (id != null) ? client.CreateDocumentQuery(docDbUrl).
                Where(d => d.Id ==
                    id.Id).AsEnumerable().FirstOrDefault() :
                null;
        else
            return id;
    }
    else
        return null;
}

```

The **Open** method makes a connection to DocumentDB when **client** (the **DocumentClient** instance) is null—i.e. when a connection has not yet been established.

Once the connection is established, **CreateDocumentAsync** is called with **docDbUrl** and an instance of **IncidentInfo**. To check if the document was successfully submitted to DocumentDB, the **check** parameter can be set to **true**.

However, in order for this to work, an instance of **IncidentInfo** needs to be created. Code Listing 10 shows how we can do that from a wrapper.

Code Listing 10: Wrapper around Incident.Open

```

using System;
using Newtonsoft.Json;
using System.Threading.Tasks;
using Microsoft.Azure.Documents;
using System.Linq;

```

```

using System.Collections.Generic;

using CrmCore.EnumUtils;
using CrmCore.Enums;
using CrmCore.DateEpoch;
using CrmCore.IncidentInfo;
using CrmCore.Incident;

namespace CrmCore
{
    public class CrmExample
    {
        private const string cStrCaseCreated =
            "Case created (as JSON) ->";
        private const string cStrTotalResults = "Total results: ";

        private const string cStrDescription = "Description: ";
        private const string cStrStatus = "Status: ";
        private const string cStrSeverity = "Severity: ";
        private const string cStrCommunication = "Communication: ";
        private const string cStrFrequency = "Frequency: ";
        private const string cStrOpened = "Opened: ";
        private const string cStrClosed = "Closed: ";

        private const string cStrDateTimeFormat =
            "dd-MMM-yyyy hh:mm:ss UTC";

        public static async Task<string> OpenCase(string description)
        {
            string id = string.Empty;

            using (Incident inc = new Incident(IncidentStatus.Reported,
                IncidentSeverity.High,
                IncidentFeedbackFrequency.Every4Hours,
                IncidentCommunicationType.Bidirectional))
            {
                var issue = await inc.Open(description);

                if (issue != null)
                {
                    var i = JsonConvert.DeserializeObject
                        <IncidentInfo>(issue.ToString());

                    Console.WriteLine(cStrCaseCreated);
                    Console.WriteLine(issue.ToString());

                    id = issue.Id;
                }
            }
        }
    }
}

```

```

        return id;
    }

    public static async Task<string> OpenCase(string description,
        IncidentStatus st, IncidentSeverity sv,
        IncidentFeedbackFrequency ff, IncidentCommunicationType ct)
    {
        string id = string.Empty;

        using (Incident inc = new Incident(st, sv, ff, ct))
        {
            var issue = await inc.Open(description);

            if (issue != null)
            {
                var i = JsonConvert.DeserializeObject<IncidentInfo>(issue.ToString());

                Console.WriteLine(cStrCaseCreated);
                Console.WriteLine(issue.ToString());

                id = issue.Id;
            }
        }

        return id;
    }

    public static void CheckCase(string id)
    {
        using (Incident inc = new Incident())
        {
            IEnumerable<Document> issues = inc.FindById(id);

            foreach (var issue in issues)
                OutputCaseDetails(issue);
        }
    }

    private static void OutputCaseDetails(object issue)
    {
        IncidentInfo i = (issue is IncidentInfo) ?
            (IncidentInfo)issue :
            JsonConvert.DeserializeObject<IncidentInfo>(issue.ToString());
    }

```

```

        Console.WriteLine(cStrDescription + i?.Description);
        Console.WriteLine(cStrStatus + i?.Status);
        Console.WriteLine(cStrSeverity + i?.Severity);
        Console.WriteLine(cStrCommunication + i?.CommunicationType);
        Console.WriteLine(cStrFrequency + i?.FeedbackFrequency);
        Console.WriteLine(cStrOpened +
            i?.Opened?.Date.ToString(cStrDateTimeFormat));
        Console.WriteLine(cStrClosed +
            i?.Closed?.Date.ToString(cStrDateTimeFormat));
    }
}

using CrmCore;
using System;
using System.Threading.Tasks;

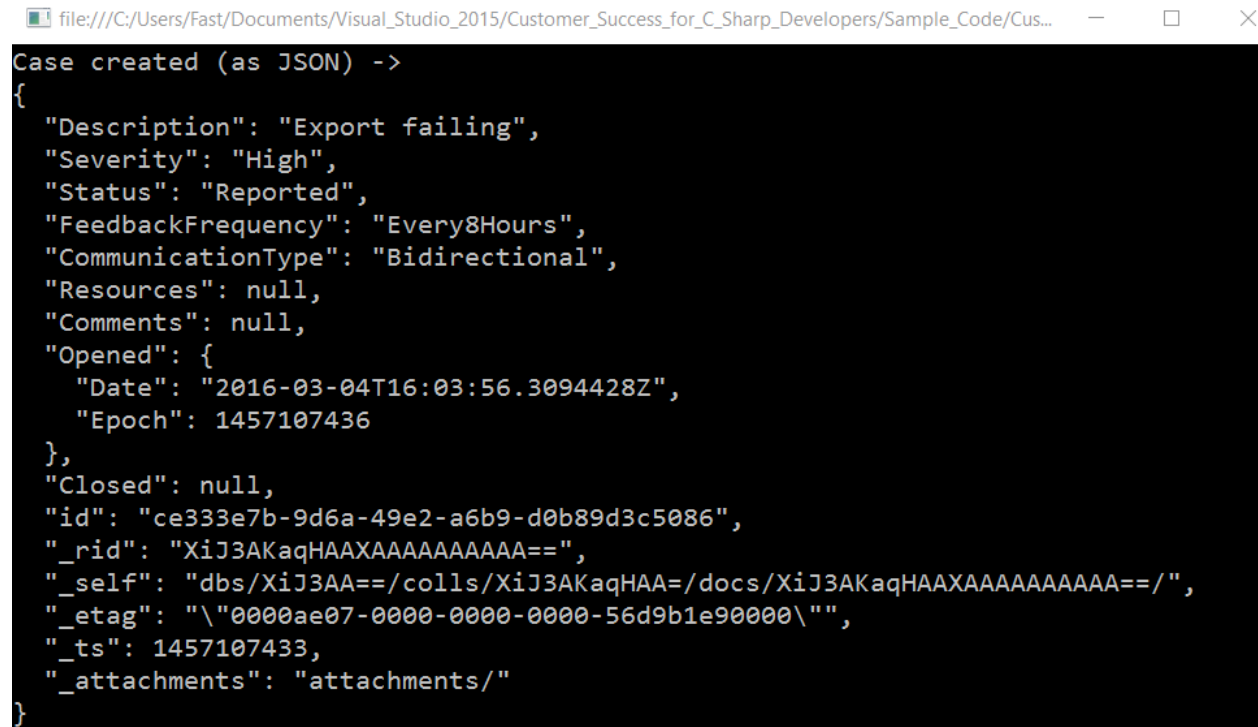
namespace CustomerSuccess
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Clear();
            OpenFindCase();
            Console.ReadLine();
        }

        public static async void OpenFindCase()
        {
            await Task.Run(
                async () =>
                {
                    string id = await CrmExample.OpenCase(
                        "Export failing",
                        IncidentStatus.Reported,
                        IncidentSeverity.High,
                        IncidentFeedbackFrequency.Every8Hours,
                        IncidentCommunicationType.Bidirectional);

                    Console.WriteLine(Environment.NewLine);
                    CrmExample.CheckCase(id);
                });
        }
    }
}

```


OpenFindCase wraps a call around **CrmExample.OpenCase**, which creates an **Incident** instance and calls the **Open** method of that instance. This returns a **Document** instance that is then deserialized into an **IncidentInfo** object. Finally, this **IncidentInfo** object is printed on the screen by **OutputCaseDetails**. Running this example will produce the result shown in Figure 12.



```
file:///C:/Users/Fast/Documents/Visual_Studio_2015/Custom_Success_for_C_Sharp_Developers/Sample_Code/Cus...
Case created (as JSON) ->
{
  "Description": "Export failing",
  "Severity": "High",
  "Status": "Reported",
  "FeedbackFrequency": "Every8Hours",
  "CommunicationType": "Bidirectional",
  "Resources": null,
  "Comments": null,
  "Opened": {
    "Date": "2016-03-04T16:03:56.3094428Z",
    "Epoch": 1457107436
  },
  "Closed": null,
  "id": "ce333e7b-9d6a-49e2-a6b9-d0b89d3c5086",
  "_rid": "XiJ3AKaqHAAXAAAAAAAAA==",
  "_self": "dbs/XiJ3AA==/colls/XiJ3AKaqHAA=/docs/XiJ3AKaqHAAXAAAAAAAAA==/",
  "_etag": "\"0000ae07-0000-0000-0000-56d9b1e90000\"",
  "_ts": 1457107433,
  "_attachments": "attachments/"
}
```

Figure 12: Incident Creation Console Output

The document can be seen on Azure, as in Figure 13.

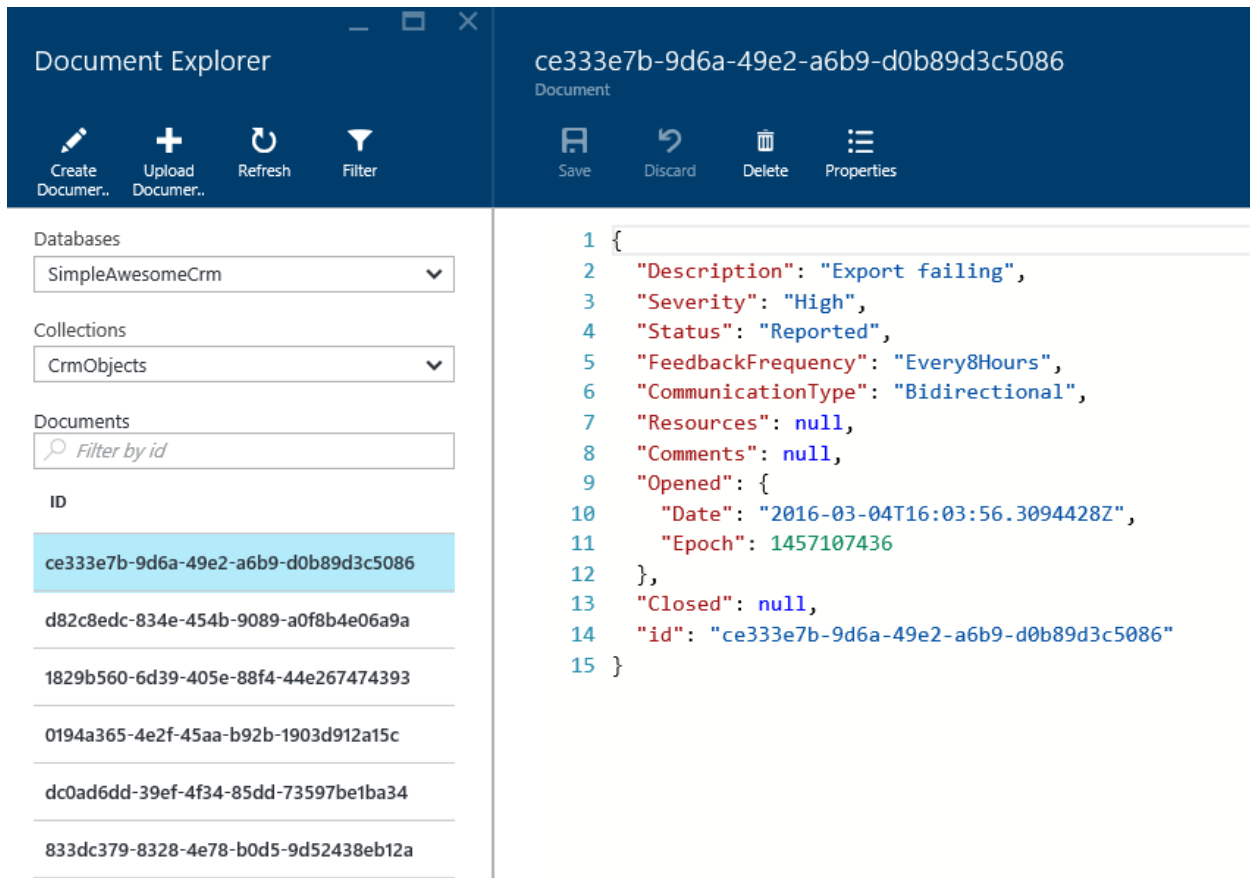


Figure 13: Incident in the Azure Document Explorer

When a document is created with no particular specifications, DocumentDB automatically assigns a random GUID to it—in this case: [ce333e7b-9d6a-49e2-a6b9-d0b89d3c5086](#).

Find methods

Now that we have the ability to create incidents, we will shift our focus to retrieving and finding documents in Azure. Given that we should be able to search on any specific criteria using any of the fields of an **IncidentInfo** object, we next need to add several **Find** methods to the **Incident** class.

Code Listing 11: Find Methods of the Incident Class

```

public IEnumerable<Document> FindById(string id)
{
    if (client == null) Connect2DocDb();

    if (info != null && client != null)
    {
        var cases =

```

```

        from c in client.CreateDocumentQuery(docDbUrl)
        where c.Id == id
        select c;

        return cases;
    }
    else
        return null;
}

public IEnumerable<IncidentInfo> FindByDescription(string description)
{
    if (client == null) Connect2DocDb();

    if (info != null && client != null)
    {
        var cases =
            from c in client.CreateDocumentQuery<IncidentInfo>(docDbUrl)
            where c.Description.ToUpper().
                Contains(description.ToUpper())
            select c;

        return cases;
    }
    else
        return null;
}

public IEnumerable<IncidentInfo> FindByDateOpenedAfter(DateTime opened)
{
    if (client == null) Connect2DocDb();

    if (info != null && client != null)
    {
        var cases =
            from c in client.CreateDocumentQuery<IncidentInfo>(docDbUrl)
            where c.Opened.Epoch >= opened.ToEpoch()
            select c;

        return cases;
    }
    else
        return null;
}

public IEnumerable<IncidentInfo> FindByDateOpenedBefore(DateTime opened)
{
    if (client == null) Connect2DocDb();

```

```

    if (info != null && client != null)
    {
        var cases =
            from c in client.CreateDocumentQuery<IncidentInfo>(docDbUrl)
            where c.Opened.Epoch < opened.ToEpoch()
            select c;

        return cases;
    }
    else
        return null;
}

public IEnumerable<IncidentInfo> FindByDateOpenedBetween(DateTime start,
    DateTime end)
{
    if (client == null) Connect2DocDb();

    if (info != null && client != null)
    {
        var cases =
            from c in client.CreateDocumentQuery<IncidentInfo>(docDbUrl)
            where c.Opened.Epoch >= start.ToEpoch()
            where c.Opened.Epoch < end.ToEpoch()
            select c;

        return cases;
    }
    else
        return null;
}

public IEnumerable<IncidentInfo> FindByStatus(IncidentStatus status)
{
    if (client == null) Connect2DocDb();

    if (info != null && client != null)
    {
        var cases =
            from c in client.CreateDocumentQuery<IncidentInfo>(docDbUrl)
            where c.Status == status.ToString()
            select c;

        return cases;
    }
    else
        return null;
}

```

```

public IEnumerable<IncidentInfo> FindBySeverity(IncidentSeverity
    severity)
{
    if (client == null) Connect2DocDb();

    if (info != null && client != null)
    {
        var cases =
            from c in client.CreateDocumentQuery<IncidentInfo>(docDbUrl)
            where c.Severity == severity.ToString()
            select c;

        return cases;
    }
    else
        return null;
}

public IEnumerable<IncidentInfo> FindByFeedbackFrequency
    (IncidentFeedbackFrequency ff)
{
    if (client == null) Connect2DocDb();

    if (info != null && client != null)
    {
        var cases =
            from c in client.CreateDocumentQuery<IncidentInfo>(docDbUrl)
            where c.FeedbackFrequency == ff.ToString()
            select c;

        return cases;
    }
    else
        return null;
}

public IEnumerable<IncidentInfo> FindByCommunicationType
    (IncidentCommunicationType ct)
{
    if (client == null) Connect2DocDb();

    if (info != null && client != null)
    {
        var cases =
            from c in client.CreateDocumentQuery<IncidentInfo>(docDbUrl)
            where c.CommunicationType == ct.ToString()
            select c;
    }
}

```

```

        return cases;
    }
    else
        return null;
}

```

All the **Find** methods of the **Incident** class are similar—they each start by attempting to connect to DocumentDB using **Connect2DocDb** and, if a connection exists (when **client** is not null), a LINQ query is executed with a specific condition. In most cases an **IEnumerable<IncidentInfo>** instance is returned with the results from the LINQ query.

IEnumerable is preferable because it describes behavior, while **List** is an implementation of that behavior. When you use **IEnumerable**, you give the compiler a chance to defer work until later, allowing for possible optimization along the way. If you use **List**, you force the compiler to check the results immediately.

Whenever you use LINQ expressions, you should also use **IEnumerable**, because only when you specify the behavior will you give LINQ a chance to defer evaluation and optimize the program.

Given that there could be one or more results, the returned **IEnumerable<IncidentInfo>** object needs to be parsed, and each result should be printed out. In order to do this, we will use our wrapper **CrmExample** class.

Let's create our first wrapper for the method **FindByDescription** within the **CrmExample** class.

Code Listing 12: FindByDescription Wrapper

```

public static void FindByDescription(string description)
{
    using (Incident inc = new Incident())
    {
        IEnumerable<IncidentInfo> issues =
            inc.FindByDescription(description);

        Console.WriteLine("FindByDescription: " + description);

        if (issues.Count() > 0)
        {
            Console.Write(Environment.NewLine);

            foreach (var issue in issues)
            {
                OutputCaseDetails(issue);
                Console.Write(Environment.NewLine);
            }
        }
    }
}

```

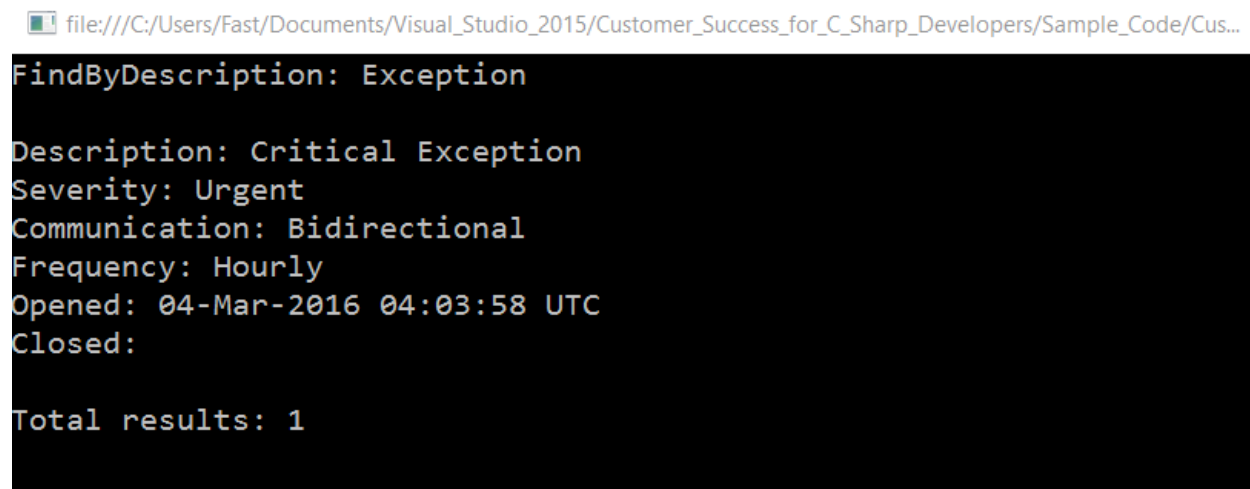
```
        Console.WriteLine(cStrTotalResults + issues.Count().ToString());  
    }  
}
```

Code Listing 13 depicts how calling it from the main program would follow.

Code Listing 13: FindByDescription Invocation

```
CrmExample.FindByDescription("Exception");
```

Figures 14 and 15 show the **FindByDescription** results on screen and in Azure.



```
file:///C:/Users/Fast/Documents/Visual_Studio_2015/Customer_Success_for_C_Sharp_Developers/Sample_Code/Cus...  
FindByDescription: Exception  
  
Description: Critical Exception  
Severity: Urgent  
Communication: Bidirectional  
Frequency: Hourly  
Opened: 04-Mar-2016 04:03:58 UTC  
Closed:  
  
Total results: 1
```

Figure 14: FindByDescription Results on Screen

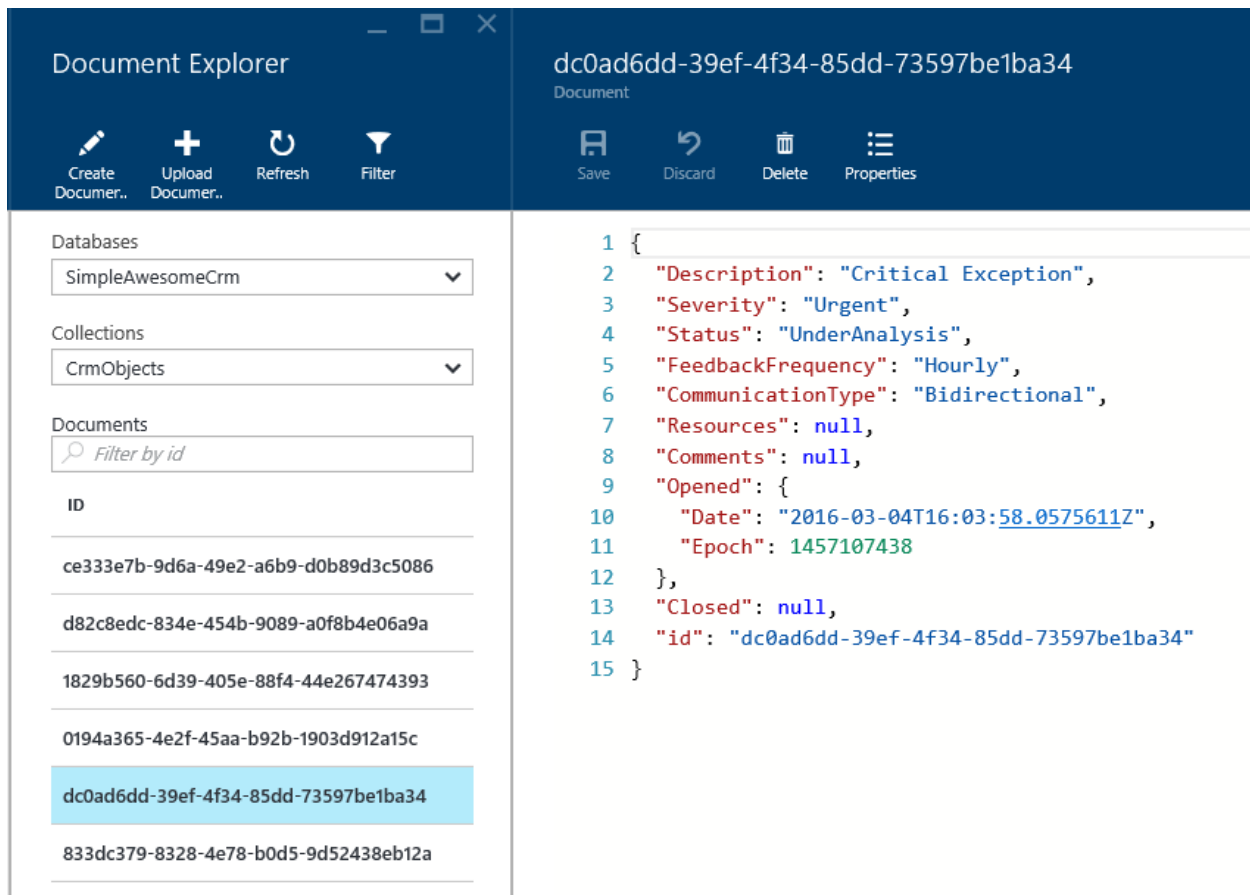


Figure 15: FindByDescription Results in Azure

Let's now examine the method **FindByDateOpenedAfter**. This will retrieve any documents from DocumentDB that were opened with a **DateTime** equal to or greater than the parameter being used for searching.

Code Listing 14: FindByDateOpenedAfter Wrapper

```

public static void FindByDateOpenedAfter(DateTime opened)
{
    using (Incident inc = new Incident())
    {
        IEnumerable<IncidentInfo> issues =
            inc.FindByDateOpenedAfter(opened);

        Console.WriteLine("FindByDateOpenedAfter: " +
            opened.ToString(cStrDateTimeFormat));

        if (issues.Count() > 0)
        {
            Console.WriteLine(Environment.NewLine);

            foreach (var issue in issues)

```



```

        {
            OutputCaseDetails(issue);
            Console.Write(Environment.NewLine);
        }
    }

    Console.WriteLine(cStrTotalResults + issues.Count().ToString());
}

```

Code Listing 15 depicts calling **FindByDateOpenedAfter** from the main program.

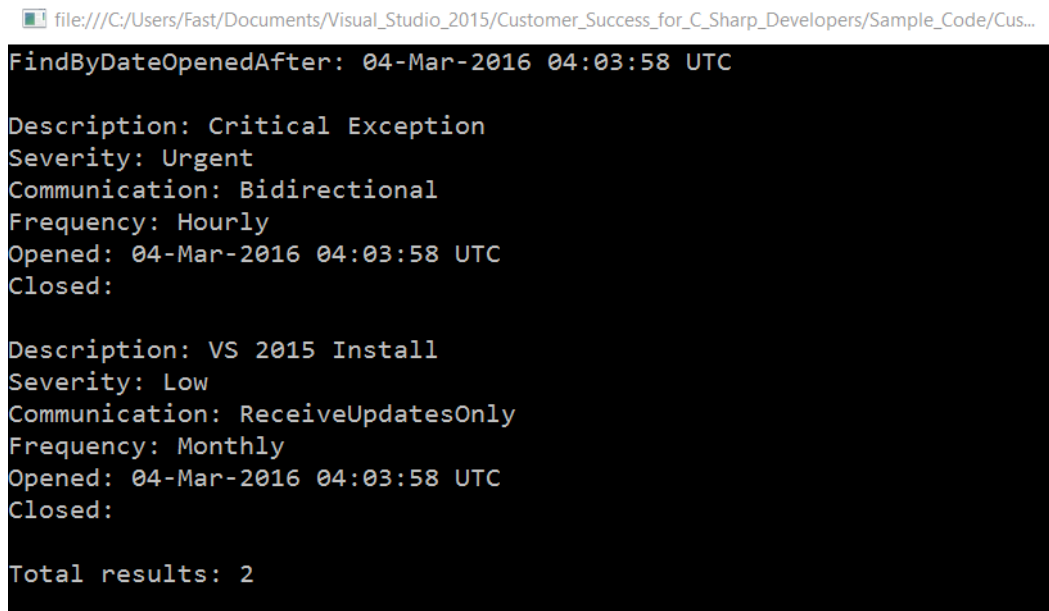
Code Listing 15: FindByDateOpenedAfter Invocation

```

CrMExample.FindByDateOpenedAfter(new DateTime(2016, 3, 4, 16, 03, 58,
DateTimeKind.Utc));

```

That action produces the results in Figures 16 and 17.



```

file:///C:/Users/Fast/Documents/Visual_Studio_2015/Custom...
FindByDateOpenedAfter: 04-Mar-2016 04:03:58 UTC

Description: Critical Exception
Severity: Urgent
Communication: Bidirectional
Frequency: Hourly
Opened: 04-Mar-2016 04:03:58 UTC
Closed:

Description: VS 2015 Install
Severity: Low
Communication: ReceiveUpdatesOnly
Frequency: Monthly
Opened: 04-Mar-2016 04:03:58 UTC
Closed:

Total results: 2

```

Figure 16: FindByDateOpenedAfter Results on Screen

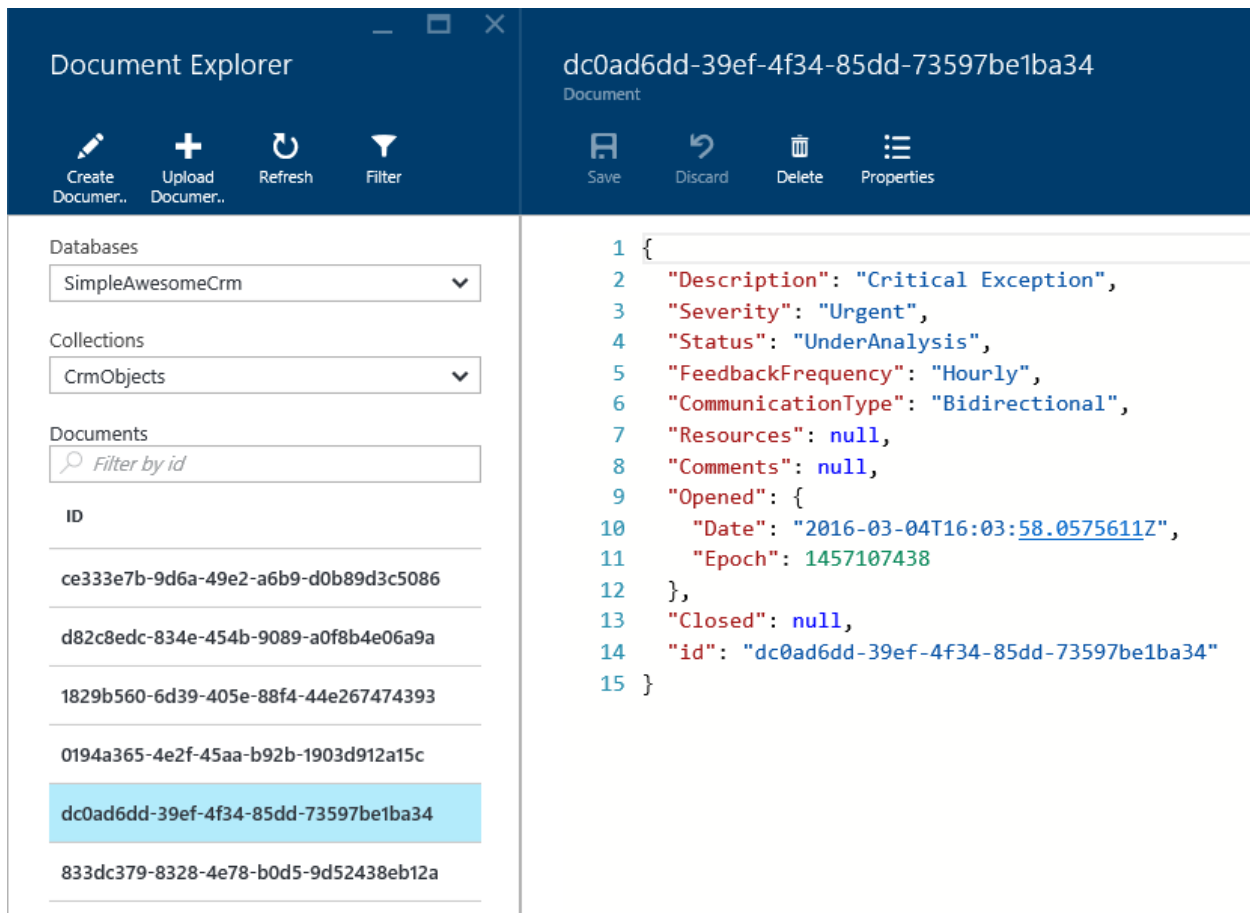


Figure 17: FindByDateOpenedAfter Results in Azure

Next, let's examine the method **FindBySeverity**. This will retrieve any documents from DocumentDB that have a specific **IncidentSeverity** assigned.

Code Listing 16: FindBySeverity Wrapper

```

public static void FindBySeverity(IncidentSeverity severity)
{
    using (Incident inc = new Incident())
    {
        IEnumerable<IncidentInfo> issues = inc.FindBySeverity(severity);

        Console.WriteLine("FindBySeverity: " +
            EnumUtils.stringValueOf(severity));

        if (issues.Count() > 0)
        {
            Console.WriteLine(Environment.NewLine);

            foreach (var issue in issues)
            {

```

```

        OutputCaseDetails(issue);
        Console.Write(Environment.NewLine);
    }
}

Console.WriteLine(cStrTotalResults + issues.Count().ToString());
}
}

```

Code Listing 17 depicts calling **FindBySeverity** from the main program.

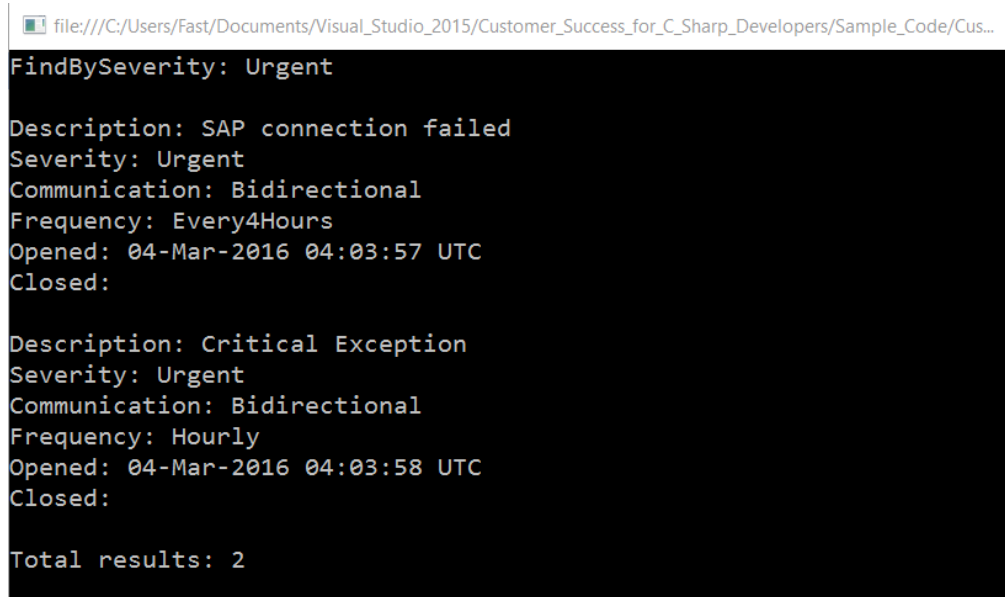
Code Listing 17: FindBySeverity Invocation

```

CrnExample.FindBySeverity(IncidentSeverity.Urgent);

```

Figure 18 shows the results.



```

file:///C:/Users/Fast/Documents/Visual_Studio_2015/Custom_Success_for_C_Sharp_Developers/Sample_Code/Cus...
FindBySeverity: Urgent

Description: SAP connection failed
Severity: Urgent
Communication: Bidirectional
Frequency: Every4Hours
Opened: 04-Mar-2016 04:03:57 UTC
Closed:

Description: Critical Exception
Severity: Urgent
Communication: Bidirectional
Frequency: Hourly
Opened: 04-Mar-2016 04:03:58 UTC
Closed:

Total results: 2

```

Figure 18: FindBySeverity Results on Screen

We've seen how easy it is to retrieve documents from DocumentDB using some of the **Find** methods of the **Incident** class wrapped around **CrnExample**. Note that the key differentiator among the **Find** methods is how they differ internally in each of their LINQ queries.

Comments and resources

At this stage we know how to open incidents and find them using different LINQ search queries throughout the **Find** methods implemented. **IncidentInfo** was designed so that any incident's allocated comments and resources can vary from none to multiple items.

Both comments and resources are arrays of the **Comments** and **AllocatedResource** classes. Both are null when a new incident is opened, so to add comments and resources, we must define a method in order to add each.

A **Comment** instance will include a **Description**, **UserId** (person submitting the comment), **AttachmentUrl** (URL location of an attachment related to the comment), and **When** (**DateTime** it was submitted).

Let's explore how to implement a method within the **Incident** class in order to add a comment to an incident.

Code Listing 18: AddComment Implementation

```
public async Task<IncidentInfo> AddComment(string id, string userId,
string comment, string attachUrl)
{
    if (client == null) Connect2DocDb();

    if (info != null && client != null)
    {
        var cases =
            from c in client.CreateDocumentQuery(docDbUrl)
            where c.Id.ToUpper().Contains(id.ToUpper())
            select c;

        IncidentInfo issue = null;
        Document oDoc = null;

        foreach (var cs in cases)
        {
            var it = await client.ReadDocumentAsync(cs.AltLink);

            oDoc = it;
            issue = (IncidentInfo)(dynamic)it.Resource;
            break;
        }

        if (oDoc != null)
        {
            Comment c = new Comment();
            c.AttachmentUrl = attachUrl;
            c.Description = comment;
            c.UserId = userId;
            c.When = new DateEpoch(DateTime.UtcNow);

            List<Comment> cMts = new List<Comment>();

            if (issue?.Comments?.Length > 0) {
                cMts.AddRange(issue.Comments);
            }
        }
    }
}
```

```

        cMts.Add(c);

        oDoc.SetPropertyValue(cStrComments, cMts.ToArray());
    }
    else {
        cMts.Add(c);
        oDoc.SetPropertyValue(cStrComments, cMts.ToArray());
    }

    var updated = await client.ReplaceDocumentAsync(oDoc);
    issue = (IncidentInfo)(dynamic)updated.Resource;
}

return issue;
}
else
    return null;
}

```

AddComment first checks whether or not there is a connection to DocumentDB, and if there isn't, it establishes a connection by invoking **Connect2DocDb**.

With the connection to DocumentDB in place, performing a LINQ query using **CreateDocumentQuery** comes next. This is done by using the ID of the incident to which we want to add the comment.

The method **CreateDocumentQuery** returns an **IOrderedQueryable<Document>** instance that we will need to loop through (even though the count on **IOrderedQueryable<Document>** might be 1) in order to perform a **ReadDocumentAsync** and retrieve the specific **Document** instance corresponding to the ID being queried.

The **ReadDocumentAsync** method expects the **AltLink** (alternative URL link) of the document that will be read from DocumentDB. **AltLink** is a property of the **Document** instance that represents the URL of the document within the DocumentDB collection we want to get.

Here's the interesting bit: to get the **IncidentInfo** instance from the returned result of the call to **ReadDocumentAsync**, we must cast the result as **issue = (IncidentInfo)(dynamic)it.Resource**; where this will be converted into an **IncidentInfo** instance.

Following that, an array instance of **Comment** is created and added to the existing **Document** instance that represents the DocumentDB document (incident) being queried. The array instance of **Comment** is added through the instance of the **Document** object by using the property **SetPropertyValue**.

Once the **Comment** object has been added to the **Document** instance, **ReplaceDocumentAsync** is called and the document with the added **Comment** is written back to the DocumentDB collection (updated in Azure). The updated **IncidentInfo** object representing that updated document is returned to the caller of **AddComment**.

To implement **AddComment**, we'll need to create a wrapper around it within **CrmExample**. Let's see how this can be done.

Code Listing 19: AddComment Wrapper

```
public static async void AddComment(string id, string userId, string
    comment, string attachUrl)
{
    using (Incident inc = new Incident())
    {
        await Task.Run(
            async () =>
            {
                IncidentInfo ic = await inc.AddComment(id, userId, comment,
                    attachUrl);

                OutputCaseDetails(ic);
            });
    }
}
```

Code Listing 20 depicts how it will be called from the main program.

Code Listing 20: AddComment Invocation

```
CrmExample.AddComment("4b992d62-4750-47d2-ac4a-dbce2ce85c12", "efreitas",
    "Request for feedback", "https://eu3.salesforce.com/...");
```

The results are shown in Figure 19.

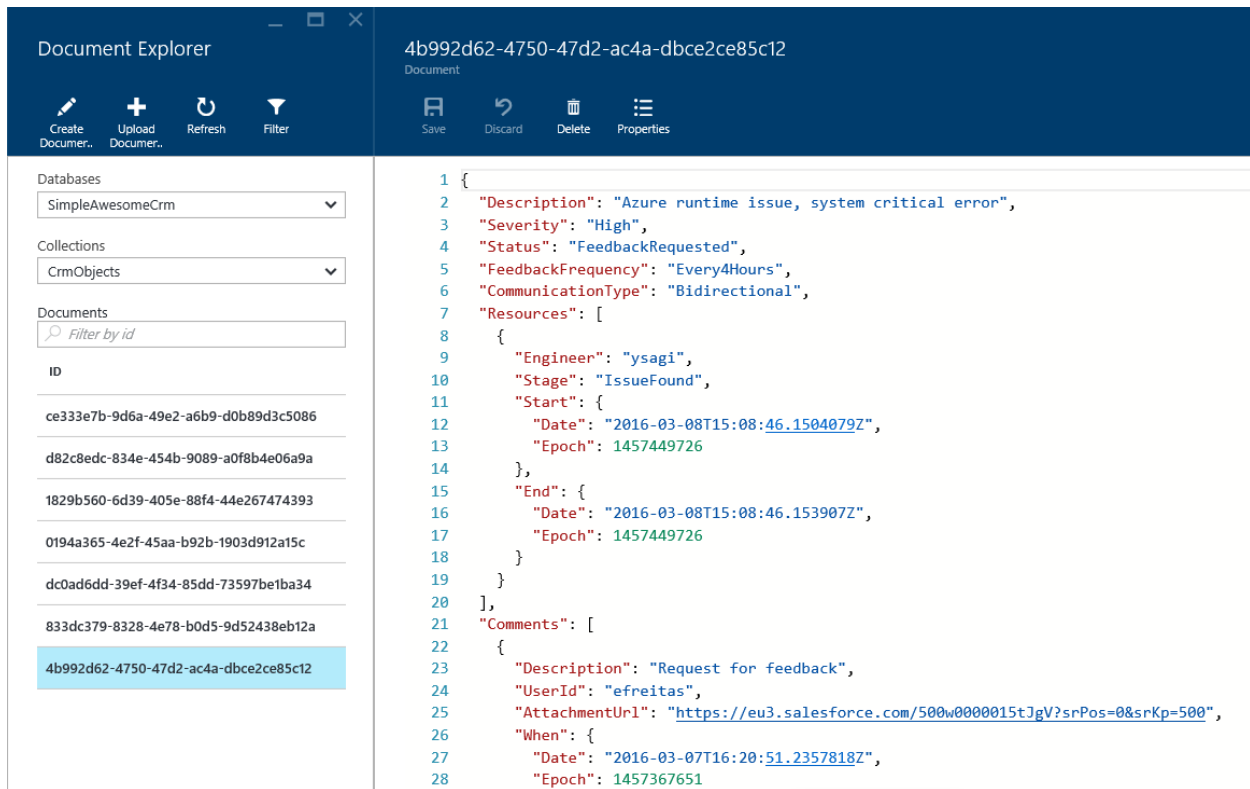


Figure 19: AddComment Results in Azure

In order to add resources to an incident, we will need to take the same approach as the **AddComment** method. This means we'll implement an **AddResource** method within the **Incident** class and also create a wrapper around it in **CrmExample**.

Code Listing 21: AddResource Implementation

```

public async Task<IncidentInfo> AddResource(string id, IncidentStatus
stage, string engineer, DateTime st, DateTime end)
{
    if (client == null) Connect2DocDb();

    if (info != null && client != null)
    {
        var cases =
            from c in client.CreateDocumentQuery(docDbUrl)
            where c.Id.ToUpper().Contains(id.ToUpper())
            select c;

        IncidentInfo issue = null;
        Document oDoc = null;

        foreach (var cs in cases)
        {

```

```

        var it = await client.ReadDocumentAsync(cs.AltLink);

        oDoc = it;
        issue = (IncidentInfo)(dynamic)it.Resource;
        break;
    }

    if (oDoc != null)
    {
        AllocatedResource rc = new AllocatedResource();

        rc.End = new DateEpoch((end != null) ? end.ToUniversalTime()
            : DateTime.UtcNow);
        rc.Engineer = engineer;
        rc.Stage = EnumUtils.stringValueOf(stage);
        rc.Start = new DateEpoch((st != null) ? st.ToUniversalTime()
            : DateTime.UtcNow);

        List<AllocatedResource> rRsc = new List<AllocatedResource>();

        if (issue?.Resources?.Length > 0)
        {
            rRsc.AddRange(issue.Resources);
            rRsc.Add(rc);
            oDoc.SetPropertyValue(cStrResources, rRsc.ToArray());
        }
        else
        {
            rRsc.Add(rc);
            oDoc.SetPropertyValue(cStrResources, rRsc.ToArray());
        }

        var updated = await client.ReplaceDocumentAsync(oDoc);
        issue = (IncidentInfo)(dynamic)updated.Resource;
    }
    return issue;
}
else
    return null;
}

```

As you can see, the implementation of **AddResource** is almost identical to the one from **AddComment**.

AddResource first checks whether or not there is a connection to DocumentDB, and if there isn't, it establishes a connection by invoking **Connect2DocDb**.

With the connection to DocumentDB in place, performing a LINQ query using **CreateDocumentQuery** comes next. This is done by using the ID of the incident to which we want to add the resource.

The method **CreateDocumentQuery** returns an **IOrderedQueryable<Document>** collection that we will need to loop through in order to perform a **ReadDocumentAsync** and retrieve the specific **Document** instance corresponding to the ID of the document being queried.

The **ReadDocumentAsync** method expects the **AltLink** to be provided.

To get the **IncidentInfo** instance from the returned result of the call to **ReadDocumentAsync**, we must cast the result as **issue = (IncidentInfo)(dynamic)it.Resource**; where this will be converted into an **IncidentInfo** instance.

An array instance of **AllocatedResource** is created and added to the existing **Document** instance that represents the DocumentDB document (incident) being queried. The array instance of **AllocatedResource** is added through the instance of the **Document** by using the **SetPropertyValue** method.

Once the **AllocatedResource** property has been added to the **Document** instance, **ReplaceDocumentAsync** is called and the document with the added **AllocatedResource** is written back to the DocumentDB collection. The updated **IncidentInfo** object representing that updated document is returned to the caller of **AddResource**.

For this to work, **AddResource** needs to be wrapped and exposed from the **CrmExample** class.

Code Listing 22: AddResource Wrapper

```
public static async void AddResource(string id, IncidentStatus stage,
    string engineer, DateTime st, DateTime end)
{
    using (Incident inc = new Incident())
    {
        await Task.Run(
            async () =>
            {
                IncidentInfo ic = await inc.AddResource(id, stage,
                    engineer, st, end);

                OutputCaseDetails(ic);
            });
    }
}
```

Finally, as Code Listing 23 shows, **AddResource** can be called from the main program.

Code Listing 23: AddResource Invocation

```
CrnExample.AddResource("4b992d62-4750-47d2-ac4a-dbce2ce85c12",  
IncidentStatus.IssueFound, "bob smith", DateTime.Now, DateTime.Now);
```

The output of this can also be seen in Figure 19.

Although it might be useful at some point to know how to delete comments and allocated resources, we will not implement that functionality in this e-book.

Changing properties

So far we've implemented the Simple Awesome CRM system so that incidents are submitted and queried along with added comments and resources allocated—all of this in order to complete a job.

The ability to change properties for an existing document (**IncidentInfo** instance) is essential, as it might be necessary to change the **IncidentStatus**, **IncidentSeverity**, or any other property within a document.

Let's implement the **ChangePropertyValue** method within the **IncidentInfo** class.

Code Listing 24: ChangePropertyValue Implementation

```
public async Task<IncidentInfo> ChangePropertyValue(string id, string  
    propValue, string propName)  
{  
    if (client == null) Connect2DocDb();  
  
    if (info != null && client != null)  
    {  
        var cases =  
            from c in client.CreateDocumentQuery(docDbUrl)  
            where c.Id.ToUpper().Contains(id.ToUpper())  
            select c;  
        IncidentInfo issue = null;  
        Document oDoc = null;  
  
        foreach (var cs in cases)  
        {  
            var it = await client.ReadDocumentAsync(cs.AltLink);  
  
            oDoc = it;  
            issue = (IncidentInfo)(dynamic)it.Resource;  
            break;  
        }  
  
        if (oDoc != null)  
        {
```

```

        switch (propName)
        {
            case cStrSeverityProp:
                oDoc.SetPropertyValue(cStrSeverityProp, propValue);
                break;

            case cStrStatusProp:
                oDoc.SetPropertyValue(cStrStatusProp, propValue);
                break;

            case cStrFrequencyProp:
                oDoc.SetPropertyValue(cStrFrequencyProp, propValue);
                break;

            case cStrCTProp:
                oDoc.SetPropertyValue(cStrCTProp, propValue);
                break;

            case cStrClosedProp:
                oDoc.SetPropertyValue(cStrClosedProp, issue.Closed);
                break;
        }

        var updated = await client.ReplaceDocumentAsync(oDoc);
        issue = (IncidentInfo)(dynamic)updated.Resource;
    }

    return issue;
}
else
    return null;
}

```

Just as with **AddComment** and **AddResource**, **ChangePropertyValue** follows the same logic. It attempts to connect to DocumentDB through **Connect2DocDb** and by calling **CreateDocumentQuery** and querying for a specific ID. An **IOrderedQueryable<Document>** is returned for the ID of the document being queried.

The specific **Document** instance is then returned through a call to **ReadDocumentAsync**, and the respective property is changed using **SetPropertyValue**.

Next, the current **Document** is updated on the DocumentDB collection by calling **ReplaceDocumentAsync**.

For this to work—for it to be called from the main program—we'll need to wrap up **ChangePropertyValue** in **CrmExample**, as shown in Code Listing 25.

Code Listing 25: ChangePropertyValue Wrapper

```

public static async void ChangePropertyValue(string id, string propValue,
    string propName)
{
    using (Incident inc = new Incident())
    {
        await Task.Run(
            async () =>
            {
                IncidentInfo ic = await inc.ChangePropertyValue(id,
                    propValue, propName);

                OutputCaseDetails(ic);
            });
    }
}

```

As Code Listing 26 shows, this can then be invoked from the main program.

Code Listing 26: ChangePropertyValue Invocation

```

CrnExample.ChangePropertyValue("4b992d62-4750-47d2-ac4a-dbce2ce85c12",
EnumUtils.stringValueOf(IncidentStatus.FeedbackRequested), "Status");

```

Summary

In this chapter, we've seen how to use DocumentDB and C# to implement a simple yet effective CRM API that you can mold to fit your needs.

Given its reliability, stability, robustness, and ease of use, DocumentDB is our preferred choice. Its API is easy to follow, needing very little documentation, and it fits perfectly with the concept of building a simple CRM API that allows us to cater to the concepts explained in [Chapter 1](#).

The CRM sample code presented here is enough to get you started, and it should be easy to expand with added functionality.

You are invited to add your own ideas to this codebase and expand upon the concepts explained in Chapter 1—feel free to keep improving it.

In the next chapters, we'll explore some ways that customer service can help your business grow and increase its value. We'll also look at how simple reverse engineering can be a valuable asset for solving problems more quickly.

Chapter 3 Helpdesk Tactics

Introduction

You might not have a full helpdesk strategy to grow your business at this moment, but we have set forth some guidelines that you can easily apply to establish the right mindset ([Chapter 1](#)). And we've looked at how you can create a baseline code repository that can be used to make a flexible and custom CRM system backed by a strong cloud-hosting platform like Azure ([Chapter 2](#)) that will help you manage incidents and communication more effectively.

You can kick-start that process right now in order to grow your business. Here's the focal point of the process—you'll need a mechanism in place to help turn your customers into evangelists so that they do the marketing work for you. As the old saying goes, there's no better way to get new customers than by word of mouth.

If you adhere to the guidelines and mindset set forth in Chapter 1, you'll find that your helpdesk will be more agile and productive. Having a leaner helpdesk that can resolve and close incidents quickly will delight your existing customers, making them feel grateful and important. This appreciation will be paid back in the form of referrals. That's when your customers become evangelists of your brand and product.

This chapter will describe the path that leads customers to become evangelists, and it will explain the tactics and process you'll initiate to make this happen.

By the end of this chapter, you should be able to apply these tactics to help you grow your business through customer service.

Steps toward evangelism

As we all know, Rome was not built in a day.

The same goes for converting a customer into an evangelist. That won't happen all at once. The conversion requires dedication, effort, consistency, and, above all, grace and patience. The key is to allow time for each customer to make the transition at their own pace. You can't force that on them.

Some customers might be very enthusiastic about your product and services right off the bat, but others might be more cautious—until you gain their full trust, they are unlikely to go out of their way to talk to others about you. Customers each go at their own pace, but they all want to be cared for and to have their issues resolved in a professional, timely, and accurate manner.

So, what does this path to evangelism look like? Let's have a closer look.

Table 6 represents what a typical path from customer to evangelist might look like.

Table 6: A Typical Customer-to-Evangelist Path

Prospect	Customer	User	Evangelist
"I heard about this cool new product..."	"I just purchased this cool software, check it out!"	"I really love this software. Their support is awesome"	"I'm treated like a partner. I fully support this company and their product."

The cool thing about this path is that the transition from prospect to evangelist can be pretty smooth if we provide reliable and top-notch support.

This path assumes that the customer is positive about the product, company, service, and the overall level of support received. However, if the customer has negative views, those can be turned around in the early stages, although the transition to evangelist will not occur as easily.

Defining and finding evangelists

In the strictest sense of the word, and in the context of today's interconnected society, an evangelist is someone who can place your product, services, or brand in a good light in front of their network.

When an effective evangelist speaks, other people tend to listen closely. Evangelists are also known as influencers—they are followed and trusted within their networks and their opinion is valued and respected.

Evangelists usually have large online and personal networks. And, apart from their network size, they tend to cultivate strong one-on-one relationships with their peers. When they talk, people tend to listen carefully.

Some other characteristics of evangelists worth considering:

- They have connections to other evangelists.
- They like to try new things.
- They appreciate when things work, so they endorse carefully.
- They might be extroverts or introverts, but what matters is how much they identify with your brand and how well they have been treated.

Take this and write it down. Read it a few times and memorize it. Act upon it: **If you want to grow your business you need to find evangelists.**

[LinkedIn](#), arguably the world's most important social network for professionals, is an invaluable resource for finding well-connected evangelists who can help you grow. However, let's clarify one crucial point: an evangelist is not a power connector. Just because someone has a large network on LinkedIn doesn't mean they can necessarily be influencers for your product and brand.

A well-connected person on LinkedIn might be a potential evangelist for your company if they work in the same industry and have more than one group in common with the groups your company deals with. Finding a person with an affinity for your industry gives you an opening to approach this person and start an informal conversation about having a look at your products and services.

You should also look for people who are not simply related to your company's groups or industry, but who have recommendations and are actively involved in the community. In other words, people who write articles, answer questions, give speeches, and are active beyond their regular 9-to-5 jobs.

You can find this information by checking a person's activity within the network and the level of recommendations they might have. Outside of social networking, you can also look into which groups of people related to your company's industry are respected by your peers. You are likely to find potential evangelists in such groups.

Last but not least, you can find evangelists at conferences. Go through the list of speakers at a conference related to your industry and do a bit of research so that you can approach the appropriate people in a friendly and non-business-oriented manner. Ask questions related to what they do or the topics of their talks. If you show interest in what those people do, they will often reciprocate and ask about what you do. If they do show further interest, you can take the conversation to the next level. Do not attempt to make a sales pitch unless there's an explicit interest from the other person. Play your cards wisely. This brings us to our next point.

How to create evangelists

In an ideal world, we could bring in all the evangelists we might ever potentially need from our existing network, be that online or offline. However, let's not overlook the potential evangelists who might already be next to us: our current customers.

How can we turn these customers into fierce advocates of our brand? Here are some actions that can move current customers in that direction:

- Build a strong sense of trust.
- Ask for opinions and ask questions.
- Use a highly empowered language.
- Keep the communication flowing—send regular updates.

- Show appreciation.
- Make a bond.

Let's explore each of these actions in detail.

The art of building trust

Building trust doesn't happen overnight. It is built every day with every little interaction we have with our users. Pay attention to your customer's details and listen more than you talk. Keep track of what you do and what you don't (and why you don't).

Over time you and your company will make mistakes when dealing with customers. Product and communication mistakes happen, and when they do, take a step back and don't be shy to ask forgiveness. It will go a long way toward nurturing trust.

Let's examine some examples of trustworthy expressions. Trust is consolidated when your customers trust that you will do the right thing, when they trust you with their contacts, when they trust you will not embarrass them, when they explicitly say they trust your product and brand, when they share their sensitive data, when they trust you will stand up for them, and, most importantly, when they put their careers on the line for you. These commitments of trust can go both ways, which means that you, as a service provider, can also demonstrate these attitudes with your customers and users.

Building and nurturing trust are perhaps the most important aspects of customer service—you cannot create loyal evangelists without trust.

The power of asking

Asking can also go a long way. What would you do if one of your customers suddenly asked for a free upgrade to a new version of one of your products? Would you turn around and simply give it to them? Consider this: somehow the fabric of the universe has been designed to work in such a way that it rewards those who give more than they take.

Imagine if everyone went into business with the mindset of providing more value to society than the value they capture from it. Wouldn't that be wonderful? Strong empathy can lead to strong business.

Asking shows two important traits: that we respect what people think and say, and that we are interested in their story, not just our own.

Of course, you are not simply going to turn around and say, "Hey, do you want to become my evangelist?" However, you can ask some of these questions:

- Will you introduce me to your colleagues or connections at other companies?
- Who else should I show this to?

- Will you introduce me to that person or company?
- What other features do you want or need from this application?
- What would make this a killer application for you?
- Which features can we enhance to make your day more productive?
- Will you share this with your peers at the event you are going to?
- Will you give us a testimonial we can use?
- Would you be willing to be a reference for us?
- Would you be willing to help us test our new version?

These are only some of the questions you might ask. If you can think of others, take a moment to write them down.

If customers are satisfied with the level of service you provide and the quality of your product, they will likely be inclined, and even happy, to help if you ask.

So, in order to find out, simply ask.

Highly empowered language

When a customer is ready to share your story with their network, they might need some help from you. Just because they are willing to take a leap and become your advocate doesn't mean they can spread the word without your assistance. If they are going to put you on the front page of their network, you should provide them with content they can use to spread your message. Think about the following:

- How do they describe your company, your services, your products, and you?
- How can they invite others to help them spread your message?
- How can they keep it simple and remove any unnecessary words that do not help get the message across quickly? If your company, product, and services cannot be understood in one sentence, it won't be understood by the average user (which is the majority of users).
- How can they share bits and pieces of your materials via email and social networks? Make sure to follow up (not in a pushy way).

Let's say your company has an amazing product called "Simple Awesome CRM." The name sounds familiar.

Because you are the visionary and the creator, you might think of it as “CRM 2.0” or “CRM Made Simple,” but does that message really convey the product’s features to everyone the way you hope it does?

If you examine both statements, “CRM 2.0” doesn’t really mean anything, and neither does “CRM Made Simple.” You might think it does, just as you might think “Web 2.0” means something, but these are simply clichés.

Even though CRM is a well-known acronym recognized by a lot of people, it is still unknown to most people.

You should use a sentence that describes your product in a super-concise and effective way that most everyone can relate to and understand.

A great way to describe your CRM product would be: “A replacement for a helpdesk ticketing spreadsheet.” Anybody who has opened a helpdesk ticket knows what this means and knows how hard it is to keep track of issues.

So, keep your message simple and clear enough that it can be understood by everyone, from new employees to the CEO.

Keep the communication flowing

Sometimes people make the mistake of assuming that because an evangelist already knows a product and company, they already know everything necessary to begin relaying the message.

Don’t make that assumption. Keep your customers in the loop. Keep them apprised of all your developments. In other words, keep the communication flowing. That doesn’t mean spamming them with information or content, but it does mean that regularly, say once or twice a month, they should hear from you and should know what you are doing. Here are some things you can share:

- New features in your products.
- New range of services.
- News or hot topics within your industry or role.
- Updates about your company.
- Industry trends.

If the time comes to send out your usual monthly update and you find you don’t have anything new to add since your last update, why not invest some effort in making a bit of news?

Surely there is something you can mention—keep your eyes open and share that. This is what public relations people do all the time. They take something typical and phrase the description in a way that makes headlines. Stand out and create some content—a new blog post or a free e-book or some new training material. Remember to keep the communication flowing.

If you do that, your customers will see you are engaged, they will appreciate your commitment, and they will be willing to spread the word.

Show appreciation

When someone goes out of their way to say something positive about your product or company, take a few minutes to put everything aside to write one or two lines saying thank you. It takes only a couple of minutes, but it can have a profound impact.

Remember, if you don't have a crystal ball, neither do your customers. Don't assume that simply because you are thinking about them, customers will be able to read your mind. Take the time to let them know you appreciate them.

Sending a thank-you is always a nice gesture, and email is the perfect mechanism for this, as it leaves written evidence.

Of course, you can always call customers. However, you should do that only after you have first sent the written message. You can also take it to the next level by sending a token of appreciation, such as a card. Kind gestures are remembered—they show the extra effort you put in.

Finally, there's one other way you can take it to the next level: make a bond.

Make a bond

This might seem a bit unusual, but making a bond doesn't mean your goal should be to establish a personal and long-lasting relationship with any of your customers or evangelists.

Making a bond means being aware that when you speak with those people, at some point they might make a comment about something personal, something that matters to them. When that happens, you should value that moment and listen carefully. Listen like you really care (and you should care). Remember what they say. If your memory is bad, take a moment to write it down.

Show them you care—when you speak to them again, ask about that topic. Be kind and ask if they sorted out their problem. They will appreciate that you thought about it, even if the subject is totally unrelated with the professional relationship you have with them. Make a bond, and you will be remembered.

Converting incidents into sales

We've seen how we can take customers and nurture them by building a strong sense of empathy and self-awareness in order to turn them into evangelists. That's an invaluable asset, but how can we generate more sales without specifically selling?

When a customer raises an incident, there are two ways to respond. You can choose to see the incident as just another problem or as an opportunity to improve and create a solution more focused on the user's needs. You can also view it as an opportunity to create new potential features and to get customer feedback on which features they might consider useful.

In essence, these incidents are opportunities that can be used to initiate new business. However, to successfully start this conversation, it is essential to move away from the money aspect of the sale and focus more on the user's needs: a solution-based sale.

While the price will be important at some point, it should not be the main driver for this conversation. You first need to ensure that the customer will get the best possible solution. The conversation should be conducted as though the customer is a strategic partner. Customers can help us understand their needs, and we can help them understand which solutions best fit their situations.

Notice how this differs from viewing the contact in terms of needing to make X or Y amount of sales this quarter. This is a more customer-friendly view in which you aim to make sure that your customers have everything they need to succeed with your product.

Once you set this as your sales mentality, you won't ever need to make direct sales again, because the customer will explicitly request that from you. With this attitude and partnership in place, sales will come naturally as customers feel that you care about them and that you want to help them succeed, not just sign a contract.

If for some reason what you have offered is not necessarily the right fit for a customer, let it go. Write it off. It's better to leave it than to give them a solution they will never be happy with. If you don't let it go, you risk the customer losing money and losing their trust in you.

A relatively simple way to transition an incident into a sales opportunity is to show customers which features and offerings your organization can provide. Don't assume that because they are your customer they know all about you and the intrinsic details of your product or brand. Even if you have introduced some of these details in the past, that might not have come at the right time. Some customers are very problem-centric and might not pay attention to tangential details when they are focused on some other issue with a higher priority.

So how can you show them your new features? Simply ask if they know about them. For instance, you might say something like, "As a matter of fact, we have a tool that should solve the very problem you are talking about."

You can also ask if any other features might complement any of the existing features they already use. Explicitly ask for value-added improvements for the features you already offer. Customers will value the fact that you asked them. And remember—their feedback is invaluable to you, as it will help you strengthen your product.

Typically, in some of these conversations with your customers, you might talk about features that are not related to the problem at hand. You'll notice that as you develop your customer relationship, you'll encounter new needs from them, and you'll be able to introduce other seemingly unrelated products and features into the conversation.

By doing this, you are broadening your customers' understanding of the scope of services you can provide. The advantage of this is that your customers will get the notion that they might not need to look elsewhere to fill in their gaps—you've already done this for them.

Remember to make sure you give customers a choice. You aren't forcing them to buy anything, and if a decision to buy is made, it's theirs, not yours. Add value for your customers and forget about ever explicitly selling again.

Company initiative and culture

In order for your company to have a successful helpdesk strategy, the initiative needs to come from the top, and as a developer with a strong sense of customer service, you should feel empowered to talk to your direct manager about a proactive helpdesk strategy. Management needs to be aligned and in sync with the concept, and this attitude should permeate throughout the entire organization. This will have an impact on financials, training, R&D, and product management.

Your organization's helpdesk strategy should be embedded into the company's fabric: its guidelines and policy manual. This way everyone will know it, and people can effectively reference it and be trained on it.

Furthermore, the company should have a way to gather strong metrics about customer engagement and should reward those actively involved in it. Incentive programs that make it easy for employees at any level to delight customers are also effective tools. In essence, the helpdesk should be seen not as a liability, but as an asset. Possibly the most important one in your company.

But how can this be propagated to all employees? The best approach is to implement orientation programs for new and current employees. Whenever a new employee is hired, they should know that customer service is a huge priority and is key to the organization's success.

People committed to showing an exceptional customer service attitude should be rewarded, either financially or through some other form of recognition. Make sure that the time employees spend with customers is respected and valued. If someone needs to spend three or four hours with a specific customer on a particular problem, let it happen—by all means don't punish that employee.

Summary

In this chapter, we examined tactics for transforming customers into evangelists. We also looked at how incidents can be viewed as opportunities for acquiring new business without going through a regular sales process. Last but not least, we saw how having a customer-centric company culture can permeate to all employees, allowing them to become internal evangelists for customer success.

By now, you should have the mindset, tools, and tactics in place to establish a proactive, customer-centric helpdesk strategy within your organization that will delight your customers, create evangelists, and help you grow your business.

In the next chapter, we'll look at how reverse engineering can assist in helping us achieve faster incident resolution.

Chapter 4 Reflection to the Rescue

Introduction

The [Reflection](#) mechanism in the .NET Framework is an extremely powerful feature. However, most developers find themselves asking when they will ever use it, and why use it at all.

At first glance, Reflection can look scary, as it mostly falls into the category of [reverse engineering](#)—a technology often used by people writing [decompilers](#) and other developer tools.

Although there is truth in that perception, Reflection offers many extremely useful features, and it can prove to be invaluable for troubleshooting or writing plugins for existing applications.

Some of these features include discovering types in [assemblies](#), dynamic loading with configuration, and better understanding of how .NET assemblies work internally. By gaining an improved understanding of these items, we can become better at troubleshooting, leading to faster and better customer service. Faster incident turnaround and resolution is the ultimate goal.

We'll also explore how some tools built around Reflection can be used to inspect code when we don't have the source code at hand, which can be essential for achieving faster incident resolution. By inspecting the code of a problematic DLL, for instance, we can narrow the location of a problem within the code much more quickly.

The goal of this chapter is to explore the practical parts of Reflection with an emphasis on safety, flexibility, performance, and proper ethical use so that you can put this extremely powerful feature to work and gain the benefits of diagnosing problems more quickly, speeding up incident-resolution time and allowing you to better understand how to build plugins for existing applications.

What is Reflection?

In .NET, Reflection consists of programmatically inspecting the metadata and compiled code within a .NET assembly. To better understand this, we need to look at each of these concepts separately.

An assembly in the .NET world is a logical unit of execution, deployment, and reuse. It can be either a library ([DLL](#)) or [executable](#) (EXE). Inside an assembly are one or more independent units called modules.

The main module of an assembly contains what is known as the assembly [manifest](#), which includes information such as the assembly name, version, culture, and lists of all the module parts of the assembly itself.

The metadata of the module contains a complete description of all the types exposed, which includes available methods, properties, fields, parameters, constructors, etc. An essential part of this metadata is the [Common Intermediate Language](#) (CIL) compiled from the source code. Assemblies can also contain resources, which can include images, text files, and strings for localization.

Assemblies can be discussed from many perspectives, but because Reflection works with metadata and CIL, that's what we'll focus on.

Assemblies are essentially self-describing, and they contain all the necessary information for the .NET runtime to execute the code. This contains information about the assembly being executed, other assemblies referenced by the assembly, all of the types within the assembly, and the CIL code.

Using a free tool called [ILSpy](#), we can dig into the details of any assembly, explore its metadata, and automatically decompile its source code from CIL to its original, high-level .NET language. Visual Studio ships with a Spy++ program, but be aware that ILSpy is very similar to .NET Reflector from Red Gate Software, which is easy and intuitive to use.

The CIL code is the actual runnable part of the assembly. Rather than compiling a high-level, human-understandable language such as C# to machine code, .NET takes an intermediate step, with the compiler generating CIL code that sits somewhere between human-readable code and machine code. CIL is somewhat familiar with other low-level languages.

In .NET, assemblies are compiled [just-in-time](#) (JIT), which means that compilation is actually done during the execution of the program rather than prior to execution. Essentially, the actual machine code is not generated until the application is executed. The .NET runtime takes care of converting the CIL into OS instructions that the CPU can understand. This is advantageous because the code is compiled for the machine it is running on (it can actually be optimized for that particular machine), and because only parts of the code that are actually run get compiled. So parts of the application that are not used will remain in CIL until executed.

ILSpy requires no installation. It is a zipped archive that contains an executable and a set of DLLs when uncompressed, which means it can be used to inspect any .NET assembly.

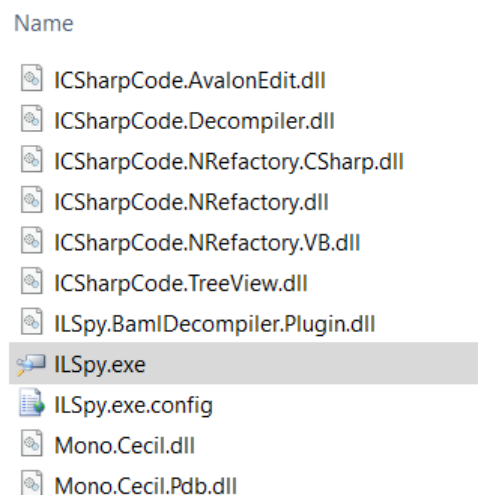


Figure 20: ILSpy after Downloading and Unzipping

With ILSpy, let's quickly inspect the metadata of the **Newtonsoft.Json** assembly we previously used in our Simple Awesome CRM project.

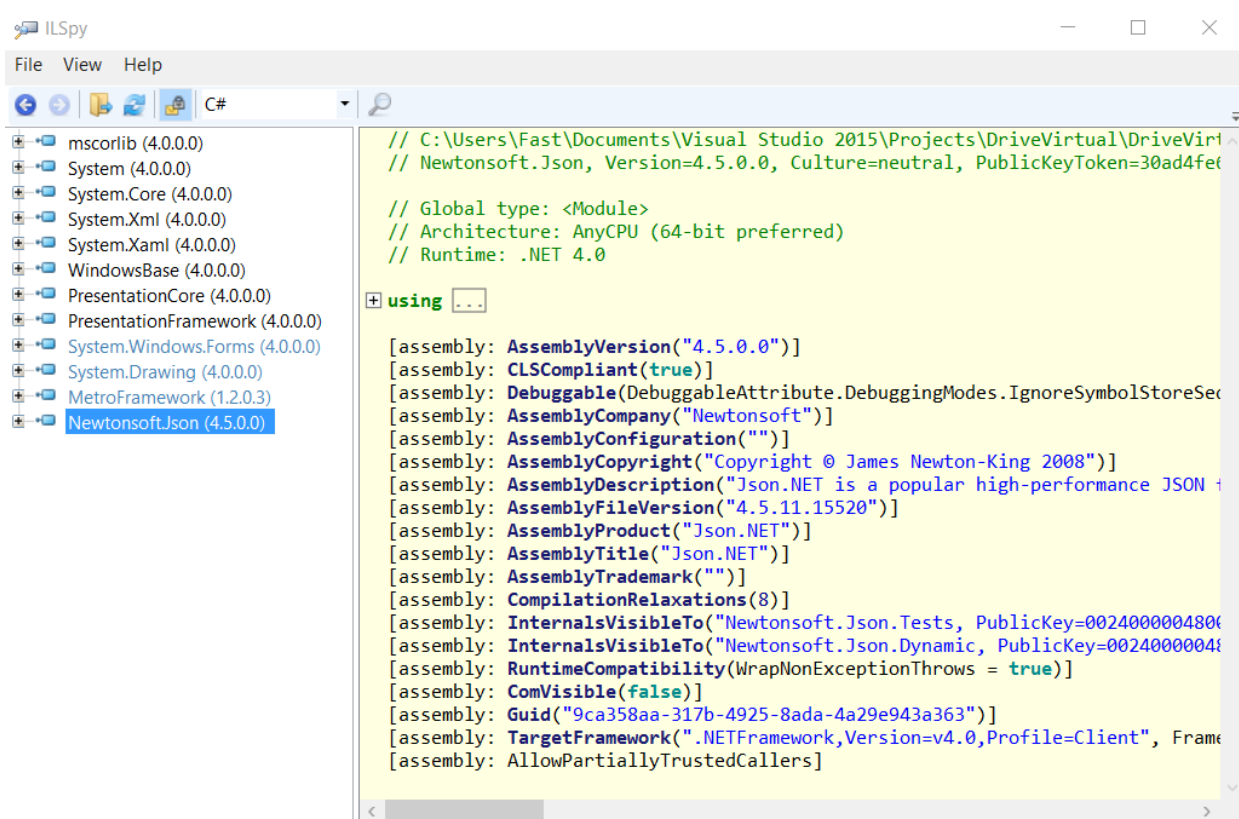


Figure 21: Assembly Metadata Properties Seen with ILSpy

Based on the metadata, Reflection allows us to dynamically create types, invoke methods, get and set the values of properties, and inspect attributes. The **System.Reflection** namespace offers all the necessary .NET classes and methods needed to perform all of these actions.

Arguably the most important feature of the **System.Reflection** namespace is the **Type** class. This class has more than 150 members, including static instance members and properties. Some of the most common methods in the **Type** class include the **Get** methods, such as **GetType**, **GetMemberInfo**, **GetPropertyInfo**, **GetFieldInfo**, and many more.

The **Activator** class is another extremely important class within the **System.Reflection** namespace. The **Activator** class has a set of static methods that can be used to create an instance of any specific type, such as the **CreateInstance** method.

The **Assembly** class can be used to load an assembly from a file or the current program context. Once the assembly is loaded, we can use Reflection to see which types are available. The **Assembly** class contains various useful methods, such as **Load**, **LoadFrom**, **GetTypes**, **GetName**, **GetFiles**, etc.

There is also an **ILGenerator** class that can be used to create our own CIL code. While it is useful to know that this class exists, that topic goes far beyond the scope of this chapter and e-book.

Going forward, we'll focus on the aspects of Reflection that are useful in everyday application development and can be helpful for troubleshooting. We won't specifically focus on features more suited for creation of advanced software development or reverse engineering tools like obfuscation.

Speed and consistency

Reflection can be well suited for daily application development and troubleshooting. But what can be done and what should be done are not always the same thing.

Prior to the introduction of the **dynamic** keyword in .NET 4.0, the only way to interact with properties on COM objects was to do Reflection on the object itself. So what could be done with Reflection prior to .NET 4.0 in order to get the value of a COM object property should no longer be done in .NET 4.0 and beyond.

It is also feasible to perform Reflection on a private field (but it's usually not a good idea) and also on any particular method.

You can also get a CLR type based on the type name (using the **Type** class), create an instance of a type (using the **Activator** class), load an assembly based on the file name (using the **Assembly** class, which is commonly used when designing applications that allow plugins), get the types exposed on an assembly (using the **Assembly** class), or check to see if a **Type** implements an **Interface** (using the **Type** class).

The key point to understand here is that you should only use Reflection when you really need it. Let's explore a few examples that show some drawbacks regarding speed and accuracy when using Reflection. We'll do this by analyzing how we can create a **List** with and without Reflection. Then we'll compare the speed of both.

Code Listing 27: Speed Comparison with and without Reflection

```
using System.Diagnostics;
using System;
using System.Collections.Generic;

namespace ReflectionExamples
{
    class Program
    {
        static void Main(string[] args)
        {
            CreateListNormally();
            CreateListReflection();
            Console.ReadLine();
        }
    }
}
```

```

    }

    public static void CreateListNormally()
    {
        Stopwatch sw = Stopwatch.StartNew();

        for(int i = 0; i <= 1000; i++)
        {
            List<string> l = new List<string>();
        }

        sw.Stop();
        Console.WriteLine("CreateListNormally -> Time taken: {0}ms",
            sw.Elapsed.TotalMilliseconds);
    }

    public static void CreateListReflection()
    {
        Stopwatch sw = Stopwatch.StartNew();

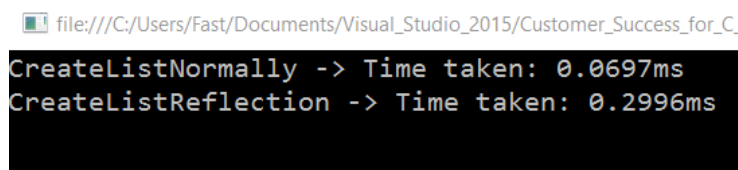
        Type lType = typeof(List<int>);

        for (int i = 0; i <= 1000; i++)
        {
            var l = Activator.CreateInstance(lType);
        }

        sw.Stop();
        Console.WriteLine("CreateListReflection -> Time taken:
            {0}ms", sw.Elapsed.TotalMilliseconds);
    }
}

```

This produces the output shown in Figure 22.



```

file:///C:/Users/Fast/Documents/Visual_Studio_2015/Customer_Success_for_C_
CreateListNormally -> Time taken: 0.0697ms
CreateListReflection -> Time taken: 0.2996ms

```

Figure 22: Output of the Speed Comparison with and without Reflection

In this example, using Reflection takes 4.298 times longer than using standard code. The power of creating the needed **Type** during runtime using **CreateInstance** comes at the price of performance.

Let's now examine the other big drawback of Reflection—consistency.

We cannot expect the same results from executing private methods and properties that we get with public methods and properties.

Code Listing 28 shows a library that we reference in our application. Let's assume this is a vendor library that we have no control over but that we need in order to troubleshoot a specific problem our customer has raised.

The vendor has simply provided the assembly we can use to reference in our application.

Code Listing 28: Vendor Library Source Code

```
using System;
using System.Collections.Generic;

namespace ReflexionVendorSampleAssembly
{
    public class VendorAssembly
    {
        private const string cStr = "dd-MMM-yyyy hh:mm:ss UTC";

        private DateTime dtDate;
        private List<string> pItems;

        public string Info
        {
            get { return dtDate.ToString(cStr); }
        }

        public List<string> Items
        {
            get { return pItems; }
            set { pItems = value; }
        }

        public VendorAssembly()
        {
            pItems = new List<string>();
            AddNewItem();
        }

        public void AddNewItem()
        {
            dtDate = DateTime.UtcNow;
            pItems.Add(dtDate.ToString(cStr));

            Console.WriteLine("AddNewItem -> " +
                pItems.Count.ToString() + " - " + dtDate.ToString(cStr));
        }

        private void AddNewItemPriv()
```

```

        {
            Console.WriteLine("AddNewItemPriv -> " +
                pItems.Count.ToString() + " - " + dtDate.ToString(cStr));
        }
    }
}

```

Now let's call this vendor library by standard means (assuming we have the source code) and by using Reflection (assuming we don't have the source code).

Code Listing 29: Calling the Vendor Library with and without Reflection

```

using System.Diagnostics;
using System;
using System.Collections.Generic;

using ReflexionVendorSampleAssembly; //If we have the vendor's code.
using System.Reflection;

namespace ReflectionExamples
{
    class Program
    {
        static void Main(string[] args)
        {
            VendorPublicItems();
            VendorPrivateItems();

            Console.ReadLine();
        }

        public static void VendorPublicItems()
        {
            Console.WriteLine("VendorPublicItems");

            VendorAssembly publicVendor = new VendorAssembly();
            publicVendor.AddNewItem();
        }

        public static void VendorPrivateItems()
        {
            Console.WriteLine("VendorPrivateItems");

            Type vendorType = typeof(VendorAssembly);

            var act = Activator.CreateInstance(vendorType);
            VendorAssembly pv = (VendorAssembly)act;
        }
    }
}

```

```

        MethodInfo dMet = vendorType.GetMethod("AddNewItemPriv",
            BindingFlags.NonPublic | BindingFlags.Instance);

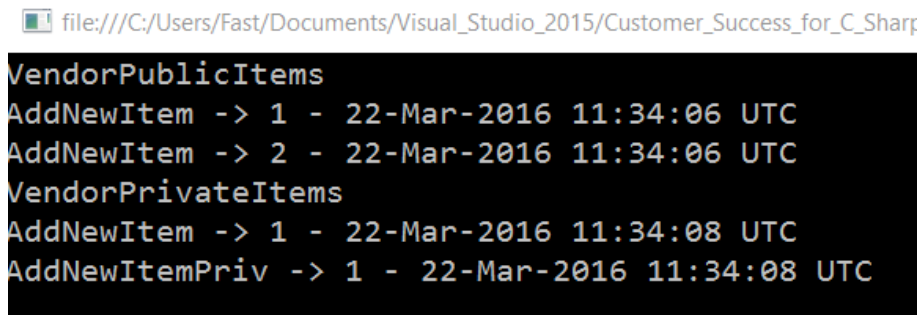
        dMet.Invoke(pv, null);
    }
}

```

With the **VendorPublicItems** method, we are assuming that we have the vendor's library source code, which means we can create an instance of the **VendorAssembly** class, as usual, and then invoke the **AddNewItem** method.

With the **VendorPrivateItems** method, we assume that we do not have the vendor's library source code, which means we must use Reflection to invoke the **AddNewItemPriv** method. In this particular example, we will explicitly call a private method within the vendor's assembly to show that although this is technically possible, it is not necessarily a good practice.

The code execution of this example results in the output shown in Figure 23.



```

file:///C:/Users/Fast/Documents/Visual_Studio_2015/Custom...
VendorPublicItems
AddNewItem -> 1 - 22-Mar-2016 11:34:06 UTC
AddNewItem -> 2 - 22-Mar-2016 11:34:06 UTC
VendorPrivateItems
AddNewItem -> 1 - 22-Mar-2016 11:34:08 UTC
AddNewItemPriv -> 1 - 22-Mar-2016 11:34:08 UTC

```

Figure 23: Output of the Vendor's Library

Let's examine this closely.

When running **VendorPublicItems**, we see that **AddNewItem** is called by the constructor and invoked immediately after an instance of **VendorAssembly** has been created. This results in having two items on the **VendorAssembly.Items** list.

However, when **VendorPrivateItems** is called, only a single item is added to the **VendorAssembly.Items** list.

This happens because after calling **CreateInstance**, **GetMethod** is executed referencing **AddNewItemPriv**, which doesn't actually add any element to the **VendorAssembly.Items** list. Instead, it simply displays the item that already exists.

Even though Reflection allows us to execute private methods within an assembly, we should avoid following this practice. Typically, private methods and properties do not execute the same logic as their public counterparts. They have been marked private for a good reason.

By executing a private method or property through Reflection (for an assembly for which we might not have the source code), we are not ensuring that our intended results will be the same as those we would get when executing a public counterpart method or property.

To illustrate this, we made it quite obvious in the vendor's code that the logic within the **AddNewItem** and **AddNewItemPriv** would be different.

In a real-world scenario, you might not have the vendor's library source code at hand, so as a rule of thumb, when using Reflection make sure to invoke public methods and properties. This will guarantee that you get the same results as when you have the source code at hand and instantiate the class normally (without using Reflection).

Let's change the **VendorPrivateItems** method to invoke the public **AddNewItem** method.

Code Listing 30: VendorPrivateItems Adjusted to Invoke the AddNewItem Method

```
using System.Diagnostics;
using System;
using System.Collections.Generic;

using ReflexionVendorSampleAssembly; //If we have the vendor's code.
using System.Reflection;

namespace ReflectionExamples
{
    class Program
    {
        static void Main(string[] args)
        {
            VendorPublicItems();
            VendorPrivateItems();

            Console.ReadLine();
        }

        public static void VendorPublicItems()
        {
            Console.WriteLine("VendorPublicItems");

            VendorAssembly publicVendor = new VendorAssembly();
            publicVendor.AddNewItem();
        }

        public static void VendorPrivateItems()
        {
            Console.WriteLine("VendorPrivateItems");

            Type vendorType = typeof(VendorAssembly);
```

```

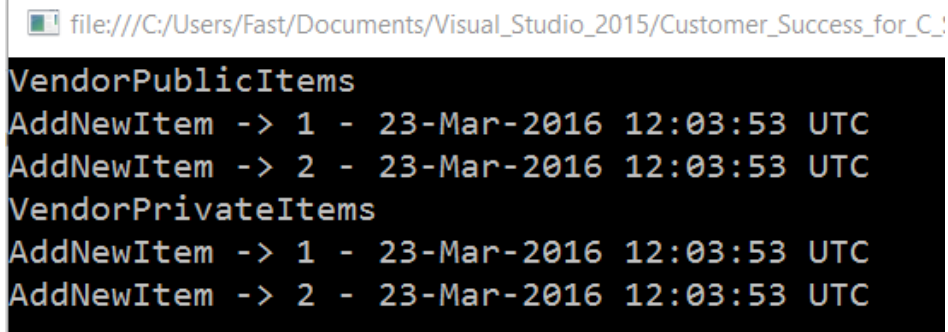
        var act = Activator.CreateInstance(vendorType);
        VendorAssembly pv = (VendorAssembly)act;

        MethodInfo dMet = vendorType.GetMethod("AddNewItem",
            BindingFlags.Public | BindingFlags.Instance);

        dMet.Invoke(pv, null);
    }
}

```

Notice that both methods **VendorPrivateItems** (using Reflection) and **VendorPublicItems** (without Reflection) produce exactly the same results.



```

file:///C:/Users/Fast/Documents/Visual_Studio_2015/Custom_Success_for_C_!
VendorPublicItems
AddNewItem -> 1 - 23-Mar-2016 12:03:53 UTC
AddNewItem -> 2 - 23-Mar-2016 12:03:53 UTC
VendorPrivateItems
AddNewItem -> 1 - 23-Mar-2016 12:03:53 UTC
AddNewItem -> 2 - 23-Mar-2016 12:03:53 UTC

```

Figure 24: Output of the Vendor's Library after Adjusting

The only actions taken were changing the name of the method being retrieved by **GetMethod** from **AddNewItemPriv** to **AddNewItem** and changing **BindingFlags.NonPublic** to **BindingFlags.Public**.

Reflection strategy

In the previous example, the main reason our results were not consistent is that we violated the principle of encapsulation.

Classes are black boxes that have clearly defined input exposed to the outside world (public methods and properties). They also have clearly defined output exposed through properties or method return values.

So long as we stick to the basic rules of encapsulation and use the exposed input and output of the class we are interacting with, we are fine. However, if we try to dig inside the box, we can run into unexpected behavior, and this is exactly what happened in the previous demo.

We should not peek inside the box. Instead, we should use the exposed interfaces, public methods, and properties. In Reflection, this means we should not interact with any nonpublic members of any class.

We also saw that our speed is reduced when we use Reflection. This can be mitigated by programming to an abstraction rather than a concrete type or, in other words, by writing your code to target an interface rather than using a specific class.

When your code works with interfaces through Reflection instead of the dynamically created type, performance will increase.

Here's a strategy you should consider when working with Reflection—if you want the flexibility of choosing functionality at runtime, you should dynamically load assemblies only once (at application start-up). Again—make sure that the assembly is loaded just once. It is also important to dynamically load any types from those assemblies at start-up.

Most importantly, once you have a dynamically loaded type, this should be cast to an interface that your application already knows about. Doing this will ensure your application will make all method calls through the interface (not using `MethodInfo.Invoke`), which will help reduce the performance hit when using Reflection.

Since interfaces have only public members, you will avoid interacting with any private members of the class and also avoid experiencing unexpected behaviors from breaking encapsulation.

Table 7 summarizes the proposed Reflection strategy.

Table 7: Proposed Reflection Strategy

Strategic Item	Remark
Dynamically Load Assemblies	Once and during application start-up
Dynamically Load Types	Once and during application start-up
Cast Types to Known Interfaces	All method calls go through the interface
	No usage of <code>MethodInfo.Invoke</code>
Stick to Encapsulation	Don't interact with private members

Following this strategy will limit the use of Reflection to only what you specifically need. You will also maximize performance and accuracy while maintaining flexibility.

This is all valid for web (ASP.NET, MVC, and WebForms), console, and desktop (Windows Forms and WPF) applications written in C#. However, because Windows Store apps are sandboxed, Reflection features are limited.

There's no access allowed to nonpublic methods or types (which is a good thing), and there's no dynamic loading of assemblies available from the file system. This is because all the assemblies that a Windows Store application needs must be submitted as a package to the Windows Store.

Pluggable agents

Having seen what Reflection can do, let's examine how we can make use of it in our applications following the strategy we have outlined.

Let's say we are working with an application that browses specific job websites to get data on skills trends in the market. We won't actually build the specific logic of gathering data from the websites, but we'll show how we can create a pluggable architecture following the principles outlined in the proposed strategy.

To achieve this, an abstraction layer must be created, and it should consist of pluggable agents that adhere to a common interface. Each pluggable agent will gather information from a different job website, but all the plugins will share the same common abstraction in order to plug into the main system.

First, let's create the interface. We will keep it simple and use only a method that all plugins should implement called **NavigateToSite**.

Code Listing 31: Plugin Interface

```
namespace TrendsAgent
{
    public interface IPluginInterface
    {
        string[] NavigateToSite(string url);
    }
}
```

This method, which will be implemented by all pluggable agents, will simply receive as a parameter the URL of the website and return a string array with all the HTML lines of the webpage retrieved.

With this interface defined, we'll use **GetType(string typeName)** from the **Type** class, where **typeName** is the assembly-qualified type name, to properly implement our Reflection strategy.

Once we have the **Type** object, we can use the **CreateInstance(Type type)** method of the **Activator** class to create an instance of a **Type** that was obtained with **GetType(string typeName)**.

The assembly-qualified type name contains five parts:

- Fully-qualified type name (namespace and type).
- Assembly name.
- Assembly version.

- Assembly culture.
- Assembly public (for strongly-named assemblies).

The fact that .NET allows us to specify an assembly version and culture is great because this allows us to use different versions of the assembly or a version specific to a certain locale (great when localization is required) during run time.

Limiting the use of Reflection to loading and creating the pluggable agents will make for better performance, however we will not use Reflection to dynamically invoke members. Instead, we'll take the instance created and interact with it using the **IPluginInterface**. By interacting with the interface, we'll be able to directly call the **NavigateToSite** method.

In order to have full separation of concerns (so we can load the pluggable agent's assembly from the main application using Reflection), the agent's code should be defined in a separate Visual Studio Project and not in the one used for the **IPluginInterface**.

The Visual Studio project that contains **IPluginInterface** is the host application that will load the pluggable agent.

Let's jump into some code and see how we can implement a pluggable agent.

Code Listing 32: Pluggable Agent

```
using System.Net;
using TrendsAgent;

namespace TrendsPlugin
{
    public class PluginDemo : IPluginInterface
    {
        public string[] NavigateToSite(string url)
        {
            string[] p = null;

            using (var client = new WebClient())
            {
                client.Headers["User-Agent"] =
                    "Mozilla/4.0 (Compatible; Windows NT 5.1; MSIE 6.0) "
                    +
                    "(compatible; MSIE 6.0; Windows NT 5.1; " +
                    ".NET CLR 1.1.4322; .NET CLR 2.0.50727)";

                using (var stream = client.OpenRead(url))
                {
                    string value = client.DownloadString(url);
                    p = value.Split('\r');
                }
            }
        }
    }
}
```

```

        return p;
    }
}

```

Here you can see how the **PluginDemo** class from **TrendsPlugin** implements the **IPluginInterface** from the host program.

In this **PluginDemo** class, **NavigateToSite** fetches the HTML content of the given URL and returns it as a string array.

Now let's use limited Reflection from the main program to examine the how the pluggable agent **PluginDemo** is called.

Code Listing 33: Using Reflection to Invoke the Pluggable Agent from the Main Method

```

using System;

namespace TrendsAgent
{
    class Program
    {
        static void Main(string[] args)
        {
            LoadRunPlugin();
            Console.ReadLine();
        }

        public static void LoadRunPlugin()
        {
            string pluginName = "TrendsPlugin.PluginDemo, TrendsPlugin,
                Version=1.0.0.0, Culture=neutral, PublicKeyToken=null";

            Type pluginType = Type.GetType(pluginName);
            object pluginInstance = Activator.CreateInstance(pluginType);
            IPluginInterface plugin = pluginInstance as IPluginInterface;

            string[] l = plugin.NavigateToSite("http://google.com");
            Console.WriteLine("Lines fetched: " + l.Length.ToString());
        }
    }
}

```

Let's use the assembly-qualified type name **TrendsPlugin.PluginDemo, TrendsPlugin, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null** to take a closer look.

We are able to load the assembly's type into the main program (using the **LoadRunPlugin** method). However, for this to happen, the TrendsPlugin.dll (which contains the **PluginDemo** class) must be in the same folder from which the main program is executed.

An easy way to get the assembly-qualified type name is to open the TrendsPlugin.dll in ILSpy. This is clearly visible at the top of the ILSpy screen in **green**.

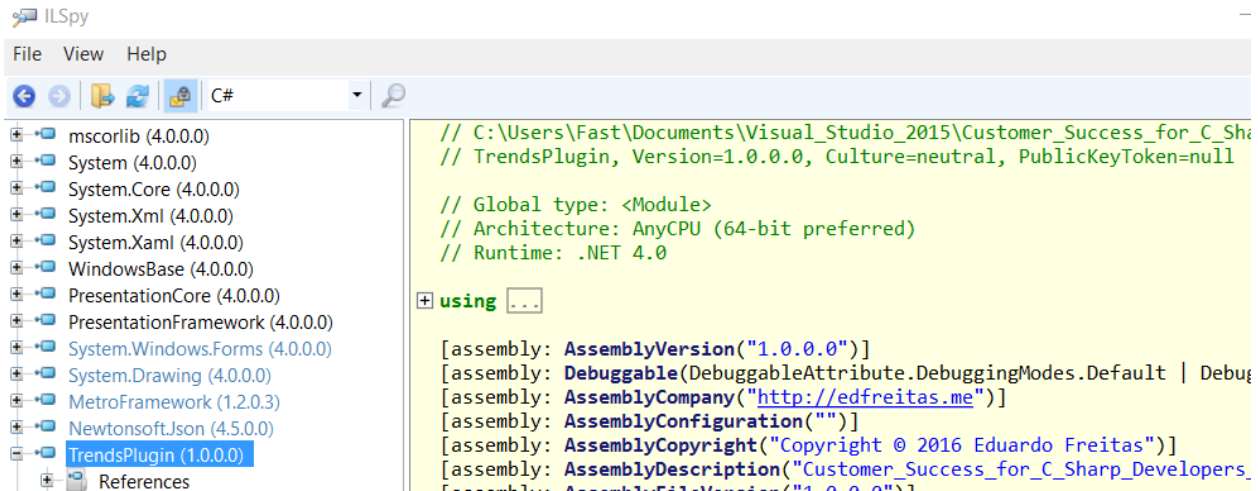


Figure 25: Using ILSpy to Get the Assembly-Qualified Type Name

Having **pluginType**, next we create an instance of it using **Activator.CreateInstance**. This will return an **object** that can be cast into an **IPluginInterface** instance on which **NavigateToSite** can be invoked.

By doing this, we are able to use minimal features from Reflection, adhere to the correct strategy, and follow the rules of encapsulation in order to load and execute the pluggable agent.

Best practices

We've seen how we can use Reflection to inspect assemblies and how to use it to invoke members of classes for which we might not have the source code.

In essence, the purpose of using Reflection in this e-book is to simply illustrate the power this technology has and how it can enable us to be more productive when troubleshooting—especially when dealing with third-party libraries.

It is true that Reflection is widely used, especially for the creation of developments tools (ILSpy is a great example of such a tool).

So, is .NET code really safe?

Frankly it's not much different from other compiled languages. If someone has access to the compiled file, this can be reversed using Reflection or any of the tools available in the market, including .NET Reflector, dotPeek, ILDASM, JustDecompile, Moq, Unity, and ILSpy.

There are many other similar tools for both Java applications and disassemblers for native Win32/64 executables. For example: Procyon, Krakatau, and CFT for Java, and Boomerang, ExeToC, and IDA for Win32/64 executables.

If the files are on the user's machine, they can be reversed and examined. However, this doesn't mean we are hopeless. There are several things that can be done to minimize the risk.

Let's explore some ways in which this can be done.

SOA

Limiting what actually runs on the client machine and keeping proprietary code (which gives us a competitive business advantage) on the server are two ways of minimizing risk.

Table 8: Client versus Server Execution

Client Machine	Server
Nonsecret code runs here	Proprietary code runs here
Calls server for secrets	Returns results to the client

When the client requests the server for the information it needs, there should be some authentication process built into the server's code that verifies that the request is coming from a valid client machine.

Once this verification takes place, the server will return the results to the client. These should be encrypted and returned. This very common architecture is known as [service-oriented architecture](#) (SOA), which you might have heard about.

Running a server service is an integral part of component thinking, and it is clear that distributed architectures were early attempts to implement service-oriented architecture.

Most of the time, web services are mistaken as the same thing as SOA, but web services are part of the wider picture that is SOA.

Web services provide us with certain architectural characteristics and benefits—specifically platform independence, loose coupling, self-description, and discovery—and they can enable a formal separation between the provider and consumer because of the formality of the interface.

Service is the important concept. Web services are the set of protocols by which services can be published, discovered, and used in a technology-neutral and standard way.

Web services are not necessarily a compulsory component of an SOA, although increasingly they have become so. SOA is much wider in its scope than simply defining service implementations. It addresses the quality of the service from the perspective of the provider (server) and the consumer (client application).

Table 9: SOA versus Web Services

SOA	Web Services
Service is abstracted from implementation	Service is consumable and discoverable
Published service interface	Use of the service itself—not copied from the server to the client
Contract between the consumer and provider	Endpoint and platform independent

A well-formed service provides us with a unit of management that relates to business usage.

The client machine requests what it needs from the server, providing us with a basis for understanding the lifecycle costs of the service and how the lifecycle is used within the business.

With this information, we can design better applications, and we can design them to be better aligned with the realities and expectations of businesses.

Secrets

When an assembly is opened with tool like ILSpy, any string contained within the assembly is easily revealed.

Neither passwords nor sensitive strings should be hard coded. You can avoid this by using a password linked to a user's authentication. However, this is not always possible or practical.

Another option is to have the client application request the application token from the server or have it run the code that requires the password on the server.

In an absolute worst-case scenario in which a password or token needs to be stored on the client application, it must be obfuscated to the extent that it cannot be exposed when the code is decompiled.

Obfuscation

Obfuscation is a way to make CIL code less readable. We cannot prevent Reflection; however, we can rename private variables, fields, properties, parameters, methods, and constants to nonsense names that will make the code harder to read when decompiled.

The code still continues to work, but it will be harder for human beings to understand. When obfuscation takes place, it might be possible that Reflection won't work anymore on private members (decompilers cannot follow the code). However, public member names usually remain unchanged.

In a way, obfuscation is similar to JavaScript minification. The main difference is that minification's purpose is for faster downloading of a JavaScript library to the browser. Minification reduces the download payload by removing all unnecessary characters in the JavaScript code.

There are also some advanced obfuscators that go a step further and try to prevent any disassembly. This is typically done by using unprintable characters (which is still valid CIL) when renaming private variables, fields, properties, parameters, methods, and constants to nonsense names.

Keep in mind that ultimately this is an ongoing battle between obfuscators and decompilers. As one evolves, so does the other.

Some well-known obfuscators include Dotfuscator (a free version with fairly limited functionality ships with Visual Studio) and two commercial products—Eazfuscator and Deep Sea Obfuscator. There are many more.

Obfuscation is a very broad topic that we won't cover in this e-book, but reviewing and understanding it is certainly worthwhile.

Summary

Throughout this chapter we've explored how Reflection can help us understand code to which we might not have access.

Sometimes, when trying to replicate production problems or trying to analyze software issues, we might be faced with the challenge of understanding third-party libraries used by our application. This is where Reflection can give us a hand.

In this e-book, we've turned upside down the notion of what a helpdesk at a software company might look like. We've also set forth guidelines to increase customer loyalty and awareness.

Here are the key reasons why customer loyalty is so important:

Incrementing Referrals: The customer lifecycle has become more complex because customers begin their buying journey even before they directly engage with your product and brand.

Therefore, customer-relationship management should begin at the very first contact. This is why having an embedded Simple Awesome CRM Tool within your product can be invaluable as a strategy.

Reduced Costs: By cultivating customer loyalty, your business can create a legion of strong evangelists who outshine your best marketing efforts. These evangelists can help save huge advertising and marketing costs for your company because word of mouth is a powerful and cost-effective marketing tool.

Furthermore, customer retention efforts are cheaper than acquiring new customers. Building loyalty forms a solid customer base that helps your business grow, potentially even exponentially.

Repeat Business: Because your organization might have already won the trust of loyal customers by applying these techniques, it is easier to cross-sell and up-sell to these customers compared to new ones.

If you maintain an open line of communication with loyal customers, they will generally provide honest feedback to help improve your brand. And because they will openly voice their comments, your team can make a concerted effort to address issues raised promptly, which will increase customer satisfaction.

Loyal customers are likely to ensure repeated business, which is critical for your company's long-term survival.

Insulation from Competitors: Loyal customers will stick to your brand because they trust you.

If your organization has a loyal customer base, you should be fairly immune to both competition and economic changes. This helps to protect your bottom line.

Customer loyalty strengthens your product and brand image, and it builds equity. A truly positive reputation can help your business scale to greater heights. Your team should focus on treating customers better and rewarding them for their loyalty. After all, customers are the biggest assets your business can have.

The code samples presented in this e-book were created with Visual Studio 2015 and .NET 4.5.2 and can be found here:

<https://bitbucket.org/syncfusiontech/customer-success-for-c-developers-succinctly>

I hope this e-book has inspired you to start a journey to achieving greater customer success.

Thank you.