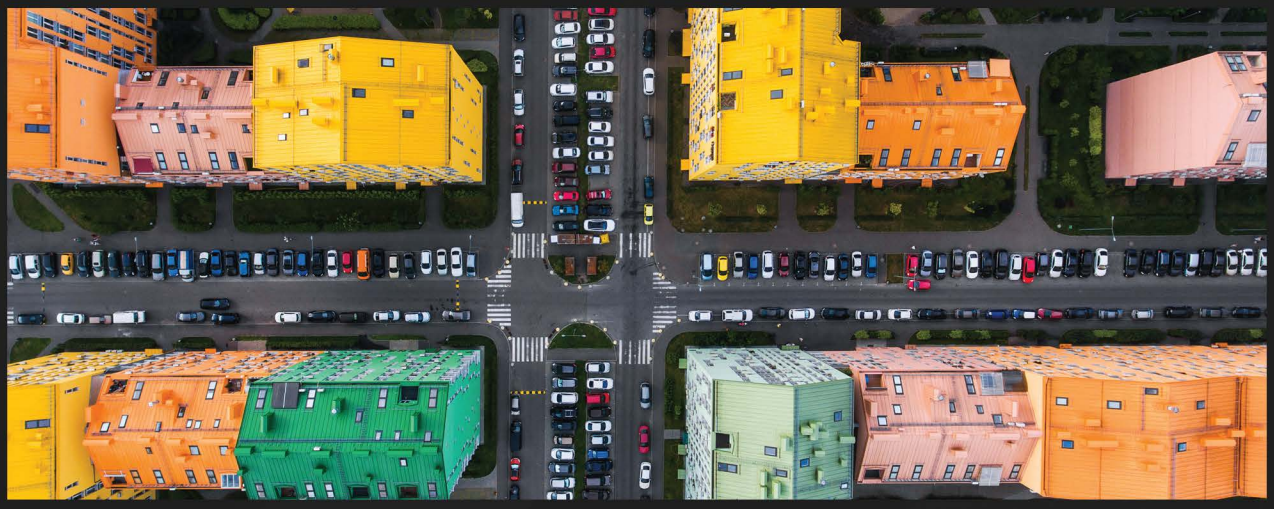


1ST EDITION

Building CLI Applications with C# and .NET

A step-by-step guide to developing cross-platform CLI apps—from coding and testing to deployment

TIDJANI BELMANSOUR

Foreword by Damian Brady, Staff Developer Advocate, GitHub

Formerly DevOps Advocate at Microsoft

Building CLI Applications with C# and .NET

A step-by-step guide to developing cross-platform CLI apps—from coding and testing to deployment

Tidjani Belmansour

‹packt›

Building CLI Applications with C# and .NET

The author acknowledges the use of cutting-edge AI, such as ChatGPT, with the sole aim of enhancing the language and clarity within the book, thereby ensuring a smooth reading experience for readers. It's important to note that the content itself has been crafted by the author and edited by a professional publishing team.

Group Product Manager: Kunal Sawant
Publishing Product Manager: Samriddhi Murarka
Book Project Manager: Prajakta Naik
Lead Editor: Kinnari Chohan
Technical Editor: Vidhisha Patidar
Copy Editor: Safis Editing
Proofreader: Kinnari Chohan
Indexer: Tejal Soni
Production Designer: Alishon Mendonca
Senior DevRel Marketing Executive: Sonia Chauhan

First published: February 2025

Production reference: 2170225

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK.

ISBN 978-1-83588-274-0

www.packtpub.com

To my wife Lamia Rarrbo and to my daughter Camélia, for being the sunshine of my life. To my family for your love and support.

– Tidjani Belmansour

Foreword

I remember when I first used Git.

I'd spent years working with Team Foundation Server (using `TFVC`), and I'd always done so through Visual Studio, so I'd become very used to the GUI. But Git felt like a different animal. I resolved to learn to use it from the Windows command line rather than relying on pointing and clicking.

It turned out to be a great decision. The CLI felt powerful and raw and exposed me to more of how Git worked. The simple request-response pattern was natural, and each command was self-contained. It meant that later, when I used GUI tools such as `GitHub Desktop`, I still understood what was happening under the hood. In subsequent jobs, no matter how unfamiliar the operating system, IDE, or development environment was, it always felt like I could drop back to the terminal to "talk to Git directly."

Git was my gateway tool, but I quickly learned to love the simplicity of a CLI. It was straightforward when it needed to be, but for more power, I learned how to chain commands together or to use tools such as `jq` to parse JSON responses. For integration with my own software or scripts, well-written CLI tools meant I could make requests of APIs without having to construct exactly the right HTTP headers (is it `text/json` or `application/json`?).

When my career moved increasingly toward DevOps and developer processes, CLI tools became a core part of my practice, particularly when constructing CI/CD pipelines. It didn't matter what tool I was using; it was easy to call a CLI tool to get the job done. CLIs were often cross-platform and consistent regardless of where you executed them. Switching from Jenkins to Azure Pipelines to GitHub Actions was not a big deal. It involved the same commands – they were just called from a different orchestrator.

Over time, I've come to love CLI tools for their simplicity, power, and directness. It brings me great joy when I see a CLI option alongside an API or GUI. It's just another way to work with the software I use every day.

Applications and websites are frequently written with beautiful user interfaces, and these interfaces are important. But I think we'd be well served if more developers thought about – and built – the CLI as a first-class interface.

Damian Brady

Staff Developer Advocate, GitHub

Formerly DevOps Advocate at Microsoft

Contributors

About the author

Tidjani Belmansour is an expert in developing and architecting solutions on the Microsoft Azure platform, particularly in .NET, with over 21 years of experience. His passion for development began early—he wrote his first program in QuickBasic at the age of eight and has never stopped since. He is currently the Director of the Azure Centre of Excellence at Cofomo, where he collaborates with organizations of all sizes, across both public and private sectors, and on projects around the globe.

Since 2019, Tidjani has been recognized as a Microsoft Azure MVP. He is also the co-host of the Azure Quebec Community, a trainer, a blogger, and an international speaker. Tidjani holds a BSc in Computer Science and a Ph.D. in Engineering.

This book is dedicated to my wife, Lamia, and my daughter, Camélia, for bringing light into my life and for being my source of support and inspiration. I love you both more than words can express.

To my family: my mother, Fatiha; my sister, Lamia; my second mom, Nacera; my sisters-in-law, Assia and Meriem; my brothers-in-law, Karim and Jordan; and my nephews, Zaki and Yani.

And to the memory of my father, Omar; my second father, Nasseradine; and my brother, Mounir.

About the reviewer

Mabrouk Mahdhi is a software engineer, Microsoft MVP, and the founder of CodeCampsis, a consulting firm specializing in innovative IT solutions and .NET technologies. With a passion for technology and a commitment to sharing knowledge, Mabrouk is also a book author and an accomplished speaker, inspiring and educating audiences on a global scale.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/BuildNETCLI`

Table of Contents

3

Basic Concepts of Console Applications in .NET 23

Part 2: Foundations of Building CLI Applications

4

Command-Line Parsing 45

5

Input/Output and File Handling 67

8

Building Modular and Extensible CLI Applications 133

9

Working with External APIs and Services 151

Part 4: Testing and Deployment

10

Testing CLI Applications 169

11

Packaging and Deployment 195

Part 5: Advanced Techniques and Best Practices

12

Performance Optimization and Tuning 223

13

Security Considerations for CLI Applications 239

14

Preface

CLI applications are everywhere! Once you start paying attention, you'll begin to notice them.

There's a reason for their ubiquity: CLI applications enhance productivity by helping us stay focused on the task at hand and enabling the automation of repetitive tasks that would be error-prone if done manually. In fact, many graphical user interface (GUI) applications include a CLI option for precisely this reason.

By learning how to develop your own CLI applications, you can deliver significant business value to your users and customers.

And that's exactly what we'll explore together in the pages of this book!

Who this book is for

This book is for developers, architects, and software or DevOps engineers who recognize the value of CLI applications and want to create their own—whether to meet personal needs or those of their customers.

To get the most out of this book, you should have a foundational understanding of .NET, along with practical experience in C# and Git.

What this book covers

Chapter 1, *Introduction to CLI Applications*, introduces CLI applications and explains why they matter.

Chapter 2, *Setting Up the Development Environment*, describes how to set up your development environment.

Chapter 3, *Basic Concepts of Console Applications in .NET*, explains that there is a console application behind every CLI application. Therefore, we will start our journey by discussing console applications.

Chapter 4, *Command-Line Parsing*, converts our console application into a real CLI application, and we learn how to parse its arguments.

Chapter 5, *Input/Output and File Handling*, delves into the essential concepts of reading from and writing to files, a common task for CLI applications.

Chapter 6, *Error Handling and Logging*, explains how to handle errors to allow for graceful handling of these errors by the application and we will learn how to log relevant information for later analysis.

Chapter 7, *Interactive CLI Applications*, teaches techniques to enhance the user experience by creating interactive and visually attractive CLI applications.

Chapter 8, *Building Modular and Extensible CLI Applications*, delves into techniques to make our CLI applications easier to maintain and extend.

Chapter 9, *Working with External APIs and Services*, covers how to consume and work with external dependencies, including web services to extend the capabilities of our CLI applications.

Chapter 10, *Testing CLI Applications*, explains how to test our CLI applications, as testing is one of the key steps of the application development lifecycle.

Chapter 11, *Packaging and Deployment*, delves into how to package and distribute our CLI applications to our users and customers using the most widely used distribution methods.

Chapter 12, *Performance Optimization and Tuning*, covers how to spot performance issues along with the common techniques to enhance our applications' performance, making our users happy.

Chapter 13, *Security Considerations for CLI Applications*, explores key security areas, explores tools that help us assess the security posture of our applications, and teaches some techniques to secure remote communications.

Chapter 14, *Additional Resources and Libraries*, explores additional material that will help you deep dive into the various concepts we covered in this book.

To get the most out of this book

The table below shows the required software to run the examples of this book. All code examples have been tested using .NET 8 on Windows. However, they should work with future version releases too.

Software/hardware covered in the book	**Operating system requirements**
.NET 8	Windows, macOS, or Linux
Visual Studio Code	
Git	

In Chapter 2, we will explain in detail the steps required to set up your development environment.

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from GitHub at `https://github.com/PacktPublishing/Building-CLI-Applications-with-C-Sharp-and-.NET`. If there's an update to the code, it will be updated in the GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "The first thing we need to create is a `BookmarkService` class."

A block of code is set as follows:

```
namespace bookmarkr;

public class BookmarkService
{
    private readonly List<Bookmark> _bookmarks = new();
}
```

Any command-line input or output is written as follows:

```
$ bookmarkr link add --name <name> --url <url>
```

> **Tips or important notes**
> Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, email us at `customercare@packtpub.com` and mention the book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata` and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Share Your Thoughts

Once you've read *Building CLI Applications with C# and.NET*, we'd love to hear your thoughts! Please `click here to go straight to the Amazon review page` for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

`https://packt.link/free-ebook/978-1-83588-274-0`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

Part 1: Getting Started with CLI Applications

In this part, you will get an overview of command-line interface (CLI) applications and understand their importance in modern software development. In addition, you will learn about setting up an efficient development environment for CLI programming, including essential tools and frameworks. Finally, you will explore the basic concepts of console applications in .NET, providing you with a solid foundation for creating powerful CLI tools.

This part has the following chapters:

- *Chapter 1, Introduction to CLI Applications*
- *Chapter 2, Setting Up the Development Environment*
- *Chapter 3, Basic Concepts of Console Applications in .NET*

1
Introduction to CLI Applications

In the realm of computing, **command-line interface** (**CLI**) applications exemplify the enduring power and efficiency of text-based user interfaces. Unlike their graphical counterparts, CLI applications offer a streamlined, no-frills approach to interacting with software, allowing users to execute commands, manipulate files, and perform a myriad of tasks right from the terminal, or automate them so these tasks do not require user interaction at all!

In this chapter, we will cover the following topics:

- An IT professional's typical day
- What CLI applications are, what their benefits are, and when to use them
- Popular CLI applications

A day in the life of an IT professional

It's a beautiful Monday morning. Today, my team and I are starting a new project.

The project is to build a web application using **ASP.NET** with **Entity Framework** as an **object-relational mapper** (**ORM**), **NuGet** for managing dependencies, **Git** as a code versioning system, and, since the application is to be deployed on Azure, **Bicep** as an infrastructure scripting language.

A few years ago, I would have used GUI applications for this, such as the full-fledged **Visual Studio**, **GitHub Desktop**, or **GitKraken**, and the **Azure portal**.

Today, I doing most of my work in **Visual Studio Code** and its integrated terminal.

So, I use commands such as `mkdir` to create my project directory, `cd` to position myself into this directory, `dotnet new` to create my project, `git init` to initialize the Git repository, `dotnet add package` to add NuGet packages as dependencies to my project, `dotnet ef dbcontext scaffold` to generate a database context and all the entity type classes for my database, `dotnet`

`build` to compile my application, and `dotnet run` to run it. When comes the time to deploy my application to Azure, I use commands such as `az login` to log into my Azure account, `az account set` to position myself onto the appropriate subscription, and finally, `az deployment group create` to deploy my Azure infrastructure as declared in my Bicep script.

When I realized that, I paused to pay careful attention to the situation. Wow, CLI applications are everywhere! They are truly part of our daily operations, no matter the role we play in an IT team.

I wondered how I missed that… then it hit me. This might be because CLI applications are a bit shy (they have no flashy UIs), and for that reason, we may not always notice them, so let me tell you about some common ones:

- If you are a developer, you have certainly used the .NET CLI (`dotnet`), the Node.js CLI (`node`), the npm package manager (`npm`), the Angular CLI (`ng`), Python (`python`), Git (`git`), Docker (`docker`), Kubernetes (`kubectl`), and many more.
- If you are a DevOps engineer, you may be using Git (`git`), GitHub (`gh`), Azure DevOps (`az devops`), Docker (`docker`), Kubernetes (`kubectl`), Ansible (`ansible`), and so on.
- If you are a system administrator, you may be regularly using package managers on various operating systems, such as `apt` on Linux, `brew` on macOS, or even `choco` or `winget` on Windows. You also most likely use shells, automation, and configuration tools such as PowerShell or Bash.
- If you are a cloud administrator or architect, you may be using the Azure CLI (`az`), the AWS CLI (`aws`), Terraform (`terraform`), or Bicep (`bicep`), among others.
- If you are a data scientist, you might be using Python (`python`), R (`R`), Pandas (`pandas`), SQL (`sql`), or Jupyter Notebooks (`jupyter notebook`).
- If you are a video or audio producer, a content creator, or simply a multimedia enthusiast, you are probably using `FFmpeg` (yes, it is a CLI application) to manipulate, convert, and analyze media files.
- And the list goes on and on…

Then, I started wondering why this happened. How come we switched from those beautiful UIs with their shiny colors and animations, inviting us to do all sorts of tasks and activities, to the blinking cursor inside of that terminal that is waiting for us to tell it what to do? This might seem like a big leap backward, right?

I know I struggled with that feeling until I figured out why CLI applications are so great! Sure, they make us look cool and smart in front of our Muggle friends and relatives. But it's not about that. Okay… not only about that. Because, when we are alone working on our project, there are not many people to impress.

So…

Why care about CLI applications?

Because they improve our productivity by keeping us focused on the task at hand.

You see, when we switch contexts between different applications, the chances are that we lose sight of what we were doing and get derailed from our task by some other unrelated activity.

By relying on CLI applications, all the commands that we type and execute happen to be within the same terminal, so we have a better chance of staying focused on what we are doing, thus achieving more.

To CLI or not to CLI?

That is the question. And the answer is quite simple: you don't have to choose. In some scenarios, CLI applications make perfect sense, while in others they make no sense. Imagine using Microsoft Teams, Slack, or any tool of the Adobe Creative Suite. Would it make sense to interact with these applications as CLI applications? Of course not! (unless it is for installing and configuring them).

So, the point here is that you become aware of the power of CLI applications, and you start taking advantage of them in your everyday workflow. They are not meant to replace those outstanding GUI applications.

As my wife, Lamia Rarrbo (who is an executive coach) says: "This is not an **or** situation but rather an **and** situation."

CLI applications as the building block for creating workstation profiles

Let me tell you a true story. A few years ago, one of my customers asked me to come up with a solution to build workstation configuration profiles for different roles within their company.

This is nothing new, you may say, and you are exactly right!

So, what makes this situation worth mentioning? What makes it special?

For many years, this used to be achieved using customized OS images, depending on the role you have in the organization. However, this comes at a cost:

- The cost of storage: These images tend to be large and hence require a lot of disk space to store them.
- The cost of OS updates: When a new OS version is introduced, the IT department must recreate all images.
- The cost of tools updates: When new versions of the tools utilized by the various profiles are introduced, the IT department must recreate the impacted images.

- The cost of frustration: Since computers are configured using these images, and because this activity is solely performed by the IT department, it causes frustrations at both ends. First, users accuse the IT department of being slow at providing them with their new machine ("*3 weeks to configure a new laptop?! You gotta be kidding me!*". We have all heard this at some point, right?). Second, the IT people have to configure these computers *in addition* to their other tasks.

The solution I proposed is to leverage CLI applications to provide users with more autonomy while ensuring that the IT department still has control over what is deployed on the workstations.

So, I built configuration profiles for each role as a PowerShell script. This script relies on *Chocolatey* (and, later, on *WinGet*) to install all the necessary tools for a given user role. These scripts were stored on a file share that users had access to according to their role (so, for example, if you are an analyst, you don't have access to the developer profile, and so on).

This solution provided multiple advantages:

- The IT department now only installs the OS, configures the user account, and hand the workstation to the user
- The user can now navigate to the file share and start the installation of their software based on their profile
- The IT department can update the configuration profiles, or provide new ones, without having an impact on the users and without making them wait

Hence, by leveraging CLI tools, we were able to automate workstation configuration in a personalized manner, according to users' profiles. Imagine how tedious this would have been if we had to install each and every application through its GUI assistant!

By relying on CLI tools, we were able to improve both the IT department's productivity and users' satisfaction.

Even heavy graphical applications have a CLI tool!

Why is the preceding story interesting?

It's not because of the use of PowerShell, WinGet, or Chocolatey. These are clearly CLI tools.

It is because we designed a way to install software from the command line without involving any GUI. This means that even if we are installing graphical applications (such as the Microsoft Office suite, internet browsers, and the Adobe suite), we are doing so by relying on their own CLI tools.

Yes, these graphical applications provide a CLI tool allowing us to install (and sometimes also configure) these applications.

Even ChatGPT has a CLI!

A quick Google search and I found that there are even CLI applications for **ChatGPT**! Can you believe that?!

You can check out some of them at `https://github.com/kardolus/chatgpt-cli` and `https://github.com/marcolardera/chatgpt-cli`.

I am telling you: CLI applications are truly everywhere, and once you start caring about them, you start noticing how much they are part of your day!

Summary

If you work in IT, you have most likely already used at least one CLI application of some sort (and the chances are that you use many of them!), and you may even be using it on a daily basis.

I am sure that, by now, you have understood the value of CLI applications and the role they play in your everyday job.

In this book, we will explore every aspect of building our very own CLI applications, that serve our needs and those of our users and customers.

Feeling excited? Then, grab your keyboard, fire up your terminal, and let's start coding!

We have only scratched the surface of what CLI applications can do and why they're an essential tool in today's tech landscape. I want this book to be your roadmap and your guide in this exciting journey.

Oh, and by the way, do not simply read this book, experience it!

Now, turn this page, and let's continue our journey together. The world of CLI applications awaits you!

Enjoy the journey.

2
Setting Up the Development Environment

Throughout this book, we'll build (and incrementally improve) a CLI application using .NET. We are going to start by setting up our development environment, which means installing the necessary tools to get started building CLI applications.

To reinforce and consolidate your learning, thereby getting the most out of this book, I strongly recommend that you practice the demos I'll be showing you and complete the suggested exercises in the *Your turn!* sections at the end of each chapter. To do that, you'll need to have your development environment set up and properly configured.

More specifically, in this chapter, we're going to do the following:

- Install Visual Studio Code
- Install the required extensions
- Install the .NET SDK
- Install and configure Git

Technical requirements

The code for this book can be found at `https://github.com/PacktPublishing/Building-CLI-Applications-with-C-Sharp-and-.NET`. The code for each chapter will be available in chapter-wise folders and will be mentioned in the *Technical requirements* section at the beginning of each chapter.

> **Note**
>
> There will be no code samples for this chapter, as we'll be focusing on installing the necessary tools and getting your development environment ready. Please also note that all the required tools are available to you free of charge.

Throughout this book, I'll be using a Windows 11 development machine. If you're running on Linux or macOS, there should not be any major differences, apart from the installation of the tools, which I'll highlight when needed throughout this chapter.

Installing Visual Studio Code

Even though .NET applications can be developed using various code editors and Integrated Development Environments (IDEs), we'll be relying on Visual Studio Code.

Visual Studio Code is an open source, free-of-charge, cross-platform, powerful code editor from Microsoft that can be used to develop applications in numerous technologies (including .NET). Its true power comes from the wide variety of extensions, from Microsoft, third-party editors, and the community, that can be added to it to extend its capabilities and make it a code editor like no other. In my humble opinion, with all the possibilities that the extensions marketplace provides, Visual Studio Code is blurring the line that separates a "simple" code editor from a complete and powerful IDE.

My number one reason for choosing Visual Studio Code is that, by being cross-platform, its usage is the same whether you're running on Windows, Linux, or macOS. Hence, even though I'll be running on Windows, you'll have no problem following along with me if you're running on a different platform, and this is a huge advantage!

To download and install Visual Studio Code, the easiest method is to visit the `https://code.visualstudio.com/` website and download it from there.

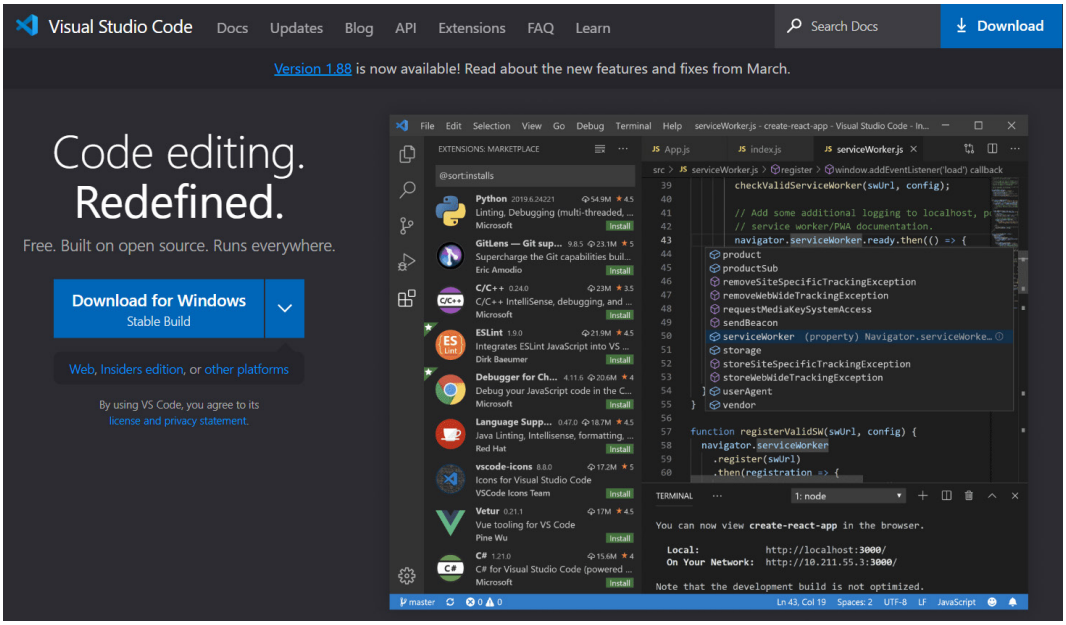


Figure 2.1 – Downloading Visual Studio Code for Windows

Important note

The **Download for…** link on the website should adapt to the platform you're visiting the website from. In other words, if your computer is running Linux, the link should state **Download for Linux**. If not, click the drop-down symbol to the right of the download link to reveal other options.

Once the download is complete, head over to your downloads folder and double-click on the installer file. This will start the installation process. After you select **Next** and **Finish**, accepting the default values, you'll have Visual Studio Code installed on your development workstation!

On the **Select Additional Tasks** screen of the installation wizard, I recommend that you check the two highlighted checkboxes in the following figure. These checkboxes will help you to open Visual Studio Code from the context of the current file/folder. I find this to be extremely useful.

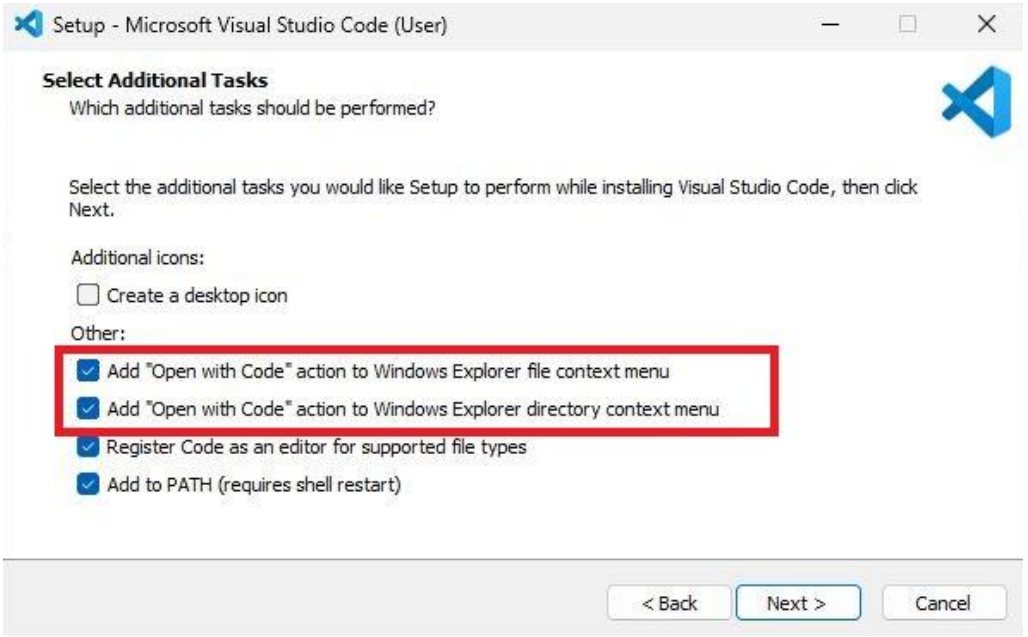


Figure 2.2 – "Open with Code" checkboxes

Our code editor is now installed and ready to be used. However, the true power of Visual Studio Code lies in its extensions. They make the development process smooth and easy, while also improving the developer's productivity.

At its essence, an extension is a software package that adds new features, functionality, or customizations to the Visual Studio Code editor.

Let's install some extensions!

Installing the required extensions

In this section, we'll install some extensions that I highly recommend.

Feel free to install additional extensions as well. (Drop me a line on social media to let me know which other extensions you installed and why you like them. I'm always keen to learn new things 😊.)

The extensions I would recommend for this book are as follows:

- **C#**: This extension from Microsoft is a no-brainer if you want to enable C# support in Visual Studio Code. It provides language support (including syntax highlighting and IntelliSense), debugging capabilities, and code completion.
- **C# Dev Kit**: This extension from Microsoft provides the Solution Explorer and Test Explorer experiences of the full-fledged Visual Studio in Visual Studio Code. If you're migrating from Visual Studio to Visual Studio Code, or if you are still using both, this extension will provide you with a similar experience in both environments.
- **IntelliCode for C# Dev Kit**: This AI-powered extension provides whole-line completion, ranked IntelliSense suggestions, and personalized insights based on your code base.
- **GitLens**: This extension by GitKraken supercharges your Git experience within Visual Studio Code by providing features such as Git blame annotations, code navigation, and commit graph navigation. It greatly improves your Git productivity.

> **Important note**
>
> If you're using extensions for AI assistants, such as GitHub Copilot, in Visual Studio Code, you may be presented with a warning message stating that IntelliSense will not work if the AI assistant is enabled.

To add extensions to your Visual Studio Code environment, use the **EXTENSIONS** window by clicking the corresponding icon on the left-hand side of the Visual Studio Code interface:

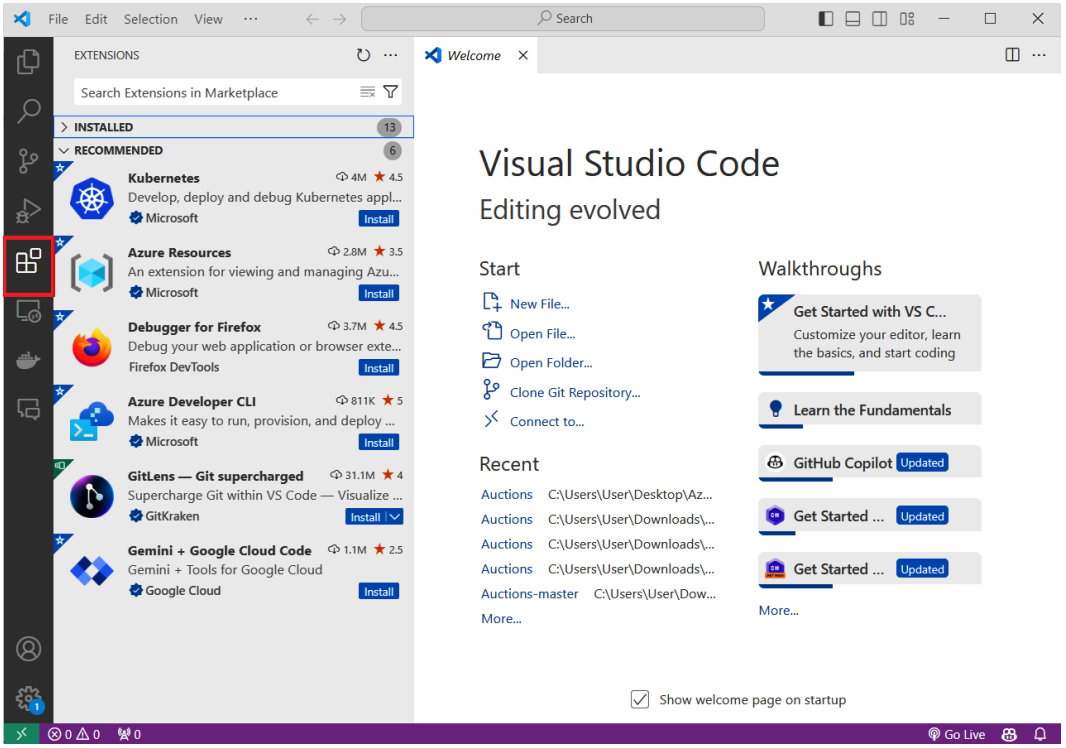


Figure 2.3 – The EXTENSIONS window in Visual Studio Code

From there, you can search (and install) various extensions. You may also notice that Visual Studio Code will suggest extensions based on the type of application you're developing. It's up to you whether to install them. Just keep in mind that the more extensions you install, the more computer resources Visual Studio Code will consume. So, you may want to find a balance between the extensions you install and the performance hit you're willing to accept.

Let's install the C# Dev Kit extension next.

First, open the **EXTENSIONS** pane, as shown in *Figure 2.3*. Then, in the search bar, type `C# Dev Kit`. Multiple results might be returned, but the one we're looking for should be at the very top. Make sure it is the right one (C# Dev Kit, developed by Microsoft) before clicking on it to select it and reveal its product page. Finally, click on the **Install** button below the name of the extension. The installation process should only take a few seconds.

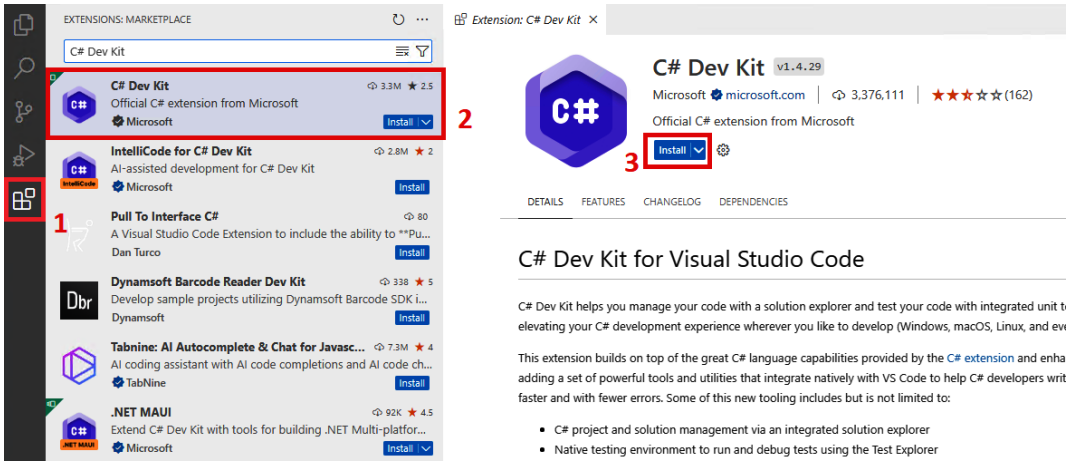


Figure 2.4 – Installing the C# Dev Kit extension

Once installation is complete, and a project has been opened, the C# Dev Kit extension will add the **SOLUTION EXPLORER** capability, which is similar to our experience with Visual Studio for Windows.

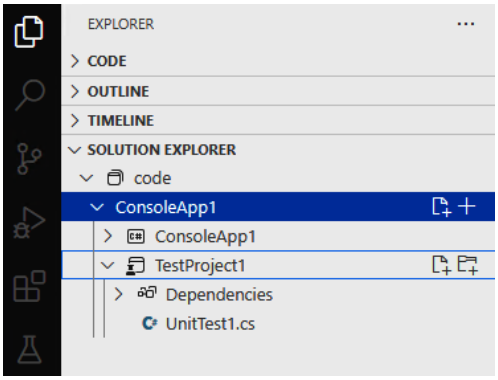


Figure 2.5 – The C# Dev Kit extension in action

> **Important note**
>
> Note that when you install the C# Dev Kit extension, the C# extension is automatically installed as well.

Other extensions can be installed by following the exact same process.

Installing the .NET SDK

As you may have guessed, the .NET SDK is required because we'll be developing CLI applications using .NET.

Although any version of .NET would do, we'll use .NET 8 for the following reasons:

- It is the latest **Long Term Support** (**LTS**) version of .NET and will be supported until November 10, 2026 (`https://dotnet.microsoft.com/en-us/platform/support/policy/dotnet-core`)
- It is cross-platform, so the CLI applications we'll be building with .NET 8 can be executed on Windows, Linux, or macOS

Before you install the .NET 8 SDK, you can verify whether it's already installed by using this command (works on Windows, Linux, and macOS):

```
$ dotnet --list-sdks
```

This will return the list of installed .NET SDKs on your machine.

The list should look like this:

```
3.1.424 [C:\program files\dotnet\sdk]
5.0.100 [C:\program files\dotnet\sdk]
6.0.402 [C:\program files\dotnet\sdk]
7.0.404 [C:\program files\dotnet\sdk]
```

If the .NET 8 SDK is not installed on your machine (as you can see from the preceding listing, it is not installed on mine), you can download it by visiting this website: `https://dotnet.microsoft.com/en-us/download`

Figure 2.6 – Downloading the .NET 8 SDK for Windows

Once the installer is downloaded, locate it in your downloads folder and double-click on it to start the installation process. This will require a couple of clicks, on **Next** and then on **Finish**.

After the installation is complete, you will notice that not only the .NET 8 SDK has been installed but also the .NET runtime.

The .NET runtime is the core component that executes compiled .NET code and provides runtime services such as memory management and exception handling. The .NET SDK, on the other hand, contains tools and libraries for developing, building, testing, and debugging .NET applications.

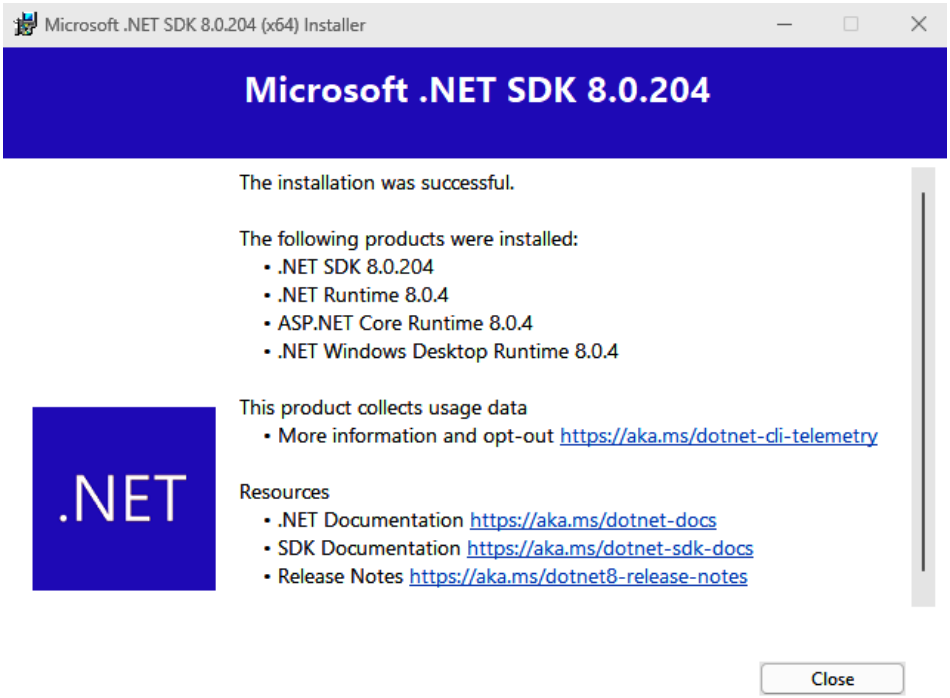


Figure 2.7 – The .NET 8 SDK is installed

We can then run the previous command to list the installed .NET SDKs:

```
$ dotnet --list-sdks
```

This time, you'll notice that the .NET 8 SDK is present:

```
3.1.424 [C:\program files\dotnet\sdk]
5.0.100 [C:\program files\dotnet\sdk]
6.0.402 [C:\program files\dotnet\sdk]
7.0.404 [C:\program files\dotnet\sdk]
8.0.204 [C:\program files\dotnet\sdk]
```

If you're running on a macOS computer, visiting the .NET SDK installation website will look like this:

Figure 2.8 – Downloading the .NET 8 SDK for macOS

We now have everything we need to develop our applications. However, if you are serious about development, you need a code management and versioning tool. This is where Git comes into play.

Installing and configuring Git

Git is a powerful and widely used distributed version control system that allows developers to track changes, collaborate on code, manage project history, and maintain different versions of their code base efficiently. It provides many benefits, including version tracking, branching, and collaboration, making it an essential tool for software development teams.

In other words, if you're serious about development, you need to use Git.

From our perspective, since we'll be dealing with a code repository that is hosted on GitHub, we'll need to use Git (hence, having it installed on our development workstation).

Git can be installed on Windows, Linux, and macOS.

Depending on what operating system you are using, Git might already be installed. You can check that by running this command in a terminal window:

```
git --version
```

If it's already installed, you can update it by running this command:

```
git update
```

Once this is done, you can skip the rest of this section.

The easiest way to install Git is to visit the `https://git-scm.com/downloads` website, where you'll get the link to download Git for your platform of choice.

Since I'm running on a Windows machine, I'll download Git for Windows by clicking on the **Download for Windows** link:

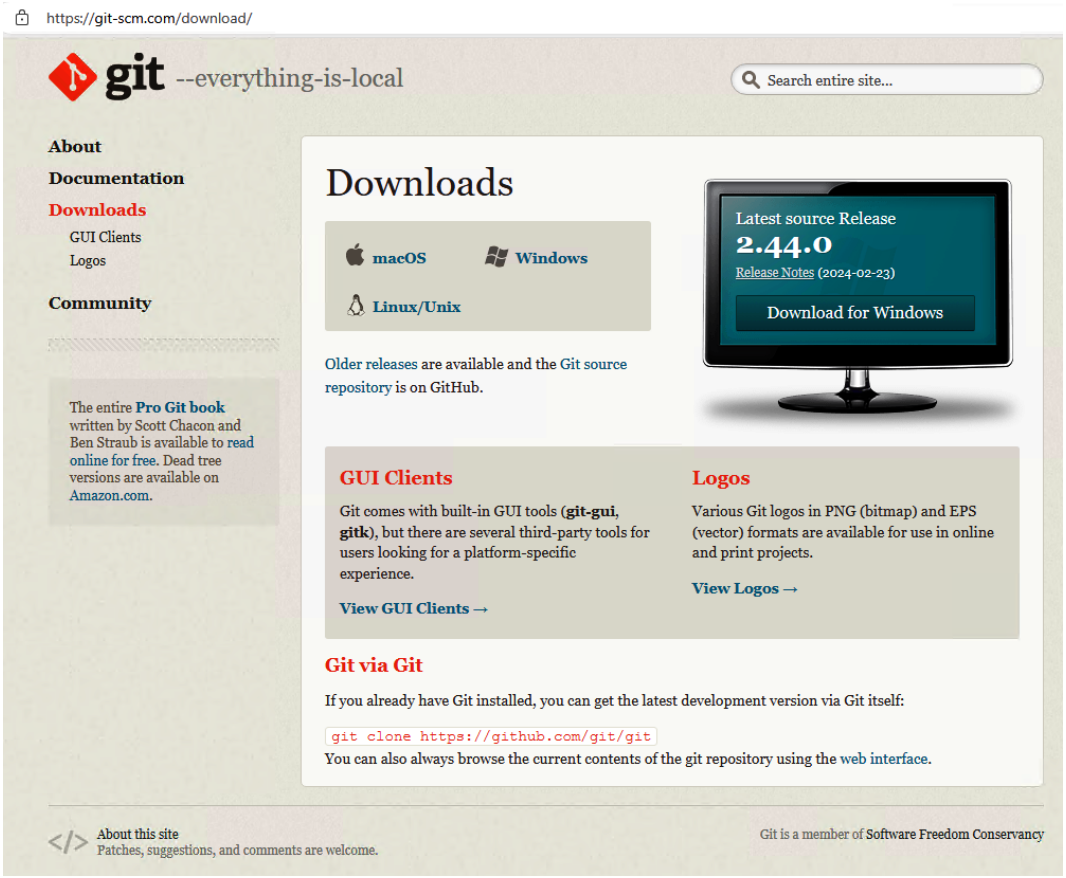


Figure 2.9 – Downloading Git for Windows

Next, I'll click on **64-bit Git for Windows Setup** under **Standalone Installer** because I'm running on an x64 machine:

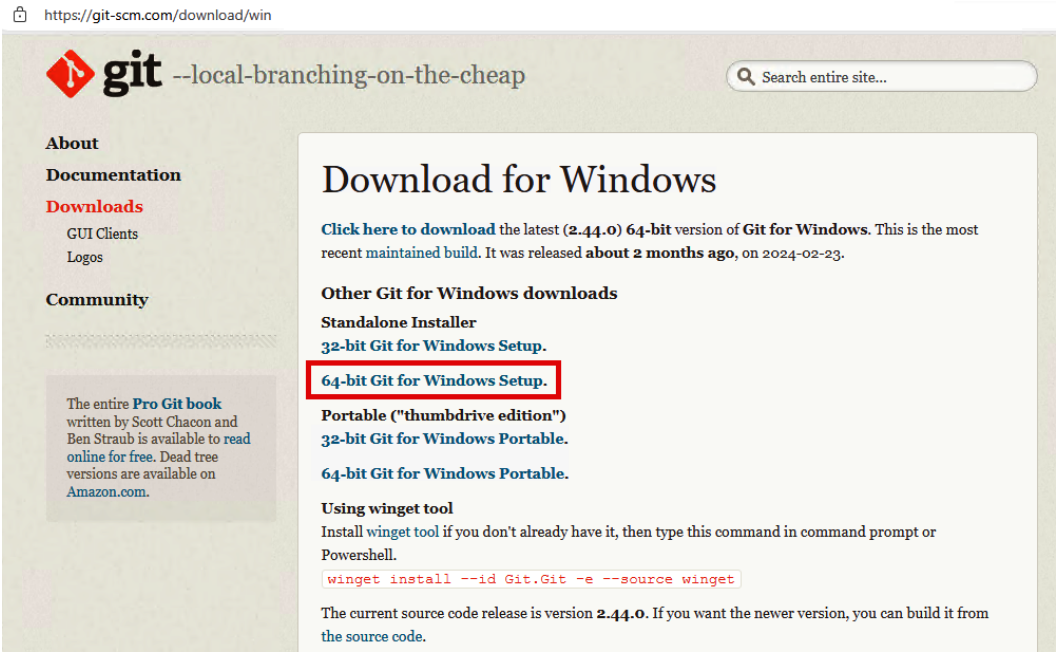


Figure 2.10 – Selecting the 64-bit version of Git for Windows

Once the download is complete, double-click on the executable file in your downloads folder to start the installation process. Once again, the installation process consists of clicking on the **Next** and **Finish** buttons. You should accept mostly default values apart from two exceptions.

The first exception is that I choose to select **Use Visual Studio Code as Git's default editor** on the **Choosing the default editor used by Git** screen of the installation wizard:

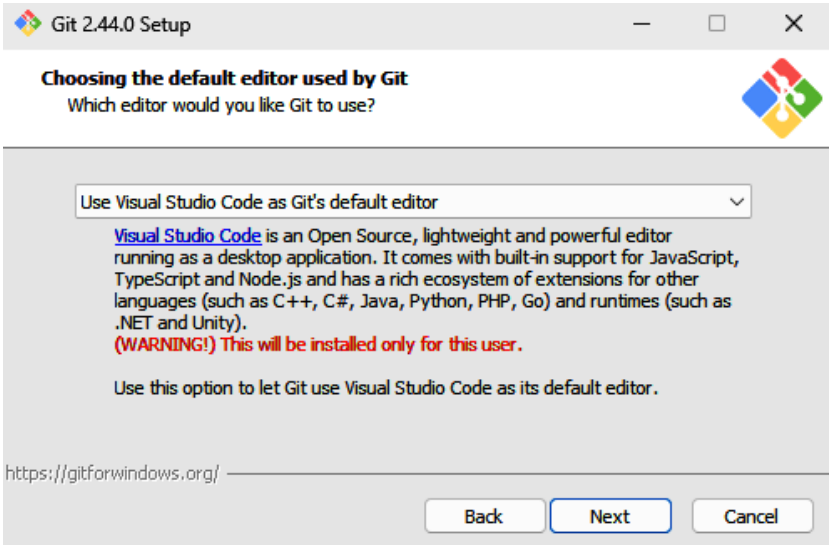


Figure 2.11 – Use Visual Studio Code as Git's default editor

The second exception is that I ask Git to override the default branch name, using `main` rather than **master** to use a more inclusive name.

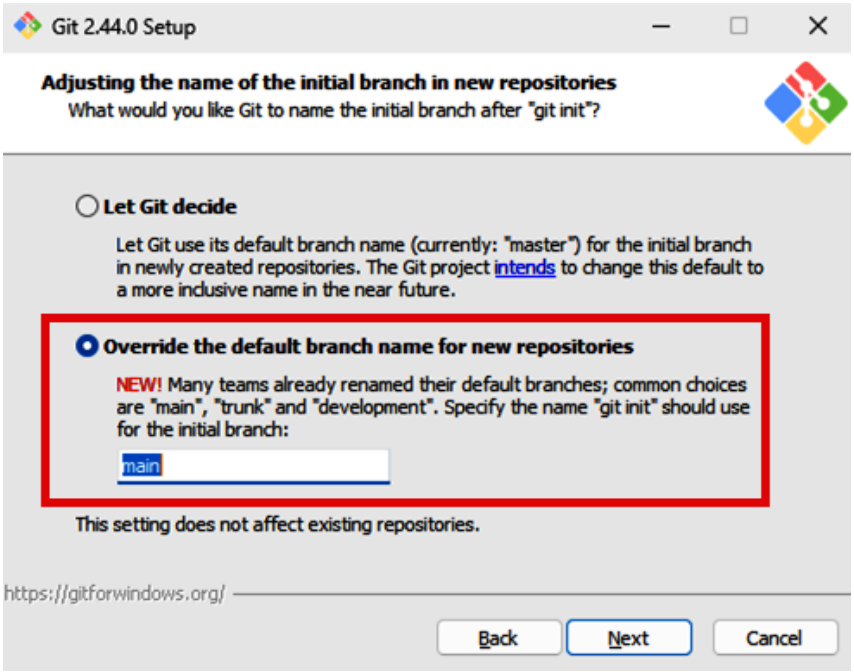


Figure 2.12 – Override the default branch name in Git

Once the installation is complete, I can validate that Git is indeed installed by opening a terminal window and typing the following:

```
$ git
```

When the installation process is completed successfully, you should see this output:

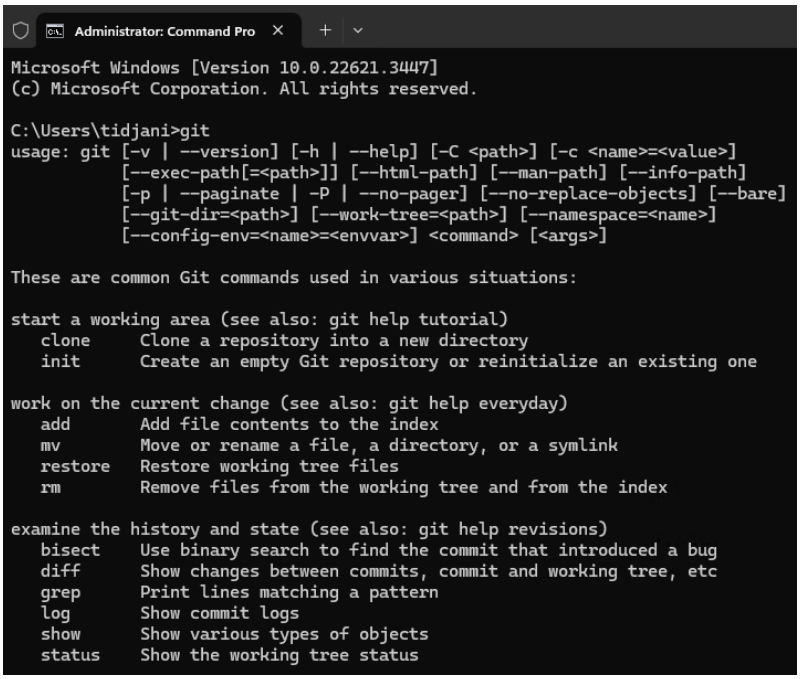


Figure 2.13 – Git is installed!

> **Important note**
>
> If you're running on a Linux workstation, the chances are that Git is already installed. So, before you try to install it, I recommend that you validate that it isn't already installed by opening a terminal and simply typing `git`. If it is already installed, the output will show the help of the command.

Once Git is installed, we'll perform some basic configurations, particularly setting a Git user identity so other developers can identify our contributions.

For this, we'll use these two commands:

```
$ git config --global user.name "Tidjani Belmansour"
$ git config --global user.email "Tidjani.Belmansour@gmail.com"
```

Perfect! We are now able to not only write our code but to version it and track its changes.

Summary

Congratulations! You now have a development environment that is properly configured and ready to be used.

As a reminder, we've installed the following:

- Visual Studio Code, which will serve as our code editor
- The C#, C# Dev Kit, IntelliCode for C# Dev Kit, and GitLens extensions for Visual Studio Code, which will make the development, debugging, and testing process smoother
- The .NET SDK, which is obviously required since we'll be developing a .NET CLI application
- Git, which is required since we'll be dealing with a GitHub repository

Now that we're fully set up and ready, we can begin our journey to the wonderful land of CLI applications. Our first stopover will lead us to explore the concepts and anatomy of CLI applications.

Your turn!

Almost every chapter of this book closes with a *Your turn!* section in which you're challenged to complete one or more tasks by applying the knowledge you gained from the chapter you just completed.

Since this chapter was all about configuring your development environment, your challenge is to configure yours so you'll be able to practice what you learn in upcoming chapters. If you have already completed this task, well done! You have no more tasks to complete here, and I will see you in the next chapter.

3

Basic Concepts of Console Applications in .NET

Now that our development environment has been set up, it's time to start our journey through the development of CLI applications using .NET.

However, first, we will explore console applications!

You're probably familiar with console applications, and you may be wondering why we need to discuss console applications in a book dedicated to CLI applications. The reason is that at the core of every CLI application is a console application. That's why, in this chapter, we'll take a moment to explore console applications. Plus, you know what they say: a reminder never hurts 😉.

Console applications can be seen as the simplest CLI applications one can build.

Hence, by exploring how we create, run, and interact with a console application, we will gain a basic understanding of how to work with CLI applications and how to build them by leveraging the console application template provided with the .NET SDK.

We will then create a very simple .NET console application, which takes some arguments as input and displays a message as an output. We will then enhance this application so it can perform some basic validations on the inputs and display the appropriate message as an output. This message will be displayed in a given color, depending on its severity.

Specifically, the chapter covers the following topics:

- Learn how to create and execute console applications
- Leverage the `System.Console` class for reading user input and outputting responses

Technical requirements

The code for this chapter can be found in the GitHub repository accompanying this book, `https://github.com/PacktPublishing/Building-CLI-Applications-with-C-Sharp-and-.NET/tree/main/Chapter03/helloConsole`.

Creating (and executing) a simple console application

Let's start by opening Visual Studio Code and displaying the integrated terminal window by going to **View | Terminal**.

Next, position yourself where you want the code folder to be created (I always create a `code` folder in my `C:` drive where all my code folders are located; I find it convenient to centralize all my code at the same location).

After ensuring that you are in the right working directory, type the following command to create a .NET console application:

```
$ dotnet new console -n helloConsole -o helloConsole --use-program-
main
```

Let's break down this command to understand what it does:

- `dotnet new console`: This will ask the .NET CLI tool to create a new console application. This will use C# as a language and .NET 8 as a framework (since these are the default values).
- `-n helloConsole`: Our application will be named `helloConsole`.
- `-o helloConsole`: A `helloConsole` folder will be created that will contain all the code files for our application.
- `--use-program-main`: The `Program` class containing the `Main` method will be added to the created `Program.cs` file. This method is the entry point of our program, and we will use its parameter to pass input values to our console application when we execute it. You can skip using `--use-program-main`, of course, but I prefer using it since it makes the `Program` class more explicit and familiar to developers coming from older versions of .NET.

Once the command completes, you should see an output that looks like this:

```
PS C:\code> dotnet new console -n helloConsole -o helloConsole
--use-program-main
The template "Console App" was created successfully.

Processing post-creation actions...
```

```
Restoring C:\code\helloConsole\helloConsole.csproj:
  Determining projects to restore...
  Restored C:\code\helloConsole\helloConsole.csproj (in 121 ms).
Restore succeeded.
```

This confirms that the application has been successfully created.

A brief tour of the generated project

The generated project contains three files:

- **Program.cs**: This file is typically the entry point for every program (through the `Main` method). While it may contain all the application's logic in very simple applications, it usually serves as a starting point, delegating to other classes and methods as needed.
- **helloConsole.csproj**: A project (`.csproj`) file is essential for .NET development as it centralizes project configuration, making it easier to manage, build, and share projects across different development environments and build systems. It's particularly important in the modern .NET ecosystem, which emphasizes cross-platform development and flexible project structures. In this file, we typically find information about the project definition, the build configuration, the dependency management (both project references and references to NuGet packages), the compilation settings, resource inclusion, any build process customization, project-wide properties (such as assembly name and version), cross-platform compatibility, and IDE integration.
- **helloConsole.sln**: A solution (`.sln`) file is a text-based file that serves as a container for organizing and managing related projects. It plays a crucial role in the development workflow, especially for larger applications that comprise multiple projects. Its purpose is to organize multiple related projects into a single solution, define build configurations and platforms for all projects, store solution-wide settings and metadata, and allow Visual Studio to load all related projects simultaneously.

Now, let's load the project into Visual Studio Code by typing this command:

```
$ code ./helloConsole
```

A new instance of Visual Studio Code will open, and you'll see the content of the newly created project, which looks like this:

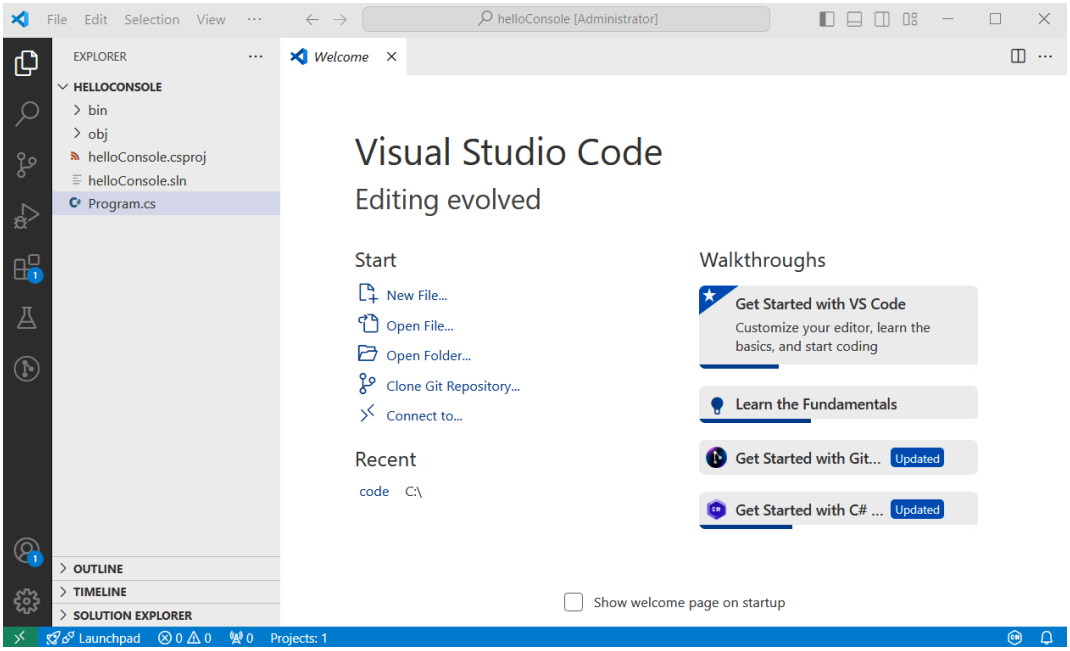


Figure 3.1 – The helloConsole project opened in Visual Studio Code

> **Do not confuse code with code!**
>
> The preceding `code` command is the executable name for Visual Studio Code. It should not be confused with the `code` folder 😉.

The code contained in the `Main` method of the `Program.cs` file doesn't do much at the moment. In fact, it only displays a `"Hello, World!"` message when the application is executed:

```
namespace helloConsole;

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello, World!");
    }
}
```

Let's execute this application and see what it returns. For that matter, we will need to type this command:

```
$ dotnet run
```

It is important to note that this command can be executed in any terminal: the PowerShell terminal, the CMD console, or the Bash terminal (if you are running Linux or macOS). However, since we are using Visual Studio Code, the easiest way to run commands is to use the integrated terminal within Visual Studio Code. That being said, you must ensure that you are in the project folder, which means that the `dotnet run` command should be executed in the same working directory as the `.csproj` file.

This will build and then execute the application. The output looks like this:

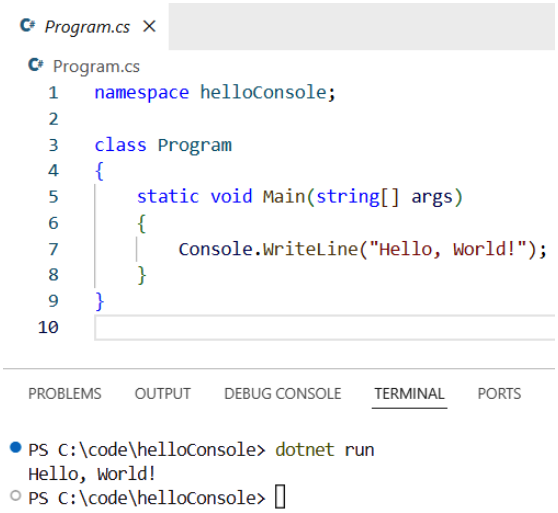


Figure 3.2 – Hello, World! output

This is not very useful at the moment, is it?

However, we can notice that the `Main` method takes an array of strings as an argument (that is, as an input parameter). So, let's use that to pass some parameters to our program.

First, we modify our application code to show the value of that parameter, as follows:

```
namespace helloConsole;

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine($"Hello, {args[0]}!");
    }
}
```

Now, once executed, our program will display the `Hello` message, followed by the value we passed in as a parameter.

Let's try it:

```
$ dotnet run Packt
```

The result will be as follows:

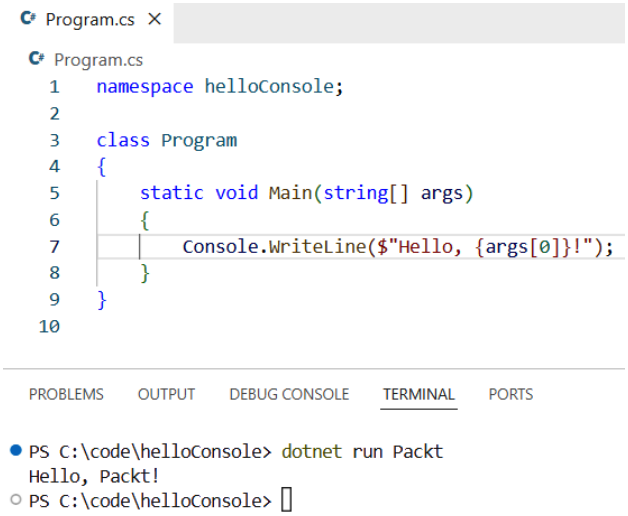


Figure 3.3 – Passing in one parameter

We can, of course, pass in more than one parameter, like this:

```
$ dotnet run Packt Publishing
```

This time, the result will be as follows:

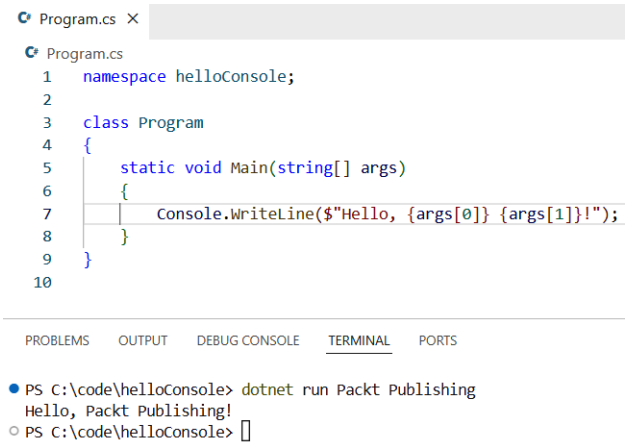


Figure 3.4 – Passing in more than one parameter

Important note

There are three remarks you need to be aware of:

1. The parameter to be passed to the `Main` method is an array of **strings**. This means that you'll need to parse these strings if the program is expecting an input of another data type (such as an integer, for example).

2. Since this parameter is an array of strings, you can use the passed-in values by specifying their index, which represents their position from the program execution.

3. If you use a parameter value in the code but don't pass it when executing the program, an exception will be thrown.

What do these remarks mean from a code perspective? Let's consider some illustrative examples.

First, we'll address the first remark. This example shows that the input parameter is of the `string` type despite its value, `42`, representing an integer. In order to use that value as an integer, we need to parse it.

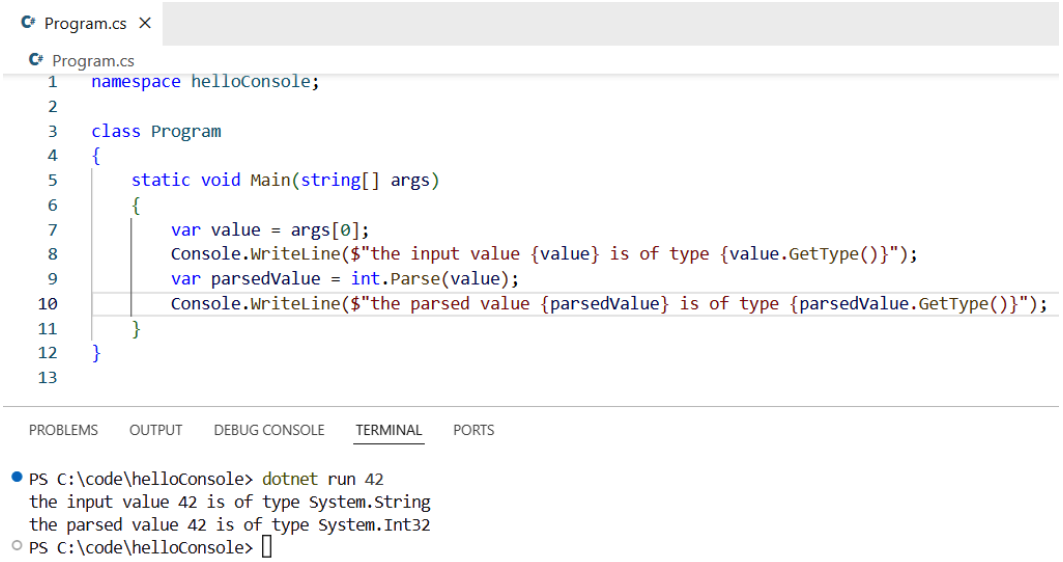


Figure 3.5 – Parsing input parameters

Next, let's consider the second remark. This example shows that by switching the indexes of the parameters, we also switch the display of their values.

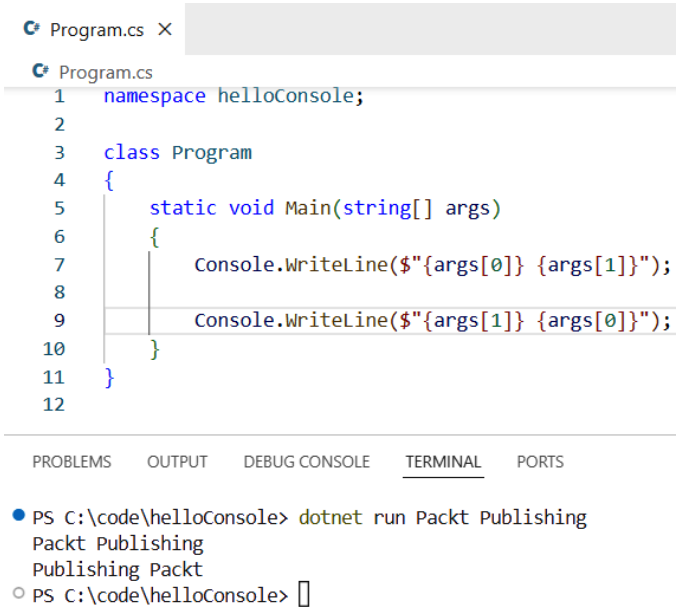


Figure 3.6 – Switching input parameters

Finally, regarding the third remark, this example demonstrates that failing to provide a value for the input parameter when executing the program will cause it to crash and throw an exception.

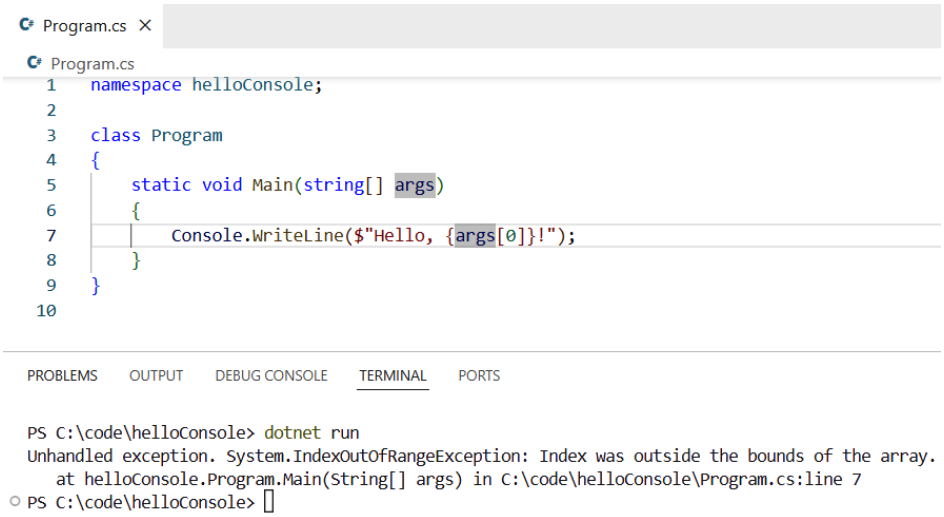


Figure 3.7 – Missing an input parameter

We now know how to create and execute .NET console applications. Let's see how we can interact with user inputs and better format our outputs.

Working with the System.Console class

You may have noticed that in all our examples up to this point, we have used the `Console` class. More specifically, we only used one of its methods (namely `WriteLine`). This method displays the value of the expression passed as a parameter, and then gets to the next line. If we don't want to get to the next line, we can use the `Write` method instead.

However, the `Console` class provides other useful properties and methods. We won't go into the details of all of them (for that matter, I'd recommend that you visit `https://learn.microsoft.com/en-us/dotnet/api/system.console`). Instead, I'll highlight the most interesting ones when it comes to console applications.

Useful properties

There are three properties I would like to tell you about in particular: `BackgroundColor`, `ForegroundColor`, and `Title`.

The first two, as their names suggest, are used to alter the background and foreground color of the terminal.

The third one, `Title`, is used to alter the title of the terminal window. Keep in mind that you will need to execute the program in an external terminal (not in the Visual Studio Code integrated terminal) to see the effect of changing the terminal's title.

Here is an illustrative example:

```
class Program
{
    static void Main(string[] args)
    {
        // performing a backup of the background and
        //foreground colors
        var originalBackroungColor = Console.BackgroundColor;
        var originalForegroundColor = Console.ForegroundColor;

        // changing the background and foreground colors
        Console.BackgroundColor = ConsoleColor.Blue;
        Console.ForegroundColor = ConsoleColor.Yellow;

     // setting the title of the terminal while the
     //application is running
        Console.Title = "Packt Publishing Console App";
```

```
        // displaying a message
        Console.WriteLine($"Hello from Packt Publishing!");

        // restoring the background and foreground colors
        // to their original values
        Console.BackgroundColor = originalBackroungColor;
        Console.ForegroundColor = originalForegroundColor;

        // waiting for the user to press a key to end the program.
        // this is useful to see the altering of the terminal's title
        Console.ReadKey(true);
    }
}
```

As you can see, at the beginning of the program, we performed a backup of both the foreground and the background colors, and we restored them at the end of the program.

There is no need to do this for the terminal's title since the set value is only effective during the execution of the program.

Useful methods

There are also a few interesting (and useful) methods I would like to talk about. These are as follows:

- `ReadLine`
- `ReadKey`
- `Clear`

Let's start with `ReadLine`. This method reads all the characters from the user input until they hit *Enter* and returns the user input as a `string`. So, it is useful to gather user inputs, such as a name, an age, or an address.

Here is an example:

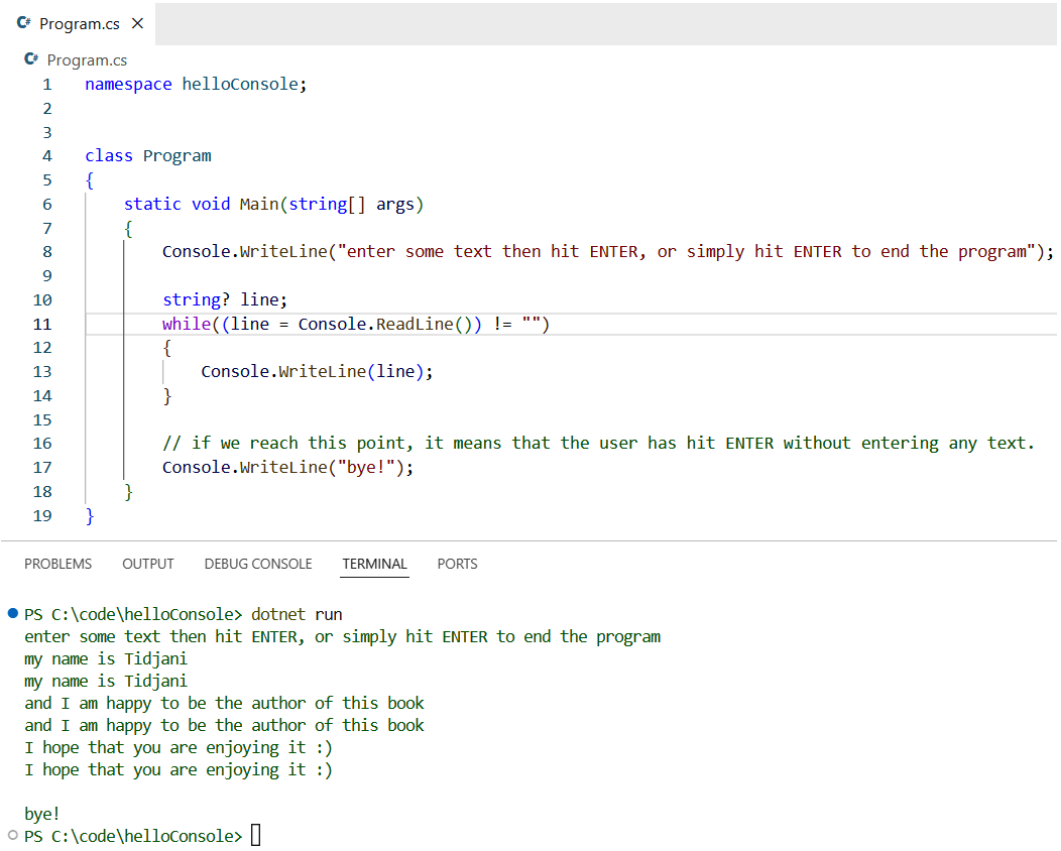


Figure 3.8 – Reading user input from the console

Next, Let's talk about `ReadKey`. This method reads the character or function key the user pressed. It returns an object of the `ConsoleKeyInfo` type, which includes information about the pressed key. It also takes an optional Boolean parameter that, if set to `true`, will not display the pressed key to the console, and if set to `false`, will display it.

Here's an example with the Boolean parameter set to `true`:

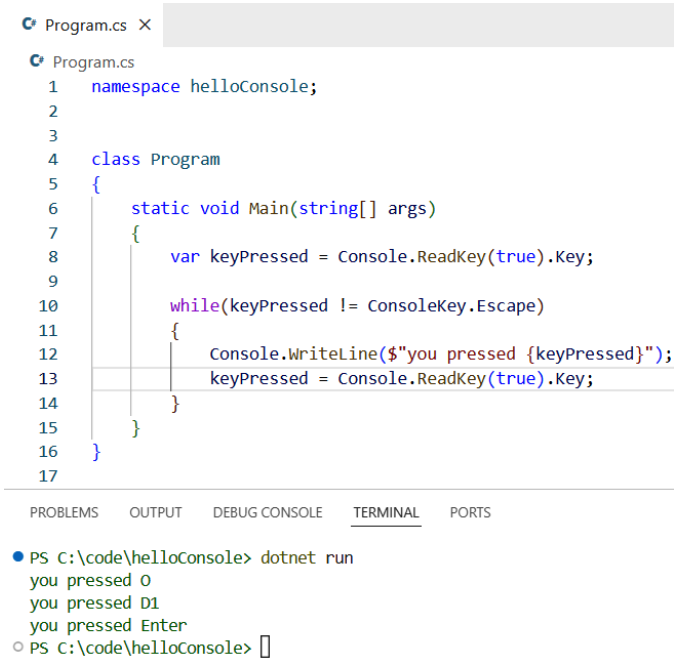


Figure 3.9 – A ReadKey method with the Boolean parameter set to true

Now, notice the output when the Boolean parameter is set to `false`:

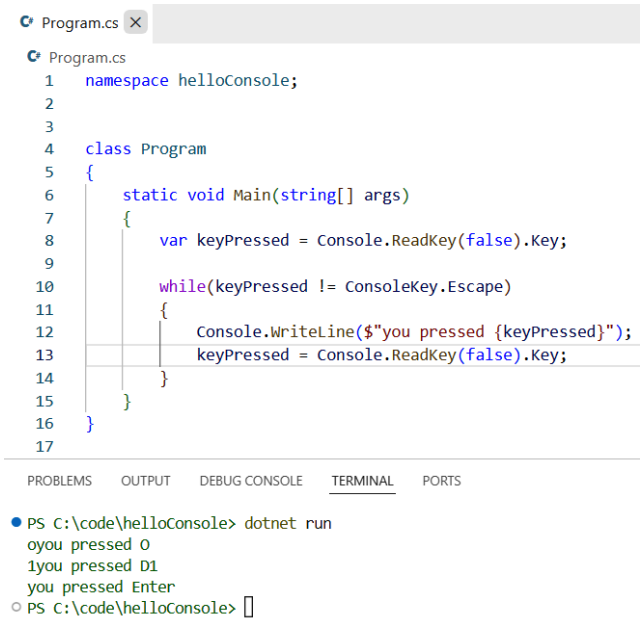


Figure 3.10 – A ReadKey method with the Boolean parameter set to false

Have you noticed that the pressed key is displayed twice: once after it is pressed, and a second time with the output message?

By the way, you can visit `https://learn.microsoft.com/en-us/dotnet/api/system.consolekey` to find a list of all of the values of the `ConsoleKey` enumeration.

Finally, let's see the `Clear` method. As its name suggests, it clears the console:

Figure 3.11 – Clearing the console with the Clear method

After the user presses the *C* key, the console is cleared.

This command is typically used in the following scenarios:

- **At the startup of the program**: It is used here so that your program gets a clean interface by removing all previous outputs from other programs or commands
- **When entering a new section or menu item**: This prevents outputs from previous sections from polluting the current one

Useful event

If you've ever run a CLI application, you're probably aware of the *Ctrl* + *C* or *Ctrl* + *Break* key combinations that make you exit the program at any time by terminating it.

In a .NET console application, these key combinations raise an event called `CancelKeyPress`. When one of the key combinations is pressed, the event is raised and interrupts the operation being executed. Our code can handle this event to allow for a graceful shutdown of the program, cleaning and freeing up resources before the shutdown.

Let's illustrate this with an example. Consider the following code:

```
using System.Threading.Tasks;

class Program
{
    static void Main(string[] args)
    {
        int counter = 1;
        while(true)
        {
            Console.WriteLine($"Printing line number {counter}");
            counter++;
            Task delayTask = Task.Run(async () => await Task.
            Delay(1000));
            delayTask.Wait();
        }
    }
}
```

As you may notice, this code uses an infinite loop with no exit condition. Hence, the only way to exit from it is by using one of the canceling key combinations (*Ctrl* + *C* or *Ctrl* + *Break*). However, when pressed, this will abruptly terminate the program without giving it a chance to perform some actions in order to exit gracefully, such as the following:

- Saving the execution state
- Logging out of services
- Closing database connections

We will then modify the previous code to handle that termination event, allowing the program to gracefully terminate:

```
using System.Threading.Tasks;

class Program
{
    static void Main(string[] args)
    {
        Console.CancelKeyPress += (sender, e) =>
        {
            e.Cancel = true; // This will prevent the program from
                             // terminating immediately
            Console.WriteLine("CancelKeyPress event raised!\
            nPerforming cleanup...");
            // Performing cleanup operations (logging out of services,
            saving progress state, closing database connections,...)
            Environment.Exit(0); // This will terminate the program
                                 // when cleanup is done
        };

        int counter = 1;
        while(true)
        {
            Console.WriteLine($"Printing line number {counter}");
            counter++;
            Task delayTask = Task.Run(async () => await Task.
            Delay(1000));
            delayTask.Wait();
        }
    }
}
```

Now, the result of the execution looks like this:

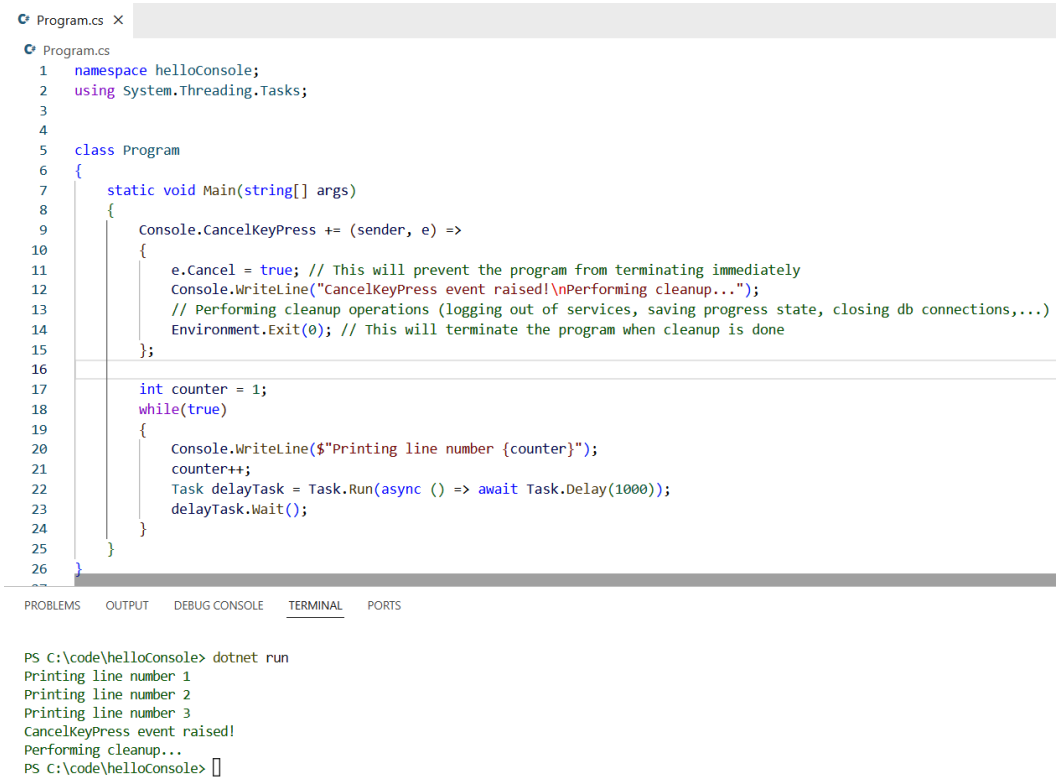


Figure 3.12 – Canceling the current operation

So, if your CLI application interacts with external resources or services (as we will do in the upcoming chapters), you should leverage the `Console.CancelKeyPress` event.

One more thing

Up to this point, we have executed the program using the .NET CLI `run` command.

However, if you are familiar with CLI applications, you will know that this type of application is usually executed by the name of its executable file.

The first question that comes to mind is "Why have we executed our program using the .NET CLI `run` command?". The answer is "because this is how you will do it when developing and testing your CLI application."

The next question that comes to mind is "Well, how do I execute my program using its executable?". The answer to that question is "by reaching the location of that executable file and running the program from there."

Let's see how we do this.

When you build the program, use the following command:

```
$ dotnet build
```

This will generate the executable file in the `bin\Debug\net8.0` folder on your hard drive.

On a Windows machine, this will look like this:

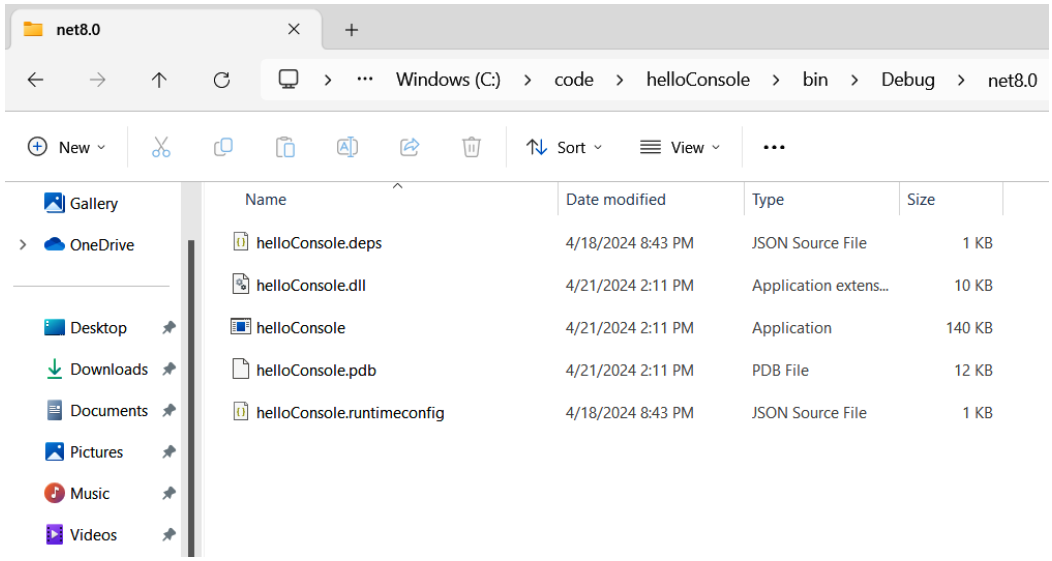


Figure 3.13 – The generated executable file

To run the program using its executable file name, open a terminal window, navigate to that location, and then type the following command (here, we are passing `42` as a parameter):

```
$ .\helloConsole 42
```

The result should be similar to this:

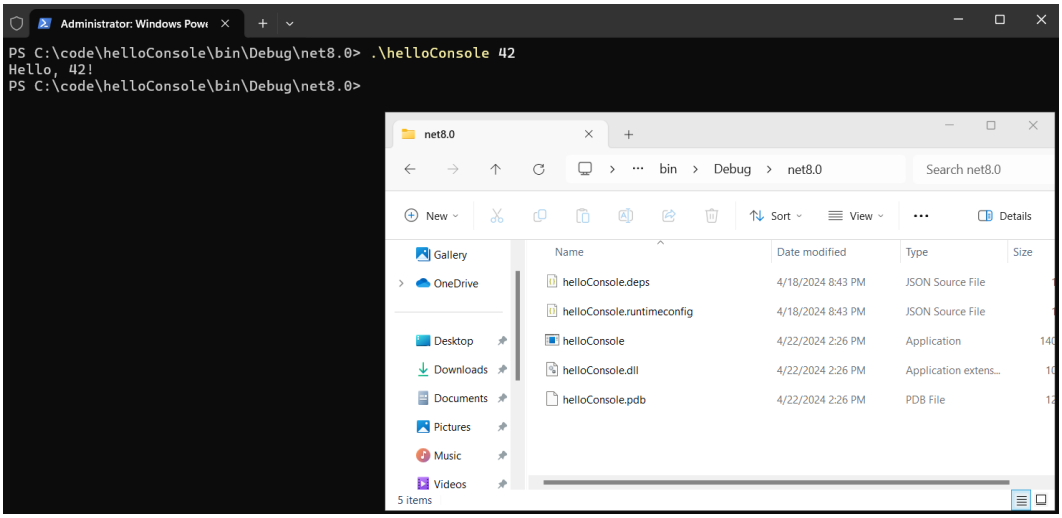


Figure 3.14 – Running the program from its executable file

Should I use dotnet run or run the executable file?

There are some differences between running the program using the `dotnet run` command and running the program from its executable file. Here are the most significant ones:

- **Build ^tprocess**: The `dotnet run` command builds the project before running it, which ensures that we are always running the most up-to-date version of the code. The executable file represents the last compiled version of the code.
- **Performance**: `dotnet run` is slower (due to the build step) compared to running the program from its executable file. While the `dotnet run` command may take a few seconds to build and execute the code, the executable file usually executes in a matter of milliseconds.

It is important to keep this in mind since, during the development phase, you'll likely rely on `dotnet run` to execute your program. However, in the testing and production phases, you'll rely on the name of the executable. So, make sure you carefully choose the name for your program (and its executable) 😉.

Summary

In this chapter, we explored what lies at the core of a CLI application: a console application! This is why it is crucial to learn how to work with console applications, as they are the foundation for building more complex CLI applications.

We saw how we can execute a console application, providing values as input parameters and parsing these input parameters' values in order to convert them into the data type expected by our program. We saw how we can gather user inputs through the use of the `ReadKey` and `ReadLine` methods of the `Console` class. Finally, we saw how we can handle exceptions raised as the result of a missing input parameter value.

However, a CLI application is more than a console one. It contains named parameters, flags, and subcommands.

In the upcoming chapters, we will see how we can leverage these capabilities into a fully functional CLI application. In the next chapter, we will start by creating the CLI application and learning how to parse its inputs, including commands, flags, and parameters.

Your turn!

Following along with the provided code is a great way to learn through practice.

A better way is by challenging yourself to achieve tasks. Hence, I challenge you to achieve the following task:

Create a console application that does the following:

- Sets the title of the terminal window to **GUESSING GAME!**
- Generates a random integer value between 1 and 20
- Tells the player that they have three chances to figure out that number
- After each player's trial, it should then do the following:
    - If the player figured out the number, the program displays, in green text, **Congratulations, you won!**, along with the number of trials it took the player to figure it out
    - If the player was not able to figure out the number within the three granted trials, the program should display, in red text, **Better luck next time! The number to be guessed was** along with the value to be guessed
    - If the player hits *Enter* without providing a number, the program should consider that the value is zero and proceed with it
    - If the player hits *Ctrl + C*, the program should exit and display, in yellow text, **Thank you for playing!**

Part 2: Foundations of Building CLI Applications

In this part, you will delve into the essential components of CLI application development. You'll explore command-line parsing techniques, learning how to effectively handle user input and parse arguments using libraries. Next, you'll learn about input/output operations and file handling, covering methods to read from and write to files, as well as manipulate file streams for efficient data processing. Finally, you'll discover best practices for error handling and logging, including implementing structured logging with different severity levels, gracefully managing exceptions, and providing informative error messages to users while maintaining detailed logs for debugging purposes.

This part has the following chapters:

- *Chapter 4, Command-Line Parsing*
- *Chapter 5, Input/Output and File Handling*
- *Chapter 6, Error Handling and Logging*

4
Command-Line Parsing

In the previous chapter, we created a console application and learned how to pass parameters to it, converting these parameters to their expected data type when needed (remember that parameters passed to a console application are of the `String` type).

However, even though a console application is at the heart of a CLI application, a CLI application is more than just a console application. A CLI application contains named parameters, switches, and subcommands to achieve the intended goal.

Armed with our knowledge of creating console applications, we'll build on top of that to learn how to create a CLI application.

To do so, in this chapter, we'll cover the following topics:

- Creating the console application
- Parsing the arguments of a console application
- From console to CLI: parsing the arguments using an existing library

By the end of the chapter, you will learn how to start with a simple console application and convert it to a powerful CLI application that handles commands, subcommands, and options.

Introducing Bookmarkr

Bookmarkr is the name of the CLI application we will be building throughout this book.

It is a command-line application for managing bookmarks.

Throughout the pages of this book, we will bring *Bookmarkr* to life and add more and more features to it.

Why a bookmark manager?

Because everyone has used one, so they are familiar with how such a tool works and what functionalities it provides.

By removing the burden of understanding the business context of *what* we are building, we can then focus all of our attention on *how* we are building it. And that is precisely why I chose this application. Plus, it could still be super useful 😉.

Technical requirements

The code for this chapter can be found in the GitHub repository accompanying this book, `https://github.com/PacktPublishing/Building-CLI-Applications-with-C-Sharp-and-.NET/tree/main/Chapter04`.

Creating the console application

Let's start by creating the console application. For this, in Visual Studio Code, display the **Terminal** window by going to **View**|**Terminal**.

Then, position yourself where you want the code folder to be created (I mentioned in the previous chapter that I always create a `C:\Code` folder that will contain all of my code projects).

From there, type the following command to create the console application:

```
$ dotnet new console -n bookmarkr -o bookmarkr --use-program-main
```

The .NET project currently looks like this when loaded in Visual Studio Code:

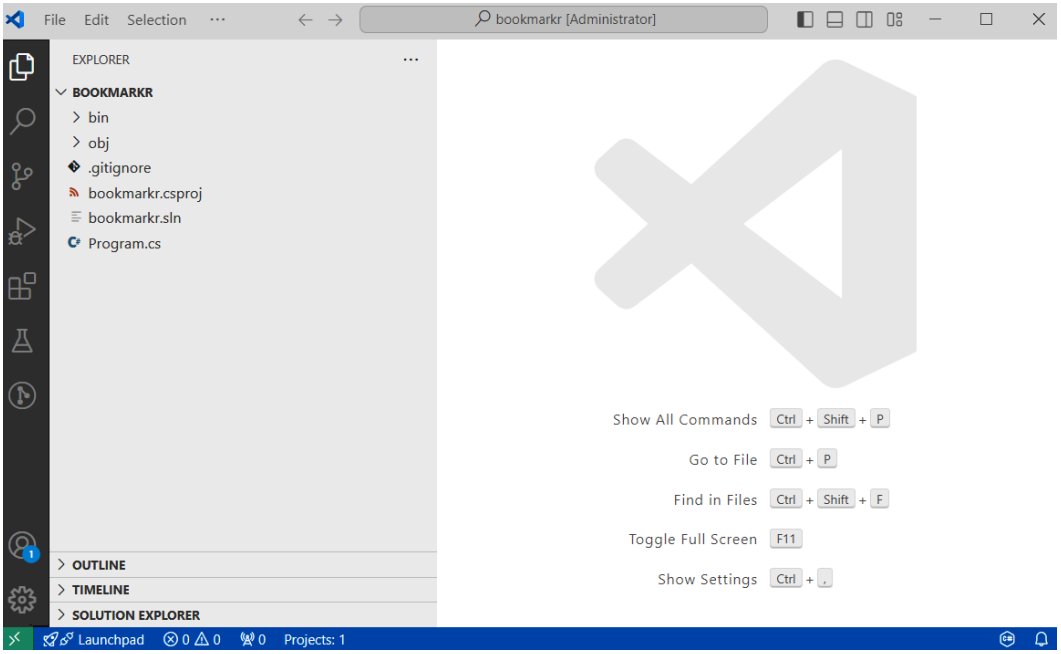


Figure 4.1 – The bookmarkr project opened in Visual Studio Code

Let's add some code to the `Program.cs` file.

The first feature we will implement is the ability to add a new bookmark to the list of bookmarks.

To do so, we need to create a `BookmarkService` class that will contain all the logic for the bookmarking operations. By following the best practices of development, we will create that class in its own code file, named `BookmarkService.cs`:

```
namespace bookmarkr;

public class BookmarkService
{
}
```

Next, we will need to add a list of `Bookmark` objects to that `BookmarkService` class:

```
namespace bookmarkr;

public class BookmarkService
{
    private readonly List<Bookmark> _bookmarks = new();
}
```

We will also need to define the `Bookmark` class. Here, again, we will follow the best practices of development, and we will create that class in its own code file, smartly named `Bookmark.cs` 😊. This class looks like this:

```
namespace bookmarkr;

public class Bookmark
{
    public required string Name { get; set; }
    public required string Url { get; set; }
}
```

Since the two properties of the `Bookmark` object cannot be `null`, we declare them with the `required` modifier.

The updated .NET project now looks like this:

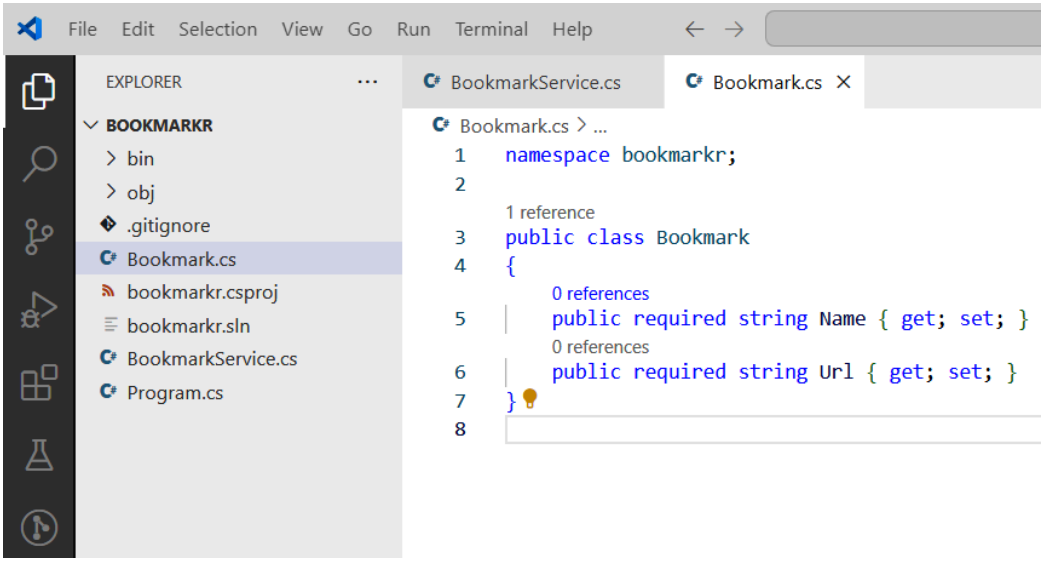


Figure 4.2 – The updated bookmarkr project opened in Visual Studio Code

Now that we have all the pieces in place, let's see how we can process user requests.

Parsing the arguments of a console application

A request to a CLI application usually contains the name of the command (and an optional subcommand) along with arguments that provide values to the parameters needed by the command.

The first command we will be adding is the ability to add a new bookmark to the list of bookmarks.

The syntax of the expected command is the following:

```
$ bookmarkr link add <name> <url>
```

So, let's modify the code to handle such a command!

We will start with the `Main` method of the `Program` class (located in the `Program.cs` file). Why? Because this is the method that receives the input parameters from the user.

Since we may have multiple commands in the future, we will add a `switch` statement to handle each one of these. Hence, the code will look like this:

```
namespace bookmarkr;

class Program
{
    static void Main(string[] args)
    {
        if(args == null || args.Length == 0)
        {
            Helper.ShowErrorMessage(["You haven't passed any argument.
            The expected syntax is:", "bookmarkr <command-name>
            <parameters>"]);
            return;
        }

        var service = new BookmarkService();

        switch(args[0].ToLower())
        {
            case "link":
                ManageLinks(args, service);
                break;
            // we may add more commands here...
            default:
                Helper.ShowErrorMessage(["Unknown Command"]);
                break;
        }
    }

    static void ManageLinks(string[] args, BookmarkService svc)
    {
        if(args.Length < 2)
        {
            Helper.ShowErrorMessage(["Unsufficient number  of
```

```
            parameters. The expected syntax is:", "bookmarkr link
            <subcommand> <parameters>"]);
        }
        switch(args[1].ToLower())
        {
            case "add":
                svc.AddLink(args[2], args[3]);
                break;
            // we may add more subcommands here...
            default:
                Helper.ShowErrorMessage(["Unsufficient number of
                parameters. The expected syntax is:", "bookmarkr link
                <subcommand> <parameters>"]);
                break;
        }
    }
}
```

Let's explain this code:

- The `Main` method dispatches the processing of each command to a specific method within the `Program` class. Since, for now, we only have one command (`link`), we have only one processing method (`ManageLinks`).
- The `ManageLinks` method handles the subcommands related to links. For now, we only have one subcommand (`add`) but we can easily imagine having more subcommands, such as `update` (to update the URL of an existing link), `remove` (to remove an existing link), and `list` (to list all existing links).
- Finally, the `ShowErrorMessage` method is a utility method used to display error messages in red-colored text. Its code can be found in the `Helper` class and is omitted here because it does not provide value to the topic we are discussing.

If you write (or copy) this code, you'll notice that it does not compile. This is due to the `AddLink` method not yet being available in the `BookmarkService` class. Let's add it!

The code of the `AddLink` method is the following:

```
public void AddLink(string name, string url)
{
        if(string.IsNullOrWhiteSpace(name))
        {
            Helper.ShowErrorMessage(["the 'name' for the link is
            not provided. The expected syntax is:", "bookmarkr link
            add <name> <url>"]);
            return;
```

```
        }

        if(string.IsNullOrWhiteSpace(url))
        {
            Helper.ShowErrorMessage(["the 'url' for the link is
            not provided. The expected syntax is:", "bookmarkr link
            add <name> <url>"]);
            return;
        }

        if(_bookmarks.Any(b => b.Name.Equals(name, StringComparison.
        OrdinalIgnoreCase)))
        {
            Helper.ShowWarningMessage([$"A link with the name '{name}'
            already exists. It will thus not be added",
            $"To update the existing link, use the command: bookmarkr
            link update '{name}' '{url}'"]);
            return;
        }

        _bookmarks.Add(new Bookmark { Name = name, Url = url});
        Helper.ShowSuccessMessage(["Bookmark successfully added!"]);
}
```

This code is pretty simple and straightforward, but let's explain it briefly:

- We first ensure that both the `name` and `url` parameters are provided. If not, we return an error message to the user with the expected syntax for the command.
- We ensure that the link to be added is not already present in the list of bookmarks. If it is present, we inform the user with a warning message and invite them to use the `update` subcommand if they intend to update an existing link.
- If the link does not already exist, we add it to the list of bookmarks and inform the user with a success message.

Pretty easy, isn't it? 😊

However, there is a problem with this approach to building CLI applications. Can you guess what it is?

It is based on positional parameters. In other words, we expect the first parameter to be the name of the command, the second parameter to be the name of the subcommand, the third parameter to be the value of the `name` parameter, and the fourth parameter to be the value of the `url` parameter.

But there are a couple of issues:

- How could the user be aware of that order?
- What if the user provided the `url` value as the third parameter and the `name` value as the fourth parameter?

If you're familiar with using CLI applications, you will know that the usual syntax for a CLI command looks like this:

```
$ bookmarkr link add --name <name> --url <url>
```

It can also look like this:

```
$ bookmarkr link add -n <name> -u <url>
```

This helps the user to know what is expected as parameters and to provide the parameters in the appropriate manner.

We can, of course, rely on the `args` list of parameters and compare each one to what is expected (for example, checking whether the third parameter's value is `--name` or `-n` so we know that the fourth parameter represents the value of the name of the link, and so on), but this will complexify our code too much.

I know what you're thinking. You're smart and you've already figured out that the best approach to solve this problem is to develop a library to parse these parameters and figure out what they represent.

But, because I know you're smart, I know that you've already searched for an existing library that does just that. After all, you do not want to reinvent the wheel, do you?

From console to CLI – parsing the arguments using an existing library

Although many libraries exist for the different programming languages, including .NET, we will focus on `System.CommandLine` throughout this book.

You may be wondering why we chose this library, especially if you are familiar with (or have heard about) `CommandLineParser`, which is another common library for that matter.

There are multiple reasons for that. In essence, `System.CommandLine` is a more modern, feature-rich, and performant library, whereas `CommandLineParser` is a simpler and more lightweight alternative.

Additionally, there are a few other reasons why I prefer `System.CommandLine`:

- `System.CommandLine` is a .NET Foundation project developed by Microsoft and the community, while `CommandLineParser` is a third-party library

- `System.CommandLine` uses a builder pattern and a more declarative approach to define commands and options, while `CommandLineParser` uses attributes and a more imperative approach
- `System.CommandLine` provides more advanced features out of the box, such as command hierarchies, response files, auto-completion, and parsing directives, whereas `CommandLineParser` is more lightweight and focused on basic command-line parsing
- `System.CommandLine` is known to be faster and more efficient, especially for large command-line structures
- Even though both libraries are cross-platform, `System.CommandLine` has better support for platform-specific conventions, such as case-insensitivity on Windows

Now, let's rewrite our application to benefit from the `System.CommandLine` library!

Important note #1

The previous code of the `Program.cs` file has been moved into a `Program.Console.txt` file for further reference.

Important note #2

While you can execute the application by its executable name (located in `bin\Debug\net8.0\bookmarkr.exe`), it's more convenient, during the development phase, to use the `dotnet run` command.

I prefer to rely on the executable name because it matches how we will use the application in production. If you prefer to use the `dotnet run` command, simply replace `bookmarkr` with `dotnet run` in the following execution syntaxes.

The first thing we need to do is to add the `System.CommandLine` NuGet package library to our project. To do this, open the Visual Studio Code terminal and type the following command:

```
$ dotnet add package System.CommandLine --prerelease
```

At the time of the writing of this chapter, this library is still in Beta. When it makes it to **general availability** (**GA**), you will not need the `--prerelease` switch anymore.

The way `System.CommandLine` works is to have a `RootCommand` object that will act as the root command for all other commands of the CLI application. This means that every command in the application has, as a parent, either the `root` command or another command whose parent is ultimately the root command.

Adding the root command

The root command is the one that gets called when the user invokes the CLI application with no parameters.

The root command is an instance of the `RootCommand` class:

```
var rootCommand = new RootCommand("Bookmarkr is a bookmark manager
provided as a CLI application.")
{
};
```

A command has a handler, which is a method that is called as the result of the user invoking that command:

```
rootCommand.SetHandler(OnHandleRootCommand);
```

The `SetHandler` method takes a delegate to the actual method that does the job:

```
static void OnHandleRootCommand()
{
    Console.WriteLine("Hello from the root command!");
}
```

Finally, and since the `System.CommandLine` library follows the `Builder` pattern, we need to build and invoke a parser to kick things in:

```
var parser = new CommandLineBuilder(rootCommand)
    .UseDefaults()
    .Build();

return await parser.InvokeAsync(args);
```

Last but not least, let's add the required `using` statements at the top of the file:

```
using System.CommandLine;
using System.CommandLine.Builder;
using System.CommandLine.Parsing;
```

We are now ready to execute our application! Type the following:

```
$ bookmarkr
```

When executed, the application will invoke the appropriate command (here, the root command since no argument was passed to the executing application), which, in turn, will invoke its handler method (`OnHandleRootCommand`), and the result of its execution will be returned to the user. In this example, the command will display the message `"Hello from the root command!"`.

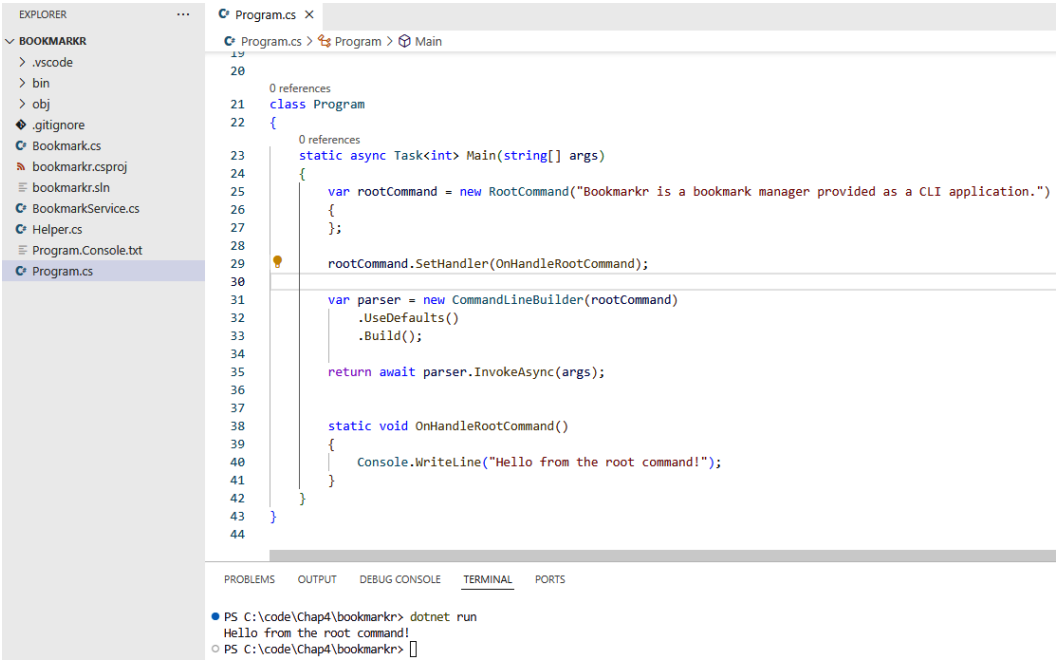


Figure 4.3 – Invoking the root command

Isn't that exciting? No? You're right… This is not (yet) looking like a CLI application. Let's add another command!

Adding the link command

The syntax for adding bookmarks using our CLI application is supposed to be like this:

```
$ bookmarkr link add <name> <url>
```

So, we need to create a `link` command. This command will have the root command as a parent:

```
var linkCommand = new Command("link", "Manage bookmarks links")
{
};

rootCommand.AddCommand(linkCommand);
```

Next, we need an `add` command, whose parent will be the `link` command:

```
var addLinkCommand = new Command("add", "Add a new bookmark link")
{
};
```

```
linkCommand.AddCommand(addLinkCommand);

addLinkCommand.SetHandler(OnHandleAddLinkCommand);
```

Now, let's execute this application:

```
$ bookmarkr link add
```

Pretty close, huh?

The only elements that are missing are the `<name>` and `<url>` parts. These are called **options**, and we will look into them in a moment. But first, let's focus on commands.

About commands

Commands are like a tree where the root command is… well, the root of that tree.

Each command has a parent, which is either another command (for example, `add` has `link` as a parent), or the root command itself (as for the `link` command).

The command tree dictates the syntax of the CLI application. For example, we cannot perform the following call:

```
$ bookmarkr add
```

This is because no `add` command has the root command as a parent.

Do all commands need to have a handler method?

The answer is, no, they don't. Only commands that actually do some processing need to have a handler method. In our case, neither the root command nor the `link` command require a handler.

Adding options to the link command

Since the `add` command requires two parameters (`name` and `url`), we will add two options to it:

```
var nameOption = new Option<string>(
    ["--name", "-n"],
    "The name of the bookmark"
);
```

```
var urlOption = new Option<string>(
    ["--url", "-u"],
    "The URL of the bookmark"
);
```

As you can see, an `Option` is defined by the following:

- The data type of its value (here, a string)
- Its aliases (for `urlOption`, these are `--url` and `-u`)
- Its description, which will be useful when requesting the help menu for that command (for `urlOption`, it is `"The URL of the bookmark"`)

Next, we need to assign these options to the command, as follows:

```
var addLinkCommand = new Command("add", "Add a new bookmark link")
{
    nameOption,
    urlOption
};
```

We then need to pass these options to the `Handler` method:

```
addLinkCommand.SetHandler(OnHandleAddLinkCommand, nameOption,
urlOption);
```

And, finally, we use the values of these options in the `Handler` method:

```
static void OnHandleAddLinkCommand(string name, string url)
{
    // 'service' is an instance of 'BookmarkService'.
    service.AddLink(name, url);
}
```

Now, if we execute the application, we get the expected result. This time, we saved about half of the code by not having to parse the arguments of the console application and delegating this to the `System.CommandLine` library:

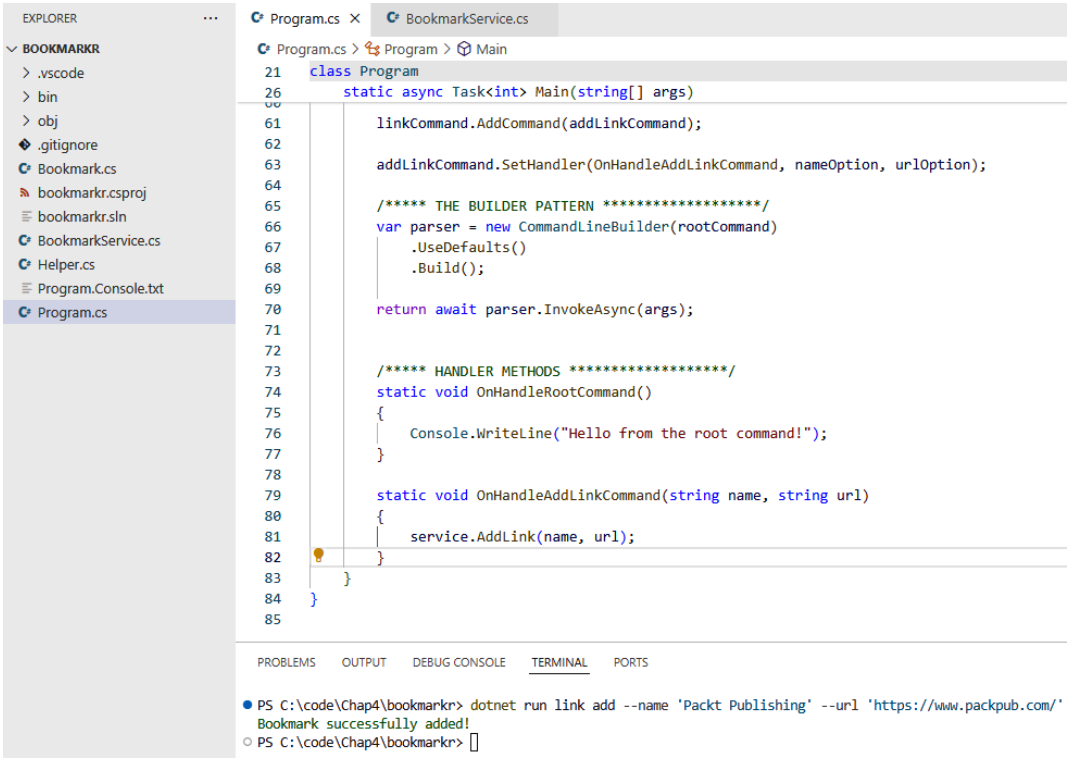


Figure 4.4 – We successfully added a bookmark

Note that, using the aliases, we can execute the CLI application using this syntax:

```
$ bookmarkr --name 'Packt Publishing' --url 'https://www.packtpub.com'
```

We can also use this syntax:

```
$ bookmarkr -n 'Packt Publishing' -u 'https://www.packtpub.com'
```

The result will be the same as the one shown in *Figure 4.4*.

What other types of options can we use?

`System.CommandLine` doesn't only support `string` options. `Option<T>` is a generic type, so you can create an `Option<T>` for any data type. This, for example, could be the following:

- `int`
- `double`

- `bool`
- `DateTime`
- `Uri`
- `TimeSpan`
- `Regex`
- `Enum`
- `IPAddress`
- `FileInfo`

It could even be your own custom class if you prefer.

Depending on the data type we use, the provided value for the option will be parsed into that data type. If parsing fails, an exception will be raised, and an error message will be displayed to the console.

Here is an illustrative example. The root command has been updated to accept an integer option. When the program is executed and a value that cannot be parsed into an integer is provided for that option, we get the following error message:

```
$ bookmarkr --number toto
Cannot parse argument 'toto' for option '--number' as expected type
'System.Int32'.
```

Hmm… Although the name of the option indicates that it is expecting a number, it is not clear what that number is all about. What is its purpose? Is there a range of valid values for that number? A little help would be… well, helpful here 😊.

Getting help

The good news is that when using `System.CommandLine`, we get the help menu automatically built for us. All we need to do is to provide meaningful names and descriptions to our commands and options and the library will do the rest.

We can get help by using either one of these options:

- `--help`
- `-h`
- `-?`

The result will look like this:

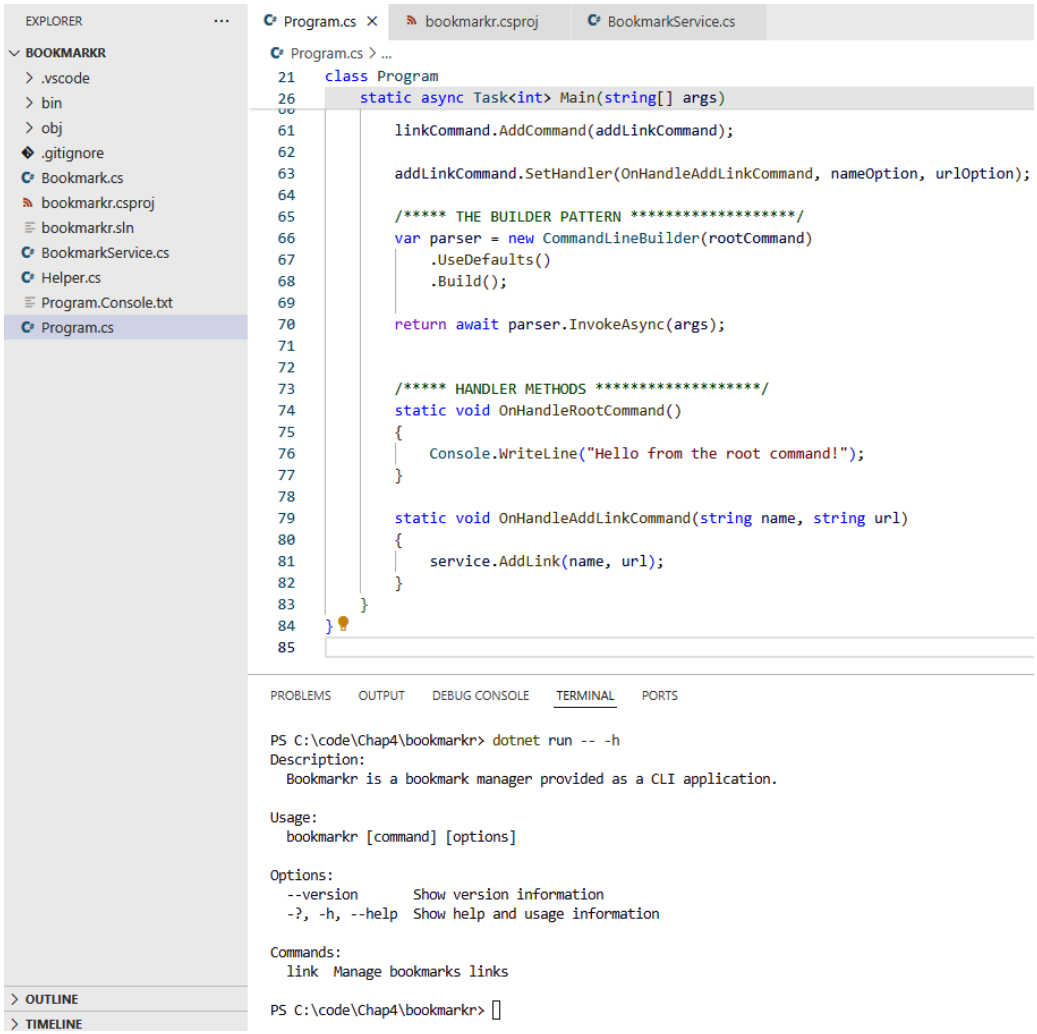


Figure 4.5 – Help!

This also works for subcommands. For example, if we want to get help regarding the `link` command, we can type the following:

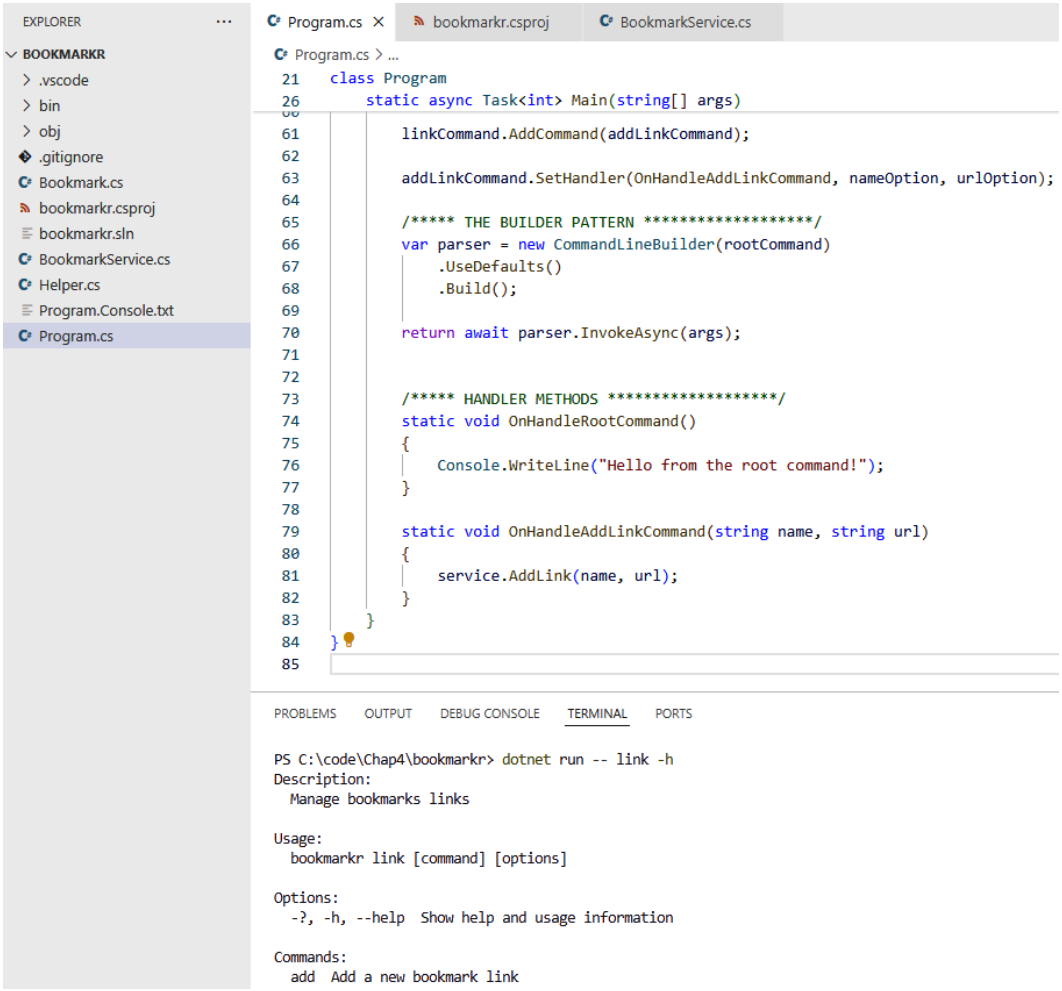


Figure 4.6 – Getting help for the link command

We can do this for one subcommand at a time, such as the `link add` command, for example:

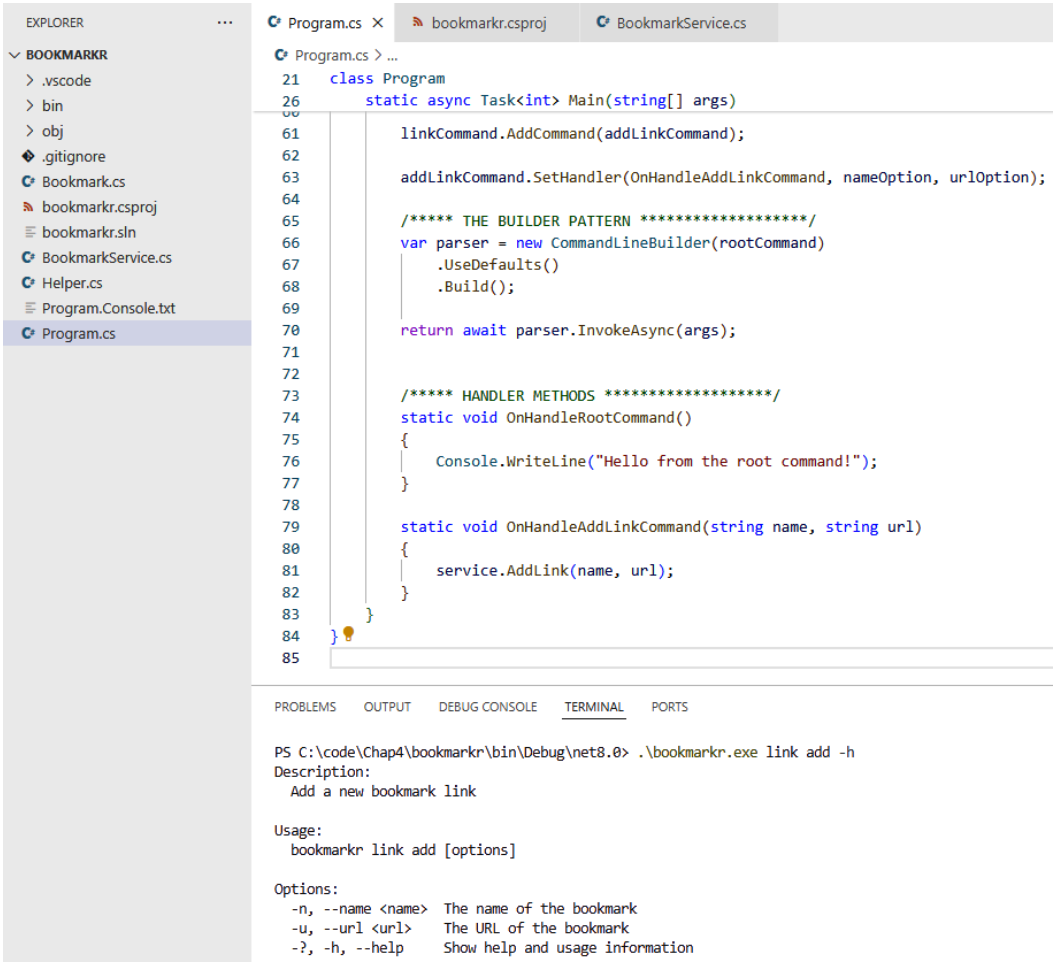


Figure 4.7 – Getting help for the link add command

Important note

You have probably noticed that the `dotnet run` syntax for getting help requires an extra `--`. This is not a typo. It is required because, otherwise, the .NET CLI tool will *think* that you are requesting help with the `dotnet` tool.

The extra `--` is used to separate the arguments that are passed to `dotnet run` from the arguments that are passed to the application being run. Everything after `--` is considered an argument to the application, not to `dotnet run`.

However, one thing we should keep in mind is that the usage of a CLI application (and hence, its help) depends on the current version of the application. But how do we get that information?

Getting the application's version number

There is a built-in option (`--version`) that displays the version number of the CLI application.

To display it, execute the command as follows:

```
$ bookmarkr --version
```

But where does this value come from?

Well, it can be found in the `.csproj` file, within the `<PropertyGroup>` element at the beginning of the file:

```
<PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <Version>2.0.0</Version>
</PropertyGroup>
```

Note that if no `<version>` element is provided, the default value to be returned will be `1.0.0`.

> **Important note**
>
> Here, too, if you want to use the `dotnet run` syntax to query the version number of the application, you'll need to use the extra `--`, as follows: `dotnet run -- --version`.

And that's it! You're now all set and ready to create your first CLI application!

Summary

In this chapter, we started to build our very own CLI application, *Bookmarkr*, which is a bookmark manager provided as a CLI application, so it can be used within a Terminal window.

We started with a console application (because remember, "At the core of every CLI application is a console application"), and then we introduced a library for parsing its command-line arguments including commands, subcommands, and options so we don't need to reinvent the wheel.

In the upcoming chapter, we will see how to control inputs and outputs and how to read data from and write data to a file so we can perform backup and restore operations for our bookmarks. This will prove especially useful to import and export bookmarks into and out of our bookmark manager application

Your turn!

Following along with the provided code is a great way to learn through practice.

A better way is by challenging yourself to achieve tasks. Hence, I challenge you to improve the *Bookmarkr* application by adding the following features.

Task #1 – Delete an existing bookmark

The syntax is as follows:

```
$ bookmarkr link remove --name <name>
```

It can also be as follows:

```
$ bookmarkr link remove -n <name>
```

If the requested link name does not exist, the application should display a warning message to the user. Otherwise, the application should delete that bookmark and display a success message to the user.

Task #2 – Update an existing bookmark

The syntax is as follows:

```
$ bookmarkr link update --name <name> --url <url>
```

It can also be as follows:

```
$ bookmarkr link update -n <name> -u <url>
```

If the requested link name does not exist, the application should display an error message to the user and invite them to use the `add` command to add a new bookmark. Otherwise, the application should update the existing bookmark with the newly provided URL and display a success message to the user.

Task #3 – List all existing bookmarks

The syntax is as follows:

```
$ bookmarkr link --list
```

It can also be as follows:

```
$ bookmarkr link -l
```

If the list of bookmarks contains no items, the application should display a warning message saying that the list of bookmarks is empty and, therefore, there is nothing to display. Otherwise, the application should present the list of bookmarks as follows:

```
# <name 1>
<url 1>

# <name 2>
<url 2>

...
```

5

Input/Output and File Handling

In the previous chapter, we laid out the foundations of **Bookmarkr**, our CLI application for managing bookmarks. We started with a basic console application, and we leveraged the `System.CommandLine` library to infuse CLI capabilities into the application.

For now, our CLI application only contains one command (`link`), which allows for managing bookmarks by adding new ones or listing, updating, or removing existing ones.

With this chapter, we are working toward two goals:

1. To go a bit deeper with options to further control input values for our CLI application's command options.
2. To see how to handle input and output files in a CLI application. This might be handy for import and export operations, making it easier to back up and restore our application's data and share it with other applications.

Specifically, we'll cover the following main topics:

- Controlling input values for an option, determining when to use required versus non-required options, setting default values for options, controlling the set of allowed values for an option, and validating input values
- Working with files passed as parameters to the CLI application, both as input and output files, which will be useful for adding import and export capabilities to our CLI application

Technical requirements

The code for this chapter can be found in the GitHub repository accompanying this book, `https://github.com/PacktPublishing/Building-CLI-Applications-with-C-Sharp-and-.NET/tree/main/Chapter05`.

Controlling input values for an option

Parameters are at the heart of any application. They allow users to indicate what command they want to execute and provide values to the input parameters. This is why, in this section and its subsections, we will cover the subtleties of dealing with these parameters.

Required versus non-required options

In its current state, adding a new bookmark requires both the name and the URL to be provided, which is, obviously, what we want.

This means that if we call the `link add` command without passing one of these options or their values, we should get an error such as the following:

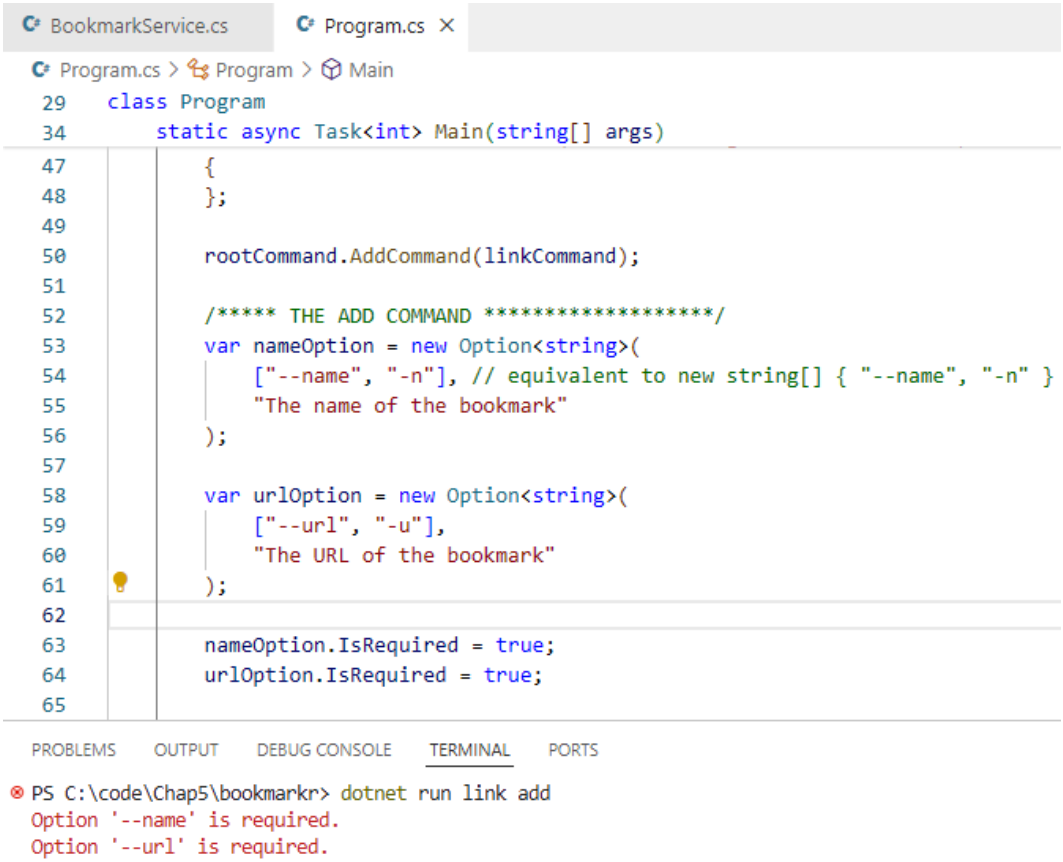


Figure 5.1 – The name and URL for a bookmark should be required

However, if we run the program without passing these two options, we currently get the following result:

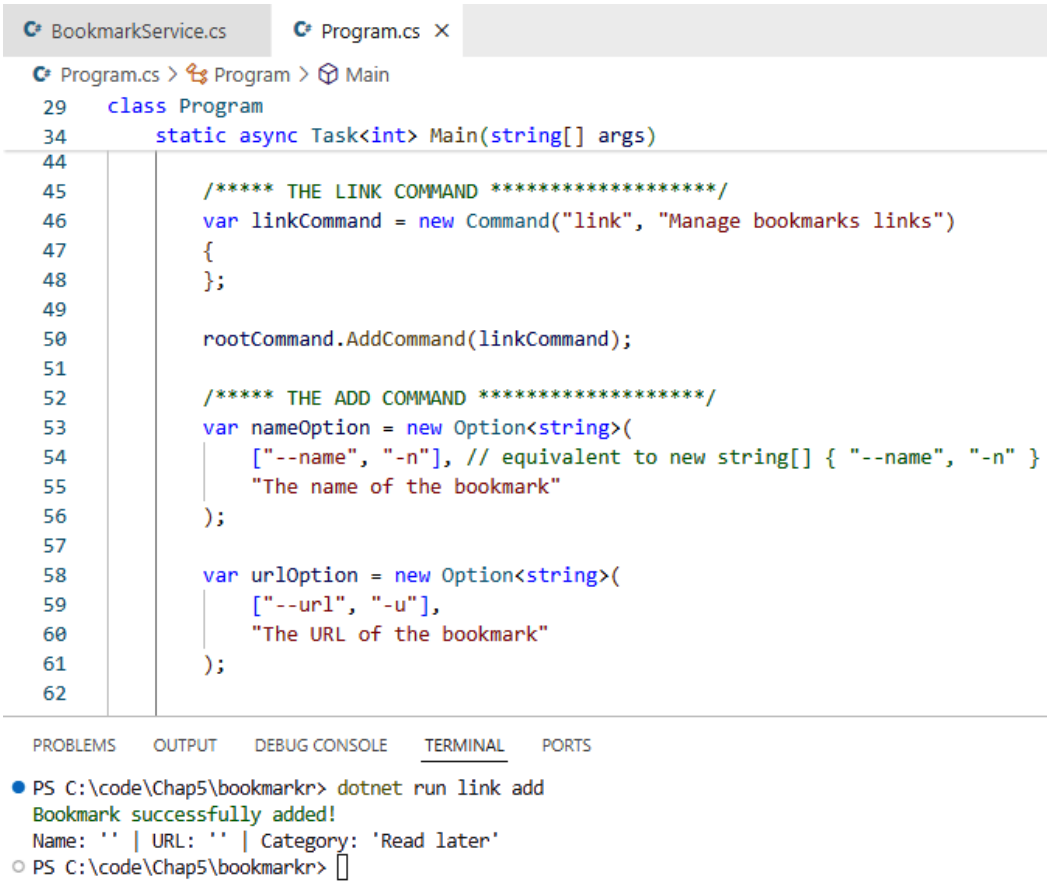


Figure 5.2 – The name and URL for the added bookmark are currently optional

Notice how a new bookmark with no name and no URL is added to the collection of bookmarks. This is clearly not what we want!

Fortunately, the `Option` class provides a Boolean value to specify whether it should be required or optional.

To make the name and URL options required, let's set their respective `IsRequired` property to `true`:

```
nameOption.IsRequired = true;
urlOption.IsRequired = true;
```

If we now run the program without passing in an option or its value, we get an error message:

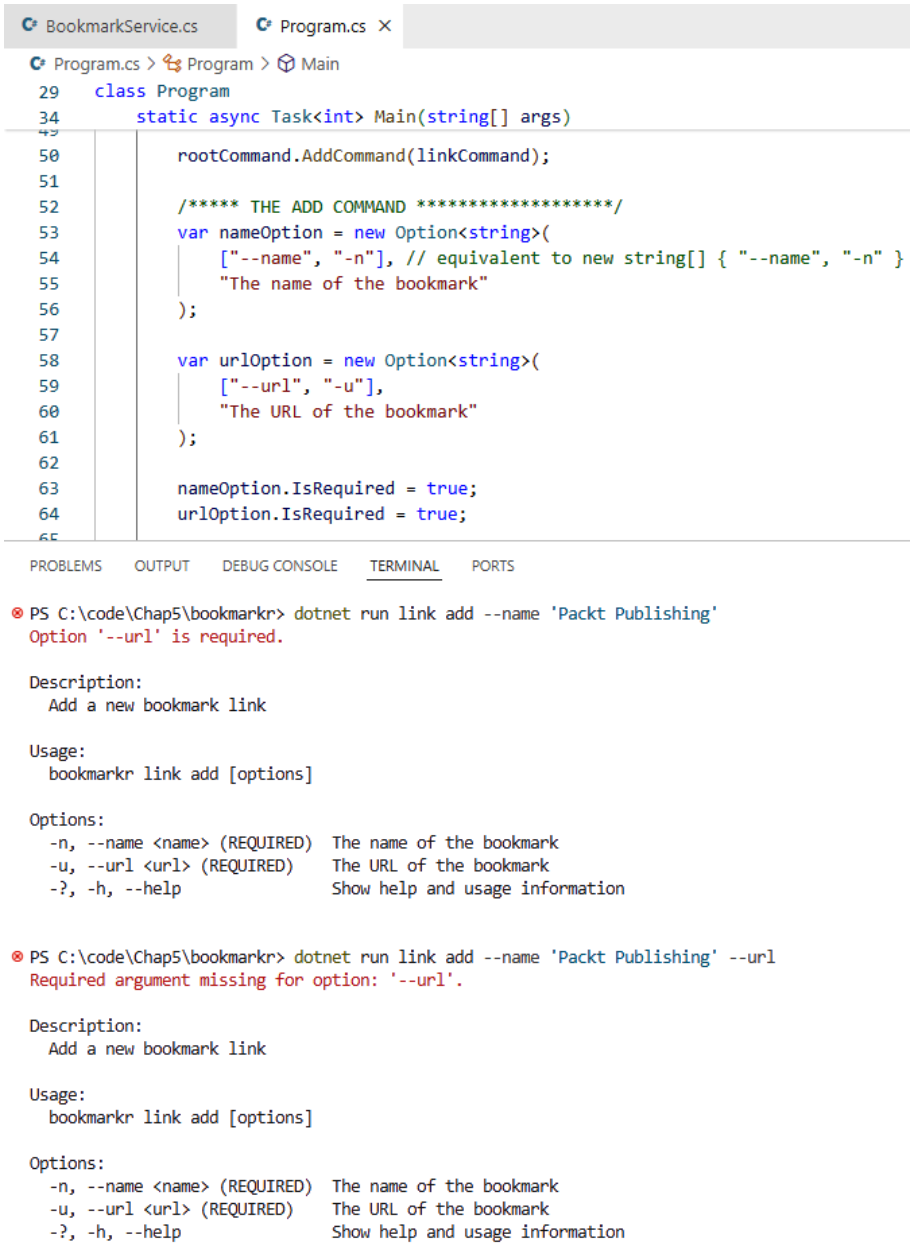


Figure 5.3 – The name and URL for the added bookmark are now required

Also note that the help menu clearly states that these two options are required.

So far, we have two required options. Let's add an optional one.

When an option is not required (i.e., optional), the application should not return an error if we don't pass that option or its value.

Let's take an illustrative example.

Let's say we want to classify our bookmarks by category. By doing so, we can imagine that we may want to list only the bookmarks that belong to a specific category.

For that matter, we will first add a `Category` property to the `Bookmark` class, as follows:

```
public class Bookmark
{
    public required string Name { get; set; }
    public required string Url { get; set; }
    public required string Category { get; set; }
}
```

Then, we will add an option for the category and pass it to the `add` command:

```
var categoryOption = new Option<string>(
     ["--category", "-c"],
     "The category to which the bookmark is associated"
);
var addLinkCommand = new Command("add", "Add a new bookmark link")
{
     nameOption,
     urlOption,
     categoryOption
};
```

Next, we update the handler method and its association with the command:

```
addLinkCommand.SetHandler(OnHandleAddLinkCommand, nameOption,
urlOption, categoryOption);

static void OnHandleAddLinkCommand(string name, string url, string
category)
{
     service.AddLink(name, url, category);
}
```

Finally, do not forget to update `BookmarkService` so it handles the `Category` property accordingly.

Now, if we execute the application without passing the category, no error is returned:

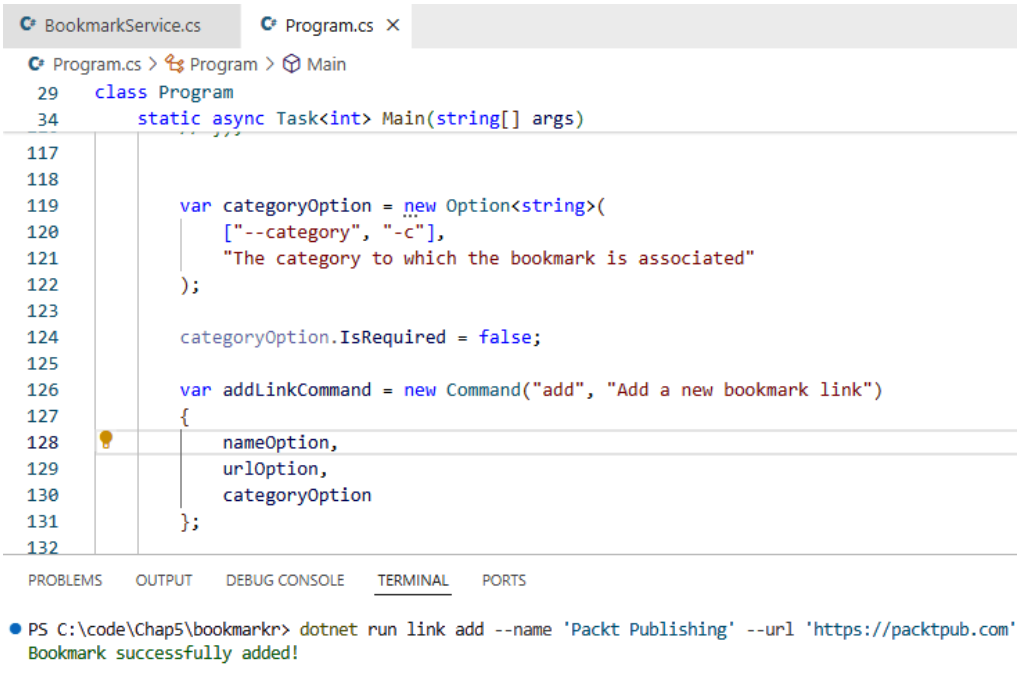


Figure 5.4 – The Category option is optional

And, of course, if we pass a category, it works too. 😊

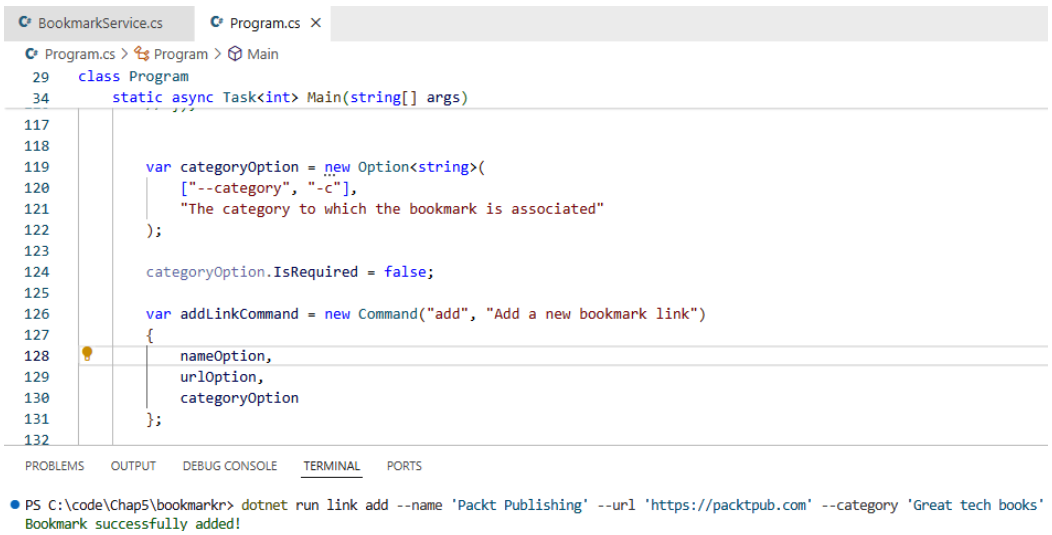


Figure 5.5 – Assigning a category to the newly added bookmark

However, since the category is now optional, if we don't pass it, what would its value be?

Double or single dash?

You might be wondering when we should use double versus single dashes. Do we even have to use both?

The answer is no! You only use both if you want to provide both a long and a short form for passing in an option, but you can definitely opt for only one of these options.

For example, while `--set-max-concurrent-requests` might be more self-explanatory to someone new to your CLI, if they often use your CLI application, having to type this long form again and again may become frustrating. That's why a short form, such as `-m`, will be more appropriate.

In the real world, you will notice that users who are just starting to use your CLI application will rely on long-form options and gradually transition to short forms as they become more experienced with your CLI application.

So, for example, a junior user of Bookmarkr will prefer this syntax:

```
bookmarkr link add --name "Packt Publishing" --url "https://
packtpub.com"
```

An experienced user, on the other hand, is likely to prefer this syntax:

```
bookmarkr link add -n "Packt Publishing" -u "https://packtpub.com"
```

What about arguments?

Ah! I can see that you've learned about arguments. Arguments are instances of the `Argument` class, and they represent required parameters that are essential to the execution of a command.

But wait… why not use arguments instead of options for required parameters?

You could, of course! But I don't like these because they are *positional* parameters and not *named* parameters. This means that only their position instructs the user about their purpose, which, to me, sacrifices the readability of the CLI request.

To illustrate my point, here is what the call to the `link add` command would look like if it relied on arguments rather than parameters:

```
$ bookmarkr link add 'Packt Publishing' 'https://packtpub.com' 'Great
tech books'
```

See how this is way less readable than our previous request (which relies on options)?

That's why I don't like arguments and prefer to use options, specifying which ones are required and which ones are optional.

So, let's get back to exploring options.

Setting a default value for an option

Well, as you may have guessed, the default value for an option will be (by default) the default value for its data type (are you still with me?).

Since the `Category` option is of type `string`, its default value is `null`. However, the `Option` class allows us to define a default value.

Let's set the default value for the `Category` option to `"Read later"`. This can be done by calling the `SetDefaultValue` method and passing in the default value:

```
categoryOption.SetDefaultValue("Read later");
```

If we run the program without providing a value for the `Category` option, we can see that its default value is used:

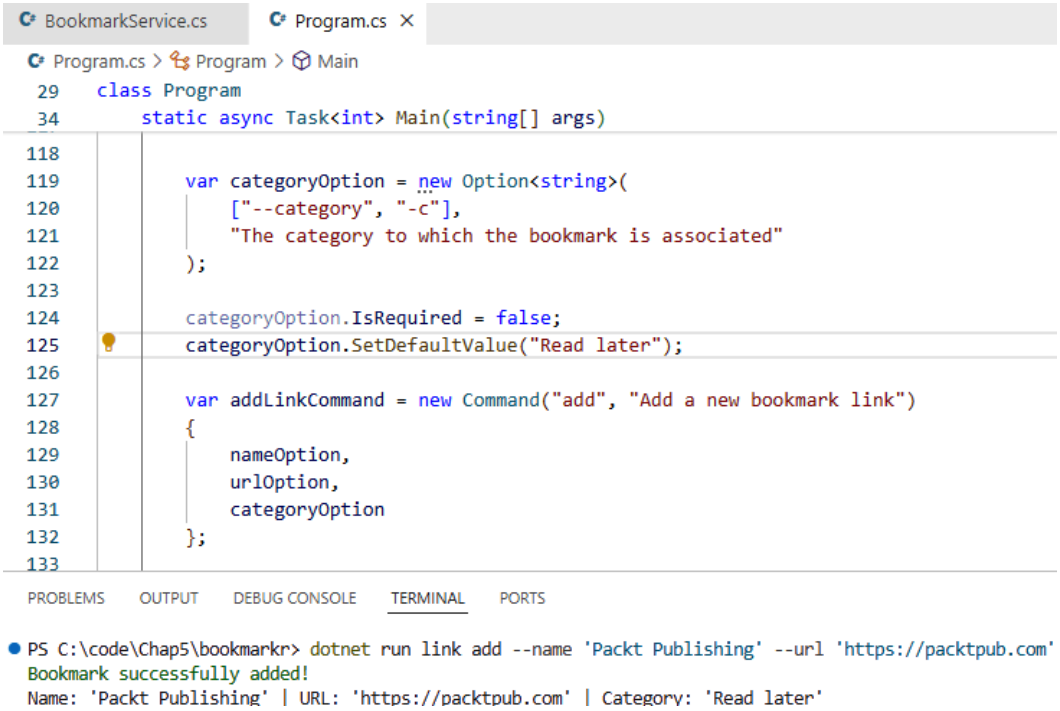


Figure 5.6 – Using the default value for the category option

However, if we do provide a value for the category, we can see that this value is actually used:

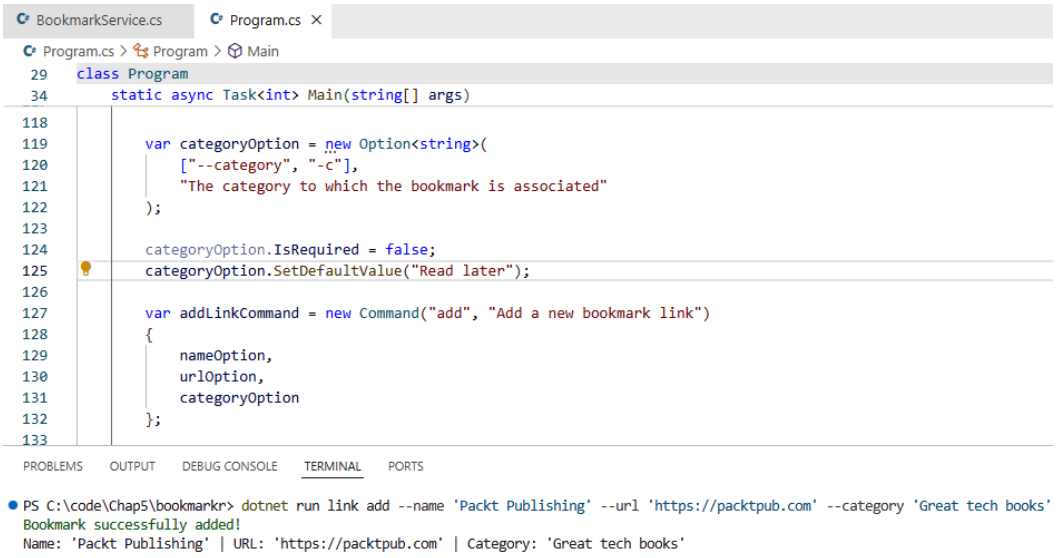


Figure 5.7 – Using the provided value for the category option

Should we provide default values for required options?

No, we should not! This is because if we do that, a required option will no longer behave as a required one but rather as an optional one.

Why? Because if we do not provide a value for it, the default value will be used.

This is why default values should only be used with optional options.

Note that in the previous example, the user can specify any string value for the `Category` option. But what if we wanted to control the set of allowed values? This is where the `FromAmong` method comes in.

Controlling the allowed values for an option

Let's pretend for a moment that we only allow a set of categories in our application. Yes, in real life, we would allow users to create as many categories as they want, but this will serve our purpose of explaining how to only allow a specific set of values for an option.

Let's say we allow the following categories:

- Read later (which serves as the default one)
- Tech books
- Cooking
- Social media

We will do this by passing these values to the `FromAmong` method as follows:

```
categoryOption.FromAmong("Read later", "Tech books", "Cooking",
"Social media");
```

If we run the application by passing in an allowed category, everything works fine:

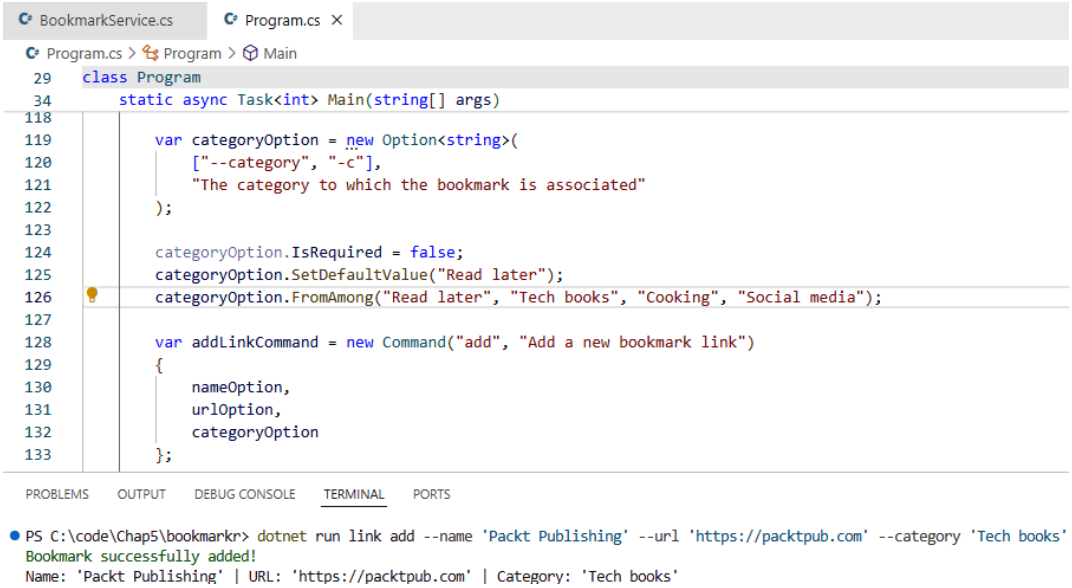


Figure 5.8 – Passing in an allowed value for the category

However, if we do pass an unallocated category value, we will get an error message:

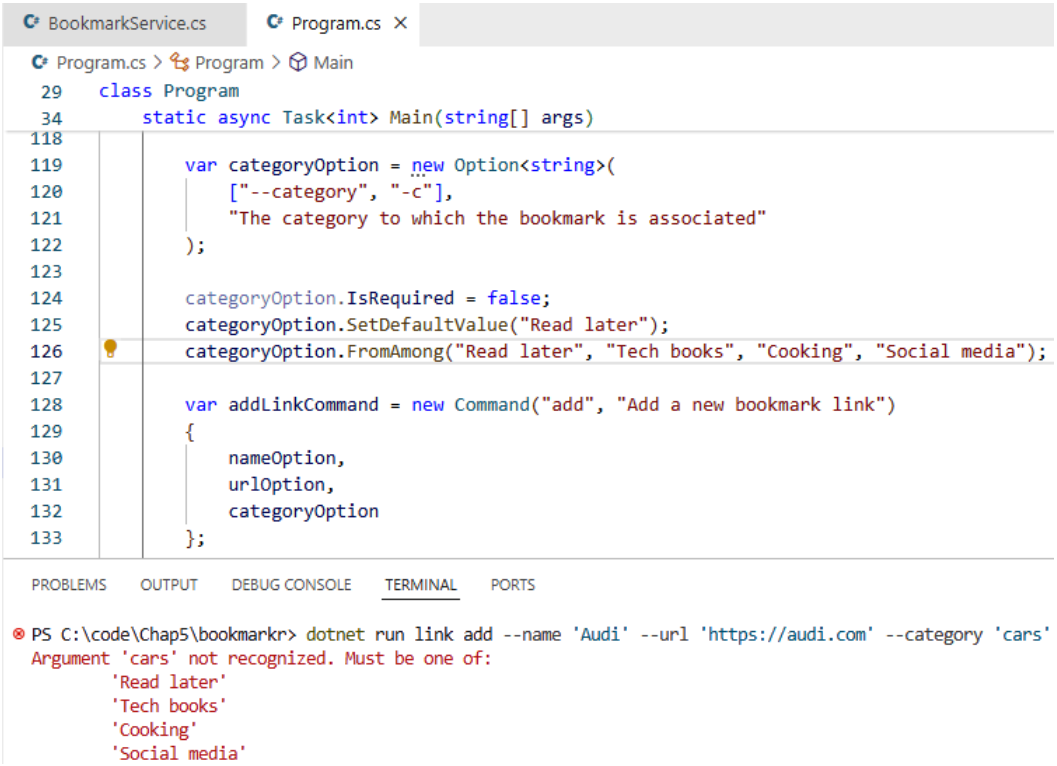


Figure 5.9 – Passing in an unallowed value for the category

Notice that the error message indicates the allowed values. We can also see the allowed values from the help menu:

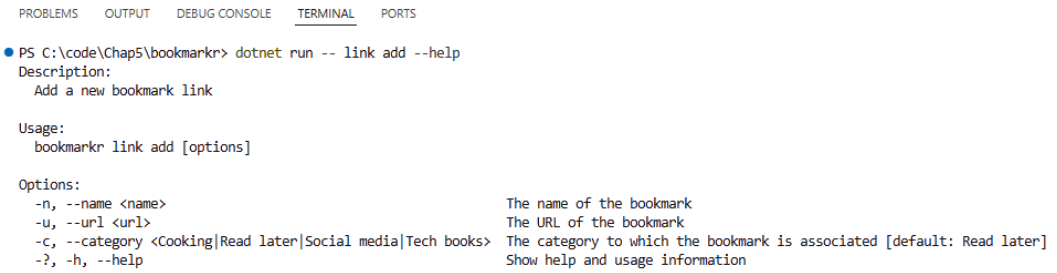


Figure 5.10 – Seeing the allowed values in the help menu

Using `FromAmong` can be particularly useful for ensuring data integrity and guiding user input, especially in scenarios where options need to conform to a predefined set of valid values.

Okay, so let's recap. Our CLI application has required and optional parameters. It specifies a default value for its optional parameter, along with allowed values. However, we are missing something, something important. Can you guess what it is?

Yes, exactly, the ability to ensure that the provided value for a specific parameter is valid.

Validating input values

When adding a new bookmark, we need to pass a URL for it. But, until now, we haven't checked whether the provided value is indeed a valid URL. Let's fix this.

The `Option` class allows us to configure a validator function. We will then add a validator method for `urlOption` to ensure it only gets valid URLs.

This can be achieved by calling the `AddValidator` method, as follows:

```
urlOption.AddValidator(result =>
{
    if (result.Tokens.Count == 0)
    {
        result.ErrorMessage = "The URL is required";
    }
    else if (!Uri.TryCreate(result.Tokens[0].Value, UriKind.Absolute,
    out _))
    {
        result.ErrorMessage = "The URL is invalid";
    }
});
```

In the preceding code snippet, the `AddValidator` method uses an inline delegate to ensure that the value provided to `urlOption` is valid. In this case, it ensures that it is actually present (that's what the `if` section is checking) and that it is a valid URL (that's what the `else if` section is checking).

So now, if we execute the program with both an invalid and a valid URL, we can see that it behaves as expected:

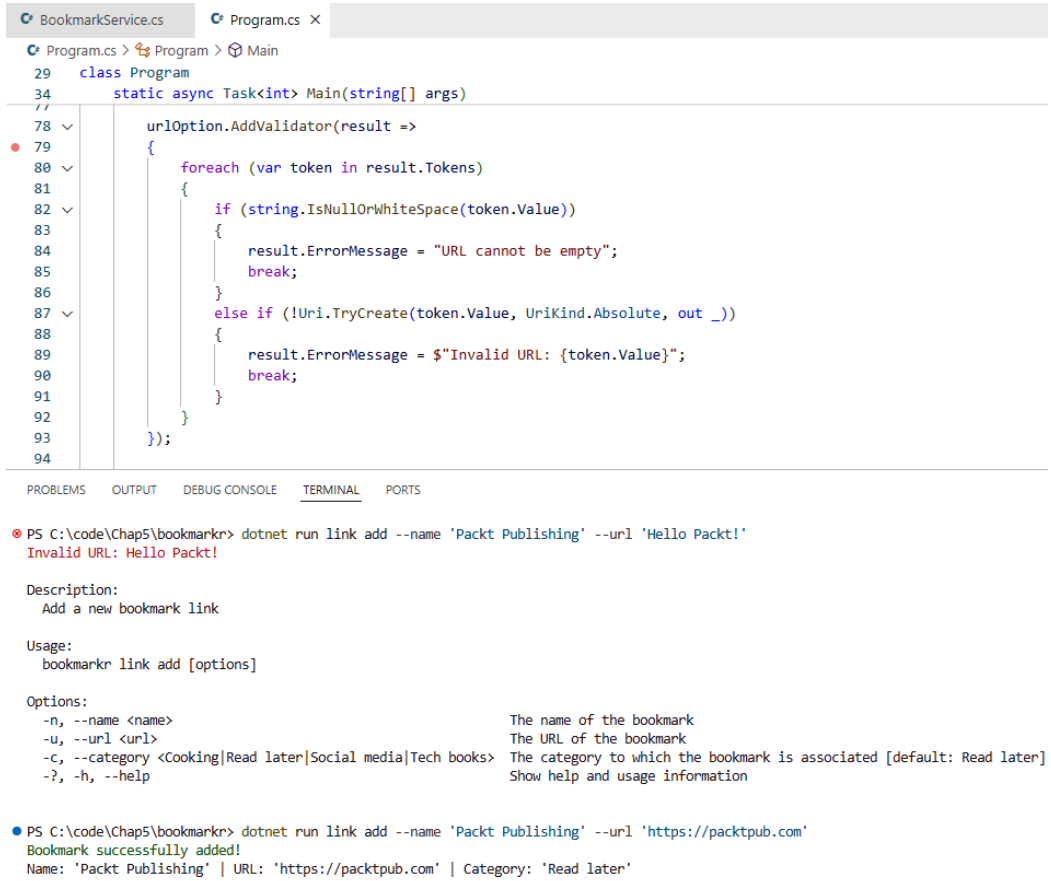


Figure 5.11 – Validating the input value for the URL option

> **More advanced validation**
>
> Validation could be more advanced than that. Our application is intended to collect bookmarks from everywhere on the web. However, if you would like to restrict its usage to, let's say, your organization only, you may want to check (in your validation process) that the bookmarked URLs are only referring to your corporate domain and dismiss everything else.

Perfect! So now, Bookmarkr allows us to manage bookmarks, ensuring that only valid information can be passed to (and stored in) the CLI application.

However, up to this point, we can still only add one bookmark at a time. Wouldn't it be nice if we could provide a set of names and URLs as part of the same request and have Bookmarkr add them in one go?

`System.CommandLine` has a feature that allows us to do just that 😉.

Adding multiple elements in one go

Let's try passing in multiple names and URLs to the same request, such as this:

```
dotnet run link add --name 'Packt Publishing' --url 'https://packtpub.
com/' --name 'Audi cars' --url 'https://audi.ca'
```

But if we do this, we will get the following error:

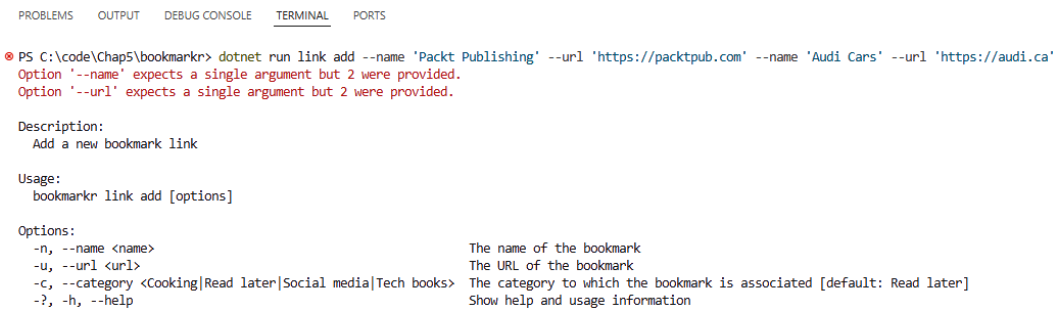


Figure 5.12 – Name and URL options expect only one value by default

This is due to the arity of these options.

What is an arity, anyway?

The arity of an option represents the number of values that can be passed if that option is specified. It is expressed with a minimum value and a maximum value.

This is of great importance if your CLI application supports bulk operations through one or many of its commands. In our example, we want to perform a bulk operation for adding multiple bookmarks at the same time.

In the case of an option of type `string`, the minimum and maximum values are both set to `1`, which means that if we specify the option, we must provide a value.

A Boolean option will have a minimum value of `0` and a maximum value of `1` since we don't need to pass in a value as both these syntaxes are valid:

```
--force
-- force true
```

In the same way, a list of elements has a minimum arity of 1 and a maximum of (by default) 100,000.

In order to specify the arity of an option, `System.CommandLine` provides an enumeration named `ArgumentArity`, which has these values:

- `Zero`, meaning no values are allowed. So, `--force` would be valid but not `--force true`.
- `ZeroOrOne`, meaning a minimum of zero and a maximum of one value is allowed.

- `ZeroOrMore`, meaning either zero, one, or many values are allowed.
- `ExactlyOne`, meaning a minimum of one and a maximum of one value is allowed. This is the case for our string options, name, and URL.
- `OneOrMore`, meaning either one or multiple values are allowed.

To set the arity of an option, we can then use one of the values provided by the `ArgumentArity` enumeration, like this:

```
nameOption.Arity = ArgumentArity.OneOrMore;
```

So now, we should be able to provide multiple values for a given option. Let's try this:

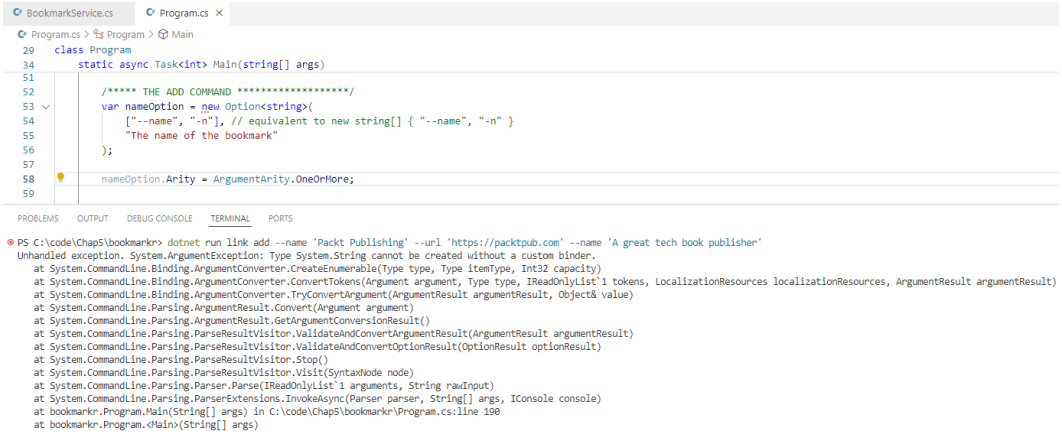


Figure 5.13 – Failing to provide multiple values for a given option

Oops, this is not what we expected, right?

The problem here is that even though `nameOption` can accept more than one value, it is not clear to the program how to convert those values into a single string. This is why the error message is referring to a custom binder (so it is told how to perform such a conversion).

In order to fix this problem, we need to tell the program to treat each of these inputs as a separate argument. This is done by setting the `AllowMultipleArgumentsPerToken` property to `true`, as follows:

```
nameOption.AllowMultipleArgumentsPerToken = true;
```

Also, let's get rid of the arity for a moment by commenting out the corresponding line of code.

Now, if we run the program, we can see that the error is gone but we are still not getting the expected result…

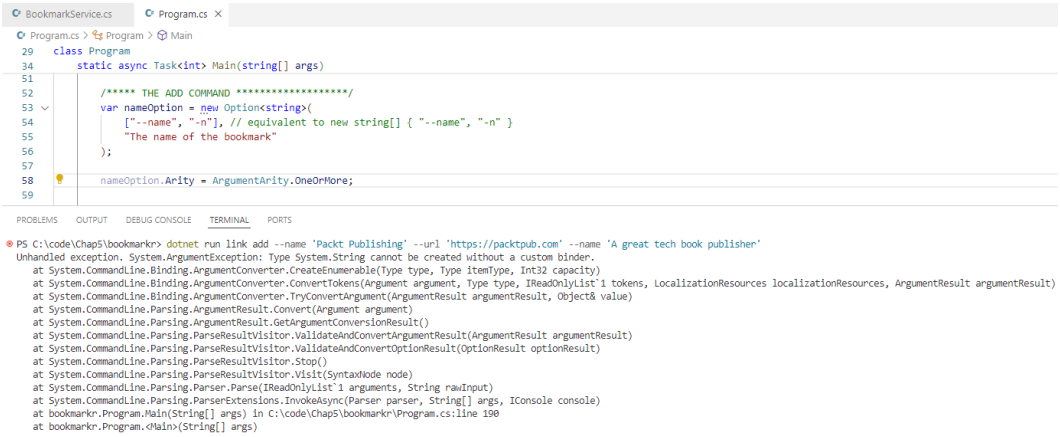


Figure 5.14 – nameOption is now accepting multiple values

Notice how only the last pair of names and URLs were considered and added to the list of bookmarks.

What happened, in fact, is that `System.CommandLine` noticed that we have two occurrences of the name and URL, so the last ones have overridden the first ones and only those last ones were actually passed to the `Handler` method. This is why we only get one bookmark added with the information of the last pair of names and URL values.

But what if we want to be able to pass a list of names and URLs and have the `Handler` method add as many bookmarks as the number of name and URL pairs?

To do this, we need two things. First, let's uncomment the lines of code that set the arities for `nameOption`, `urlOption`, and `categoryOption`.

Next, let's change the declaration of the name, URL, and category options along with the validator and the signature of the `Handler` method so that they accept a list of strings rather than a single string:

```
var nameOption = new Option<string[]>(
    ["--name", "-n"], // equivalent to new string[] { "--name", "-n" }
     "The name of the bookmark"
);
nameOption.IsRequired = true;
nameOption.Arity = ArgumentArity.OneOrMore;
nameOption.AllowMultipleArgumentsPerToken = true;


var urlOption = new Option<string[]>(
    ["--url", "-u"],
```

```
    "The URL of the bookmark"
);
urlOption.IsRequired = true;
urlOption.Arity = ArgumentArity.OneOrMore;
urlOption.AllowMultipleArgumentsPerToken = true;
urlOption.AddValidator(result =>
{
    foreach (var token in result.Tokens)
    {
        if (string.IsNullOrWhiteSpace(token.Value))
        {
            result.ErrorMessage = "URL cannot be empty";
            break;
        }
        else if (!Uri.TryCreate(token.Value, UriKind.Absolute, out _))
        {
            result.ErrorMessage = $"Invalid URL: {token.Value}";
            break;
        }
    }
});
var categoryOption = new Option<string[]>(
    ["--category", "-c"],
    "The category to which the bookmark is associated"
);

categoryOption.Arity = ArgumentArity.OneOrMore;
categoryOption.AllowMultipleArgumentsPerToken = true;

categoryOption.SetDefaultValue("Read later");
categoryOption.FromAmong("Read later", "Tech books", "Cooking",
"Social media");
categoryOption.AddCompletions("Read later", "Tech books", "Cooking",
"Social media");

static void OnHandleAddLinkCommand(string[] names, string[] urls,
string[] categories)
{
    service.AddLinks(names, urls, categories);
    service.ListAll();
}
```

Now, if we run the program, things work (finally) as expected! 😊

Figure 5.15 – Bookmarkr accepts a list of bookmarks

Since each option accepts multiple values, let's take a look at whether we can simplify the following CLI request:

```
$ dotnet run link add --name 'Packt Publishing' --url 'https://
packtpub.com/' --category 'Tech books' --name 'Audi cars' --url
'https://audi.ca' --category 'Read later'
```

We'll simplify it as follows:

```
$ dotnet run link add --name 'Packt Publishing' 'Audi cars' --url
'https://packtpub.com/' 'https://audi.ca' --category 'Tech books'
'Read later'
```

Notice that we only need to specify `--name`, `--url`, and `--category` once.

Since both CLI requests are equivalent, they lead to the same result:

Figure 5.16 – Simplified CLI request

Excellent! This works just great!

But… typing a list of names, URLs, and categories might quickly become tedious as the list grows.

Wouldn't it be nice if we could simply provide the path to a file as a parameter that contains all the names, URLs, and categories and let the application read that file and create the bookmarks accordingly?

In the same way, wouldn't it be nice if we could specify the path to an output file to store all the bookmarks our CLI application is holding?

Working with files passed in as options values

Files can be provided as options values to serve as input or output parameters.

As an input parameter, a file's content can be read to import data into the CLI application. In our case, we could import bookmarks from other browsers, such as Chrome or Firefox, into Bookmarkr.

As an output parameter, a file can be created to export the data that is held by Bookmarkr, which in turn can be imported into other browsers, such as Chrome or Firefox.

Together, these two capabilities can enable backup and restore but also data sharing and exchange scenarios.

Let's build these features into Bookmarkr!

> **Important note**
>
> Browsers, such as Chrome or Firefox, have their own proprietary structure to import and export bookmarks.
>
> We won't be performing parsing or conversion to these formats for the sake of simplicity. Our goal is to focus on working with input and output files as part of a CLI application. We will, however, import and export bookmarks in JSON format.

Let's begin with the `export` command.

This command is meant to take all bookmarks managed by Bookmarkr and save them in a JSON file whose path is specified as a value to the `--file` option. This option is, of course, required.

First, we will need to create an option of type `FileInfo`, and it will be required:

```
var outputfileOption = new Option<FileInfo>(
    ["--file", "-f"],
    "The output file that will store the bookmarks"
)
{
    IsRequired = true
};
```

Next, we will need to create a new command and add it to the `root` command:

```
var exportCommand = new Command("export", "Exports all bookmarks to a
file")
{
    outputfileOption
};

rootCommand.AddCommand(exportCommand);
```

Then, we need to set a `Handler` method for the `export` command:

```
exportCommand.SetHandler(OnExportCommand, outputfileOption);

static void OnExportCommand(FileInfo outputfile)
{
    var bookmarks = service.GetAll();
    string json = JsonSerializer.Serialize(bookmarks, new
    JsonSerializerOptions { WriteIndented = true });
    File.WriteAllText(outputfile.FullName, json);
}
```

The `Handler` method calls `BookmarkService` to get the list of all bookmarks, then converts them to JSON and saves that JSON content into the provided file. If the file already exists, it is overwritten.

Note that you'll need to import this namespace for the code to compile:

```
using System.Text.Json;
```

Now, let's try it and see whether it works as expected!

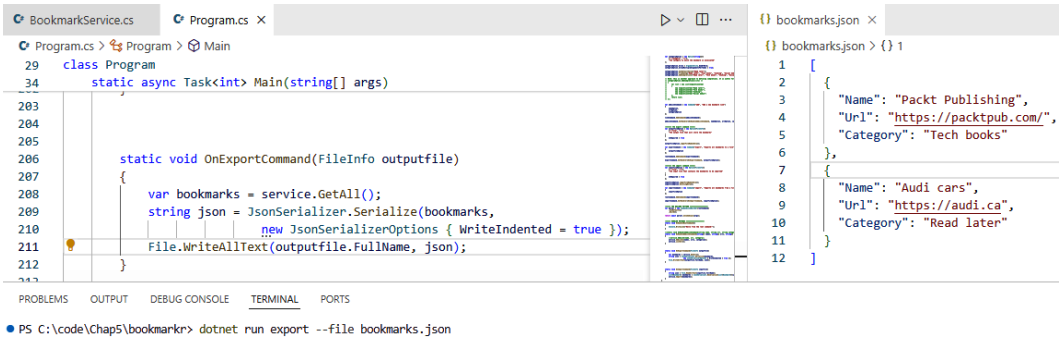


Figure 5.17 – Exporting all bookmarks

Perfect! This is exactly what we expected!

But how can we ensure that the provided file has a valid name?

We could certainly create a validator method to check this, but `System.CommandLine` already provides an extension method for that matter (and I wanted to let you know 😉):

```
outputfileOption.LegalFileNamesOnly();
```

Let's try calling the `export` command with an invalid file:

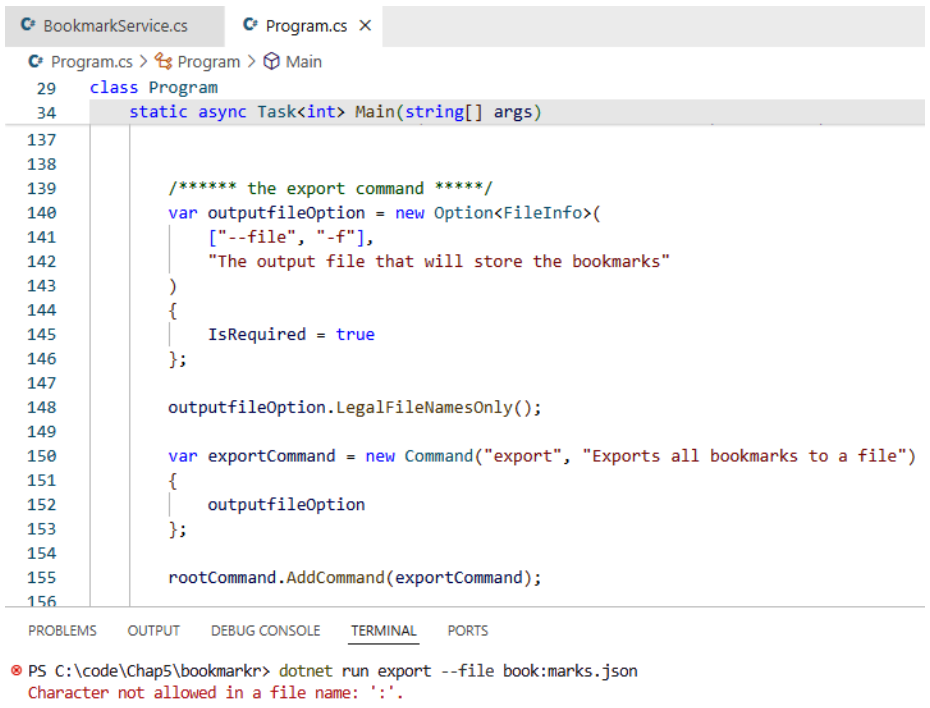


Figure 5.18 – Handling invalid files

See? That error has been raised because of the call to the `LegalFileNamesOnly` method.

Okay! Now let's move on to adding the `import` command!

As a reminder, the syntax to import bookmark data from an existing file is as follows:

```
$ bookmarkr import --file <path to the input file>
```

Since many of the steps involved are very similar to the ones we followed to create the `export` command, let's just share the code here and discuss the differences:

```
var inputfileOption = new Option<FileInfo>(
    ["--file", "-f"],
    "The input file that contains the bookmarks to be imported"
)
{
    IsRequired = true
};

inputfileOption.LegalFileNamesOnly();
inputfileOption.ExistingOnly();

var importCommand = new Command("import", "Imports all bookmarks from
a file")
{
    inputfileOption
};

rootCommand.AddCommand(importCommand);

importCommand.SetHandler(OnImportCommand, inputfileOption);

static void OnImportCommand(FileInfo inputfile)
{
    string json = File.ReadAllText(inputfile.FullName);
    List<Bookmark> bookmarks = JsonSerializer.
    Deserialize<List<Bookmark>>(json) ?? new List<Bookmark>();
    service.Import(bookmarks);
}
```

The main difference is the call to the `ExistingOnly` method. This method ensures that `inputfileOption` will only accept values corresponding to existing files, otherwise an error is raised.

The other difference is how the `OnImportCommand` handler method operates: it reads the content of the file, converts it from JSON to a list of items of type `Bookmark`, and then passes it to `BookmarkService` to add these items to the list of bookmarks it manages (by calling its `Import` method).

Now, let's try this code!

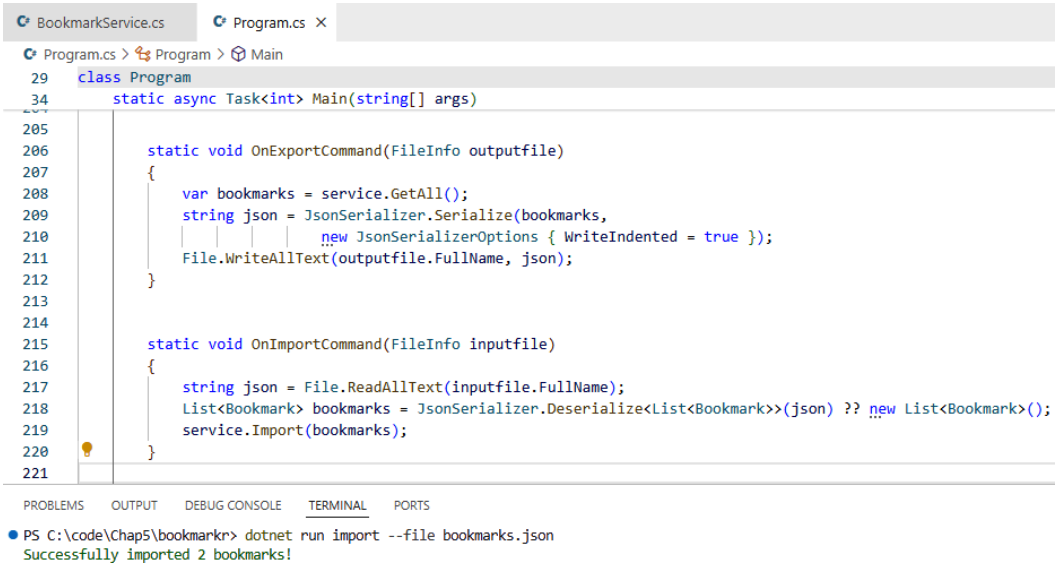


Figure 5.19 – Importing bookmarks from a file

What happens if the file doesn't exist?

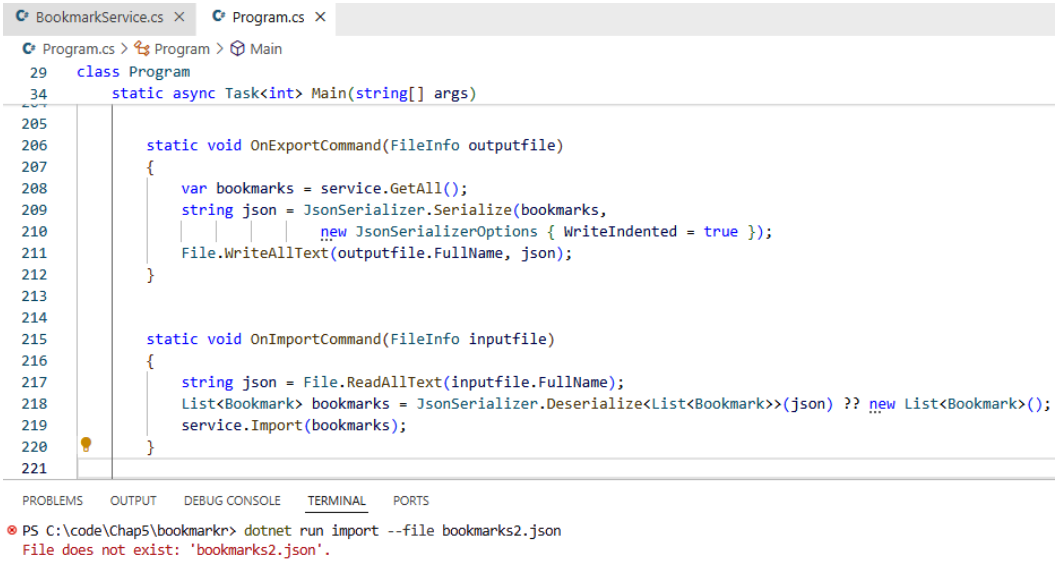


Figure 5.20 – Handling a non-existent file

Once again, we can see that we are getting the expected result! 😊

And that's a wrap! You now know how to work with input and output files in your CLI applications. Congratulations! Let's now conclude this chapter.

Summary

In this chapter, we improved our CLI application, Bookmarkr, by adding better control for the input values for its command's options (by explicitly indicating what options are required, setting default values where appropriate, designing validators to ensure the input values comply with the expected type, format, or range of values, and enabling auto-completion to make things simpler for the user).

We also added the ability to import and export application data from and to a file. This makes it easier to back up and restore data and even share it offline.

In the upcoming chapter, we will see how to implement a very important feature in every application: logging and error handling.

Your turn!

Following along with the provided code is a great way to learn through practice.

A better way is by challenging yourself to achieve tasks. Hence, I challenge you to improve the Bookmarkr application by adding the following features.

Task #1 – validating the format and the ability to access the input file

As a reminder, the syntax to import bookmark data from an existing file is as follows:

```
$ bookmarkr import --file <path to the input file>
```

If the input file cannot be accessed, or if its data is not in the expected format, then the application should display a corresponding error message to the user. Otherwise, the application should import all the bookmarks from the input file and display a success message to the user indicating how many bookmarks have been imported.

Task #2 – merging existing links from the input file

When importing bookmarks from an existing file, it is possible that some of them already exist in the bookmarks held by the application.

In such a situation, it is a best practice for a CLI application to provide the user with an option to control whether they want to merge those existing links or simply discard them and not import them.

In this task, I challenge you to implement this best practice by adding an optional `--merge` option to the `import` command.

Hence, the syntax for the `import` command with the `--merge` option will be as follows:

```
$ bookmarkr import --file <path to the input file> --merge
```

When the `--merge` option is specified, the expected behavior of the `import` command is that for each bookmark in the provided input file, the following apply:

- If its URL already exists in the list of bookmarks held by the application, the name of the existing bookmark should be updated with the name corresponding to this URL in the input file
- Otherwise, the bookmark should simply be added to the list of bookmarks held by the application

6

Error Handling and Logging

Logging and error handling are two important concepts to consider (and implement) when building any application, and CLI applications are no exceptions.

While error handling ensures graceful behavior of the application in the face of unexpected events, logging provides crucial insights into the application's runtime behavior and facilitates troubleshooting and debugging.

That's why, in this chapter, we will cover these two concepts, starting with error handling.

Specifically, we'll cover the following main topics in this chapter:

- Error handling in CLI applications
- Logging in CLI applications

Technical requirements

The code for this chapter can be found in the GitHub repository accompanying this book, `https://github.com/PacktPublishing/Building-CLI-Applications-with-C-Sharp-and-.NET/tree/main/Chapter06`.

Handling errors in CLI applications

Error handling may take two forms:

- An exception is raised, due to an unexpected event (such as an invalid input or an inaccessible dependency)
- The program is terminated, and we want to prevent it crashing by allowing it to gracefully shut down

In this section, we will cover both topics. Let's start with exception handling.

Handling exceptions

There is nothing special about handling exceptions in a CLI application compared to other kinds of applications, as it follows the same guidelines and best practices. That's why, in this section, you might find that you already know all the concepts that we are going to talk about, and that's perfectly normal because you'd likely have implemented them in other applications, whether web, APIs, or desktop applications.

However, it is important to note that a robust error-handling strategy will have a significant impact on the quality, reliability, and resilience of your application. This is why it is worth taking the time to design a good error-handling strategy.

As you certainly know, every error-handling strategy relies on a `try-catch-finally` block. But not always! Not every method needs to implement a `try-catch-finally` block. In fact, best practices of error handling state that only the *caller* method (usually the top-level method) should handle exceptions, while the *callee* method should let the exceptions bubble up to be caught, and handled, by the *caller* method. This results in leaner, cleaner, and more focused methods.

Another best practice is to never swallow exceptions unless it is absolutely necessary. Why? Because swallowing exceptions hides errors, making code appear functional while it fails. This obscures valuable error information, allowing unpredictable behavior and data corruption. It complicates debugging and violates the principle of failing fast. In other words, it leads to silent failures that are hard to detect and fix, making it a poor practice.

The `finally` block is important, although I see it often forgotten. It is important to remember that this block is used to ensure that resources are freed, even if an exception occurs.

When catching exceptions, use multiple `catch` blocks, catching exceptions from the most specific to the most generic. This will ensure far better error handling than catching all exceptions as a generic exception and applying the same error-handling processing to all of them. An example of that is when working with files: we don't want to handle the situation where the file is not found the same way as handling the situation where the file cannot be written to, because of a lack of permissions. By distinguishing each of these situations, we can apply a specific error-handling process and, ultimately, provide the user with appropriate details of what happened rather than having generic processing and telling the user that we couldn't write to the file.

We can also create our own exceptions. I do this often because it helps improve code readability. It is easy to understand what happens when the `CreateNewUser` method throws an instance of the `UserAlreadyExistsException` exception, isn't it?

Here's how this custom exception would look:

```
public class UserAlreadyExistsException : Exception
{
    public string UserId { get; }
```

```
    public UserAlreadyExistsException(string userId)
        : base($"User with ID '{userId}' already exists.")
    {
        UserId = userId;
    }

    public UserAlreadyExistsException(string userId, Exception
    innerException)
        : base($"User with ID '{userId}' already exists.",
        innerException)
    {
        UserId = userId;
    }
}
```

To catch or not to catch exceptions?

There's a movement nowadays that is going against throwing exceptions because of the performance cost this may have, and I totally understand this. One great video about this topic is the one by Nick Chapsas, titled *Don't throw exceptions in C#. Do this instead*, which you can find on YouTube. I encourage you to go and watch it and make up your own mind.

However, whether you choose to throw the exception or handle it without throwing it, you will likely be dealing with exceptions. Also, keep in mind that both the .NET framework and some other libraries that you might be using are probably throwing exceptions, and you will need to catch these to handle them. Because of that, the principles described here are still valid and worth knowing.

When catching exceptions, you can also filter them. This is because some exceptions may require a different handling mechanism depending on the reason they were thrown.

A good example of this is the `HttpResponseException` exception type, as illustrated here:

```
try
{
    using var client = new HttpClient();
    var response = await client.GetAsync("https://api.packtpub.com/
    data");
    response.EnsureSuccessStatusCode();
}
catch (HttpRequestException ex) when (ex.StatusCode == System.Net.
HttpStatusCode.NotFound)
{
    Console.WriteLine("Resource not found (404)");
}
```

```
catch (HttpRequestException ex) when (ex.StatusCode == System.Net.
HttpStatusCode.Unauthorized)
{
    Console.WriteLine("Server error (401)");
}
```

As you can see here, we are catching the same exception (`HttpResponseException`) twice, but in each `catch` block, we focus on a very specific situation: in the first one, we are handling the situation where the resource was not found, whereas, in the second one, we are handling the situation where the user accessing the resource was not authenticated.

One final best practice when handling exceptions is to avoid losing the stack trace of the exception when throwing it.

To illustrate this principle, let's consider this example:

```
try
{
      // Attempt to read from a file
      string content = File.ReadAllText(fileName);
      Console.WriteLine($»File content: {content}»);
}
catch (FileNotFoundException ex)
{
      // Handle the specific exception
      Console.WriteLine($"File not found: {fileName}");
      Console.WriteLine($"Exception details: {ex.Message}");

      // Rethrow the exception
      throw ex;
}
```

If we throw the exception using the `throw ex;` statement, we lose the stack trace containing the details of what happened up to this point. The proper way to do this is to simply use `throw` to ensure the stack trace is preserved.

However, in some cases, we may need to catch the exception, handle it, and rethrow it by encapsulating it into another exception type, as shown here:

```
// Rethrow the exception by encapsulating it while preserving the
// stack trace
throw new IOException($"File process error{fileName}", ex);
```

In this case, the stack trace is preserved.

Now, let's apply these principles to *Bookmarkr*, and more specifically, to the ability to export bookmarks out of *Bookmarkr*.

If you recall from the previous chapter, the export handler method (`OnHandleExportCommand`) looked like this:

```
static void OnExportCommand(FileInfo outputfile)
{
    var bookmarks = service.GetAll();
    string json = JsonSerializer.Serialize(bookmarks, new
    JsonSerializerOptions { WriteIndented = true });
    File.WriteAllText(outputfile.FullName, json);
}
```

However, note that the `WriteAllText` method may throw many exceptions, such as the following:

- `UnauthorizedAccessException` will be thrown in the case of insufficient permissions to access the file
- `DirectoryNotFoundException` will be thrown if the path is invalid
- `PathTooLongException` will be thrown if the path exceeds the system-defined maximum length

So, let's handle these exceptions. The code would look like this:

```
static void OnExportCommand(FileInfo outputfile)
{
    try
    {
         var bookmarks = service.GetAll();
         string json = JsonSerializer.Serialize(bookmarks, new
         JsonSerializerOptions { WriteIndented = true });
         File.WriteAllText(outputfile.FullName, json);
    }
    catch(JsonException ex)
    {
        Helper.ShowErrorMessage([$"Failed to serialize bookmarks to
        JSON.",
        $"Error message {ex.Message}"]);
    }
    catch (UnauthorizedAccessException ex)
    {
         Helper.ShowErrorMessage([$"Insufficient permissions to access
         the file {outputfile.FullName}",
          $"Error message {ex.Message}"]);
    }
    catch (DirectoryNotFoundException ex)
    {
```

```
            Helper.ShowErrorMessage([$"The file {outputfile.FullName}
            cannot be found due to an invalid path",
            $"Error message {ex.Message}"]);
        }
        catch (PathTooLongException ex)
        {
            Helper.ShowErrorMessage([$"The provided path is exceeding the
            maximum length.",
              $"Error message {ex.Message}"]);
         }
         catch (Exception ex)
         {
            Helper.ShowErrorMessage([$"An unknown exception occurred.",
              $"Error message {ex.Message}"]);
         }
}
```

In the preceding example, we are handling the most common exceptions, and we are also handling the general exception in case of an unexpected exception (yes, exceptions are exceptional situations but are nonetheless expected to happen, at least most of the time).

Note that we handled exceptions for both the serialization process and the file-writing process.

If you want to learn more about best practices for handling exceptions, I recommend that you visit this page: `https://learn.microsoft.com/en-us/dotnet/standard/exceptions/best-practices-for-exceptions`.

Handling errors doesn't necessarily mean handling exceptions

As surprising as it may seem, this is true, and exceptions might be avoided (at least most of the time) by applying defensive programming techniques.

By validating inputs, enforcing preconditions, and proactively identifying potential failure scenarios, we can significantly reduce error occurrences and enhance the overall resilience of our application.

Let's see what we can do here in terms of defensive programming:

- **Validate inputs**: We could ensure that the input file exists, hence avoiding an exception being raised if the file doesn't exist.
- **Enforce preconditions**: We can ensure that the `bookmarks` list is not null. Otherwise, `JsonSerializer` will throw a `NullReferenceException`. We can also ensure that the list is not empty because, if it is empty, although the serialization will return an empty JSON array, we may not want to write this to a file, especially if this means overwriting the existing file.
- **Identify potential failure scenarios**: We have already done this by catching the most common exceptions and handling them.

Okay. So, at this point, we know how to handle exceptions in our CLI applications, and we implemented this in *Bookmarkr*.

However, there exists one other kind of unexpected event that represents, in fact, normal behavior. I'm talking about **program termination**.

Handling program termination

A program might be terminated at any moment by pressing a specific keyboard combination (usually *Ctrl* + *C* or *Ctrl* + *Break*). When this happens, the operating system sends a signal to the program, instructing it to immediately stop its execution. This signal, often referred to as an interrupt or a termination signal, allows the program to perform any necessary cleanup operations, such as closing files, releasing resources, or saving state, before it exits. If the program has a signal handler for this specific signal, it can execute custom code to handle the termination gracefully. Otherwise, the program will terminate abruptly, and any unsaved data or incomplete operations may be lost.

Program termination allows one to gracefully stop a program that is taking too long to execute or that has become unresponsive.

`System.CommandLine` provides a mechanism to handle program termination and execute custom code, allowing our CLI application to gracefully terminate.

Let's implement it to handle the situation where the user terminates the program while the export operation is ongoing.

In order to handle program termination, we will need to modify the delegate of the `SetHandler` method to retrieve the cancellation token and pass it to the handler method itself:

```
exportCommand.SetHandler(async (context) =>
{
    FileInfo? outputfileOptionValue = context.ParseResult.
    GetValueForOption(outputfileOption);
    var token = context.GetCancellationToken();
    await OnExportCommand(outputfileOptionValue!, token);
});
```

Now, we can modify the handler method so it handles the program termination (i.e., catching the `OperationCanceledException` exception):

```
static async Task OnExportCommand(FileInfo outputfile,
CancellationToken token)
{
     try
     {
        var bookmarks = service.GetAll();
        string json = JsonSerializer.Serialize(bookmarks, new
```

```
            JsonSerializerOptions { WriteIndented = true });
            await File.WriteAllTextAsync(outputfile.FullName, json,
            token);
        }
        catch(OperationCanceledException ex)
        {
            var requested = ex.CancellationToken.IsCancellationRequested ?
            "Cancellation was requested by you.": "Cancellation was NOT
            requested by you.";
            Helper.ShowWarningMessage(["Operation was cancelled.",
            requested, $"Cancellation reason: {ex.Message}"]);
        }
        catch(JsonException ex)

// The rest of the method has been removed for brevity.
```

If we run the program now and then terminate it by pressing the *Ctrl* + *C* keyboard combination, we get the following console output:

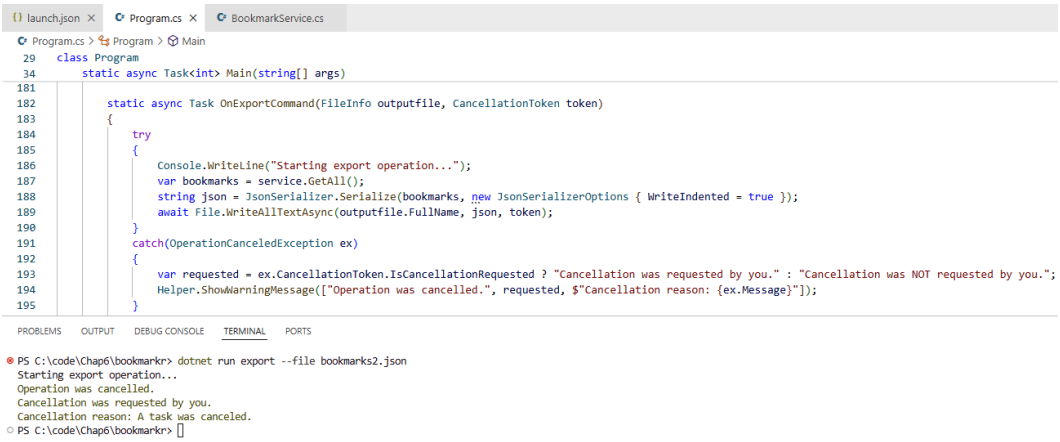


Figure 6.1 – Handling program termination

One thing worth mentioning here is that we need to use the asynchronous version of the `WriteAllText` method (aka `WriteAllTextAsync`) in order to be able to pass the cancellation token we received as a parameter, and for that reason, we needed to declare the `OnExportCommand` method as `async`.

Note that by handling program termination, we can gracefully handle the situation where the user abruptly stops the program. This results in a graceful shutdown of the program and release of the used resources, hence avoiding a crash and an error message.

Why are we passing the cancellation token as a parameter to the handler method and not using it?

This is an excellent question! As you have noticed, even though we receive the `CancellationToken` object as a parameter to the `OnExportCommand` method, we seem not to be using it. Then, why did we pass it in the first place?

This is related to how cancellation tokens work in .NET. Let me explain!

When `OperationCanceledException` is created in response to a cancellation request, it typically includes information about the `CancellationToken` object that triggered the cancellation.

For that matter, the `OperationCanceledException` exception class has a constructor that takes a `CancellationToken` object as a parameter.

When .NET framework asynchronous methods create this exception, they usually use this constructor and pass the cancellation token that triggered the cancellation request.

The `OperationCanceledException` class has a `CancellationToken` property that can be then accessed to get the token associated with the cancellation request.

In our case, we are accessing it to check whether the cancellation was requested by the user or not.

Having established a robust framework for error handling, it is equally important to ensure that these errors are recorded and monitored. That's where logging comes into play!

Effective logging not only helps in diagnosing and resolving issues but also provides valuable insights into the application's behavior, performance, and usage. In the upcoming section, we will dive into the best practices and strategies for implementing comprehensive logging mechanisms that complement our error-handling strategy.

At this point, we have covered a lot of information on error handling. However, error handling works together with logging to improve application reliability and maintainability. So, let's shift our focus to logging, and explore how to capture and preserve valuable information not only about errors but also about other significant events that will happen during program execution.

Logging in CLI applications

While error handling is more of a "just-in-time" compensation mechanism when an unexpected event or error occurs, we may want to keep track of what happened so we can reproduce the issue, analyze it, understand why it happened in the first place, and fix it.

By "keeping track of what happened," we either mean the sequence of events that led to that unexpected behavior or error and/or the call stack of the exception raised when the error occurred.

Selecting a logging format is important. We want to find a balance between the amount of data we are logging (and storing) and the usage we intend to do with it. Logging unnecessary information will complexify log analysis, increase storage (and retention) costs, and may also slow down the logging

process. We will also need to ensure that we are not logging sensitive information (such as credit card data), and if we do, that we are doing it in a safe manner. Some of the popular log formats include XML, JSON, CSV, and syslog.

Choosing the logging destination is equally important. We need to understand that there are no good or bad options, only appropriate and inappropriate options depending on our context and needs. If our intent is to analyze the logs, we may want to store these logs in a solution that provides log analysis mechanisms out of the box, so we don't need to write code for that. Examples of such solutions are Azure Log Analytics, Splunk, Datadog, Dynatrace, Serilog, and Elasticsearch.

However, note that by relying on cloud solutions (such as Azure Log Analytics), our application needs either to be running in the cloud or to have a constant connection to the internet. Sure, we can also build our application to follow the **occasionally connected application** (**OCA**) pattern so it keeps logs locally when it is running offline and sends them to Azure Log Analytics when it goes back online, but the idea here is that we should select a logging solution that is coherent with the usage pattern of our application. So, for an application that is intended to run locally, we will favor a logging mechanism that runs locally as well.

Finally, it is also important to define the log retention period. This can either be enforced by the organization's compliance rules or by the relevance of the logged data: do you still need to analyze data from a bug or customer behavior that occurred three years ago? If not, you don't need to keep this data.

Either way, it is important to separate the logging format from the solution you use to store and analyze these logs.

Since our application is intended to run locally, we will select JSON as a logging format and Serilog as a logging mechanism.

Why JSON?

JSON-structured logs are easy to read and can easily be parsed by machines. Many modern log management solutions can ingest logs in JSON format, making it a good choice for structured logging.

In addition, JSON is less verbose than XML, which results in files that are lightweight in size, which, in turn, reduces the amount of storage we need to store them.

Why Serilog?

Serilog is a diagnostic logging library for .NET. It is built with powerful structured event data in mind and supports various "sinks," which are the destinations where log events can be written to. Examples of such sinks include files, the console, databases or log management tools (such as Elasticsearch, Application Insights, Datadog, and Splunk). It's easy to set up, has a clean API, and is portable.

One of the key features of Serilog is its ability to log structured data, which allows for more meaningful and queryable logs. It uses message templates, a simple DSL extending .NET format strings, which can capture properties along with the log event.

I do like the way the NuGet packages for Serilog are structured. First, there is the base package, which provides Serilog's functionalities in our code. Then, there are "sink" packages, one for each sink, and there are tons of them. We can even create our very own sink should we need it. I haven't found a usage for that as there are already sinks for almost everything you can think of…

Oh! And, by the way, the Serilog NuGet package has been downloaded (to this day) more than 1.24 billion times! That should mean something, right? 😉

Before we add Serilog to our CLI application, we need to modify our code to expose the `IServiceCollection` property so we can configure our services.

Accessing IServiceCollection

The first step we need to take is to add the `System.CommandLine.Hosting` NuGet package to our project. From the Visual Studio Code terminal, we can do so by typing this command:

```
dotnet add package System.CommandLine.Hosting --prerelease
```

Then, we need to update our instantiation of the `CommandLineBuilder` class as follows:

```
using System.CommandLine.Hosting;
using Microsoft.Extensions.Hosting;

var parser = new CommandLineBuilder(rootCommand)
             .UseHost(_ => Host.CreateDefaultBuilder(),
             host =>
             {
                 host.ConfigureServices(services =>
                 {

                 });
             })
             .UseDefaults()
             .Build();

return await parser.InvokeAsync(args);
```

We now have access to `IServiceCollection` and can thus add services to this collection and configure their behavior.

Adding Serilog to IServiceCollection

Adding Serilog to this collection will require the `Serilog.Extensions.Hosting` NuGet package. So, let's add it!

```
dotnet add package Serilog.Extensions.Hosting
```

This allows us to add Serilog to `IServiceCollection` by adding this line of code to the preceding code sample:

```
services.AddSerilog();
```

Adding (and configuring) the required Serilog sinks

As mentioned earlier, there are tons of sinks available with Serilog. However, since we are building a CLI application, we will only use two sinks: `Console` and `File`.

We also mentioned that Serilog is structured in such a way that each sink has its own NuGet package. We will then need to add the appropriate NuGet packages using these commands:

```
dotnet add package Serilog.Sinks.Console
dotnet add package Serilog.Sinks.File
```

We can start using Serilog at this point by configuring the sinks in the code. The updated code for the instantiation of the `CommandLineBuilder` class would then be as follows:

```
using Serilog;
using System.CommandLine.Hosting;
using Microsoft.Extensions.Hosting;
var parser = new CommandLineBuilder(rootCommand)
            .UseHost(_ => Host.CreateDefaultBuilder(),
            host =>
            {
                host.ConfigureServices(services =>
                {
                    services.AddSerilog((config) =>
                    {
                        config.MinimumLevel.Information();
                        config.WriteTo.Console();
                        config.WriteTo.File("logs/bookmarkr-.txt",
                        rollingInterval:RollingInterval.Day,
                        restrictedToMinimumLevel:Serilog.Events.
                        LogEventLevel.Error);
                        config.CreateLogger();
                    });
                });
```

```
            })
            .UseDefaults()
            .Build();

return await parser.InvokeAsync(args);
```

Let's take a closer look at the part of the code where Serilog is configured (i.e., the delegate function within the `AddSerilog` method):

1. This code acts on an instance of the `LoggerConfiguration` class that is used to configure the behavior of Serilog and its sinks.
2. We define the minimum log level as `Information`. This means that every event that is informational or above (such as warnings and errors) will be logged unless overridden by a specific sink.
3. We notice that the `File` sink has overridden the log level in a way that only errors or higher severity events (such as `Fatal`) are logged.
4. We can also notice that the `File` sink has defined the location of the files (the `logs` folder) and the naming convention for the log files (`bookmark-.txt`). The dash symbol in the filename is not a misspelling but rather intentional! It is there because Serilog will append a unique identifier to that filename. Since we defined our rolling interval to be on a daily basis, Serilog will create a new log file every day. Hence, our log folder will contain files that will be named `bookmark-20240705.txt`, `bookmark-20240706.txt`, and so on.
5. We notice that we are also explicitly telling Serilog to log to the console. This is because even though we added a reference to the `Serilog.Sinks.Console` NuGet package, we still need to tell Serilog to actually use this sink.
6. Finally, we call the `CreateLogger` method so that all this configuration is taken into account.

Configuring sinks in appsettings.json

Although configuring the behavior of Serilog and its sinks directly in the code works perfectly fine, it is less flexible.

What if we want to add a new sink or update the configuration of an existing one? Well, you guessed it, we need to update and redeploy the code.

Moving this configuration into a configuration file (such as `appsettings.json`) brings more flexibility to our application.

Let's see how we can do this!

We first need to add the `Serilog.Settings.Configuration` NuGet package to our application by typing this command into the Visual Studio Code terminal:

```
dotnet add package Serilog.Settings.Configuration
```

We then need to add an `appsettings.json` file to our application. This can easily be done by adding a new file to the project in Visual Studio Code or in your favorite code editor.

For the `appsettings.json` file to be deployed with our application, we need to ensure that its latest version is always copied to the output directory. This can be done in the `bookmarkr.csproj` file by adding this snippet just before the closing `</Project>` element:

```
<ItemGroup>
    <None Update="appsettings.json">
          <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
    </None>
</ItemGroup>
```

Now, let's move the configuration of Serilog and its sinks from the code to the `appsettings.json` configuration file. The content of this file will then be the following:

```
{
  "Serilog": {
    "Using": [ "Serilog.Sinks.Console", "Serilog.Sinks.File" ],
    "MinimumLevel": {
      "Default": "Information",
      "Override": {
        "Microsoft": "Warning",
        "System": "Warning"
      }
    },
    "WriteTo": [
      {
        "Name": "Console",
        "Args": {
          "outputTemplate": "[{Timestamp:HH:mm:ss} {Level:u3}]
          {Message:lj}{NewLine}{Exception}"
        }
      },
      {
        "Name": "File",
        "Args": {
          "path": "logs/log-.txt",
          "rollingInterval": "Day",
          "outputTemplate": "[{Timestamp:yyyy-MM-dd HH:mm:ss.fff zzz}
          {Level:u3}] {Message:lj}{NewLine}{Exception}"
```

```
        }
      }
    ],
    "Enrich": [ "FromLogContext", "WithMachineName", "WithThreadId" ]
  }
}
```

This file is easy to read and pretty self-explanatory. It describes the sinks to be used and their configuration and defines the minimum log level. There is one thing to pay attention to here, however! Notice that we have overridden the minimum log level for libraries of the `Microsoft` and `System` namespaces. This is because these libraries tend to be chatty, which may result in a large amount of log data that is not very useful. By setting their minimum log level to `Warning`, we can ensure to only capture relevant events such as warnings or errors.

The `Enrich` section is a new one, however. As the name suggests, it is there to enrich the log data with extra information such as the machine name and the thread ID. If your CLI application is intended to be executed on multiple computers within your organization, knowing on which machine you got the error helps you narrow the search area. If that application is intended to run in multiple instances on the same machine, the thread ID will let you know which instance logged that information. This can be helpful in a concurrent execution scenario.

Are we done?

Not exactly… We still need to update the code inside the `ConfigureServices` method.

Since all the configuration has been moved to the `appsettings.json` file, the code becomes simpler, as you can see here:

```
using Microsoft.Extensions.Configuration;

// Only the body of the ConfigureServices is shown here for clarity.
// To see the full version of the code, please refer to the GitHub
// repo of the book.

services.AddSerilog((config) =>
{
      var configuration = new ConfigurationBuilder()
              .SetBasePath(Directory.GetCurrentDirectory())
              .AddJsonFile("appsettings.json")
              .Build();
      config.ReadFrom.Configuration(configuration);
});
```

Configuration in code or in a file?

You may be wondering whether to configure your logger in the code or in a configuration file. That is a great question!

I personally rely on both: I configure my sinks in the code and the log level in the `appsettings.json` file. This way, I can ensure that my "base sinks" are always in use even though more may be added later.

This is a personal choice, however. You are, of course, welcome to use whatever works best for you.

Keep in mind that configuration in the code takes precedence over configuration in the configuration file.

Let's log something!

Finally! Up to this point, all we did was configure our logger and its sinks. Let's now see how this works!

Once everything is in place and well configured, logging information using Serilog is quite straightforward.

To illustrate this, let's take an example.

When we implemented the `import` command back in *Chapter 5*, if an existing bookmark was to be updated (because a bookmark with the same URL but a different name already existed in the list of bookmarks held by the application), we had no way to track the name of that conflicting bookmark before the update.

If this was a piece of critical information (for compliance reasons, for example), logging would come in handy.

We will revisit this functionality and implement logging to keep track of the name before and after the update, along with its URL and the date and time of the update.

The log format would then be as follows:

```
<date and time> | Bookmark updated | name changed from '<old name>' to
'<new name>' for URL '<Url>'
```

The first thing we will do is to create a new version of the `import` method for the `BookmarkService` class. This new version will take a bookmark as a parameter and check whether a bookmark with the same URL but with a different name already exists in the list of bookmarks held by the application. If this is actually the case, it replaces the existing bookmark with the new name and then returns an instance of the `BookmarkConflictModel` type, which contains the original and updated name and the URL. If no conflict is detected, the method simply adds the bookmark and returns `null`.

Here is the code for this method:

```
public BookmarkConflictModel? Import(Bookmark bookmark)
{
```

```
    var conflict = _bookmarks.FirstOrDefault(b => b.Url == bookmark.
    Url && b.Name != bookmark.Name);
    if(conflict is not null)
    {
        var conflictModel = new BookmarkConflictModel { OldName =
        conflict.Name, NewName = bookmark.Name, Url = bookmark.Url };
        conflict.Name = bookmark.Name; // this updates the name of the
                                       // bookmark.
         return conflictModel;
    }
    else
    {
        _bookmarks.Add(bookmark);
        return null;
    }
}
```

The code of the `BookmarkConflictModel` class is as follows:

```
public class BookmarkConflictModel
{
    public string? OldName { get; set; }
    public string? NewName { get; set; }
    public string? Url { get; set; }
}
```

Finally, the code of the handler method of the `Import` command is updated to process each bookmark read from the file such that if a conflict is detected, Serilog is used to keep track of it.

Here is the updated code:

```
static void OnImportCommand(FileInfo inputfile)
{
    string json = File.ReadAllText(inputfile.FullName);
    List<Bookmark> bookmarks = JsonSerializer.
    Deserialize<List<Bookmark>>(json) ?? new List<Bookmark>();

    foreach(var bookmark in bookmarks)
    {
        var conflict = service.Import(bookmark);
        if (conflict is not null)
        {
              Log.Information($"{DateTime.Now.ToString("yyyy-MM-
dd HH:mm:ss")} | Bookmark updated | name changed from '{conflict.
OldName}' to '{conflict.NewName}' for URL '{conflict.Url}'");
```

```
            }
        }
    }
```

Now, if we run this program, we can see that was logged in case of a conflict:

Figure 6.2 – Logging in case of a conflict

As we can see, this information was both logged to the console and to the file.

We are now able to use Serilog to log important information. But what happens if the application is closed or is terminated? In such a situation, we can rely on the `CloseAndFlush` method.

Closing and gracefully disposing of Serilog

When using Serilog, the `Log.CloseAndFlush` method is called to ensure that all pending log event messages are flushed out to the sinks and that the logging system is properly shut down. This is particularly important in applications that have a definite end to their life cycle, such as console applications or batch jobs, to make sure that no log entries are missed due to the application closing before the logs are fully written out.

When calling this method, two things happen:

- **Close**: This sends a signal to the logging subsystem to stop accepting new log events. This is important to prevent any new logs from being generated after we have decided to shut down the logging system.
- **Flush**: This ensures that all log events that have been captured and are currently buffered are written out to their respective sinks. Serilog may buffer events in memory for efficiency, and flushing ensures that these buffered events are not lost.

I recommend calling this method when exiting the application, either by shutting it down or by terminating it.

For that matter, I always create a method (which I call `FreeSerilogLoggerOnShutdown`) that will subscribe to two events:

- `AppDomain.CurrentDomain.ProcessExit`: This event is raised when the process is about to exit, allowing us to perform cleanup tasks or save data.
- `Console.CancelKeyPress`: This event is triggered when the user presses *Ctrl* + *C* or *Ctrl* + *Break*, terminating the currently running application.

In both cases, these subscription calls the same delegate method (which I call `ExecuteShutdownTasks`) that will call the `CloseAndFlush` method from Serilog.

Here is the code for these two methods:

```
static void FreeSerilogLoggerOnShutdown()
{
    // This event is raised when the process is about to exit,
    // allowing you to perform cleanup tasks or save data.
    AppDomain.CurrentDomain.ProcessExit += (s, e) =>
    ExecuteShutdownTasks();
    // This event is triggered when the user presses Ctrl+C or
    // Ctrl+Break. While it doesn't cover all shutdown scenarios, it's
    // useful for handling user-initiated terminations.
    Console.CancelKeyPress += (s, e) => ExecuteShutdownTasks();
}


// Code to execute before shutdown
static void ExecuteShutdownTasks()
{
    Console.WriteLine("Performing shutdown tasks...");
    // Perform cleanup tasks, save data, etc.
    Log.CloseAndFlush();
}
```

The call to the `FreeSerilogLoggerOnShutDown` method is the first instruction of the `Main` method of the `Program` class.

While this is not a book about Serilog (which, in my opinion, deserves a book on its own), in this section, we covered the basics, which is enough for the purpose of the book. If you want to find more information about Serilog, visit `https://serilog.net/`.

Summary

In this chapter, we improved our CLI application, *Bookmarkr*, by adding error handling and logging into the application.

With error handling, we implemented graceful degradation into our CLI application. This means that our application is now more fault-tolerant and will not crash abruptly should an unexpected event occur.

With logging, we can record application activities, errors, and exceptions so that they can be analyzed at a later point in time in order to understand the sequence of events that led up to that error or unexpected behavior. But logging also enables the monitoring of application health and performance over time.

In the upcoming chapter, we will see how to make our CLI application more interactive and user-friendly.

Your turn!

Following along with the provided code is a great way to learn through practice.

A better way is by challenging yourself to achieve tasks. Hence, I challenge you to improve the *Bookmarkr* application by adding the following features.

Task #1 – Handling errors for the Import command

If the input file cannot be accessed, or if its content cannot be deserialized, it is likely that the code will throw exceptions. Your mission is to identify what exceptions are likely to be thrown and handle them accordingly.

Task #2 – Logging errors to a file

In the previous task, the goal was to handle the exceptions. However, it might be useful to log the details of these exceptions to a file so we can review them later and use this information to improve the robustness of our application.

Your mission here is to use Serilog to log exception data on a daily rolling interval and store these log files in the `logs/errors` folder.

You are also asked to customize the output template so that logs contain the following information:

- Date and time of the event
- The name of the machine on which the event happened
- The type of event (warning, error, and so on)
- The exception's details, including its stack trace

Part 3: Advanced Topics in CLI Application Development

In this part, you will explore the world of interactive command-line applications, learning how to create engaging user experiences using libraries like `Spectre.Console`. You'll discover techniques for implementing rich prompts, colorful output, and interactive menus that enhance user interaction. Next, you'll delve into building modular and extensible CLI applications, focusing on architectural patterns that promote maintainability and scalability. This includes structuring your code and organizing your project into logical components. Finally, you'll learn about integrating external APIs and services into your CLI applications. By the end of this part, you'll have the skills to develop sophisticated CLI tools that can consume various external services and APIs.

This part has the following chapters:

- *Chapter 7, Interactive CLI Applications*
- *Chapter 8, Building Modular and Extensible CLI Applications*
- *Chapter 9, Working with External APIs and Services*

7
Interactive CLI Applications

So far, the interaction with our CLI application (*Bookmarkr*) has mostly been text-based, meaning that the application responds to a text input with a text output. In a sense, it was mostly a request-response kind of application.

Yes, we've also added some colors to the text output so that the user can easily and instantly know whether the request was processed successfully or it ended with errors or warnings.

However, even though CLI applications don't provide a **graphical user interface** (**GUI**), that doesn't mean they can't be fun to use! 😉

In this chapter, we'll learn how to enhance the output of our CLI application to make it more user-friendly. We'll learn how to add the following elements:

- Progress bars and checkmarks to let the user know of the progress of their request. This is especially useful for time-consuming, long-running operations (such as downloading or encoding a file).
- Lists of items to make it easier to pick a selection from a list of predefined items. Note that this list doesn't have to be static; it can dynamically adapt to the current context of the user (for example, their permissions) and the request (for example, the requested command and the values of its options).

By relying on such enhancements, not only are we making our CLI application more fun to use, but we're also making it more **interactive** as it can have a conversation with the user, keeping them updated on the progress of their operations or presenting them with a list of options to choose from that are tailored to their specific context.

Technical requirements

The code for this chapter can be found in the GitHub repository accompanying this book, `https://github.com/PacktPublishing/Building-CLI-Applications-with-C-Sharp-and-.NET/tree/main/Chapter07`.

Building interactive command-line applications

As you may have noticed – and I'm sure that you did 😉 – we've already created some helper methods to display text in different colors for different scenarios: green for success messages, yellow for warnings, and red for errors. The code for these helper methods can be found in the `Helper.cs` file.

We could certainly add more methods to this file to support other features, such as progress bars and drop-down lists. However, as I mentioned earlier, I trust that you're smart, so you won't want to reinvent the wheel and rather rely on an existing library that fits the job 😊.

Although multiple libraries may exist to fit this purpose, the one we'll be using in this chapter is `Spectre.Console`.

The `Spectre.Console` library is designed to enhance the creation of visually appealing console applications by going way beyond colored text. It also allows you to render trees, drop-down lists, tables, progress bars, and many other graphical elements.

This wide range of features allows us to create rich user interfaces within the console environment of our CLI applications.

For more details about `Spectre.Console`, you can visit its GitHub page at `https://github.com/spectreconsole/spectre.console`. In particular, I recommend checking out the two following repositories:

- `spectre.console`: The official repository of the project, which means it contains the code of the library
- `Examples`: This repository contains various examples of using the library

So, let's start by adding the `Spectre.Console` library to our application. This can be achieved with the following command:

```
dotnet add package Spectre.Console
```

Let's start by adding a new command that we'll call `interactive`. This command will be a child of the `root` command. Let's also add a handler for the command:

```
var interactiveCommand = new Command("interactive", "Manage bookmarks
interactively")
{
};
rootCommand.AddCommand(interactiveCommand);
interactiveCommand.SetHandler(OnInteractiveCommand);
```

Let's add the handler method for this new command:

```
static void OnInteractiveCommand()
{
}
```

Since we are using the `Spectre.Console` library, let's reference it:

```
using Spectre.Console;
```

Adding a FIGlet

A FIGlet is a way to generate large text banners using ordinary screen characters, in the form of ASCII art. Originally released in 1991, it became popular for creating eye-catching text in terminal sessions. So, it's perfect for making our CLI applications more user-friendly!

So, let's add a FIGlet to *Bookmarkr*!

We don't need to create the ASCII art ourselves since `Spectre.Console` already provides this capability. So, let's take advantage of it.

Let's update our command handler method, as follows:

```
static void OnInteractiveCommand()
{
    AnsiConsole.Write(new FigletText("Bookmarkr").Centered().
    Color(Color.SteelBlue));
}
```

The preceding code creates some FIGlet text, centers it, and displays it in a shade of blue.

Now, let's run the application and see what we've got so far:

```
dotnet run -- interactive
```

We should be presented with the following output:

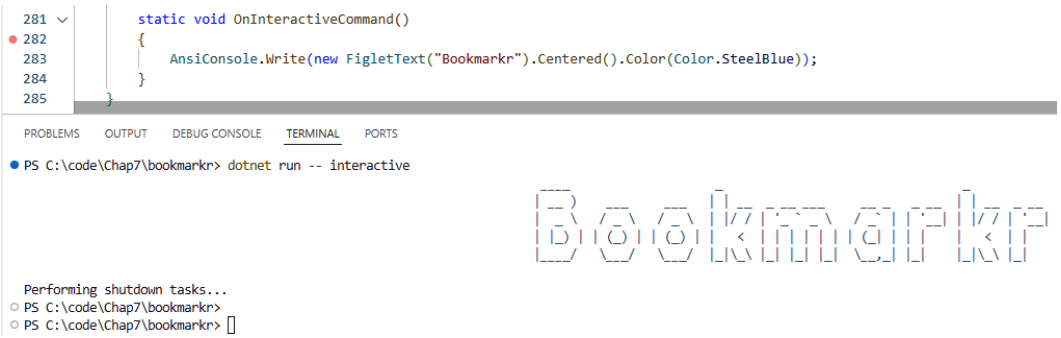


Figure 7.1 – A FIGlet for Bookmarkr

Wow! This is awesome. This is a great start for our interactive CLI application, isn't it? 😊

AnsiConsole versus Console

As you may have noticed, we aren't using the `Console.Write` method provided by .NET but the one provided by the `AnsiConsole` class, which is part of the `Spectre.Console` library. This is due to the following reasons:

1. **Enhanced functionality**: The `AnsiConsole` class provides a much richer set of features compared to the `Console` class, such as advanced text formatting, colors, styles, and interactive elements.

2. **Cross-platform compatibility**: The `AnsiConsole` class is designed to work consistently across different operating systems and terminal emulators. It automatically detects the capabilities of the current terminal and adjusts its output accordingly.

3. **Improved rendering capabilities**: The `AnsiConsole` class offers support for 24-bit colors, text styling (such as bold and italic), and various widgets, such as tables, trees, and even ASCII images.

4. **Interactive elements**: The `AnsiConsole` class enables the creation of interactive prompts, selection menus, and other user input mechanisms that are more sophisticated than what's possible with the standard `Console` class.

5. **Live rendering**: The `AnsiConsole` class supports live-rendering capabilities, allowing you to dynamically update content such as progress bars and status indicators.

6. **Markup language**: The `AnsiConsole` class provides a rich markup language that makes it easy to apply colors and styles to text without complex code.

7. **Abstraction for testing**: By using the `IAnsiConsole` interface instead of the static `AnsiConsole` class, it becomes possible to unit test command handlers and other console-related code.

At this point, we've laid the foundations for an interactive version of our CLI application. In the next section, we'll add more features to make *Bookmarkr* even more user-friendly!

Designing user-friendly CLI applications

Although tons of features are provided by `Spectre.Console` (and I encourage you to check them out by visiting the documentation at `https://spectreconsole.net`), we'll be focusing on a subset of these capabilities to add more interactivity to our *Bookmarkr* application.

Enhancing text display using markup

Let's improve our `Helper.cs` class by leveraging the markup capabilities provided by `Spectre.Console`. These capabilities allow us to style and format text and even render emojis! However, note that some terminal environments (especially older systems or restricted environments, such as CI/CD systems) might not support emoji rendering.

First, let's review our three methods: `ShowErrorMessage`, `ShowWarningMessage`, and `ShowSuccessMessage`.

These methods differ in the color in which they display the text that's received as a parameter, but they're similar in the way that they all follow the same process:

- First, they save the current foreground color to a temporary variable.
- Then, they change the foreground color to the intended color (red, yellow, or green).
- Next, they display the received text using that color.
- Finally, they restore the foreground color to the saved color.

This is made easier by the `Spectre.Console` library. We don't have to save and restore the current foreground color (the library does this for us). It also opens some new possibilities. Let's explore some while redefining these methods!

The ShowErrorMessage method

We still want the error message to be displayed in red, but we also want to use emojis to make it more eye-catching.

The code looks like this:

```
public static void ShowErrorMessage(string[] errorMessages)
{
    Console.OutputEncoding = System.Text.Encoding.UTF8;
    AnsiConsole.MarkupLine(
        Emoji.Known.CrossMark + " [bold red]ERROR[/] :cross_mark:");
        foreach(var message in errorMessages)
    {
        AnsiConsole.MarkupLineInterpolated($"[red]{message}[/]");
    }
}
```

There are many things worth mentioning in this code sample:

- First, we set the encoding to UTF-8. This ensures that emojis are rendered properly. Otherwise, they'll be replaced by question marks.
- Speaking about emojis, notice that we can display them in two ways: we can either use the `Known` enumeration under the `Emoji` class or use markup code (here, we used `:cross_mark:`). The full list of supported emojis, along with their markup codes and enumeration constants, can be found at `https://spectreconsole.net/appendix/emojis`.

- We use `AnsiConsole.MarkupLine(...)` to display the word *ERROR* in bold and red surrounded by cross marks. The syntax is based on `BBCode` (`https://en.wikipedia.org/wiki/BBCode`).
- Since the `ShowErrorMessage` method receives an array of strings, we display each string in red. However, this time, we rely on the `MarkupLineInterpolated` method of the `AnsiConsole` class because we're performing string interpolation.

Let's see what happens if we call the `ShowErrorMessage` method:

```
212     /***** HANDLER METHODS ******************/
213     static void OnHandleRootCommand()
214     {
215         Helper.ShowErrorMessage(["This is an example of an error message.",
216                                  "The detail of the error, along with the exception message might go here."]);
217         Console.WriteLine("Hello !!");
218     }
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\code\Chap7\bookmarkr> dotnet run
 ✖ ERROR ✖
This is an example of an error message.
The detail of the error, along with the exception message might go here.
Hello !!
Performing shutdown tasks...
PS C:\code\Chap7\bookmarkr>
```

Figure 7.2 – The updated ShowErrorMessage method in action

Notice that the *Hello!* message isn't displayed in red but rather in the previous foreground color of the terminal, without us having to handle this.

Now, let's update the `ShowWarningMessage` and `ShowSuccessMessage` methods.

The ShowWarningMessage method

We still want to display the warning message in yellow, but let's also align it so that it's in the center of the screen (and, yes, we'll use emojis too 😊).

The updated code looks like this:

```
public static void ShowWarningMessage(string[] errorMessages)
{
    Console.OutputEncoding = System.Text.Encoding.UTF8;
    var m = new Markup(
        Emoji.Known.Warning + " [bold yellow]Warning[/] :warning:"
    );
    m.Centered();
    AnsiConsole.Write(m);
    AnsiConsole.WriteLine();
    foreach(var message in errorMessages)
    {
```

```
            AnsiConsole.MarkupLineInterpolated(
                $"[yellow]{message}[/]"
            );
        }
    }
```

This code is very similar to the code for `ShowErrorMessage`. However, there's a slight difference: instead of calling the `MarkupLine` method to display the *Warning* text surrounded by emojis, we instantiate an object of the `Markup` type, then call its `Centered()` method before passing it as a parameter to the `Write` method. This is required so that we can center that text on the screen.

Also, notice that just after that, we call the `WriteLine` method without any parameter. This ensures that a line break is performed before the warning information is displayed.

Let's see what happens if we call the `ShowWarningMessage` method:

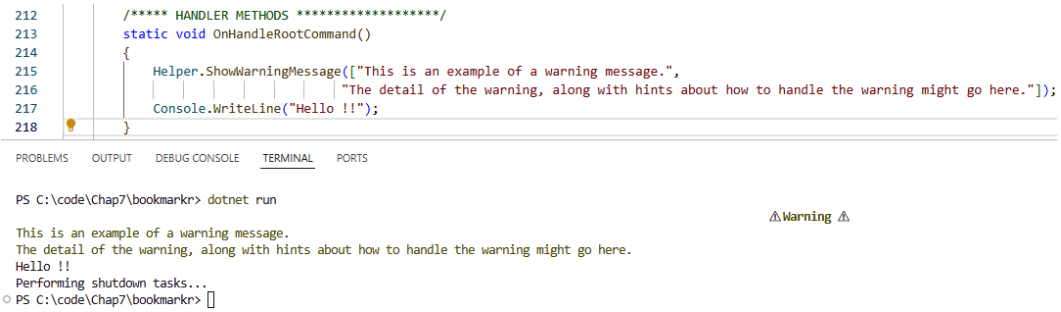


Figure 7.3 – The updated ShowWarningMessage method in action

Once again, notice that the *Hello!* message isn't displayed in red but rather in the previous foreground color of the terminal, without us having to handle this.

The ShowSuccessMessage method

We still want to display the success message in green.

The updated code looks like this:

```
public static void ShowSuccessMessage(string[] errorMessages)
{
    Console.OutputEncoding = System.Text.Encoding.UTF8;
    AnsiConsole.MarkupLine(Emoji.Known.BeatingHeart + " [bold green]
    SUCCESS[/] :beating_heart:");
    foreach(var message in errorMessages)
    {
        AnsiConsole.MarkupLineInterpolated($"[green]{message}[/]");
    }
}
```

There's absolutely nothing worth mentioning here, except maybe that we decided to use heart emojis to celebrate the success of the operation! 😊 Congratulations – you're starting to master the skills of styling and formatting text using `Spectre.Console`!

Let's see what happens if we call the `ShowSuccessMessage` method:

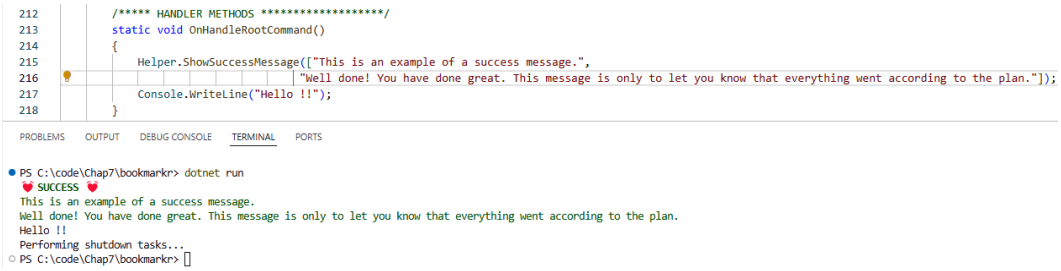


Figure 7.4 – The updated ShowSuccessMessage method in action

Once again, notice that the *Hello!* message isn't displayed in red but rather in the previous foreground color of the terminal, without us having to handle this.

Now that we can make our text more eye-catching, let's improve our interactive command by adding more features.

Offering choices to the user using selection prompts

Showing text in a visually pleasant manner is important, but interacting with the user and getting their input is equally important. Fortunately, `Spectre.Console` provides many ways to interact with the user. Let's explore one of these.

For now, our interactive command only shows a FIGlet. It's certainly great, but let's make it more valuable to the user. One thing we can add is a list of all operations a user can perform. To do so, we will use a **selection prompt**. This will allow the user to navigate the list using the up and down arrows on their keyboard and confirm the operation to be executed.

Here's what it looks like in action:

Figure 7.5 – The selection prompt in action

Here's the updated code of the `OnInteractiveCommand` method:

```
static void OnInteractiveCommand()
{
    bool isRunning = true;
    while(isRunning)
    {
        AnsiConsole.Write(
            new FigletText("Bookmarkr")
                .Centered()
                .Color(Color.SteelBlue)
        );

        var selectedOperation = AnsiConsole.Prompt(
            new SelectionPrompt<string>()
                .Title("[blue]What do you wanna do?[/]")
                .AddChoices([
                    "Export bookmarks to file",
                    "View Bookmarks",
                    "Exit Program"
            ])
        );

        switch(selectedOperation)
        {
            case "Export bookmarks to file":
                ExportBookmarks();
                break;
            case "View Bookmarks":
                ViewBookmarks();
                break;
            default:
                isRunning = false;
                break;
        }
    }
}
```

Let's take a closer look at this code:

- We want the user to be able to keep picking up operations until they decide to exit the program (by selecting the *Exit Program* option from the selection prompt). That's why we placed our code inside a `while` loop that relies on the value of a Boolean variable that's initially set to `true`. When the user selects *Exit Program*, the Boolean variable is set to `false` and the program is exited. Otherwise, the selection prompt is displayed to the user after each completed operation.

- We instantiate the `SelectionPrompt` class (provided by `Spectre.Console`), specifying a title and a list of available options that will be displayed to the user. The user can then navigate this list using the up and down arrows and confirm their selection by hitting the *Enter* key. When they do, the `selectedOperation` variable is set to the selected value.
- Finally, we use a `switch` statement to call a specific method, depending on the value of the `selectedOperation` variable. That method will process the requested operation. Note that these methods haven't been implemented yet – we'll focus on them in the upcoming sections.

Multi-selection

The `Spectre.Console` library also provides multi-selection prompts, which allow the user to select more than one item from a list of possible choices. You can find an illustrative example, along with a code sample, at `https://spectreconsole.net/prompts/multiselection`.

Showing live progress of the export command

The previous iteration of our export command was a little bit… dry! It certainly exported the bookmarks to the specified output file as expected, but it didn't give us information on the progress of the operation. We couldn't know whether it was halfway done or whether it was still at the very first few items out of a thousand. Thanks to the capabilities of `Spectre.Console`, we can display live progress along the way.

There's both a synchronous and an asynchronous method for showing live progress. Since our export code is asynchronous, we'll use the latter.

So, let's implement the `ExportBookmarks` method we saw in the previous section. Here's the code:

```
static void ExportBookmarks()
{
    // ask for the outputfilePath
    var outputfilePath = AnsiConsole.Prompt(
          new TextPrompt<string>("Please provide the output file name
          (default: 'bookmarks.json')")
          .DefaultValue("bookmarks.json"));

    // export the bookmarks to the specified file, while showing
    // progress.
    AnsiConsole.Progress()
        .AutoRefresh(true) // Turns on auto refresh
        .AutoClear(false)   // Avoids removing the task list when
                            // completed
        .HideCompleted(false)   // Avoids hiding tasks as they are
```

```
                            // completed
        .Columns(
        [
            new TaskDescriptionColumn(),    // Shows the task
                                            // description
            new ProgressBarColumn(),        // Shows the progress bar
            new PercentageColumn(),         // Shows the current
                                            // percentage
            new RemainingTimeColumn(),      // Shows the remaining
                                            // time
            new SpinnerColumn(),            // Shows the spinner,
                                            // indicating that the
                                            // operation is ongoing
        ])
        .Start(ctx =>
        {
            // Get the list of all bookmarks
            var bookmarks = service.GetAll();

            // export the bookmarks to the file
            // 1. Create the task
            var task = ctx.AddTask("[yellow]exporting all bookmarks to
            file...[/]");

            // 2. Set the total steps for the progress bar
            task.MaxValue = bookmarks.Count;

            // 3. Open the file for writing
            using (StreamWriter writer = new
            StreamWriter(outputfilePath))
            {
                while (!ctx.IsFinished)
                {
                    foreach (var bookmark in bookmarks)
                    {
                        // 3.1. Serialize the current bookmark as JSON
                        //and write it to the file asynchronously
                        writer.WriteLine(JsonSerializer.
                        Serialize(bookmark));

                        // 3.2. Increment the progress bar
                        task.Increment(1);
```

```
                        // 3.3. Slow down the process so we can see
                        // the progress (since this operation is not
                        // that much time-consuming)
                        Thread.Sleep(1500);
                    }
                }
            }
        });
    AnsiConsole.MarkupLine("[green]All bookmarks have been
    successfully exported![/]");
}
```

Let's take a closer look at this code:

1. First, we ask the user for the filename. We achieved this by using `AnsiConsole.Prompt` and passing an instance of `TextPrompt`. This class also allows us to specify a default value if no value has been provided by the user or if they're fine with it.
2. Next, we display the progress bar by calling `AnsiConsole.Progress`. The processing of the export operation is implemented as a delegate within the `Start` method:
    A. We start by retrieving the list of bookmarks to be exported.
    B. Next, we create an instance of the `Task` class, which will be responsible for tracking the processing of the export operation and incrementing the progress bar's percentage. We also set the maximum value for that task to the number of bookmarks to be exported.
    C. Then, we open a stream writer to the export file. While the task hasn't been completed (indicated by the `ctx.IsFinished` Boolean value), we serialize each bookmark as JSON and write it to the file, increment the task (which, in turn, will update the progress bar), and wait 1.5s (this is optional, but since the export operation isn't that time-consuming, adding a delay allows us to see the animation of the progress bar 😊).
3. You may have noticed some lines of code before the `Start` method is called. These are here to configure the look and behavior of the progress bar:
    - `AutoRefresh(true)`: We enable auto-refresh of the progress bar. Otherwise, even if we increment the value of the task, the progress bar won't be animated to reflect the updated value.
    - `AutoClear(false)`: This prevents the task from being removed once completed. This is especially useful if we display the progress of multiple, concurrent operations.

- `HideCompleted(false)`: This prevents the task from being hidden once completed. This is especially useful if we display the progress of multiple, concurrent operations.
- `Columns`: This collection controls the look of the progress bar. In this case, we decided to display the description of the task, the progress bar, the current percentage of the operation, the remaining time to complete the operation, and a spinner indicating that the operation is ongoing (this proves helpful in situations where dealing with time-consuming operations for which the progress bar may be updated at longer intervals but since the spinner keeps, well… spinning, the user has that sentiment that the operation is still ongoing).

Now that the code has been implemented, let's run the program and see what we get:

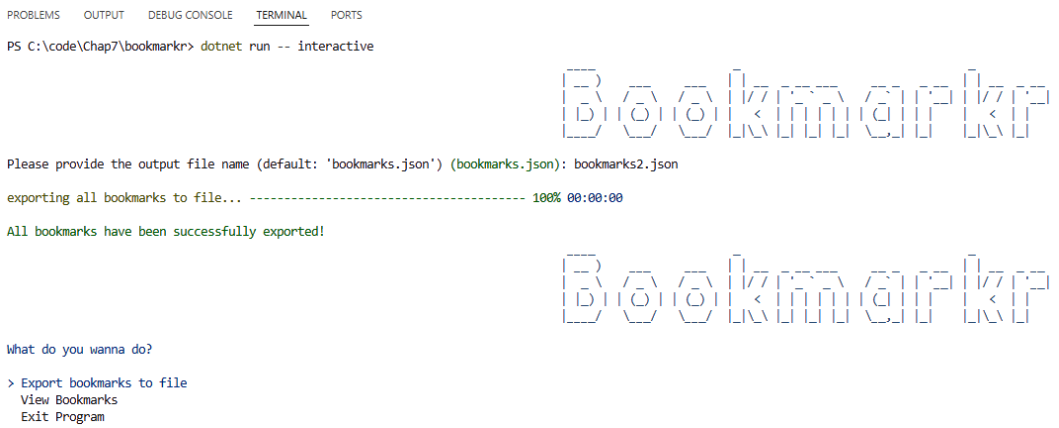


Figure 7.6 – Showing live progress while exporting bookmarks

Now, our export operation is more user-friendly (and, let's face it, more fun to use 😊). The user is informed of the progress of the export operation as it continues to run.

> **Important note**
>
> If the terminal isn't considered interactive (for example, when running in a continuous integration system), any progress will be displayed in a simpler way (such as the current percentage value being displayed on a new line).

With that, we've exported our bookmarks to a file and we can view them from there. Awesome! But what if we wanted to view them right from within the CLI application? Let's see how we can have a visually pleasant representation of these bookmarks.

Displaying bookmarks in a tree view

The `Spectre.Console` library provides various options to display a list of elements. We can use tables, trees, layouts, panels, grids, and so on.

Since we want to display the list of our bookmarks based on the categories to which they belong, we'll use a tree representation. So, let's implement the `ViewBookmarks` method.

Here's the code for this method:

```
static void ViewBookmarks()
{
    // Create the tree
    var root = new Tree("Bookmarks");

    // Add some nodes
    var techBooksCategory = root.AddNode("[yellow]Tech Books[/]");
    var carsCategory = root.AddNode("[yellow]Cars[/]");
    var socialMediaCategory = root.AddNode("[yellow]Social Media[/]");
    var cookingCategory = root.AddNode("[yellow]Cooking[/]");

    // add bookmarks for the Tech Book category
    var techBooks = service.GetBookmarksByCategory("Tech Books");
    foreach(var techbook in techBooks)
    {
        techBooksCategory.AddNode($"{techbook.Name} | {techbook.
        Url}");
    }

    // ... do the same for the other categories ;)

    // Render the tree
    AnsiConsole.Write(root);
}
```

Let's explain this code:

1. First, we create the root element of the tree, and we label it *Bookmarks*. This label will be displayed to indicate what elements this tree is presenting.
2. Next, we add four nodes (one for each category). These labels are children of the root node, and their labels are displayed in yellow.
3. Then, we call the `GetBookmarksByCategory` method of `BookmarkService` to retrieve the list of bookmarks associated with the specified category (in this case, *Tech Books*). After, we iterate over this list and add each bookmark as a child node to the `techBooksCategory` node.

4. We do the same thing for the other categories. The preceding code sample has been simplified and only shows the code for the *Tech Books* category for clarity. However, the complete code can be found in this book's GitHub repository.
5. Finally, we display the label in the console.

Pretty straightforward, isn't it?

Now, let's run the program and see how it looks:

Figure 7.7 – Showing bookmarks as a tree view, grouped by categories

Wow. What an improvement! The interactive version of our CLI application looks great and is more user-friendly and engaging, don't you think? 😉

To be or not to be interactive?

Providing an interactive version of a CLI application certainly enhances the user experience. So, shouldn't we always provide it? Shouldn't it be the default version? These are great questions. Thanks for asking! 😊

Let's start with the first one: **shouldn't we always provide it?** We certainly should as it (as mentioned previously) improves the user experience and engagement.

Shouldn't it be the default version? It could! It depends on your target audience:

- If your CLI application is primarily intended to be run by humans, then yes! You should make the interactive version the default one. In this case, your commands could provide a `--non-interactive` option to disable this interactive behavior when that command is executed by a program (in a CI/CD pipeline, for example).

- If your CLI application is primarily intended to be run by programs (such as one for processing a large batch of files or a CI/CD pipeline), your commands could provide a `--interactive` option to enable interactive behavior when they're executed by humans.

In other words, interactivity is fantastic, but use it wisely!

Summary

In this chapter, we made *Bookmarkr* more user-friendly and graphically appealing by introducing progress bars, checkmarks, and lists of items for easier user selection and ensuring that only valid values are selected.

We learned that these additions, in combination with text coloring, help make our CLI applications more compelling and fun to use and showed that CLI applications have nothing to be ashamed of in front of GUI applications.

But that's not all! We also learned that these additions help make our CLI applications more conversational (that is, interactive) with the user.

In the next chapter, we'll learn how to make our CLI application have a more modular design so that it's easier to extend.

Your turn!

Following along with the provided code is a great way to learn through practice.

A better way is by challenging yourself to achieve tasks. Hence, I challenge you to improve the *Bookmarkr* application by adding various features.

Task 1 – present a bookmark in a user-friendly way

You've been asked to add a `show` command that takes the name of a bookmark and displays it in a three-column grid – one for the name, one for the URL, and one for the category.

The grid should contain a row for the headers (name, URL, and category).

The name should be displayed in yellow and bold; the URL should be presented as a link; and the category should be presented in italics and green.

The syntax of the command should be as follows:

```
bookmarkr link show --name <name of the bookmark>
```

Task 2 – change the category of a bookmark interactively

You've been asked to implement a new command called `category change` that changes the category of an existing URL.

The command must display the list of existing categories as a selection menu; the user will have to select which one will be set as the new category for that bookmark based on its URL. This update will then be saved to the database.

The syntax of the command should be as follows:

```
bookmarkr category change --for-url <url of the bookmark>
```

8

Building Modular and Extensible CLI Applications

Throughout the pages of this book, we have added more and more functionalities to **Bookmarkr**, our beloved CLI application.

The problem is that we have also added more and more lines of code to the `Program.cs` file. The length of this file has grown from 191 lines of code by the end of *Chapter 3* to 479 lines of code by the end of *Chapter 7*.

In this chapter, we will take a step back and refactor our code to make it more modular. This will make it easier to extend, test, and maintain.

Refactoring is an essential part of the development life cycle. It should happen periodically in order to ensure that the quality of code is up to the standards.

By taking this necessary step, we will greatly simplify adding more features, enhance the readability and stability of our application, and even introduce testability into it.

More specifically, in this chapter, we'll cover the following topics:

- Building the code map of the current application
- Deciding where to start refactoring
- Designing the project structure to support refactoring
- Refactoring a command
- Applying the dependency inversion principle
- Refactoring the `Program` class
- Running the program to validate that the refactoring didn't break anything
- Pushing our refactoring to new boundaries

Technical requirements

The code for this chapter can be found in the GitHub repository accompanying this book, `https://github.com/PacktPublishing/Building-CLI-Applications-with-C-Sharp-and-.NET/tree/main/Chapter08`.

Step 1 – building a code map of the application

The first thing when you refactor an application is to get to know what you are about to refactor. This means having a high-level view of the application and its dependencies. This helps you visualize all the moving parts involved and better plan your refactoring activities by identifying where to start.

If you have Visual Studio Enterprise edition, you can use its great architecture capabilities (such as code maps and dependency graphs) to visualize your code and its dependencies. However, since we are using Visual Studio Code (or if you don't have the Enterprise edition of Visual Studio), we can do something else…

Sure, we can run through the code and identify every moving part of it, but since this is a CLI application, let's do something smarter. 😉

Using the Help menu to build the code map

The **Help** menu that comes out of the box with `System.CommandLine` is certainly great for learning how to use the application, but it is also great for figuring out the code map of the application.

Let's start by displaying the help menu at the root command by typing this command:

```
dotnet run -- -h
```

We will get this result:

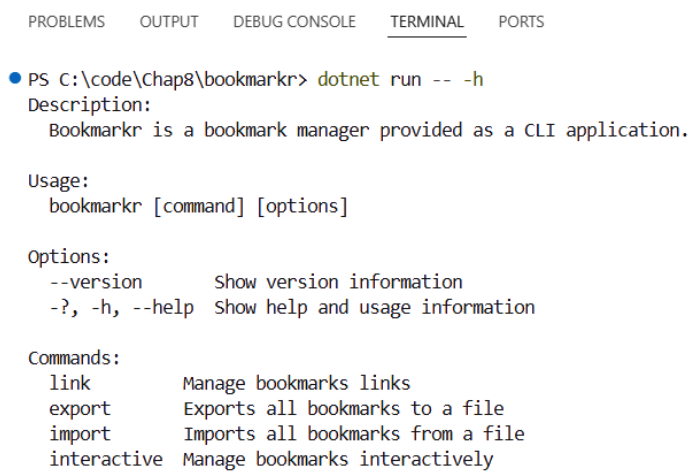


Figure 8.1 – The help menu of the root command

Next, we will repeat this operation (aka displaying the help menu) for each of the subcommands of the root command, then for each subcommand of every subcommand, then for… Okay, you get the idea! 😉

Here is an example of the `link` command:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

PS C:\code\Chap8\bookmarkr> dotnet run -- link -h
Description:
  Manage bookmarks links

Usage:
  bookmarkr link [command] [options]

Options:
  -?, -h, --help  Show help and usage information

Commands:
  add  Add a new bookmark link
```

Figure 8.2 – The help menu of the link command

When should we stop? Well, when the current command has no more subcommands. Here is an example with the `link add` command:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

PS C:\code\Chap8\bookmarkr> dotnet run -- link add -h
Description:
  Add a new bookmark link

Usage:
  bookmarkr link add [options]

Options:
  -n, --name <name> (REQUIRED)                                  The name of the bookmark
  -u, --url <url> (REQUIRED)                                    The URL of the bookmark
  -c, --category <Cooking|Read later|Social media|Tech books>   The category to which the bookmark is associated [default: Read later]
  -?, -h, --help                                                Show help and usage information
```

Figure 8.3 – The help menu of the link add command

After completing this exercise, we will get the following code map:

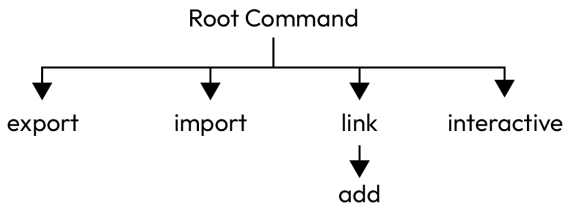


Figure 8.4 – The code map of the Bookmarkr application

Okay, now that we have a clearer view of the moving parts in our application, what should we do next?

Step 2 – deciding where to start

It's now time to decide what to refactor first.

I would recommend not to start with the root command first, but rather with the subcommands of that root command.

There is no right or wrong decision from there. You can pick up any subcommand you would like to start with. We will take the `export` command as an example in the remainder of this chapter.

Although the `export` command does not have a subcommand, it will still help us lay out the foundation of the refactored version of *Bookmarkr*. More specifically, it will help us do the following:

- Define the project structure to support our refactoring
- Refactor it and hide its "complexity" (aka moving parts to the root command)
- Refactor the `Program` class and make it leaner, cleaner, and more concise
- Set up dependency injection for interacting between commands and external services (such as `BookmarkService`)

Let's start with designing the project structure that will support our refactoring activities.

Step 3 – designing the project structure

Although one can design their project structure according to their tastes, I design mine in a way that makes it easy for whoever looks at my project to understand what it does and where each moving part is located.

Following this principle, all commands will be grouped in a folder named `Commands`. This folder will be created at the root of the project structure.

Since we will be refactoring the `export` command, let's create a subfolder named `Export` where all the code artifacts involved in the `export` command will be located.

Once we start refactoring another command, we will create a specific folder for it.

> **What about subcommands of a command?**
>
> Following the principle of **encapsulation** in object-oriented programming, and since a subcommand can only be invoked through its parent command, I recommend locating subcommands in the same folder as their parent command.
>
> An example of that is the `link add` command. The `add` subcommand can only be called through its parent (`link`) command. Hence, their life cycles are closely related to each other.
>
> For that matter, the code artifact of the `add` command will be located close to the code artifact of its parent command (`link`), within the `Link` folder.

Step 4 – refactoring the export command

Within the `Export` folder, let's create a new C# file named `ExportCommand.cs`.

Every command class (including `RootCommand`) derives from the `Command` base class. Furthermore, that base class provides an `AddCommand` method that takes a parameter of the `Command` type, which also means any class that derives from the `Command` class.

Armed with this, we can start refactoring the `export` command by making the `ExportCommand` class derive from `Command`.

After importing the required `using` statement, specifying the `namespace` name, and adding the required class constructor, the first iteration of our class looks like this:

```
using System.CommandLine;
namespace bookmarkr.Commands;

public class ExportCommand : Command
{
    #region Constructor

    public ExportCommand(string name, string? description = null)
        : base(name, description)
    {
    }

    #endregion
}
```

The first things to move into this class are the options. The `export` command has only one option, `outputfileOption`.

I like the fact that every component of my class is well-segmented. That's why I am a fan of regions. For that reason, let's add a region dedicated to options and move the code for the `outputfileOption` option within this region.

The code looks like this:

```
#region Options

private Option<FileInfo> outputfileOption = new Option<FileInfo>(
    ["--file", "-f"],
    "The output file that will store the bookmarks"
)
{
    IsRequired = true
```

```
}.LegalFileNamesOnly();

#endregion
```

We then need to associate this option with the command. We will do this by calling the `AddOption` method from within the constructor body, as follows:

```
#region Constructor

public ExportCommand(string name, string? description = null)
    : base(name, description)
{
    AddOption(outputfileOption);
}

#endregion
```

The next thing to move is the call to the `SetHandler` method, which connects the command to its handler method. So, the updated version of the constructor looks like this:

```
#region Constructor

public ExportCommand(string name, string? description = null)
    : base(name, description)
{
    AddOption(outputfileOption);
    this.SetHandler(async (context) =>
    {
        FileInfo? outputfileOptionValue = context.ParseResult.
        GetValueForOption(outputfileOption);
        var token = context.GetCancellationToken();
        await OnExportCommand(outputfileOptionValue!, token);
    });
}

#endregion
```

Finally, the last piece of code to move into the new class is the command handler method. Once again, we will create a new region for that and move that final piece of code. We will also change the `static` modifier to `private`. The reason is that the class is not static (hence the removal of the `static` keyword) and the command handler method is private to that class (hence the use of the `private` keyword):

```
#region Handler method

private async Task OnExportCommand(FileInfo outputfile,
```

```
    CancellationToken token)
{
    // method body removed for brevity.
    // It is exactly similar to the one from the previous chapters.
}

#endregion
```

If you are typing (or copying and pasting 😉) the code along the way, you can see at this point that the code does not compile because of two errors.

The first one is straightforward to solve. It suffices to add the following statement at the top of the C# file:

```
using System.Text.Json;
```

The second one is less obvious to solve. It indicates that the class can't find an instance of the `BookmarkService` class.

Of course, we could simply create an instance of that service within the current class. However, since `BookmarkService` is an external dependency to the `ExportCommand` class, doing so will break the principle of **dependency inversion** advocated by object-oriented programming.

As a reminder, the principle of dependency inversion is one of the five SOLID principles of object-oriented programming and design. It states that high-level modules should not depend on low-level modules; both should depend on abstractions. Additionally, abstractions should not depend on details; details should depend on abstractions. This principle helps to decouple software modules, making the system more modular, flexible, and easier to maintain.

What does that practically mean? It means that we should inject an instance of `BookmarkService` into the `ExportCommand` class.

Let's do this!

Step 5 – applying the dependency inversion principle

If you are familiar with the dependency inversion principle, you will certainly have already noticed that the `BookmarkService` class does not implement any interface.

Let's start by fixing this.

I am not familiar with the dependency inversion principle!

If you are not, there are plenty of great resources to explore this principle. It is not a complicated principle to understand, and quite frankly, after you learn about it, it will seem so obvious to you that you will be wondering why you did not know about it earlier.

A great explanation of that principle can be found at `https://www.c-sharpcorner.com/article/dependency-inversion-principle-in-c-sharp/`.

I highly encourage you to go and review the implementation of the `BookmarkService` service before and after applying the dependency inversion principle in order to have a clear understanding of the benefits.

Back to our project structure discussion, we will start by creating a new folder called `Services` that will group all our services classes. Within that folder, let's create a specific folder for every service. In our case, we only have one service, so let's create the `BookmarkService` folder. This folder will contain both the interface and the concrete implementation of our service.

The folder structure for our service will then look like this:

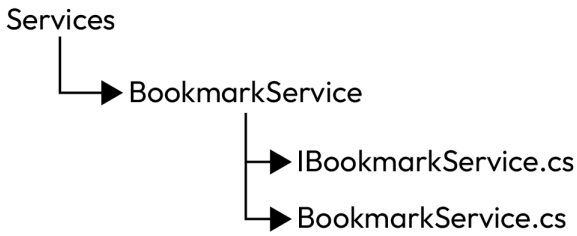


Figure 8.5 – The folder structure for the BookmarkService service

Next, let's extract the `IBookmarkService` interface out of the `BookmarkService` class. The code for that interface looks like this:

```
namespace bookmarkr.Services;

public interface IBookmarkService
{
    void AddLink(string name, string url, string category);

    void AddLinks(string[] names, string[] urls, string[] categories);

    void ListAll();

    List<Bookmark> GetAll();

    void Import(List<Bookmark> bookmarks);
```

```
    BookmarkConflictModel? Import(Bookmark bookmark);

    List<Bookmark> GetBookmarksByCategory(string category);
}
```

Now, let's make the `BookmarkService` class implement the `IBookmarkService` interface:

```
namespace bookmarkr.Services;

public class BookmarkService : IBookmarkService
{
    // method body removed for brevity.
    // It is exactly similar to the one from the previous chapters.
}
```

Notice that we changed the namespace name for these artifacts in order to better convey their intention.

All that is left to do now is to inject that service into the `ExportCommand` class. This means two things:

1. We will add a `private` property of the `IBookmarkService` type in the `ExportCommand` class that will allow us to invoke the methods of that service from within the command class (more specifically, from within the `OnExportCommand` method).
2. We will inject an instance of that service through a constructor parameter.

The updated code of the `ExportCommand` class looks like this now:

```
using System.CommandLine;
using System.Text.Json;
using bookmarkr.Services;

namespace bookmarkr.Commands;

public class ExportCommand : Command
{
    #region Properties
    private IBookmarkService _service;
    #endregion

    #region Constructor

    public ExportCommand(IBookmarkService service, string name,
    string? description = null)
        : base(name, description)
    {
```

```
        _service = service;

        AddOption(outputfileOption);
        this.SetHandler(async (context) =>
        {
            FileInfo? outputfileOptionValue = context.ParseResult.
            GetValueForOption(outputfileOption);
            var token = context.GetCancellationToken();
            await OnExportCommand(outputfileOptionValue!, token);
        });
    }
    #endregion

  // The "Options" region hasn't changed and removed for brevity.

#region Handler method

    private async Task OnExportCommand(FileInfo outputfile,
    CancellationToken token)
    {
    // …
    var bookmarks = _service.GetAll();
    // …
    }
    #endregion
}
```

This code is very straightforward to understand and does not require any particular explanation.

> **The dependency inversion principle may introduce complexity!**
>
> Dependency injection can introduce some overhead in simple applications by requiring additional setup and configuration, such as adding extra interfaces, classes, and indirection, which can be unnecessary for straightforward projects with few dependencies.
>
> Therefore, it is important to find a balance between applying this principle without adding too much complexity to the code base.

That is wonderful. We have come a long way since we started our refactoring journey!

There is one last piece of code we haven't refactored yet. It is the one for which we started this journey in the first place: the `Program` class.

Let's turn our attention to this class now…

Step 6 – refactoring the Program class

By refactoring the commands into their dedicated classes, the code to create and handle these commands will be removed from the `Program` class.

Hence, the `Program` class will now only be used to compose our application. More specifically, the `Program` class will do the following:

1. Instantiate the root command and register its subcommands.
2. Instantiate and configure the `CommandLineBuilder` class and start the program.
3. Configure logging.
4. Configure dependency injection of the `BookmarkService` service.

Here is the refactored code for the `Program` class (note that some parts of the code, including `using` statements, are not listed here for brevity and clarity):

```
using Microsoft.Extensions.DependencyInjection;

class Program
{
    static async Task<int> Main(string[] args)
    {
        FreeSerilogLoggerOnShutdown();

        /** DECLARE A VARIABLE FOR THE IBookmarkService **/
        IBookmarkService _service;

        /** INSTANTIATE THE ROOT COMMAND **/
        var rootCommand = new RootCommand(
            "Bookmarkr is a bookmark manager provided as a CLI
            application.")
        {
        };
        rootCommand.SetHandler(OnHandleRootCommand);

        /** CONFIGURE DEPENDENCY INJECTION FOR THE IBookmarkService
        **/
        var host = Host.CreateDefaultBuilder(args)
            .ConfigureServices((hostContext, services) =>
            {
                // Register your services here
                services.AddSingleton<IBookmarkService,
                BookmarkService>();
            })
```

```
            .Build();

        _service = host.Services.GetRequiredService<IBookmarkService>();

        /** REGISTER SUBCOMMANDS OF THE ROOT COMMAND **/
        rootCommand.AddCommand(new ExportCommand(_service, "export",
        "Exports all bookmarks to a file"));

        /** THE BUILDER PATTERN **/
        // code removed for brevity.
    }

    /** HANDLER OF THE ROOT COMMAND **/
    static void OnHandleRootCommand()
    {
        Console.WriteLine("Hello from the root command!");
    }

    static void FreeSerilogLoggerOnShutdown()
    {
        // code removed for brevity.
    }

    static void ExecuteShutdownTasks()
    {
        // code removed for brevity.
    }
}
```

This code is mostly straightforward to understand.

The only part that is worth an explanation is how we perform dependency injection of services (here, with `BookmarkService`):

- We declare a variable of the `IBookmarkService` type that will be used to retrieve an instance of the injected service.
- We configure the dependency injection by leveraging the `HostBuilder` class provided by .NET and registering services to the `IServiceCollection` collection.

- We retrieve the instance of the registered services by calling `GetRequiredService` on the `IServiceCollection` collection and store a reference to the retrieved service into the variable we declared earlier.
- When creating an instance of a new command, we pass that variable as a parameter to the command's constructor so that the new command receives an instance of the service.

And voila! The service is automatically instantiated and injected into the various commands that require it.

What's great about this approach is that if we need to change the service implementation, all we need to do is modify the service registration to `IServiceCollection` and the rest will be magically taken care of.

Note how the `Program.cs` file has shrunk from 479 lines of code to 115 lines of code!

And the best part? Registering a new command to the root command is a matter of one extra line of code (aka, calling `AddCommand` on the root command and passing an instance of the new command to be registered), while injecting a new service is a matter of two lines of code: one for adding the service into the services collection and the other for getting a reference to that service in order to pass it to classes that require it.

> **Beware of the pitfalls!**
>
> Common pitfalls in dependency injection include circular dependencies, where classes depend on each other, and unintended singleton behavior, which can arise from improper service lifetimes. Over-injection of dependencies can violate the Single-Responsibility Principle, while excessive reliance on service locators complicates testing. To avoid these issues, it's crucial to manage service lifetimes carefully and follow best practices.
>
> I highly recommend that you check out the reading recommendations in *Chapter 14* in order to explore this topic more deeply when necessary.

Step 7 – running the program

Perfect! We have completed the refactoring of the `export` command. Let's run the code to ensure that it still works as expected.

The syntax to invoke the `export` command hasn't changed. So, let's invoke it the same way as before by typing the following:

```
dotnet run export --file 'bookmarks33.json'
```

We will get the following results:

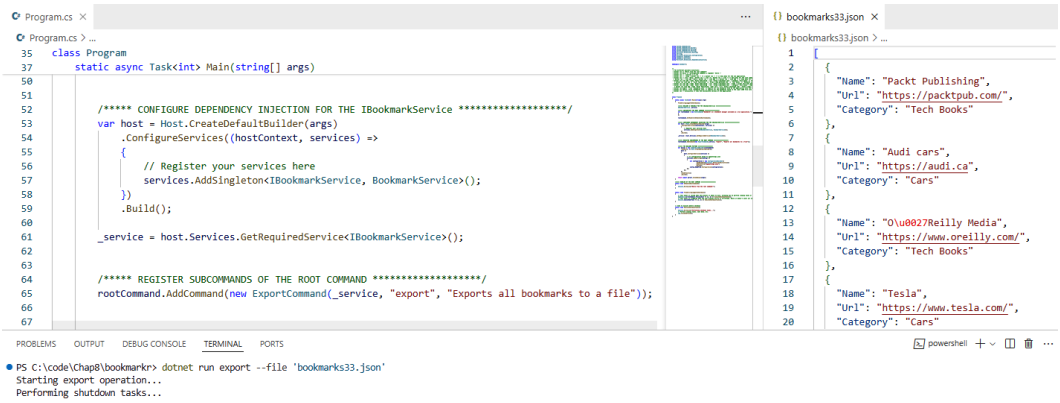


Figure 8.6 – Invoking the export command after refactoring

Wonderful! The application still works as expected.

We have done a lot of refactoring to our application so far. But is that all? Or can we take it to another level?

Taking refactoring to new heights

You may have been wondering why we didn't extract options and handler methods into their own code artifacts (such as classes).

The reason is that options and handler methods (and also arguments) are usually unique to a specific command. For this reason, they are defined in the command class.

However, in situations where they would need to be used by more than one command, we would have extracted them into their own code artifact. This reasoning is important to keep in mind in order to avoid over-complexifying our design by over-abstracting it.

In the case of an option, we would have created a dedicated class. Here's an example of `outputfileOption` we used in our `ExportCommand` class:

```
using System.CommandLine;

namespace bookmarkr.Options;

public class FileInfoOption : Option<FileInfo>
{
    public FileInfoOption(string[] aliases, string? description =
    null, bool onlyAllowLegalFileNames = true, bool isRequired = true)
        : base(aliases, description)
```

```
    {
        if(onlyAllowLegalFileNames == true)
        {
            this.LegalFileNamesOnly();
        }

        this.IsRequired = isRequired;
    }
}
```

We could then use this option in any command by creating an instance of it, as follows:

```
var outputfileOption = new FileInfoOption(["--file", "-f"], "The
output file path");
```

Here is how it would have looked for `ExportCommand`:

```
public class ExportCommand : Command
{
    #region Constructor

    public ExportCommand(IBookmarkService service, string name,
    string? description = null)
        : base(name, description)
    {
        _service = service;

        outputfileOption = new FileInfoOption(["--file", "-f"], "The
        output file path");
        AddOption(outputfileOption);

        // remaining of the code removed for brevity.
    }

    #endregion

    #region Options

    private FileInfoOption outputfileOption;

    #endregion

    // remaining of the code removed for brevity.
}
```

Pay special attention to how the `outputfileOption` property is declared (in the `Options` region) and how it is instantiated and initialized in the constructor. Its usage is no different from before.

In the case of a handler method, we would have created a base class that derives from `Command` (let's call it `CommandWithBaseHandler`), add the handler method to it (allowing it to be overridden), and make our command classes derive from that `CommandWithBaseHandler` class rather than from the `Command` class.

The `CommandWithBaseHandler` class could have looked like this:

```
using System.CommandLine;

namespace bookmarkr.Commands.Base;

public class CommandWithBaseHandler : Command
{
    public CommandWithBaseHandler(string name, string? description =
    null)
        : base(name, description)
    {
    }

    public virtual async Task OnExportCommand(FileInfo outputfile,
    CancellationToken token)
    {
        // method body removed for brevity.
    }
}
```

Notice that the `OnExportCommand` method has been marked as `virtual`. This means that it provides a default implementation in the `CommandWithBaseHandler` class but allows that implementation to be overridden if needed.

We could then have modified the `ExportCommand` class as follows:

```
using System.CommandLine;
using System.Text.Json;
using bookmarkr.Services;
using bookmarkr.Options;
using bookmarkr.Commands.Base;

namespace bookmarkr.Commands;

public class ExportCommand : CommandWithBaseHandler
{
```

```
    // the remaining code is not shown for brevity.
    // the OnExportCommand method is removed from this class since it
    // has been moved to the CommandWithBaseHandler base class.
}
```

Perfect! But where would these new code artifacts fit into our project structure? Great question!

Let's update our project structure to accommodate these new artifacts.

Updating the project structure

Following the same principle regarding project structure that we applied so far, I suggest the following:

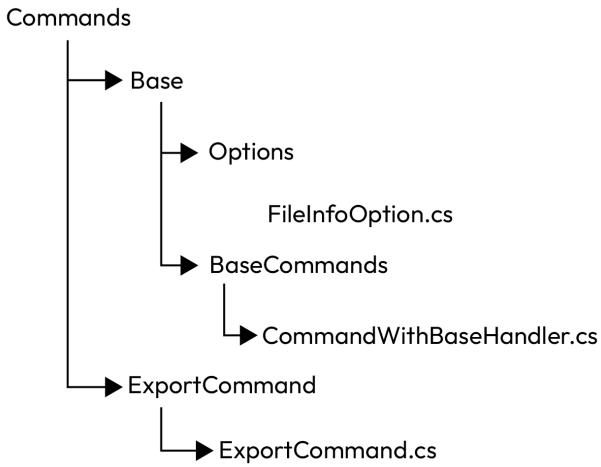


Figure 8.7 – The project structure for commands

Awesome! We now have an application that is more modular and easily extensible. Everything has its own place so it's easier to read and navigate the application's code.

Wait, extensible?!

You may not have noticed, but the refactoring exercise we have done throughout this chapter not only enhanced our application from a modularity standpoint but also from an extensibility standpoint.

Think about it: we can now easily involve other teammates in the development of our application, making delivering new features to our users even faster.

Every team member can focus on their very own command, impacting only a small subset of code artifacts and, in most cases, they won't modify the same files, reducing the number of merge conflicts that may occur when pushing their code into the source control.

This refactoring also allows to speed up the onboarding process of new team members. Since every code artifact has its proper place, the code is easier to understand and to own. If you are looking for contributors to your application, this is a very important point to keep in mind!

Summary

In this chapter, we refactored *Bookmarkr* to make it more modular. Each command is now described in its own code file.

By taking the time to refactor our CLI application, we have greatly enhanced its readability, maintainability, testability, and extensibility. It is now easier to add new capabilities, such as new commands (of course) as well as new features to existing commands.

Speaking about that, in the next chapter, we will see how to call external services and APIs to extend the capabilities of our application.

Your turn!

Following along with the provided code is a great way to learn through practice.

A better way is by challenging yourself to achieve tasks. Hence, I challenge you to improve the *Bookmarkr* application by adding the following features.

Task #1 – refactor the remaining commands

Even though this chapter proposes only one challenge, it will require effort on your side!

Throughout the pages of this chapter, we have refactored the `export` command. You are now tasked with refactoring the other commands of the *Bookmarkr* application.

For that matter, you can follow the same strategy and steps we used in our previous refactoring activity. By practicing it again and again, you will gain mastery of this process.

You will find the version of the code that hasn't been refactored in the `Program.Unrefactored.cs` file.

Let's do this!

9

Working with External APIs and Services

Although an autonomous application can provide great value to its users, consuming external APIs and services can make it provide even greater value to those users by integrating the functionalities of the application with other applications!

However, consuming external APIs and services creates new dependencies for your application. While this might make perfect sense, you have to know how to interact with these dependencies and how to integrate them into your application, so you don't couple your application too much to that external dependency and avoid having to change your application's code every time that dependency changes.

More specifically, in this chapter, we will discuss the following:

- The benefits of consuming external APIs
- Extending **Bookmarkr**'s capabilities by consuming an external API
- The proper way in .NET to consume external APIs
- How to avoid tight coupling between our application's commands and the external API

Technical requirements

The code for this chapter can be found in the GitHub repository accompanying this book, `https://github.com/PacktPublishing/Building-CLI-Applications-with-C-Sharp-and-.NET/tree/main/Chapter09`

Why consume external APIs?

When building your application, you have to consider multiple factors and sometimes implement multiple features that go well beyond your expertise.

Does that mean you should not build an application if you do not master every feature? No! Many applications rely on code developed by other people who are more skilled and experienced in a very specific area. These pieces of code are packaged as APIs and services so we can use (aka consume) them without having to understand every line of code they contain.

We have already come across this situation when we added logging into Bookmarkr. We didn't develop the logging engine ourselves. We instead relied on an existing service provided by an organization that knows how to do it (and do it well!). By relying on that service, our application was able to benefit from logging functionality without having to be experts in the logging business domain.

Now, I can hear your thoughts (yes, I can – that is my sixth sense 😉). You are thinking that it doesn't seem that complicated to develop your own logging engine, and you might be right. This is a business decision: if it is part of your core business, then yes, it makes sense to invest time, resources, and money in developing, testing, and maintaining your own logging engine. But remember, while it might be cool to develop it, you will have to maintain it, and that is what hurts many organizations in the long run! You know what they say… You build it, you run it! 😉

Also keep in mind that building your own "dependencies" (aka, services that are not part of your core business) is not always easy. An example of that is a payment gateway. There is a lot of regulation involved in building and offering such a service. If it's not your core business (in other words, if you are not Stripe or such a company), don't do it! Consume an existing service.

By consuming external APIs and services, we can then focus on what we do best, and in our case, it is managing bookmarks! That is the key to consuming external APIs and services: having the ability to focus on our core business and delegating other concerns to whom those concerns are their core business.

How to consume an external API

.NET provides a way to interact with external APIs and services by abstracting the need to construct our own HTTP requests, handle the underlying networking details, send the request, and receive the response while performing serialization and deserialization and handling communication issues.

So, in order to interact with these external APIs and services, .NET provides us with the `HttpClient` class. However, the proper way to deal with this class is through the `IHttpClientFactory` interface. This allows us to create and manage `HttpClient` instances for optimal performance and resource management.

Benefits of using IHttpClientFactory

Using `IHttpClientFactory` provides several advantages:

- **Connection management**: It manages the lifetime of `HttpMessageHandler` instances, which helps prevent issues such as socket exhaustion
- **Connection reuse**: It reuses underlying HTTP connections, improving performance

- **Resilience**: It adds resilience to transient faults
- **Configurability**: It allows for easy configuration of `HttpClient` instances

Bookmarkr: your bookmarks, anywhere!

Up to this point, Bookmarkr has been managing our bookmarks locally. This means that we are tied to the physical boundaries of our computer.

But what happens if we want to access these bookmarks from another computer?

To make this happen, we will need to extend the capabilities of Bookmarkr beyond the local computer. To do this, we will make Bookmarkr call an external API that will be responsible for storing and retrieving our bookmarks.

For that matter, we will add a new command called `sync` that will be responsible for synchronizing the local bookmarks with the ones stored by the external service.

> **About the external service**
>
> When you consume an external service, you don't have to know its internals (aka, its architecture, technology stack, application code, and dependencies). This is in accordance with the encapsulation principle of object-oriented programming.
>
> All you need to know is how to send requests to it and how to interpret the responses it returns.
>
> However, since I know that you are curious to know more about it, I have provided the details of its architecture and its application and infrastructure code in the `appendixA-bookmarkr-syncr` branch of the GitHub repository.

Let's start by adding the new command!

The sync command

Following the project structure that we designed in the previous chapter, let's add a new folder named `Sync` under the `Commands` folder, and within this folder, let's add a new code file named `SyncCommand.cs`.

The startup code for this command is the following:

```
public class SyncCommand : Command
{
    #region Properties
    private readonly IBookmarkService _service;
    #endregion

    #region Constructor
```

```
    public SyncCommand(IBookmarkService service, string name, string?
    description = null)
        : base(name, description)
    {
        _service = service;
        this.SetHandler(OnSyncCommand);
    }
    #endregion

    #region Options
    #endregion

    #region Handler method
    private async Task OnSyncCommand()
    {
    }
    #endregion
}
```

This code is pretty straightforward and needs no explanation.

The synchronization process comprises the following steps:

1. The local bookmarks are sent by Bookmarkr's `sync` command to the external service (called BookmarkrSyncr).
2. BookmarkrSyncr will then perform the synchronization between the local bookmarks it receives from Bookmarkr with the ones it has in its data store.
3. BookmarkrSyncr will send the synchronized bookmarks back to the Bookmarkr `sync` command's handler method.
4. The `sync` command's handler method will store the received bookmarks in the local data store. Keep in mind that if the application is dealing with large datasets or rate-limiting APIs, batching and retry techniques will be needed.

So, the `sync` command needs to have a reference to `IHttpClientFactory`. Let's add this:

```
#region Properties
    private readonly IBookmarkService _service;
    private readonly IHttpClientFactory _clientFactory;
#endregion

#region Constructor
    public SyncCommand(IHttpClientFactory clientFactory,
    IBookmarkService service, string name, string? description = null)
```

```
        : base(name, description)
    {
        _service = service;
        _clientFactory = clientFactory;
        this.SetHandler(OnSyncCommand);
    }
#endregion
```

In this code, we are adding a `private` property of type `IHttpClientFactory` and we are injecting it through the constructor.

We are then using it in the command's method handler:

```
#region Handler method
    private async Task OnSyncCommand()
    {
        var retrievedBookmarks = _service.GetAll();
        var serializedRetrievedBookmarks = JsonSerializer.
        Serialize(retrievedBookmarks);
        var content = new StringContent(serializedRetrievedBookmarks,
        Encoding.UTF8, "application/json");

        var client = _clientFactory.CreateClient("bookmarkrSyncr");
        var response = await client.PostAsync(«sync», content);

        if (response.IsSuccessStatusCode)
        {
            var options = new JsonSerializerOptions
            {
                PropertyNameCaseInsensitive = true
            };

            var mergedBookmarks = await JsonSerializer.
            DeserializeAsync<List<Bookmark>>(
                await response.Content.ReadAsStreamAsync(),
                options
            );

            _service.ClearAll();
            _service.Import(mergedBookmarks!);

            Log.Information("Successfully synced bookmarks");
        }
        else
```

```
        {
            switch(response.StatusCode)
            {
                case HttpStatusCode.NotFound:
                    Log.Error("Resource not found"); break;
                case HttpStatusCode.Unauthorized:
                    Log.Error("Unauthorized access"); break;
                default:
                    var error = await response.Content.
                    ReadAsStringAsync();
                    Log.Error($"Failed to sync bookmarks | {error}");
                    break;
            }
        }
    }
#endregion
```

This code is pretty easy to understand and conforms to the synchronization process we described earlier.

There is, however, one segment in that code that requires an explanation:

- We create an HTTP client from the `IHttpClientFactory` instance by relying on the named clients approach. As you can see, we are providing the name of the client configuration (here, `bookmarkrSyncr`) to the `CreateClient` method. We will get back to this configuration later.
- Next, we issue a `POST` request to the `sync` endpoint of the remote web service, passing the list of local bookmarks that have been previously serialized as JSON using an instance of the `StringContent` class:
  - If the request is successful, we deserialize the returned list of bookmarks (which represents the list of synchronized local and remote bookmarks) and we replace the local list of bookmarks with this new list
  - If the request is not successful, we display an error message corresponding to the returned HTTP status code

In order to import the `IHttpClientFactory` interface, we need to reference the `Microsoft.Extensions.Http` NuGet package. As we already know by now, we can do this by typing this command:

```
dotnet add package Microsoft.Extensions.Http
```

Before we can use our new command, let's register it within the `Program` class!

Registering the sync command

Let's register the `sync` command in the `Program` class. It's only a matter of one line of code:

```
rootCommand.AddCommand(new SyncCommand(_clientFactory, _service,
"sync", "sync local and remote bookmark stores"));
```

But wait! Where did the `_clientFactory` variable come from?!

Well done! You spotted it! 😊

As you may have guessed, this is a reference to the `HttpClient` that we need to configure to make the magic happen. This is where we will talk about the named clients approach that we mentioned earlier.

The `_clientFactory` variable is of type `IHttpClientFactory`. So, we first need to declare it within the `Main` method of the `Program` class:

```
IHttpClientFactory _clientFactory;
```

This will allow us later to retrieve a reference to it and pass it to the constructor of `SyncCommand` during its registration (as we saw earlier). We can retrieve that reference as follows:

```
_clientFactory = host.Services.
GetRequiredService<IHttpClientFactory>();
```

Finally, let's register the HTTP client for the BookmarkrSyncr service. We do this within the `ConfigureServices` block as follows:

```
services.AddHttpClient("bookmarkrSyncr", client =>
{
    client.BaseAddress = new Uri("https://bookmarkrsyncr-api.
    azurewebsites.net");
    client.DefaultRequestHeaders.Add("Accept", "application/json");
    client.DefaultRequestHeaders.Add("User-Agent", "Bookmarkr");
});
```

Let's explain what this code does:

- A name (`bookmarkrSyncr`) is provided for the registered HTTP client. This is why we call this approach "`named clients`". Notice that this is the same name that is passed to the `CreateClient` method in the `SyncCommand` class we saw earlier. This is how the appropriate HTTP client is selected.
- We then specify the base address for the service and a couple of request headers. Note that the base address does not specify the `sync` endpoint. It is specified when performing the request. This allows a web service to have different endpoints and for those endpoints to be invoked as needed without having to specify the base address over and over again.

> **About the base address**
>
> You may have noticed that the base address points to an external URL. I deployed the code of the BookmarkrSyncr service in App Service on Azure.
>
> I will keep this service up and running as long as I can but keep in mind that you have access to both its infrastructure and application code in the `appendixA-bookmarkr-syncr` branch if you need to redeploy it.

Everything is now set up. We can run the program and see what happens.

Running the program

To run the program, we simply need to execute this command:

```
dotnet run sync
```

The result will be as follows:

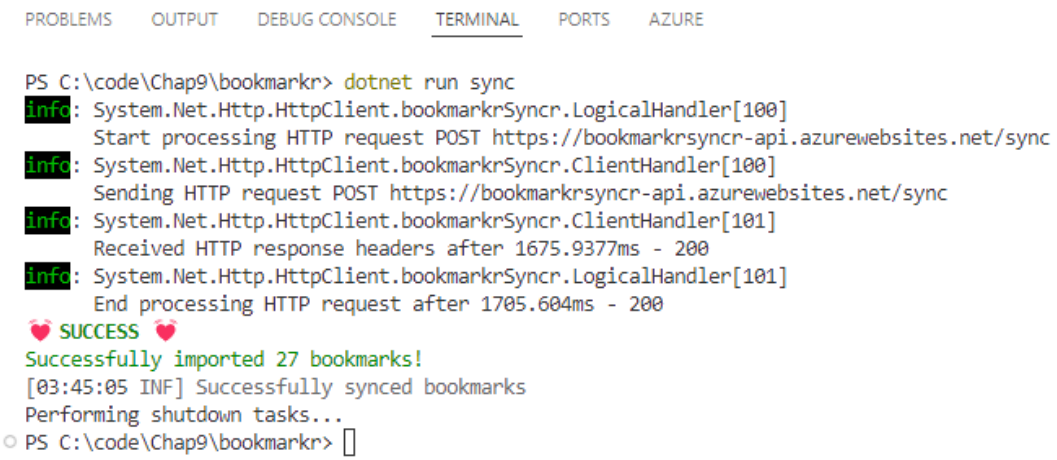


Figure 9.1 – The sync command in action

Great, isn't it?

If we list all available local bookmarks, we will notice that they have indeed been synchronized with the remote list of bookmarks.

What about security?

You may certainly have noticed that the web service can be used without any authentication. In other words, anonymous requests are allowed, which may be a security concern.

You are totally right, and this is intentional for now as security will be addressed in *Chapter 13* where we will see how we can authenticate users using a technique called "Personal Access Tokens," which resembles using API keys.

The code works great but there is actually a drawback to it.

Reducing the coupling between our application and the external dependency

In the previous section, although we applied the best practices of consuming external APIs, we created a coupling between our application and that dependency…

Notice that our application actually knows about the data type and structure that is returned by the API. This means that whenever this API changes, we will need to update our code accordingly.

This also means that our application is responsible for handling the different HTTP codes that the API may return. Can't we abstract this complexity somewhere so that eventual changes are scoped to a small portion of our code?

Of course we can! And there is a pattern for that, which is called **Service Agent**.

About the Service Agent pattern

The Service Agent pattern abstracts away the details of HTTP communication into a dedicated service, allowing other services (or, in our case, commands) to interact with external systems without directly dealing with HTTP requests and responses.

There are many benefits to the Service Agent pattern, among which are the following:

- **Abstraction**: It abstracts the complexity of HTTP communication, including constructing the HTTP request, handling the response, and managing errors
- **Encapsulation**: It encapsulates all the logic related to communicating with a specific external service or API
- **Reusability**: Service Agent can be reused by multiple components or services within the application
- **Separation of concerns**: It separates the business logic (in the `sync` command) from the communication logic (in Service Agent)
- **Maintainability**: Changes to the external API or communication protocol only need to be made in one place (Service Agent)

I believe that it's clear to you by now that our CLI application can greatly benefit from leveraging the Service Agent pattern. Let's now see how we can implement it!

Implementing the Service Agent pattern

This pattern is often implemented using `IHttpClientFactory` and named or typed `HttpClient` instances.

We are already using these artifacts, so it will be quite straightforward for us to abstract the HTTP details away from the `sync` command and into a dedicated Service Agent class.

The first step we will perform is to create a folder structure for Service Agents. Following the project structure we have laid out in the previous chapter, let's create a folder named `ServiceAgents` and a subfolder named `BookmarkrSyncrServiceAgent`.

Within this subfolder, let's create two code artifacts: an interface file named `IBookmarkrSyncr-ServiceAgent.cs`, and a class file named `BookmarkrSyncrServiceAgent.cs`.

Here is the code of the `IBookmarkrSyncrServiceAgent` interface:

```
namespace bookmarkr.ServiceAgents;

public interface IBookmarkrSyncrServiceAgent
{
    Task<List<Bookmark>> SyncBookmarks(List<Bookmark> localBookmarks);
}
```

This interface only exposes one operation, `SyncBookmarks`, which takes the list of local bookmarks (held by the Bookmarkr CLI application) and returns the synced list of bookmarks that includes the bookmarks from the remote web service, BookmarkrSyncr.

Let's now implement this interface:

```
namespace bookmarkr.ServiceAgents;

public class BookmarkrSyncrServiceAgent : IBookmarkrSyncrServiceAgent
{
    private readonly IHttpClientFactory _clientFactory;

    public BookmarkrSyncrServiceAgent(IHttpClientFactory
    clientFactory)
    {
        _clientFactory = clientFactory;
    }

    public async Task<List<Bookmark>> Sync(List<Bookmark>
```

```
localBookmarks)
    {
        var serializedRetrievedBookmarks = JsonSerializer.
        Serialize(localBookmarks);
        var content = new StringContent(serializedRetrievedBookmarks,
        Encoding.UTF8, "application/json");

        var client = _clientFactory.CreateClient("bookmarkrSyncr");
        var response = await client.PostAsync(«sync», content);

        if (response.IsSuccessStatusCode)
        {
            var options = new JsonSerializerOptions
            {
                PropertyNameCaseInsensitive = true
            };

            var mergedBookmarks = await JsonSerializer.
            DeserializeAsync<List<Bookmark>>(
                await response.Content.ReadAsStreamAsync(),
                options
            );

            return mergedBookmarks!;
        }
        else
        {
            switch(response.StatusCode)
            {
                case HttpStatusCode.NotFound:
                    throw new HttpRequestException($"Resource not
                    found: {response.StatusCode}");
                case HttpStatusCode.Unauthorized:
                    throw new HttpRequestException($"Unauthorized
                    access: {response.StatusCode}");
                default:
                    var error = await response.Content.
                    ReadAsStringAsync();
                    throw new HttpRequestException($"Failed to sync
                    bookmarks: {response.StatusCode} | {error}");
            }
        }
    }
}
```

As you may have noticed, this implementation is reusing the code that was located in the body of the `Sync` command's handler method, hence abstracting it from this method and encapsulating it into the Service Agent class.

For this reason, the code of this class does not need a lot of explanation. However, it is worth mentioning that in case of an unsuccessful request, we return an instance of `HttpRequestException` with the details about the issue.

Next, we need to update the code of the `SyncCommand` class to abstract the use of `IHttpClient-Factory` and use our new Service Agent instead. The updated code is the following:

```
public class SyncCommand : Command
{

    #region Properties
    private readonly IBookmarkService _service;
    private readonly IBookmarkrSyncrServiceAgent _serviceAgent;
    #endregion

    #region Constructor
    public SyncCommand(IBookmarkrSyncrServiceAgent serviceAgent,
    IBookmarkService service, string name, string? description = null)
        : base(name, description)
    {
        _service = service;
        _serviceAgent = serviceAgent;
        this.SetHandler(OnSyncCommand);
    }
    #endregion

    #region Options
    #endregion

    #region Handler method
    private async Task OnSyncCommand()
    {
        var retrievedBookmarks = _service.GetAll();
        try
        {
            var mergedBookmarks = await _serviceAgent.
            Sync(retrievedBookmarks);
            _service.ClearAll();
            _service.Import(mergedBookmarks!);
```

```
            Log.Information("Successfully synced bookmarks");
        }
        catch(HttpRequestException ex)
        {
            Log.Error(ex.Message);
        }
    }
    #endregion
}
```

This code is quite simple and easy to read. What we did basically was replace the use of `IHttpClientFactory` with the use of `IBookmarkrSyncrServiceAgent` and remove all the code, in the `OnSyncCommand` method, that was dealing with the HTTP communication (which we abstracted away into the Service Agent class) by the call to the `Sync` method of the Service Agent. Hence, the `OnSyncCommand` method is also leaner, shrinking from 41 lines of code down to 16.

> **Note**
>
> For your reference, we provide a copy of the `SyncCommand` class as it was before introducing the use of the Service Agent class. By doing so, you can easily compare the differences between the two implementations. Look for the file named `SyncCommand_NoServiceAgent.txt` in the `Commands/Sync` folder.

The final step is to register the Service Agent into the list of services in the `ConfigureServices` section of the `Program` class.

As we have seen previously, this can be easily done by adding this line of code:

```
services.AddScoped<IBookmarkrSyncrServiceAgent,
BookmarkrSyncrServiceAgent>();
```

We need not forget to declare a variable for the Service Agent:

```
IBookmarkrSyncrServiceAgent _serviceAgent;
```

Next, we need to retrieve the instance of that Service Agent:

```
_serviceAgent = host.Services.
GetRequiredService<IBookmarkrSyncrServiceAgent>();
```

We pass it to the constructor of the `SyncCommand` class:

```
rootCommand.AddCommand(new SyncCommand(_serviceAgent, _service,
"sync", "sync local and remote bookmark stores"));
```

Everything is now in place. Let's make sure that the application still works as expected.

Rerunning the program

We can run the program the same way we did before, by typing this command:

```
dotnet run sync
```

We will get the exact same result, proving that the application still works as it is supposed to:

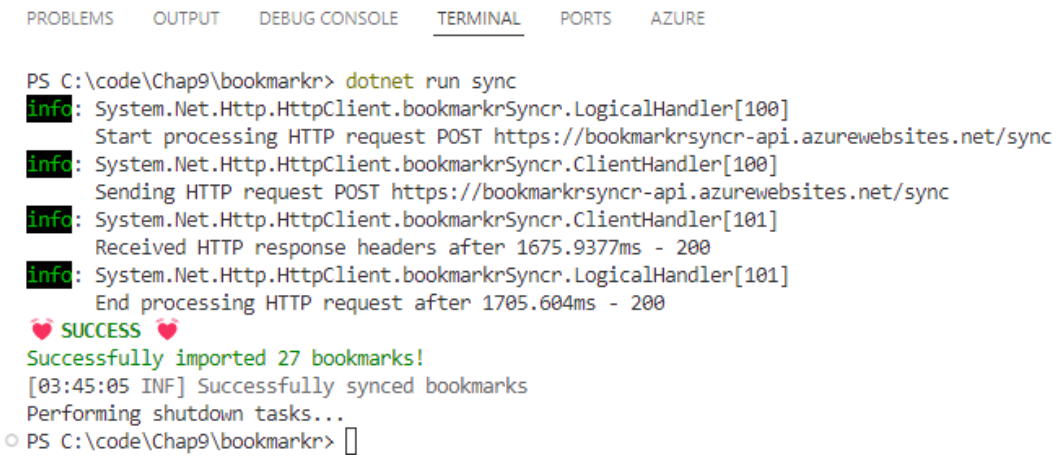


Figure 9.2 – The sync command in action using the Service Agent

Awesome! By leveraging the Service Agent pattern, we have been able to provide a clean separation between the business logic and the details of HTTP communication. Hence, we can consume the BookmarkrSyncr web service in any other command (using the `BookmarkrSyncrServiceAgent` class), without this command having to deal with the HTTP communication details.

Summary

In this chapter, we learned how to extend the functionalities of Bookmarkr by consuming external APIs and services.

We explored the best practices of communicating with external dependencies, handling response data, codes, and errors, and designing that integration in a way that does not create a heavy coupling between the external dependency and our application, making it easy to replace that dependency over time if this proves necessary.

In the next chapter, we will cover one of the key aspects of building applications, which is testing those applications.

Your turn!

Following along with the provided code is a great way to learn through practice.

A better way is by challenging yourself to achieve tasks. Hence, I challenge you to improve the Bookmarkr application by adding the following features.

Task #1 – adding SQLite as a data store

Who said that APIs are the only external dependencies an application can rely on? Certainly not me! 😊

Until now, our application has stored its bookmarks in memory. You will certainly agree with me that this is not an ideal solution as bookmarks will be lost as soon as the application terminates or restarts.

You are asked to add a new dependency to the Bookmarkr application – a **SQLite** database! This will allow bookmarks to be stored in a more permanent manner by Bookmarkr, making it more useful to our users 😉.

Why SQLite? You may ask…

SQLite is a versatile and lightweight database solution, designed to be both simple and easy to use while requiring minimal setup and administration. One of its most significant advantages is its portability: the entire database is stored in a single file, which makes it easy to move, back up, and distribute. Its self-contained nature also means that SQLite doesn't require a separate server process or system configuration, simplifying its deployment. That is why it is a great fit for CLI applications!

Now, you will also need to modify the code of `BookmarkService` to retrieve bookmarks from and store bookmarks in the SQLite database.

Consider using the `Microsoft.Data.Sqlite` library for .NET, as it is a reliable and lightweight library. Consider adding migrations and ensuring thread-safe access for SQLite in concurrent CLI scenarios.

Task #2 – retrieving the web page name based on its URL

Until now, when adding a new bookmark, we had to pass both the web page name and URL.

Now that we know how to work with external dependencies, let's tweak the `link add` command so that it makes an HTTP request to retrieve the name of the web page to bookmark based on the provided URL. If the name can't be retrieved, we can then use the name that was passed as a command option.

If the web page cannot be found, the bookmark's name should be `Unnamed bookmark`. If the request takes more than 30 seconds, terminate it and also set the name to `Unnamed bookmark`.

Part 4: Testing and Deployment

In this part, you will explore the critical aspects of testing, packaging, and deploying CLI applications. You'll learn various strategies for testing CLI tools, including unit testing individual components. Next, you'll delve into packaging your CLI application for distribution, using the most commonly used mechanisms such as Docker, .NET Tool, and Winget. You'll understand how to specify entry points, define dependencies, and structure your project for optimal packaging. Finally, you'll explore deployment methods, learning how to distribute your CLI tool through package managers and ensure it works consistently across different environments.

This part has the following chapters:

- *Chapter 10, Testing CLI Applications*
- *Chapter 11, Packaging and Deployment*

10
Testing CLI Applications

Testing is an important phase of any software development project. The purpose of testing is to ensure that the application we put into the hands of our users behaves as expected and doesn't cause any harm to users (by leaking their personal information or by allowing malicious actors to take advantage of a security issue to harm users).

In this chapter, we will discuss why testing is so important and explore different testing techniques and tools that will help us achieve this goal. More specifically, we will discuss the following:

- Why testing is important
- Different types of tests
- What to test, how to do it, and when to run these tests
- How to mock external dependencies when writing unit tests

Technical requirements

The code for this chapter can be found in the GitHub repository accompanying this book, `https://github.com/PacktPublishing/Building-CLI-Applications-with-C-Sharp-and-.NET/tree/main/Chapter10`.

Why is testing so important?

Over the years, testing has proven to be very valuable when it comes to providing great software and digital experiences to users. Any developer, team, or organization that is serious about their software development project and truly cares about their users and the experience they have when using their applications will invest in software testing.

Testing allows us to ensure that the applications we put into the hands of our users are of great quality, reliable, performant, and secure.

Here are some key benefits you can expect from testing your application:

- **Ensuring your application's quality, usability, and reliability**: Testing ensures that the application behaves as expected, meets stakeholders' needs, adheres to business requirements and technical specifications, and provides value to users. This helps increase user (and customer) satisfaction and prevents a poorly designed or developed application from having a negative impact on the organization's reputation. This can, for example, be achieved by validating API responses and verifying the output formatting of the CLI application.
- **Ensuring your application's security and compliance**: Testing plays a crucial role in validating the security of the application. It helps identify potential vulnerabilities and weaknesses that could be exploited by malicious actors. It also ensures that the application complies with industry standards, regulations, and other critical requirements that an organization (or an industry) might be subject to.
- **Acting as your application's documentation**: If you have ever worked on IT projects, you know how hard it is to maintain accurate and up-to-date documentation. A by-product of software testing is that it also acts as live documentation for your application: you can run it at any time to understand the behavior of the application.
- **Simplify the application's evolution**: If you have ever worked on IT projects, you know how scary it is when you have to modify the code of an application that works fine. We even have a saying for that: "If it's not broken, don't fix it!". However, by having an efficient suite of tests, modifying that code is way less scary because we know that we can rely on that test suite to ensure that we didn't introduce any bug in the existing code and functionalities (we call these *regressions*).
- **Achieving cost savings and efficiency**: It is well-known that the cost of finding and fixing a bug in production might be 100 times more expensive than finding it and fixing it during the development or testing phase. This cost also includes the cost of loss of efficiency as the team has to stop working on new features and concentrate on fixing that bug.

> **This is not a book about testing!**
>
> During my career, I have coached and trained development teams, both for the organizations I worked for and for our customers, on testing and raising awareness about its importance. That's why I couldn't write a book about development without covering this topic.
>
> However, although we will provide very valuable information and guidance throughout this chapter, keep in mind that this is not a book about software testing. Hence, concepts such as test-driven development, behavior-driven development, and code coverage, will not be covered.
>
> That being said, I provide many references in *Chapter 14* if you want to dig into this fascinating topic.

As you can see, there are various types of tests. Let's highlight them!

Types of tests

There is a wide variety of test types. They can be classified into two categories:

- **Functional tests**: This category of tests verifies that the application performs its intended function, according to its specifications. In other words, it ensures that the application does what it is supposed to do (what it has been designed for).
- **Non-functional tests**: This category of tests verifies that the application does what is intended to do in a way that meets user expectations and quality standards.

Each of these categories is composed of various types of tests. This figure illustrates this relationship:

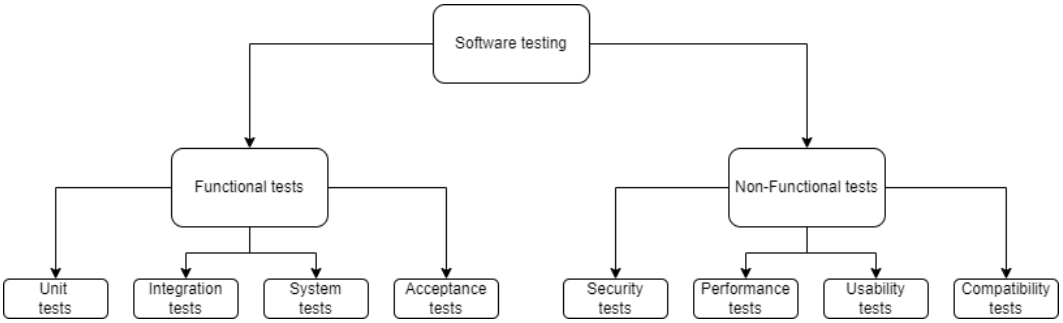


Figure 10.1 – Categories and types of software tests

Let's briefly describe each of these types of test:

- **Unit tests**: Here, we focus on testing methods in isolation (i.e., without relying on their dependencies such as databases or external services. To achieve this, we use *mocking* techniques (more on that later) to avoid relying on these dependencies. These tests are usually fast and provide immediate feedback. They therefore help ensure that a very specific portion of the code (a method) is doing exactly what it is intended to do. In the event that a bug is discovered, this type of test proves very useful as it narrows the issue down to the line of code that causes it! It is however important not to overuse mocks as it can lead to tests that do not represent reality, making them worthless.
- **Integration tests**: This type of test verifies the interactions between different components of the application, ensuring that the integrated parts work together as expected. It also helps ensure that the flow of data and communication between these components is correct and can surface integration issues, usually due to interface defects.

- **System tests**: This type of test verifies that the complete application meets the specified requirements. This covers end-to-end functionality and behavior and relies on external dependencies. These tests are performed in an environment that is similar to the production environment (usually the staging or pre-production environment).
- **Acceptance tests**: This type of test validates that the application meets the users' needs and business requirements. The main difference is that acceptance tests are usually performed by users or stakeholders and serve as a final approval before going to production.

We just described the types of functional tests. Let's now describe the types of non-functional tests:

- **Security tests**: This type of test is intended to reveal vulnerabilities and security breaches in the application and aims to protect users against data breaches, unauthorized access, and cyber-attacks in general.
- **Performance tests**: This type of test is intended to identify performance issues and bottlenecks by measuring response times and resource usage and identifying scalability or capacity limits under various workloads.. This provides valuable insights into what parts of the application require special attention, such as redesigning or refactoring, in order to meet the performance requirements and users' expectations.
- **Usability tests**: This type of test focuses on assessing the user-friendliness and ease of use of the application. This involves asking real users to test the application by completing tasks, gathering feedback and metrics along the way (such as the user experience in general, how easy it is to navigate the application, how long it takes to complete a task, and the user's appreciation in general).
- **Compatibility tests**: This type of test verifies that the application works as expected across different environments. For a cross-platform CLI application, this means that we need to ensure that it works correctly whether it is running on Windows, Linux, or macOS. Another example is to ensure that it gracefully degrades when necessary. Remember that we designed an `interactive` command, which means that the terminal it is running in is compliant with the requirements. We should then ensure that it works as expected when these requirements are met and gracefully degrades (e.g., converts to text-based output) when they are not.

About usability tests

As you may have guessed, usability tests are intended to be performed manually. While it may not always be possible to gather real users together to perform these tests, one way to achieve this goal in the context of a CLI application is by implementing a command in the CLI application that allows users to provide feedback.

Here is, as an example, how the Azure CLI team does it: they provide a `survey` command that directs the user to an online form where they can provide feedback.

Figure 10.2 – Allowing users to provide feedback

The pyramid of (software) testing

Many of you may not be familiar with the concepts of functional and non-functional tests, but you may be familiar with the pyramid of testing. It is worth mentioning that this pyramid includes many of the functional and non-functional types of test that we just discussed.

As a reminder, the pyramid of testing looks like this:

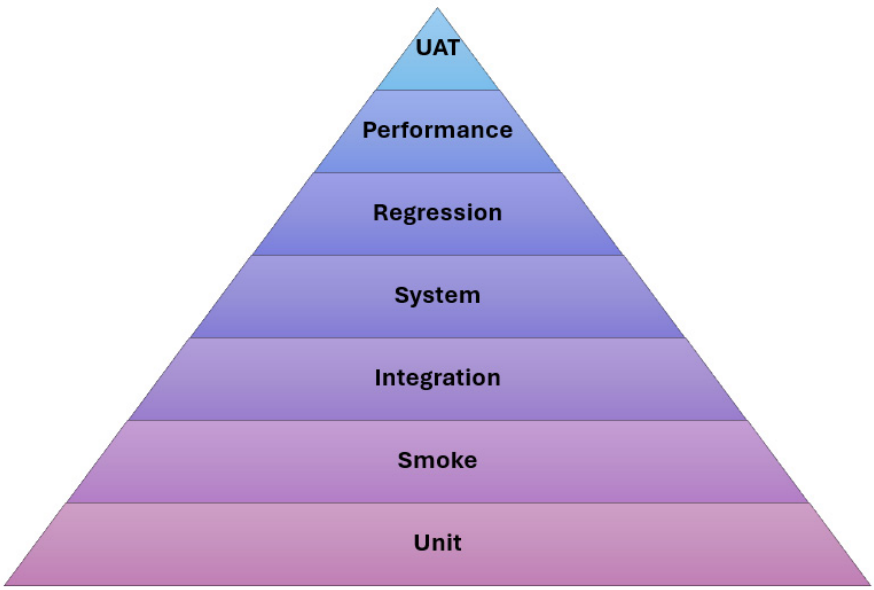


Figure 10.3 – The pyramid of (software) testing

The reason this is presented as a pyramid is to illustrate the quantity of tests to be expected at each step. The larger the step, the more tests it is expected to contain. As an example, a project is likely to have more unit tests than system tests and more system tests than UAT tests. This is due to the cost of creating and maintaining such tests.

It is important to keep in mind that what separates these various types of tests is their scope and their intent, not the frameworks or libraries used to implement them.

It is also important to keep in mind that you can create your own types of tests if you need to. Let me give you some examples.

About custom types of tests

In my career, I have worked with organizations that had their own custom types of tests that suited their needs and policies.

For example, some organizations may have *architectural tests*, which are tests written to ensure that an application (and its components) are developed following their architectural standards, such as what components can reference what components, ensure that every service class exposes an interface, and so on.

Another type of test I have seen can be referred to as *naming conventions tests*. These tests are meant to ensure that every component (such as classes, services or libraries) is named according to the organization's naming conventions and standards.

These two types of tests are intended to simplify the code review process and can be automated as part of validating a pull request.

We now have a better understanding of the various types of tests. The next logical question (the one that I get every time I discuss software testing with clients) is: *"What should we test?"* Let's discuss this now.

What should we test?

This is a great question!

It is easy to say that you should test every possible scenario in your application. However, we need to consider the following:

- How do you define "every scenario"?
- Can you really test "every scenario"? How many tests does that mean?

In a nutshell, your suite of tests should cover both of the following:

- **The happy path**: This means testing the situation in which the required input for your application is provided and is in the expected format
- **The unhappy path**: Here, we test the behavior of the application under unexpected circumstances, such as input in the wrong format, user errors, network issues (when relying on external dependencies), the user canceling a task, and so on

Once again, this is not a book about software testing, but I would like to give you some guidance on what to test in different situations:

Input parameters	Test both valid and invalid values. As an example, if a method takes only one integer parameter, and the range of valid values is between 1 and 100, we should also test it with values outside of this range, such as `-1`, `0`, and `2000` (we usually call these **boundary values**), along with values in the wrong format, such as `"bonjour"` or `1.23`.
Lists	When dealing with lists, we should ensure that the list only contains the expected elements, no more, no less. It is not sufficient to check the number of elements in the list because if a bug in the code inserts an element more than once or inserts inappropriate elements, the count could meet the expected value, but the list might not contain the appropriate elements.
Exceptions	If the application throws exceptions, this should be tested too to ensure that the right type of exception is thrown and with the expected details.
Methods/services returned values	It is easy to verify that a numerical or string value matches the expected value, but if the return value is an object (or a list of objects), we should verify that the values of all meaningful properties match the expected ones.
Methods/services behavior	While most developers verify the returned values of their methods or services when implementing tests, they fail at verifying that these methods or services behave as expected. Verifying the behavior of a method or service means that we need to ensure that: • The correct subsequent methods are called with the expected parameters. These could be logging or caching methods or calls to external dependencies such as database or external APIs for example. • The state changes within the object, the class, or the service are accurate and expected. • Side effects, such as database updates or changes to files, are accurate and expected. • Idempotency occurs when appropriate, which means that if the same method is called more than once, it maintains the coherence of the system. Think of a method that performs a call to a payment gateway or to a reservation system. We certainly don't want the customer to be charged twice for the same purchase, nor for multiple reservations to be made for the same appointment.

Figure 10.4 – What to test

It is just as important to know what to test as it is to know what *not* to test.

What not to test

You should not test external frameworks and libraries because this is the responsibility of their creators and maintainers. It is very likely that they have already been tested before you had a chance to use them. So, don't do this, please!

Other code artifacts that do not need to be tested are model classes and **Data Transfer Objects** (**DTOs**) as they are supposed to only contain properties, not methods, since they do not perform any processing of any kind, only move data around.

Also, you may have heard that it is not recommended to test private methods. There is a heated debate about whether we should test private methods. My personal opinion is that you should not, for two main reasons:

- **A private method is intended to be invoked by at least one public method**: Hence, when testing that public method, you are also testing the private one.
- **Technical difficulties when testing private methods**: Testing private methods relies on reflection in order to invoke the method and also relies on an array of `Object` to pass parameters to it. It is hence very easy to break these tests at runtime (they will compile, though) since the name of the method is usually passed as a string, and since every data type inherits from the `Object` class, if we change the data type or structure for a given parameter, it will still inherit from that same base class even though it is not expected by the private method anymore.

Testing is a safety net

As I tell my customers and students, a test suite is like a safety net:

- The more test cases you cover, the wider the safety net. If you fall from 30 feet and your safety net is 2 inches by 2 inches, the chances are that it won't help.
- On the other hand, if you have a safety net that is 50 feet by 50 feet but its meshes are 3 feet by 3 feet large, it won't help either. What I mean here is that if you have an extensive test suite in terms of a number of tests, but these tests do not cover meaningful situations, your test suite is of no use.

So, we are now aware of the different types of tests, and we know what to test. But when should we run these tests? Let's discuss this.

When should we run tests?

The various types of tests we discussed are intended to be run at different points during the development lifecycle.

An organization (or even a developer or a team) may have policies and preferences but, in general, the following recommendations are adopted by the industry:

- **Unit tests**: These tests are intended to be run during development. In other words, the developer should run them as they write the code. There are some IDEs (such as Visual Studio Enterprise) that even allow you to configure unit tests to be run in the background as you type your code! This has to be configured and used carefully as it might quickly become cumbersome. You have to, however, run them before committing your changes, and before creating or updating a pull request. Unit tests are also usually run as part of a CI/CD pipeline. For these reasons, unit tests should always be automated and are actually very easy to automate.
- **Integration tests**: These tests are to be run after unit tests have passed (that is, run successfully).
- **Smoke tests**: These tests should be run after a new build has been deployed and before QA testers engage in more extensive testing. For this reason, these tests are usually triggered by the CI/CD pipeline, right after the deployment operation is completed.
- **System tests**: These tests are run on a staging or pre-production environment, right before releasing the application to production. These tests are to be triggered after integration tests have passed.
- **Regression tests**: These tests should be run at least before committing your changes. They should also be part of the CI/CD pipeline.
- **Performance tests**: These should be run when new features (especially the ones dealing with external dependencies) or any significant code changes have been made.
- **UAT tests**: These special tests are performed by users (or their representatives, such as stakeholders) and are intended to obtain final approval before releasing the application to production. Hence, these tests are usually manual.

As you may have noticed, we haven't covered all the possible types of tests. We have only focused on the ones that are part of the pyramid of software testing.

Okay. Now that we understand the importance of software testing, and we know what to test, let's implement testing into our CLI application.

Adding a test project to Bookmarkr

In order to add a test project for our CLI application, I needed to make a slight change to the project structure, which was to extract the solution file (`.sln`) from the project directory and edit it to update the path to the `.csproj` file. This allows us to create a test project and add it to the solution.

Next, let's type the following command to create the test project:

```
dotnet new mstest -n bookmarkr.UnitTests
```

This will create a new directory, named `bookmarkr.UnitTests`, in which the content of the test project will reside.

Right now, this directory only contains two files:

- `Bookmarkr.UnitTests.csproj`, which describes the project, its configuration, and its dependencies
- `UnitTest1.cs`, which acts as a sample test class

It is interesting to note that the `.csproj` file already references some testing libraries and frameworks, particularly the **MS Test** testing framework, which is the one we will be using throughout this chapter.

> **About testing frameworks**
>
> While there are many testing frameworks, the most common ones being **NUnit**, **xUnit**, and **MS Test**, we decided to use the latter for many reasons:
>
> - MS Test is Microsoft's testing framework and is widely known and used
>
> - MS Test has evolved over the years and has a rich feature set, such as built-in support for testing parallel code, data-driven testing, and test grouping capabilities
>
> That being said, no matter what your preferred testing framework is, the concepts are similar and the one covered in this chapter will apply as well. The main difference will be in the keywords provided by each testing framework.

It is also interesting to note that the test project is defined as *non-packable*. This means that this project will not be packaged and distributed as part of the application, which makes perfect sense.

But before we go any further, let's add the test project to the solution using this command (which has to be run at the same location as the `.sln` file):

```
dotnet sln add bookmarkr.UnitTests/bookmarkr.UnitTests.csproj
```

Now, if we open the solution in Visual Studio, it will contain both the code and the test projects.

The next step is to make our test project reference the actual project. This is required so that we can test the actual code. So, let's navigate to the test project's directory and type this command:

```
dotnet add reference ../bookmarkr/bookmarkr.csproj
```

There is one last step we need to take before we start implementing unit tests, which is to define a structure for our test project.

Structuring the test project

Each developer, team, or organization will have preferences when it comes to structuring their test projects. In this section, I will introduce you to my way of structuring test projects, which I implemented during my career and found valuable.

First, if your application is composed of multiple components and each component has its own Visual Studio project, you will want to create a separate unit tests project for each component while having a single integration tests project. This structure may be similar to this:

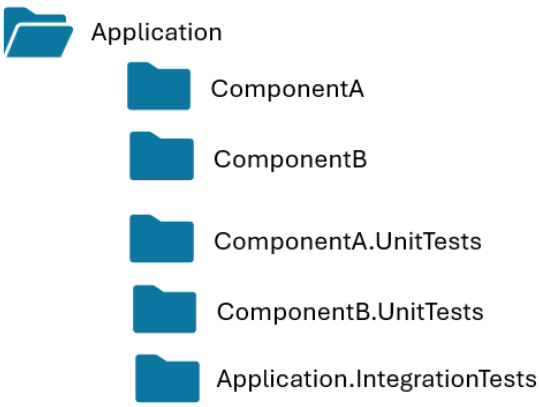


Figure 10.5 – Structure of test projects

In this chapter, we will only focus on unit tests, but the same principles apply to integration tests. This is the reason why we only have a `bookmarkr.UnitTests` project.

However, if you create multiple test projects within the solution, remember to always use the same version of the testing framework across all projects to prevent side effects due to compatibility issues.

Now, let's structure our unit tests project. Here, again, there are unlimited possibilities depending on personal choices and teams'/organizations' policies. My approach is to have one test class for each code class. The test class will have the same name as the code class completed with the `Tests` suffix. I also like to replicate, in the test project, the same folder structure as in the code project as I find it easier to navigate in the test project because of the structure parity between the two projects.

After applying these principles, the structure of our test project looks as follows:

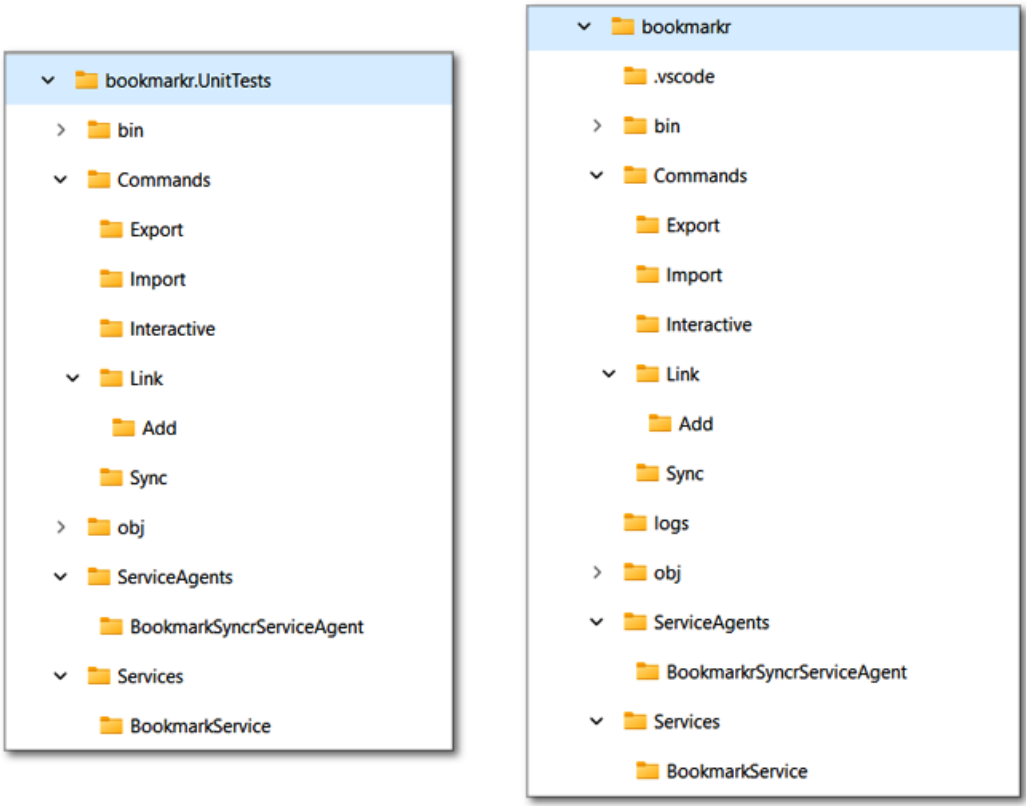


Figure 10.6 – Test project's structure

Now that the structure of our test project is set up, we can start implementing our unit tests. But wait! Are there any code artifacts that should not be tested? Yes, indeed!

Code artifacts that should not be tested

The following artifacts do not need to be tested since they do not perform any processing:

- `Bookmark.cs` and `BookmarkConflictModel.cs` since they are only model classes and hence only serve the purpose of moving data around.
- `Program.cs`: The purpose of this class is to configure the CLI application, configure logging, identify which command is the root command and build the command hierarchy.
- `Helper.cs`: This helper class's methods are used to format the text outputs using different colors and formatting. Hence, this class is more suitable for UI testing rather than unit testing. For this reason, it is excluded from unit testing. However, it could be tested as part of manual tests or end-to-end tests.

Keep in mind that even though we decided not to test these code artifacts for valid reasons, MS Test will tell us, as part of the test results, that these artifacts were not tested.

We can inform MS Test that we have chosen not to test these artifacts using the `[ExcludeFromCodeCoverage]` attribute. This attribute is very flexible: it can be applied to the property, method, class, or even assembly level. This attribute also allows us to pass a string to justify our decision.

As an example, this is how we will apply it to the `Program` class:

```
using System.Diagnostics.CodeAnalysis;
//…
[ExcludeFromCodeCoverage(Justification="CLI application configuration.
No processing is performed in this class.")]
class Program
{
    // …
}
```

We are finally ready to start implementing some tests. Let's dive in!

Writing effective tests

We will learn how to implement tests by writing tests for the `link` and the `import` commands.

The first thing we need to do is to add a test class for each command. We already have the folder structure in place, so let's add the test classes. As I mentioned earlier, I find it useful to name my test classes after the actual classes with an added suffix of `Tests`. So, our test classes will be named `LinkCommandTests` and `ImportCommandTests`, respectively.

Now, let me introduce you to the best practices of structuring a test class and its test methods (yes, we will talk about structure once again! 😊):

- Using MS Test, a test class is decorated with the `[TestClass]` attribute. If you don't provide this attribute, the class will not be considered a test class and the test methods it contains will not be run.
- A test class usually consists of multiple test methods. The name of a test method should convey its intent. This is important since the test report will only present the names of the methods along with an icon indicating the result of this test method (pass, fail, skipped, and so on). The usual approach is that the name of test a method is composed of its name, its input parameters' values, and the expected result. Examples of such names are `GetEmployeeById_ValidId_ReturnsTheExpectedEmployeeObject` and `GetEmployeeById_InvalidId_ThrowsEmployeeNotFoundException`.
- Using MS Test, a test method is decorated with the `[TestMethod]` attribute. If you don't provide this attribute, the class will not be considered as a test method and will not be run.

- A test method should test one and only one outcome (whether this is a result or a behavior). This is important since we need to be able to know that, if a test method fails, it is because it didn't achieve the expected outcome (a result or a behavior). To achieve this, however, a test method may contain more than one assertion as long as these multiple assertions serve the purpose of validating that one outcome.
- To maximize clarity and readability, it is advised that a test method's body be chunked into three parts (also known as the 3As):
  - **Arrange**: where all the objects required to perform the test are instantiated and initialized.
  - **Act**: where the code artifact to be tested is invoked and the result is gathered.
  - **Assert**: where the obtained result (usually referred to as the "actual" result) is compared to the expected result. If both match, the test is considered to be a success. However, the test is considered to be a failure.

Armed with this new knowledge, we have everything we need to write our first test. Let's start by writing tests for the `link` command.

Looking at the code of the `LinkCommand` class, we notice that it does not have any methods. However, we see that its constructor calls the `AddCommand` method to set the `LinkAddCommand` as a sub-command of `Linkcommand` (that is why `add` appears as a child of the `link` command).

In this case, our test method will not be verifying a result but rather a behavior. In this case, we want to verify that `LinkAddCommand` is actually a sub-command of `LinkCommand`.

Here is the code for this test method:

```
[TestMethod]
public void LinkCommand_CallingClassConstuctor_
EnsuresThatLinkAddCommandIsTheOnlySubCommandOfLinkCommand()
{
    // Arrange
   IBookmarkService service = null;
   var expectedSubCommand = new LinkAddCommand(service, "add", "Add a
   new bookmark link");

    // Act
    var actualCommand = new LinkCommand(service, "link", "Manage
    bookmarks links");
    var actualSubCommand = actualCommand.Subcommands[0];

    // Assert
   Assert.AreEqual(1, actualCommand.Subcommands.Count);
   CollectionAssert.AreEqual(actualSubCommand.Aliases.ToList(),
   expectedSubCommand.Aliases.ToList());
```

```
    Assert.AreEqual(actualSubCommand.Description, expectedSubCommand.
    Description);
}
```

Although this code is self-explanatory and easy to understand, I wanted to point out a couple of key points:

- Notice the naming convention applied to the name of the test method. It clearly indicates its intent: we are testing the `LinkCommand` class, and our test consists of calling the class constructor and ensuring that `LinkAddCommand` is its only sub-command.
- Notice that we applied the 3A principle to structure the body of the test method.
- Notice that we performed three assertions to verify the expected behavior. Also, note the usage of `CollectionAssert`, which helps with asserting collections and their items. It is way more convenient than using `Assert` when dealing with lists and collections of elements. This is my little secret for you, as many developers are not aware of it or don't naturally tend to use it 😉.

We are now ready to run our tests. Let's see how we do this.

Running our tests

The .NET CLI provides a command for this purpose:

```
dotnet test
```

This command will compile the code and the test project, discover the test classes and test methods, execute the tests, and return the results.

Visual Studio Code also provides a **graphical user interface** (**GUI**) for listing and executing the tests. This pane can be displayed by clicking on the appropriate icon, as shown in the following figure:

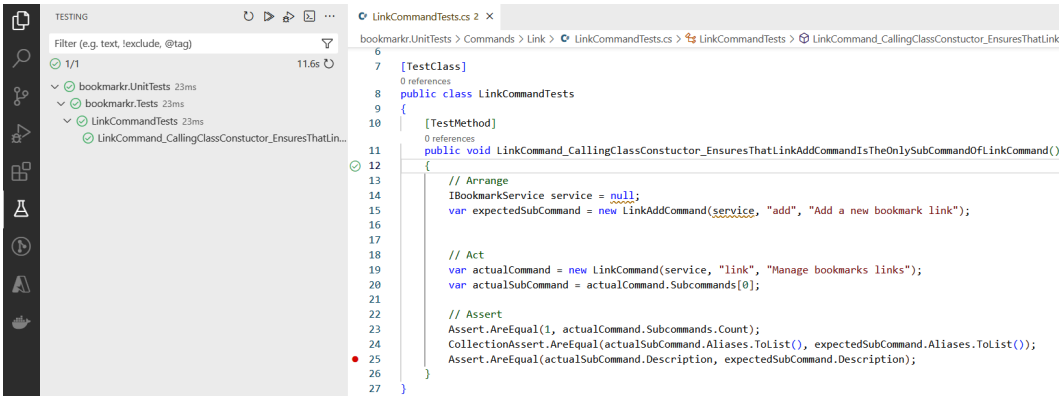


Figure 10.7 – Running tests

This GUI also provides a visual identification of the states of the test methods. In the preceding screenshot, we can see that our test method has been completed successfully.

From this GUI, we can also debug tests! This is a fantastic capability that helps us understand why a test is failing by strategically applying breakpoints and re-executing it in debug mode.

Excellent! We can implement more tests.

But wait!

Have you noticed that we passed a `null` instance of the `BookmarkrService` as a parameter? This is fine since we are not relying on that parameter in the test we are conducting. But if we did (as we will when testing the `import` command), we would like to provide an instance for it.

We obviously don't want to use an actual instance of that service as it is a dependency of the command and it may also rely on an external dependency, such as a database where the bookmarks are stored.

We will then need to provide a fake representation of it. This is where **mocks** come into play!

Mocking external dependencies

Mocking is useful for simulating the behavior of dependencies without actually relying on them. This is powerful because it allows us to test our application in isolation from its environment. The reason we want to do this is to have tests that ensure that the application's code works correctly, irrespective of the state of its dependencies.

The role of mocking

Let me clarify this with an example. Let's say you have a method that stores a bookmark in the database. You write a test method to verify that, and it fails. You run it again, and it passes. Can you tell, without investigating, whether this was due to a transient database connectivity problem or due to a bug in the code? You can't! But if you remove the dependency (that is, the database) from the equation and the same behavior occurred, you can tell (with a high level of confidence) that this was due to a bug in the code.

It is worth mentioning that we usually write tests that take external dependencies out of the equation (namely, unit tests) and tests that take these external dependencies into account (such as integration or system tests). So, when facing such an issue, we can look at unit and integration or system tests to figure out if the problem is due to the dependency (such as a communication problem) or due to the code itself.

How to mock an external dependency

As mentioned earlier, we can write our own fake implementation of a dependency that simulates the behavior of the real dependency, but the major drawback of this approach is that we have to maintain (and possibly rewrite) these implementations if the behavior of the real dependency changes.

It is wiser to rely on a mocking framework that will perform this task for us. What we need to do is to provide an interface for that dependency and the mocking framework will create a fake representation of it at runtime. We can also pass instructions to the mocking framework to configure the behavior of the fake dependency in a certain way, depending on the test we need to perform. For example, we can instruct the mocking framework to simulate a certain error or exception when invoking a dependency with some parameter values to verify the behavior of our application under these circumstances.

> **About mocking frameworks**
>
> Just as there are many testing frameworks, there are also many mocking frameworks for you to choose from. One of the most commonly used ones, and my personal favorite, is **NSubstitute**. I like it because it is both powerful and easy to learn and use.
>
> You can learn more about NSubstitute by visiting its website: `https://nsubstitute.github.io/help/getting-started/`.

Let's mock the `BookmarkService` service using NSubstitute.

Mocking the BookmarkService service

The first thing we need to do is to add NSubstitute to the test project. We can achieve this by navigating to the directory of the test project and typing this command:

```
dotnet add package NSubstitute
```

Now, let's update the `LinkCommandTests` test class by mocking the `BookmarkService` service. The updated code is as follows:

```
using NSubstitute;
…
namespace bookmarkr.Tests;

[TestClass]
public class LinkCommandTests
{
    [TestMethod]
    public void LinkCommand_CallingClassConstuctor_
    EnsuresThatLinkAddCommandIsTheOnlySubCommandOfLinkCommand()
    {
        // Arrange
        IBookmarkService service = Substitute.For<IBookmarkService>();
        …
    }
}
```

As you can see, we haven't done much here. We merely added a `using` statement for `NSubstitute` and instead of initializing the service to a `null` value, we asked `NSubstitute` to provide a simulation of it based on its interface (which is referred to as a *mock*). The result is a temporary, in-memory object built based on the structure of the `IBookmarkService` interface for which we can configure the behavior depending on the test we are performing.

If we now execute the test again, it still passes. Since we are not calling the service during our test, we do not need to configure its behavior. But we will need to do that for the upcoming test methods we are going to implement.

Using the mock version of the BookmarkService service

Let's start by creating a test class for the `import` command. We first create the `ImportCommandTests.cs` file in the `Commands\Import` directory. This file will contain all the test methods related to the `import` command.

Next, we lay out the base structure of the test class, as follows:

```
using bookmarkr.Commands;
using bookmarkr.Services;
using NSubstitute;

namespace bookmarkr.Tests;

[TestClass]
public class ImportCommandTests
{
}
```

This command has no subcommands, so we do not need to test that, or we can write a test that verifies this fact. If you want to do this, you can follow the same procedure as in the `LinkCommandTests` class.

Before we start implementing tests, we need to identify the test cases we want to perform. In invite you to reflect on that, but in the meantime, here is a non-exhaustive list of test cases:

- **Test case #1**: Calling the `OnImportCommand` handler method will call the `Import` method of `BookmarkService`.
- **Test case #2**: If the file name is invalid, an error message should be returned (for example, indicating that there is an forbidden character in the filename).
- **Test case #3**: If the file is not found, an error message, indicating that the file does not exist, should be returned.

- **Test case #4**: When importing bookmarks, if no conflict is detected, the imported bookmarks are found in the local collection of bookmarks.
- **Test case #5**: When importing bookmarks, if a conflict occurs, the conflicting bookmark's name is updated but the URL remains unchanged. The `Log` method is also called.

For the purpose of this chapter, we will only implement test cases 1 and 5. The remaining test cases are left for you as a challenge.

However, before we can implement these test cases, some changes to the application's code have to be made.

Changes to the code must be made!

Here is the idea: sometimes, changes to your code will have to be made so that the code can be testable. This is okay because some classes of the .NET framework aren't testable by nature.

In our situation, this is the case with the `FileInfo` class, which is a sealed class and exposes no interface, hence it cannot be overridden nor mocked.

Fortunately, there is a library that allows us to work around this limitation. We will then need to add the following NuGet package to both the application and the test projects:

```
dotnet add package System.IO.Abstractions
```

For the test project, we will also need to add this NuGet package, which will help with testing:

```
dotnet add package System.IO.Abstractions.TestingHelpers
```

We will also need to make the following changes to the `ImportCommand` class:

1. We will add a private property of type `IFileSystem`.
2. We will initialize this property to an instance of the `FileSystem` class in the default constructor.
3. We will add a second constructor that will only be used for testing. This constructor will take an extra parameter, of type `IFileSystem`.
4. Finally, we will add an overload of the `OnImportCommand` handler method that takes an `IFileInfo` parameter and whose only purpose is to call the original version of the `OnImportCommand` method, passing an instance of `FileInfo` based on the `IFileInfo` object it received.

Invoking the `import` command once again, we find out that it still works as expected.

We can now implement those test cases.

Going back to implementing the test cases

Let's start with test case 1. Here is the associated test method:

```
[TestMethod]
public void OnImportCommand_PassingAValidAndExistingFile_
CallsImportMethodOnBookmarkService()
{
    // Arrange
    var mockBookmarkService = Substitute.For<IBookmarkService>();

    string bookmarksAsJson = @"[
        {
            ""Name"": ""Packt Publishing"",
            ""Url"": ""https://packtpub.com/"",
            ""Category"": ""Tech Books""
        },
        {
            ""Name"": ""Audi cars"",
            ""Url"": ""https://audi.ca"",
            ""Category"": ""See later""
        },
        {
            ""Name"": ""LinkedIn"",
            ""Url"": ""https://www.linkedin.com/"",
            ""Category"": ""Social Media""
        }
    ]";

    var mockFileSystem = new MockFileSystem(new Dictionary<string,
    MockFileData>
    {
        {@"bookmarks.json", new MockFileData(bookmarksAsJson)}
        });

    var command = new ImportCommand(mockBookmarkService,
    mockFileSystem, "import", "Imports all bookmarks from a file");

    // Act
    command.OnImportCommand(mockFileSystem.FileInfo.New("bookmarks.
    json"));

    // Assert
    mockBookmarkService.Received(3).Import(Arg.Any<Bookmark>());
```

```
    mockBookmarkService.Received(1).Import(Arg.Is<Bookmark>(b =>
    b.Name == "Packt Publishing" && b.Url == "https://packtpub.com/"
    && b.Category == "Tech Books"));
    mockBookmarkService.Received(1).Import(Arg.Is<Bookmark>(b =>
    b.Name
    == "Audi cars" && b.Url == "https://audi.ca" && b.Category == "See
    later"));
    mockBookmarkService.Received(1).Import(Arg.Is<Bookmark>(b =>
    b.Name == "LinkedIn" && b.Url == "https://www.linkedin.com/" &&
    b.Category == "Social Media"));
}
```

This code deserves an explanation, so here we go:

1. We first create a mock of the `BookmarkService`, as we did in the previous example.
2. Then, we create a string representation of the JSON content for three bookmarks that will be needed for our test.
3. Next, and this is the reason why we performed the changes to the code that we described above, we create a mock of the filesystem and simulate the existence of a file named `bookmarks.json` that contains the JSON representation we created in the previous step.
4. After that, we create an instance of the `ImportCommand` class using the new constructor we have added, which allows us to pass the mock filesystem as a parameter.
5. We are now ready to invoke the `OnImportCommand` by relying on the mock filesystem and passing the name of the `bookmarks.json` file we simulated earlier. It is important to note here that if we pass the name of a file that was not part of the simulation, the test will fail.
6. We are now ready to verify whether our assertions are correct. Pay careful attention to how we did it: we first ensured that calling the `OnImportCommand` method triggered three calls to the `Import` method of `BookmarkService` (here, these calls are in fact made to the mock version of the service since we do not want the service to be activated but rather, we only want to validate that it was invoked as expected). This is, however, not sufficient to validate that the test is successful as these three calls could include unexpected calls. To make sure that these calls are legitimate, we verify them, one by one, ensuring that each of their meaningful properties matches what is expected.

And that's it for test case #1. Let's now move on to test case #5.

Here is the code for this test case:

```
[TestMethod]
public void ImportCommand_Conflict_
TheNameOfTheConflictingBookmarkIsUpdated()
{
    // Arrange
```

```
var bookmarkService = new BookmarkService();
bookmarkService.ClearAll();
bookmarkService.AddLink("Audi Canada", "https://audi.ca", "See
later");

string bookmarksAsJson = @"[
    {
        ""Name"": ""Packt Publishing"",
        ""Url"": ""https://packtpub.com/"",
        ""Category"": ""Tech Books""
    },
    {
        ""Name"": ""Audi cars"",
        ""Url"": ""https://audi.ca"",
        ""Category"": ""See later""
    },
    {
        ""Name"": ""LinkedIn"",
        ""Url"": ""https://www.linkedin.com/"",
        ""Category"": ""Social Media""
    }
]";

var mockFileSystem = new MockFileSystem(new Dictionary<string,
MockFileData>
{
    {@"bookmarks.json", new MockFileData(bookmarksAsJson)}
});

var command = new ImportCommand(bookmarkService, mockFileSystem,
"import", "Imports all bookmarks from a file");

// Act
command.OnImportCommand(mockFileSystem.FileInfo.New("bookmarks.
json"));
var currentBookmarks = bookmarkService.GetAll();

// Assert
Assert.AreEqual(3, currentBookmarks.Count);
Assert.IsTrue(currentBookmarks.Exists(b => b.Name == "Packt
Publishing" && b.Url == "https://packtpub.com/" && b.Category ==
"Tech Books"));
Assert.IsTrue(currentBookmarks.Exists(b => b.Name == "Audi cars"
&& b.Url == "https://audi.ca" && b.Category == "See later"));
```

```
    Assert.IsTrue(currentBookmarks.Exists(b => b.Name == "LinkedIn"
    && b.Url == "https://www.linkedin.com/" && b.Category == "Social
    Media"));
    Assert.IsFalse(currentBookmarks.Exists(b => b.Name == "Audi
    Canada" && b.Url == "https://audi.ca" && b.Category == "See
    later"));
}
```

The code for this test case is very similar to the code for test case #1, with two noticeable differences:

- We are calling the real implementation of `BookmarkService`, not a mock. The reason is that we want to ensure that the bookmarks have been imported correctly and that the conflicting one has been renamed accordingly. If that service was relying on a database, we could have mocked that database.
- The last assertion, although not necessary, ensures that the original, conflicting bookmark does not exist anymore since it has been updated.

Now, if you have been coding along with me, you have certainly noticed that this code does not work. In fact, it does not even compile! Don't worry, this was intentional 😊. The intention is to teach you how to control visibility to test artifacts.

Internals visibility

In the `ImportCommand` class, we added a second constructor (which takes a parameter of type `IFileSystem`) and an overload for the `OnImportCommand` handler method. These two methods have been marked as `internal`, which simply means that they are visible to all parts of the code within the current project but aren't visible outside of that project unless we specify otherwise.

This is the recommended approach when you add artifacts specifically for testing purposes.

The `internal` accessor is very interesting. It allows us to control its visibility. In this case, we only want the test project to see these internal code artifacts.

To do that, we will need to update the `bookmarkr.csproj` file (the one where these code artifacts marked as `internal` are located) to indicate that we only want the test project to be able to access them. We can achieve this by adding this entry:

```
<ItemGroup>
    <InternalsVisibleTo Include="bookmarkr.UnitTests" />
</ItemGroup>
```

This means that code artifacts that are marked as `internal` in the `bookmarkr` project can only be "seen" by the `bookmarkr.UniTests` project.

Now, you will notice that the code compiles and executes as expected.

Looking back at the test methods of the `ImportCommandTests` class, you have certainly noticed that in both test methods, we instantiated and initialized `MockFileSystem` and the string representation of the JSON structure of the bookmarks in the same way. Hence, this code is redundant and, if it changes over time, we need to update this code in both places. This will get even worse as the number of test methods increases.

Fortunately, MS Test provides a way to centralize this initialization. Let's see how this works.

Centralizing test initialization

MS Test provides a `[TestInitialize]` attribute that can be used to decorate a method where any common instantiation, initialization, or configuration can be centralized.

This method is then automatically invoked by the MS Test framework before calling every test method. This has another benefit: every test method gets fresh instances of the objects instantiated in the initialization method, preventing the execution of a test method to have an impact and an influence on the execution of the next test method.

The code of the test initialization method looks like this:

```
public required IBookmarkService _bookmarkService;
public required MockFileSystem _mockFileSystem;

[TestInitialize]
public void TestInitialize()
{
    string bookmarksAsJson = @"[
       {
           ""Name"": ""Packt Publishing"",
           ""Url"": ""https://packtpub.com/"",
           ""Category"": ""Tech Books""
        },
        {
            ""Name"": ""Audi cars"",
            ""Url"": ""https://audi.ca"",
            ""Category"": ""See later""
        },
        {
            ""Name"": ""LinkedIn"",
            ""Url"": ""https://www.linkedin.com/"",
            ""Category"": ""Social Media""
        }
    ]";

    _mockFileSystem = new MockFileSystem(new Dictionary<string,
```

```
        MockFileData>
        {
            {@"bookmarks.json", new MockFileData(bookmarksAsJson)}
        });
    }
```

Fantastic! Now we know everything we need to know to implement meaningful and efficient tests for our CLI application.

But before closing this chapter, there is one last thing I want to discuss with you. I want to tell you how you can use these tests that you are implementing to identify and eliminate a bug.

How to hunt a bug

Tests play a key role in hunting down a bug and ensuring it doesn't come up again. To do that, you should follow a process.

Whenever a bug is discovered (by you or your team) or reported (by the user), you should write tests that reproduce it. These tests can be of different types. These are to ensure that the bug doesn't come up again.

Now, run your tests and you should notice that tests are failing. Tests that cover wider ranges (such as system or integration tests) will tell you in what component of the application the bug is happening. Then, more granular tests (such as unit tests) will tell you in what class and ultimately in what method the bug is hiding.

By using a smart combination of both wide-range and granular tests, you will be able to hunt down the bug. Keep in mind that breakpoints will be great allies.

And that's it! We are now armed with everything we need to enhance the quality of our application by leveraging software testing.

Summary

In this chapter, we learned why testing is an important step in the process of developing any application, including CLI applications. I like to call tests your safety nets: they ensure not only that your new features behave as expected but also that you don't accidentally introduce bugs in existing functionalities (we call these **regressions**). I highly encourage you to write effective tests and run them often.

We also explored the categories and roles of tests, and we learned techniques for testing applications and applied them to **Bookmarkr**, our CLI application.

Our application now has the required functionalities, and we ensured, through testing, that these functionalities behave as expected. It is now time to deliver the application to its users!

This is why, in the next chapter, we will explore different techniques that will allow us to package, distribute, and deploy our application.

Your turn!

Following along with the provided code is a great way to learn through practice.

A better way is by challenging yourself to achieve tasks. Hence, I challenge you to improve the Bookmarkr application by adding the following features.

Task #1 – Write the required unit tests for the remaining functionalities

In this chapter, we wrote tests only for the `link` and `import` commands. You are hence challenged to write tests for the other commands. You will have to figure out what test cases are to be considered and implement them.

Task #2 – Write integration tests for the sync command

The `sync` command deals with a database. For the purpose of unit testing, you can mock the database using NSubstitute. However, when implementing an integration test, you need a real database. You are then challenged to write integration tests for the `sync` command. You will have to provide a test database and use the appropriate connection string depending on whether the application is running in production or in testing mode.

11
Packaging and Deployment

Now that we have completed the development and testing of our application, it is time to release it to the world! We will need to package and deploy it in order to distribute it to (millions of) users worldwide.

Each platform (such as Windows, macOS, and Linux) has its own approach to distributing applications. Since .NET 8 is cross-platform, we can distribute Bookmarkr to even more users, no matter what platform they are using.

However, before we package and distribute the application, it is important that we test it on every target platform.

In this chapter, we will explore different packaging and deployment techniques that will help us achieve this goal. More specifically, we will do the following:

- Explore the different options when it comes to packaging and distributing a CLI application
- Learn how to package and distribute a cross-platform CLI application
- Learn how to deploy that CLI application to multiple platforms
- Learn how to manage versions of a distributed application

Technical requirements

The code for this chapter can be found in the GitHub repository accompanying this book, `https://github.com/PacktPublishing/Building-CLI-Applications-with-C-Sharp-and-.NET/tree/main/Chapter11`.

A bit of terminology

Throughout this chapter, you will come across the terms "packaging," "distribution," and "deployment." For those of you who are not familiar with these terms, here is a brief definition of each:

- **Packaging**: Packaging refers to the process of preparing the application for release. This includes bundling all necessary files, libraries, and resources into a single unit that can be easily installed or executed by our users. Effective packaging ensures that the application is compatible with various environments and simplifies the installation process. It often involves creating installers or archives that streamline the deployment of the application.
- **Distribution**: Distribution is the method by which a packaged application is delivered to users. This can involve various channels, including online downloads, physical media (such as USB drives), or cloud-based services (such as NuGet, NPM, WinGet, and `apt-get` repositories). The goal of distribution is to make the application accessible to its intended audience while ensuring that it reaches them in a secure and efficient manner.
- **Deployment**: Deployment is the mechanism by which the distributed application is installed and made operational on the user's computer. This can involve configuring settings, integrating with existing systems, and ensuring that all components work appropriately. Deployment can be done manually or can be automated through various tools and scripts. The aim is to allow users to access and utilize the application effectively.

As you may have figured out, making an application (including a CLI application) available to a user is a three-step process that can be summarized by this diagram:

Figure 11.1 – The process of making an application available to users

Now that we understand the terminology, let's start by exploring the available options when it comes to the packaging, distribution, and deployment of a CLI application.

Packaging and distribution options for CLI applications

When it comes to packaging a CLI application, several methods exist, and choosing the most appropriate one depends on the way we intend to distribute it.

The most common options are as follows:

- **MSI installer**: This option allows for a more traditional installation experience and can be achieved using tools such as WiX or Visual Studio Installer Projects. Keep in mind that this option only works for Windows. Hence, if we intend to distribute our CLI application to multiple platforms, this option may not be the best one.
- **.NET tool**: We can distribute our application as a library or tool by packaging it as a NuGet package. This allows users to install it through the `dotnet tool install` command. Since our CLI application is built using a version of .NET that is cross-platform, we can distribute it as a .NET tool to various platforms. The downside of this approach is in the installation mechanism: it requires the .NET CLI. This is great if our audience is developers or IT professionals but not that appropriate otherwise. We should only consider this approach if our CLI application is a developer or IT administrator tool, which is not the case with our bookmark management application.
- **Docker container**: This is also a great option for multiplatform distribution. A noticeable advantage of a Docker container is that it has a lower footprint on the local machine since no local installation is performed and limited access to the system is required. A Docker container is a self-contained file. However, as with the .NET tool option, this option mainly targets developers or IT administrators since users need to have knowledge of using Docker in order to deploy our application.
- **Platform-specific packaging**: All the major platforms provide a package management system. Linux is famous for its `apt-get` package manager, while macOS provides Homebrew and Windows provides WinGet. These options are great since users of each platform are familiar with them no matter their technical knowledge. This means that these distribution mechanisms don't just target developers and IT administrators but everyone! Once again, since our CLI application is built with cross-platform technology (.NET), we can use the same code and package it for distribution on each of these platforms.

As you can see, several packaging and distribution options are provided to us, and you can use whichever best suits your situation. In this chapter, we will explore the last three packaging and distribution options: .NET tool, Docker container, and WinGet (as a platform-specific packaging option).

Let's get started!

Packaging and distributing a CLI application

In this section, we will explore the subtleties of packaging and distributing our application, Bookmarkr, using three different options. We will take this opportunity to explain when each approach is most appropriate.

Option #1 – as a .NET tool

By packaging and distributing our application as a .NET tool, our users will be able to install it using the .NET CLI. It is, however, important that users ensure they have the appropriate .NET version installed to avoid version mismatches, which may cause unexpected behaviors in the application.

Step 1 – packaging

The first step is to modify the `.csproj` file to add properties that indicate that it should be packaged as a tool. These properties should be added to the `<PropertyGroup>` section:

```
<PackageId>bookmarkr</PackageId>
<Version>1.0.0</Version>
<Authors>Tidjani Belmansour</Authors>
<Description>Bookmarkr is a bookmarks manager provided as a CLI
application.</Description>
<PackAsTool>true</PackAsTool>
<ToolCommandName>bookmarkr</ToolCommandName>
<PackageOutputPath>./nupkg</PackageOutputPath>
<PackageLicenseExpression>MIT</PackageLicenseExpression>
<PackageReadmeFile>README.md</PackageReadmeFile>
<Copyright>Tidjani Belmansour. All rights reserved.</Copyright>
<PackageProjectUrl>https://github.com/PacktPublishing/Building-CLI-
Applications-with-C#-and-.NET</PackageProjectUrl>
<RepositoryUrl>https://github.com/PacktPublishing/Building-CLI-
Applications-with-C#-and-.NET</RepositoryUrl>
<PackageTags>.net cli;bookmark manager;.net 8</PackageTags>
```

Let's explain what we have just added here:

- `PackageId`: This represents the unique identifier for our package.
- `Version`: This indicates the version of our package. We will need to change this value when we need to package a newer version.
- `Authors`: This represents the author (or the list of authors) of a package.
- `Description`: This provides a brief description of what the application does.
- `PackAsTool`: Set to `true`, this indicates that the application should be packaged as a .NET tool.
- `ToolCommandName`: This is the name of the command users will type into their terminal to execute our application.
- `PackageOutputPath`: Since a .NET tool is packaged as a NuGet package, an `.nupkg` file is then generated. This property indicates where this file will be generated.

- `PackageLicenseExpression`: I chose to provide the code as an MIT license since it is a permissive license that allows for the reuse of the code in any project as long as the original copyright notice and license are included in all copies or substantial portions of the software.
- `PackageReadmeFile`: This points to a Markdown file in which we explain the purpose of the application, how to get started using it, and a link to its documentation, among other things. The content of this Markdown file will be displayed on the package page on the NuGet site for the user to read. You will find that file in the code repository.
- `Copyright`: This presents the copyright details of the application.
- `PackageProjectUrl`: This points to the home page of the project's website.
- `RepositoryUrl`: This points to the repository where the application's code resides.
- `PackageTags`: This presents a semicolon-delimited list of keywords that can be used when searching for packages.

To specify the location and how to treat the `README.md` file, we need to add the following XML code to the `.csproj` file:

```
<ItemGroup>
    <None Include="README.md" Pack="true" PackagePath="\"/>
</ItemGroup>
```

The second step is to package the application. This is achieved by running this command:

```
dotnet pack --configuration Release
```

Remember that it will be generated at the location indicated by the value of the `PackageOutputPath` property.

Our package is now ready for distribution.

Step 2 – distribution

The most common way to distribute a .NET tool is to provide it through the NuGet site, located at `https://www.nuget.org`.

So, let's head to the NuGet site and click on the **Sign in** link in the top-right corner of the page:

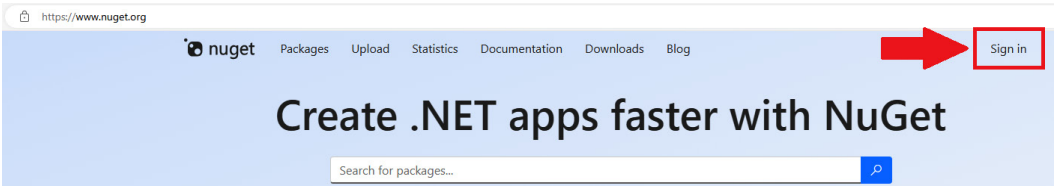


Figure 11.2 – Signing in to the NuGet site

I will be signing in with my personal account and granting the required permissions to the NuGet site, as shown here:

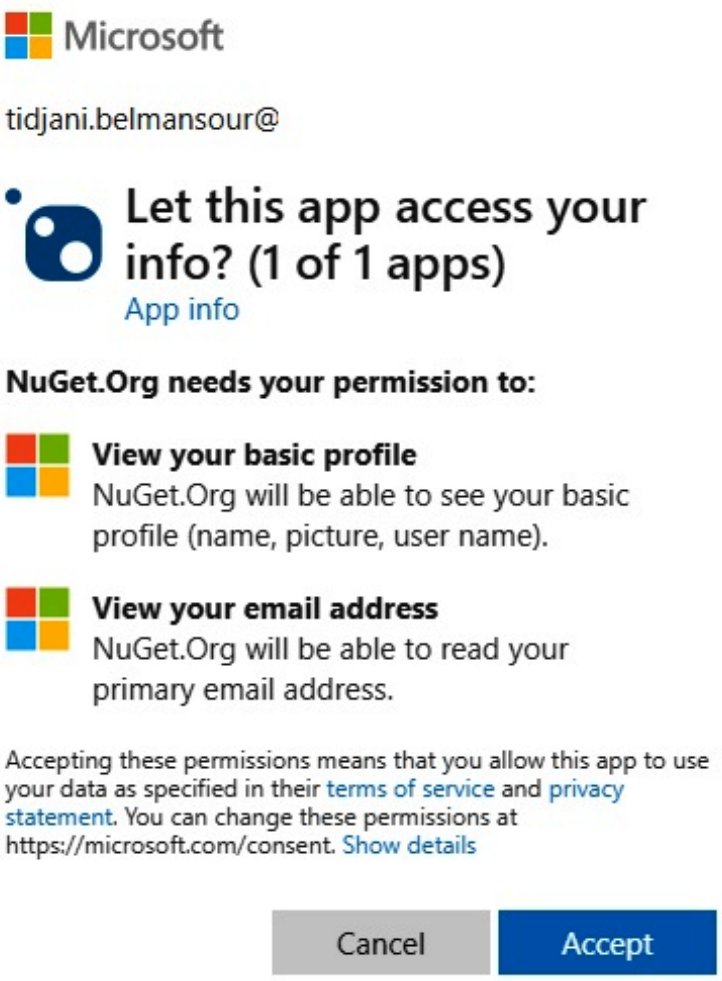


Figure 11.3 – Granting required permissions to the NuGet site

Since this is the first time I have signed in with this account, the NuGet site asks me to provide a username:

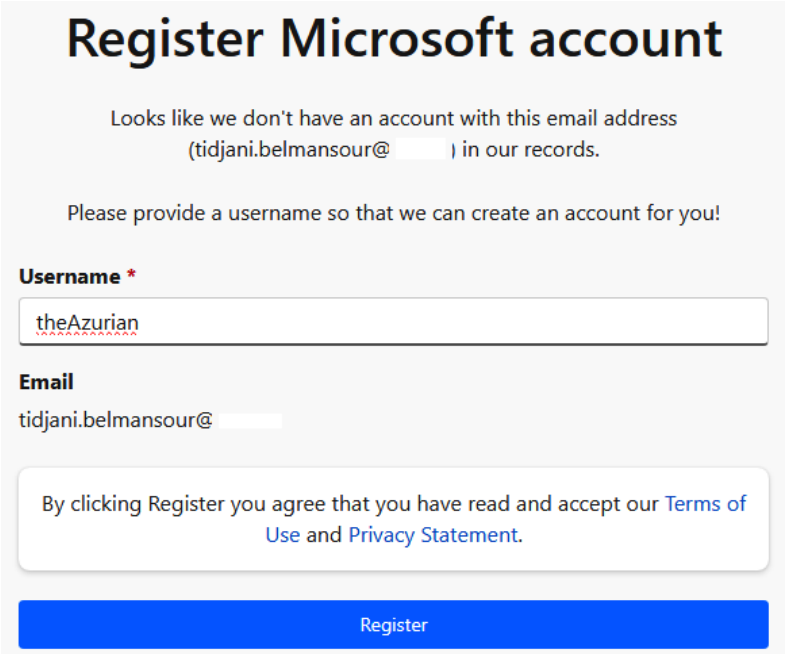


Figure 11.4 – Choosing a username for the NuGet site

And that's it! As a publisher of NuGet packages, I am now all set, and I can start uploading my packages:

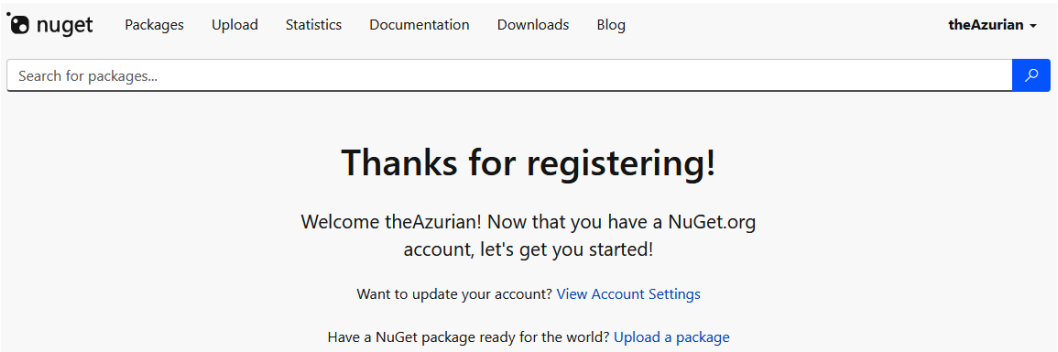


Figure 11.5 – All set as a NuGet package publisher

Let's now upload our package!

All we need to do is to click on **Upload** and browse to the `.nupkg` file we generated earlier. The package is then analyzed and the validation results are presented:

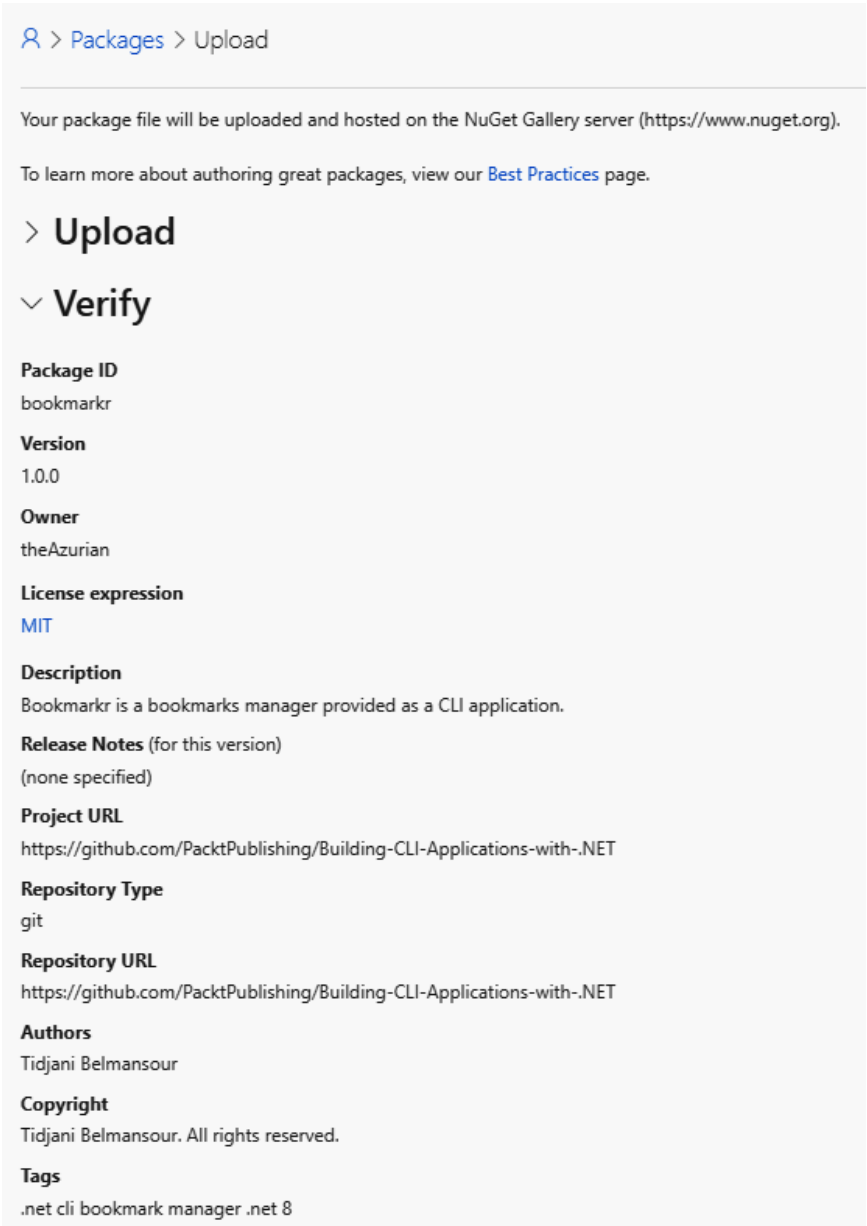


Figure 11.6 – Uploading a package to the NuGet website

Since our package is valid, we can submit it by clicking the **Submit** button at the bottom of the page.

Once uploaded, it usually takes around 15 minutes (but can sometimes take up to an hour) for the package to be validated and indexed before it appears in search results:

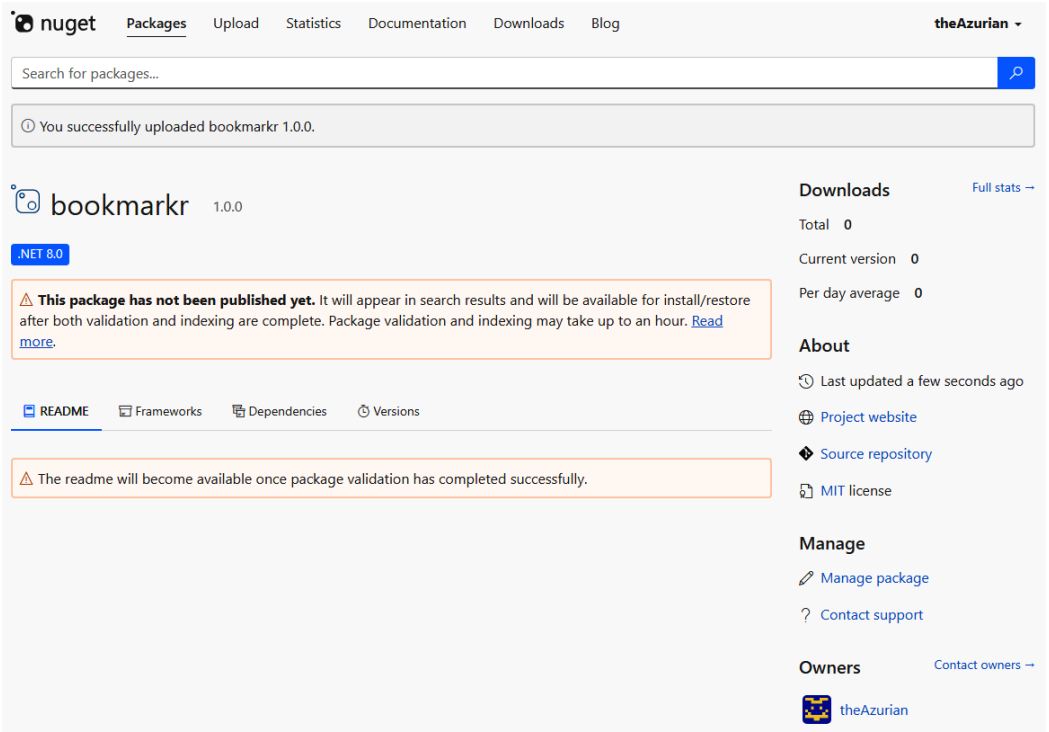


Figure 11.7 – Package awaiting validation and indexing

Once the package validation and indexing have been completed, it will appear on the NuGet website just as any other NuGet package:

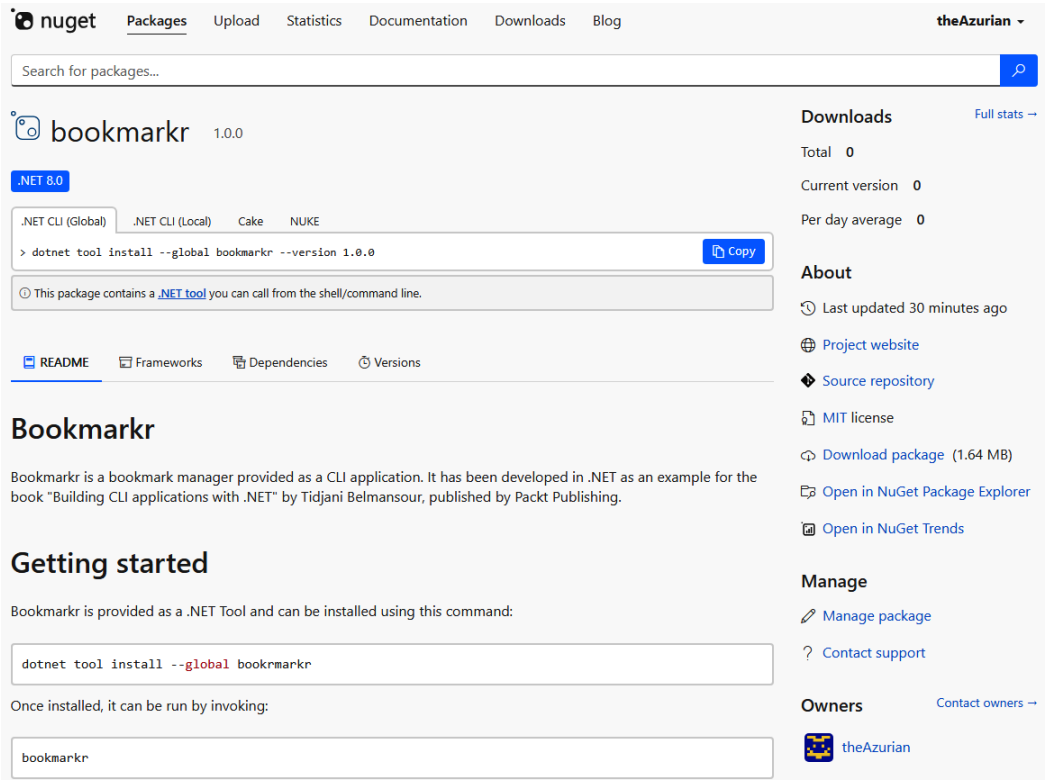


Figure 11.8 – Bookmarkr is available on the NuGet website!

Now that our application can be found by users, let's see how it can be deployed.

Step 3 – deployment

A user can deploy (that is, install) our application on their machine very easily by typing this command:

```
dotnet tool install --global bookrmarkr
```

After the installation is complete, the user can execute our application by typing this command:

```
bookrmarkr
```

And that's it! We've packaged, distributed, and deployed Bookmarkr as a .NET tool.

Let's now see how we can deliver Bookmarkr to our users as a Docker container.

Option #2 – as a Docker container

Packaging and distributing our application as a Docker container allows our users to install and use our application by reducing the footprint of the application on their environment (i.e., operating system and data).

Step 1 – packaging

If you are familiar with containers, you may already know that in order to create a container image, a Dockerfile is required.

A Dockerfile is a file with no extension that should be located at the root of the project directory. For our application, its content is the following:

```
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
WORKDIR /app
COPY *.csproj ./
RUN dotnet restore
COPY . ./
RUN dotnet publish -c Release -o out

FROM mcr.microsoft.com/dotnet/runtime:8.0
WORKDIR /app
COPY --from=build /app/out .
ENTRYPOINT [«dotnet», «bookmarkr.dll»]
```

In essence, this file instructs Docker to build and publish the application (the first six lines of it), and then to build a container image out of the published application (the remaining five lines).

Before we run the command to actually build the image, we need to ensure that both Docker Desktop and **Windows Subsystem for Linux** (**WSL**) are installed and running. Note that Docker Desktop requires administrator privileges on the local machine.

Docker Desktop can be installed using the following command:

```
winget install Docker.DockerDesktop
```

WSL can be installed using the following command (Windows needs to be rebooted after WSL is installed):

```
wsl --install
```

The command for building a Docker image is as follows:

```
docker build -t bookmarkr .
```

The `-t bookmarkr` parameter is used to tag the Docker image to be generated with a name and an optional version number (more on that later).

The last dot character is neither a typo nor is it optional. It refers to what we call the **build context**. More specifically, it instructs Docker where to look for the Dockerfile, which, in this case, is the current directory.

The operation should take about five minutes, and once it is complete, the Docker image will be created, and it can be retrieved by typing the following:

```
docker images
```

Note that the container image has been generated on our local machine. We should, however, distribute it through a location that everyone can easily find.

Step 2 – distribution

The most common way to distribute Docker images is through Docker Hub.

To do that, we need to head to the Docker Hub portal, located at `https://hub.docker.com`. If you don't already have a Docker Hub account, you can create one from there. I already have such an account, and my username is `theAzurian`.

So, let's follow the steps to push our local Docker image to Docker Hub.

First, let's log in to our Docker Hub account using this command:

```
docker login -u theazurian -p ****
```

I'm passing my **Personal Access Token** (**PAT**) to the `-p` parameter. This PAT was created through the Docker Hub portal.

Next, we need to tag the image to include the Docker Hub username of its author, the name of the application, and its version, such as the following:

```
docker tag bookmarkr theazurian/bookmarkr:1.0.0
```

Finally, we need to push the tagged image to Docker Hub using this command:

```
docker push theazurian/bookmarkr:1.0.0
```

We can ensure that the image has effectively been pushed to Docker Hub by heading to the portal and looking for it in our Docker Hub profile:

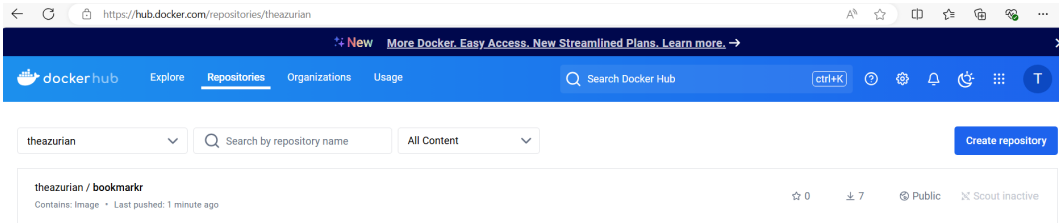


Figure 11.9 – Bookmarkr is available in the Docker Hub portal!

We can also perform a search for it in the Docker Hub portal:

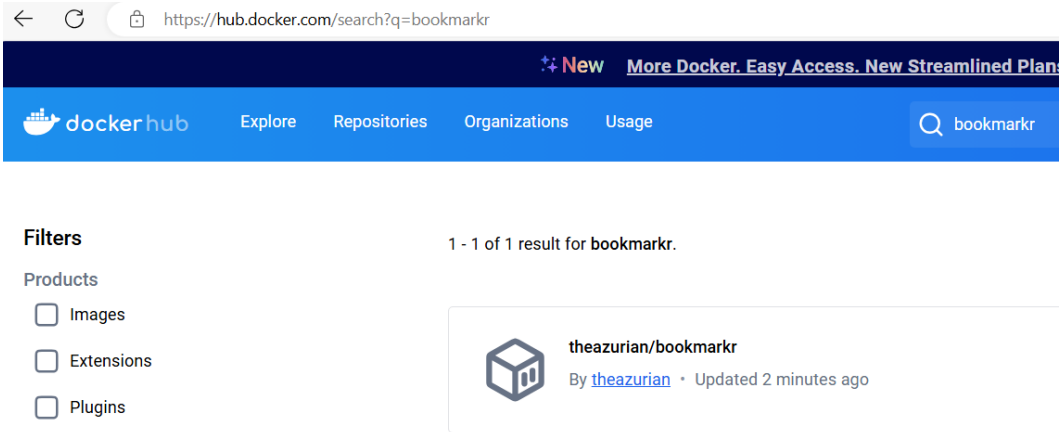


Figure 11.10 – Searching for Bookmarkr in the Docker Hub portal

Our application can now be found by our users. Let's see how it can be deployed.

Step 3 – deployment

In order for a user to run Docker on a Windows machine, they also need to have both Docker Desktop and WSL installed.

Docker Desktop can be installed using the following command:

```
winget install Docker.DockerDesktop
```

WSL can be installed using the following command (Windows needs to be rebooted after WSL is installed):

```
wsl --install
```

Now, our user can obtain the application from Docker Hub by typing this command:

```
docker pull theazurian/bookmarkr:1.0.0
```

They can execute it by typing this command:

```
docker run theazurian/bookmarkr:1.0.0
```

Bookmarkr can then be run on the user's computer as a Docker container:

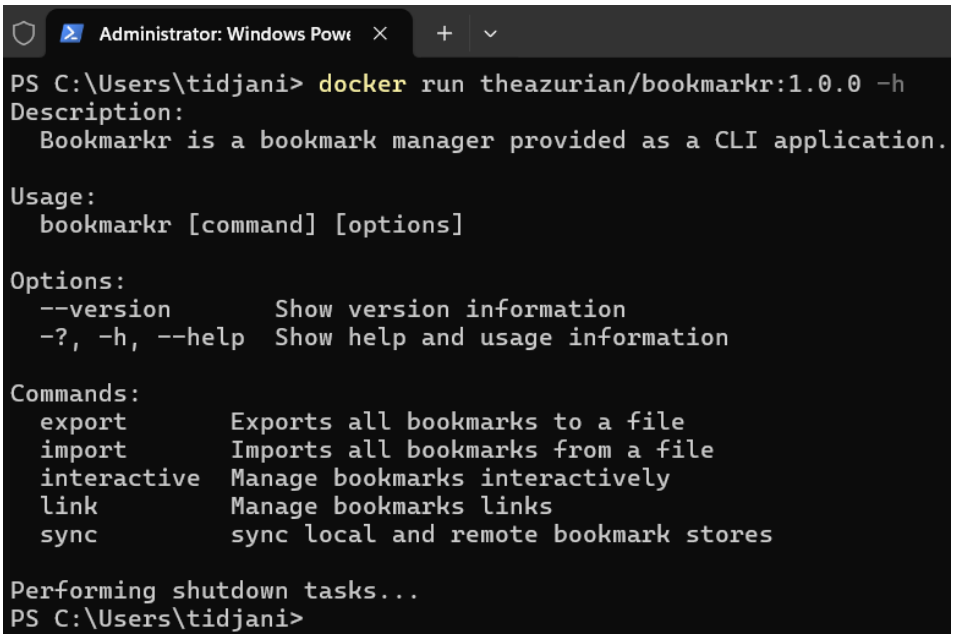


Figure 11.11 – Bookmarkr running as a Docker container

And that's it! We packaged, distributed, and deployed Bookmarkr as a Docker container.

Let's now see how we can deliver Bookmarkr to our users as a WinGet package.

Option #3 – as a WinGet package

By packaging and distributing our application as a WinGet package, we allow our users to install it as any other application they have installed using WinGet, Microsoft's package manager.

Packaging

To package a .NET CLI application for distribution through WinGet (the official Windows package manager), we first need to create a manifest file.

Although it is possible to manually create and submit the manifest to the WinGet package repository on GitHub (`https://github.com/microsoft/winget-pkgs`), the easiest way to do so is using the `WingetCreate` CLI.

Let's first install it using this command:

```
winget install wingetcreate
```

Before we create the new manifest, we first need to build our CLI application as a self-contained .NET application using this command:

```
dotnet publish -c Release -r win-x64 -p:selfcontained=true
-p:IncludeNativeLibrariesForSelfExtract=true -p:PublishSingleFile=true
```

Let's take a closer look at this command:

- `-c Release`: Since this is a production-ready version of the application, we want to publish it using the Release configuration to ensure that it is optimized for performance.
- `-r win-x64`: Since WinGet is the package manager for Windows (and Windows only), we specify the target runtime to be the 64-bit version of Windows.
- `-p:selfcontained=true`: A self-contained application already includes the .NET runtime, so the user's machine doesn't need to have it installed. The application will then carry everything it needs to run, including the runtime, libraries, and dependencies.
- `-p:IncludeNativeLibrariesForSelfExtract=true`: This ensures that platform-specific libraries, along with unmanaged native libraries, are included in the published application. This is useful if we use some specific Serilog sinks and for the SQLite library.
- `-p:PublishSingleFile=true`: This instructs .NET to bundle everything (including the application code, the .NET runtime, and the dependencies) into a single executable file. While this makes it more convenient for distribution (because we are distributing a single file), it results in a file that is larger in size than framework-dependent publishing.

The application will be generated in the `bin\Release\net8.0\win-x64\publish` directory.

Next, we will upload it to a location that should be accessible to the WinGet tool. It is common to pick a remote, publicly accessible, read-only location. I decided to use an Azure Storage account. Hence, the location of the executable will be `https://bookmarkr.blob.core.windows.net/releases/1.0.0/`.

About GitHub releases

If your application is built as an open source project on GitHub, you will likely make your executable versions available as releases. However, releases on GitHub follow certain guidelines that go way beyond the scope of this book. If this is a topic you are interested in, I recommend that you explore these guidelines by visiting `https://github.com/github/docs/blob/main/content/repositories/releasing-projects-on-github/about-releases.md`.

Let's now create our manifest files! We can do so using this command:

```
wingetcreate new https://bookmarkr.blob.core.windows.net/
releases/1.0.0/bookmarkr.exe
```

The tool will ask a series of questions required to generate the manifest files. Here is an example of what it looks like:

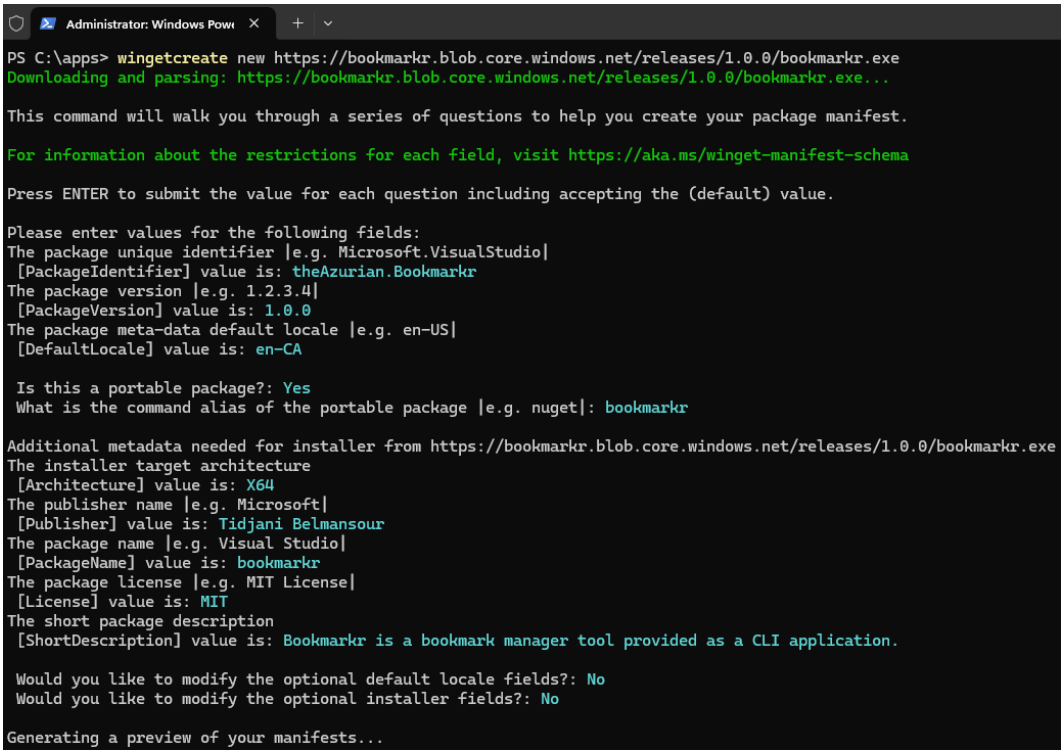


Figure 11.12 – Generating the WinGet manifest files

There will be three files generated:

- **Version manifest (theAzurian.bookmarkr.yaml)**: Contains metadata about the specific version of the application being packaged.
- **Installer manifest (theAzurian.bookmarkr.installer.yaml)**: Details the installation specifics of the application.
- **Default locale manifest (theAzurian.bookmarkr.locale.en-CA.yaml)**: Defines localization settings for the application. It ensures that users receive a version of the application that is appropriately localized, enhancing user experience by presenting information in their preferred language.

I personally like to keep these files inside my Visual Studio project, inside the following folder structure that I create at the root of the project:

```
/manifests/ApplicationName/Version
```

So, in our case, this folder structure will look like this:

```
/manifests/Bookmarkr/1.0.0
```

Before we submit our manifest to the WinGet team, it is recommended that we test it locally to ensure that it works as expected. This is important as submitting our package could lead to a delay during the WinGet approval process if the manifest contains issues.

To do this, we first need to activate the ability to install applications from local manifests. This can be done by executing the following command in a terminal running as administrator:

```
winget settings --enable LocalManifestFiles
```

Next, we run this command, providing the path to the `manifests.json` file:

```
winget install --manifest "C:\code\Chap11\bookmarkr\manifests\
Bookmarkr\1.0.0\"
```

As we can see, the application is installed and runs as expected:

```
PS C:\apps> winget install --manifest "C:\apps\manifests\t\theAzurian\Bookmarkr\1.0.0"
Found bookmarkr [theAzurian.Bookmarkr] Version 1.0.0
This application is licensed to you by its owner.
Microsoft is not responsible for, nor does it grant any licenses to, third-party packages.
Downloading https://bookmarkr.blob.core.windows.net/releases/1.0.0/bookmarkr.exe
  ████████████████████████████████  67.2 MB / 67.2 MB
Successfully verified installer hash
Starting package install...
Command line alias added: "bookmarkr"
Path environment variable modified; restart your shell to use the new value.
Successfully installed
PS C:\apps> bookmarkr
Hello from the root command!
Performing shutdown tasks...
```

Figure 11.13 – Testing the WinGet package locally before submission

We are now ready to submit our manifest to the WinGet team!

Submitting our manifest to the WinGet package repository requires us to generate a PAT for our GitHub account. We can do this using the `wingetcreate token` command, or we can skip this step and, when submitting the manifest, `wingetcreate` will prompt us to authenticate to our GitHub account. Let's do it this way!

Let's run the following command:

```
wingetcreate submit "C:\code\Chap11\bookmarkr\manifests\
Bookmarkr\1.0.0\"
```

This will open the browser and take us to the GitHub sign-in page. We will have to log on to our account. Once done, we will need to provide the required authorization:

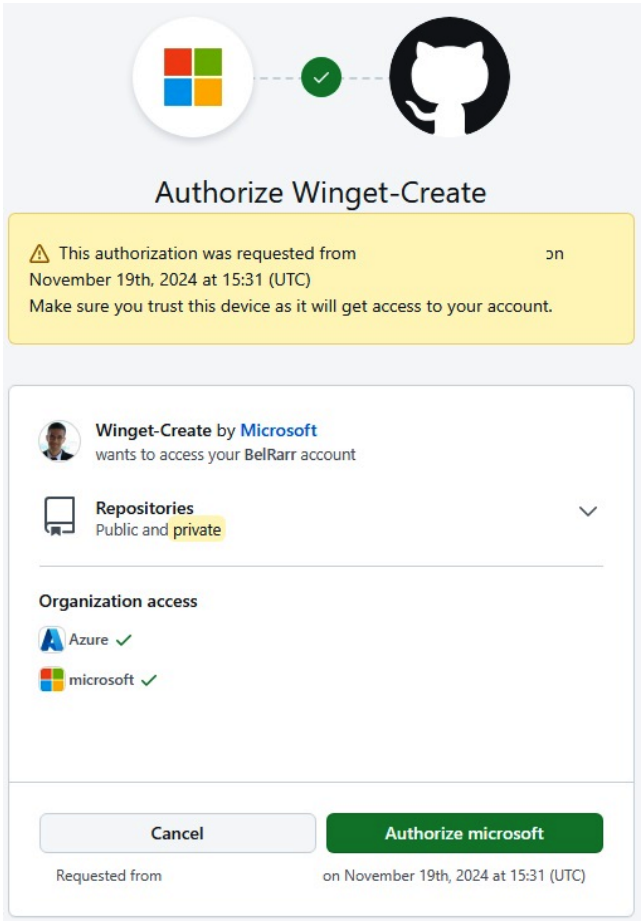


Figure 11.14 – Authorizing WingetCreate for our GitHub account

It will then take us to the Pull Request page where we can follow its progress. After about 30 minutes, the pull request is completed, and the package is available for our users to install.

Users can then install Bookmarkr using WinGet by typing this command:

```
winget install --id theAzurian.Bookmarkr
```

And voilà! We have packaged, distributed, and deployed Bookmarkr as a WinGet package.

So, we have seen three different approaches to packaging, distributing, and deploying our application. But how do we manage multiple versions of that application? That's what we are going to explore in the next section.

Managing versions of the application

All the options that we presented earlier provide version management mechanisms. Version management is as important as the packaging and distribution mechanism that we select.

As our application evolves and new features are added, modified, or removed, we want to offer a way for our users to consume these updates at their convenience. That is where versioning comes into play.

Currently, only a single version of our application is distributed. We can hence install it by either omitting its version number or explicitly indicating it.

But what if we update the application? How can we distribute the new version? And what if the new version introduces a bug and we need to roll back to the previous version?

Let's explore how we can achieve this for each of the distribution methods that we covered earlier.

Semantic versioning primer

Before we dive into managing different versions of an application, let us start by introducing **semantic versioning**.

If you are familiar with this approach to versioning applications, you know that it is probably the most common and widely adopted approach in the industry. If you haven't heard about it before, let me give you a quick introduction. If you want to go deeper in your exploration of semantic versioning, I recommend that you visit its official website at `https://semver.org`.

In essence, semantic versioning expresses a version number using this format:

```
Major.Minor.Patch
```

Here, we have the following:

- **Major**: Indicates that this version of the application contains breaking changes that are incompatible with the previous major version
- **Minor**: Indicates that this version of the application only adds new functionalities that are backward compatible with the previous versions of the same major version
- **Patch**: Indicates that this version of the application contains bug fixes that are backward compatible with the previous versions of the same major version

Each part is represented as a number that is incremented with each new version.

Now that we understand semantic versioning, let's use it to manage different versions of our application.

Managing versions of a .NET tool

If you ran the version of Bookmarkr that we provided as a .NET tool, you will certainly have noticed that it returns the following error message:

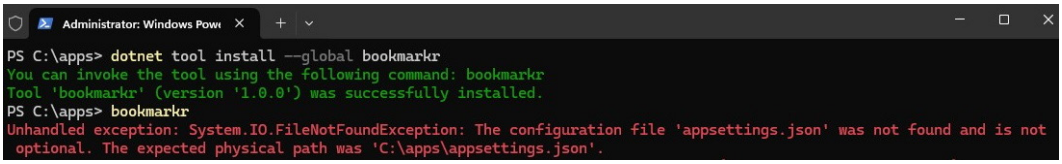


Figure 11.15 – Bookmarkr as a .NET tool fails to execute

Let's then fix the problem and distribute a new version.

The error comes from the fact that the `appsettings.json` file is not being made part of the package when the `dotnet pack` command is executed.

Fortunately, fixing this problem is straightforward. Locate the following entry in the `.csproj` file:

```
<None Update="appsettings.json">
```

Replace it with the following:

```
<None Update="appsettings.json" Pack="true" PackagePath="\">
```

Now, since the new version we are about to package and distribute only provides a bug fix, we shall increment the patch number, so the version number looks like the following:

```
<Version>1.0.3</Version>
```

We can now package and distribute the new version by following the same steps that we described earlier.

However, before we distribute it, it is recommended to test it locally using the following command:

```
dotnet tool install --global bookmarkr --version 1.0.3 --add-source
"C:\code\Chap11\bookmarkr\nupkg"
```

The `--add-source` parameter allows us to specify a location from which the package will be deployed. Here, I am specifying the path where the NuGet package was generated on my local machine.

After ensuring that this new version works successfully on the local machine, we can proceed with pushing it to the NuGet website.

Users can get a specific version of the tool by providing its version number as a parameter. In this case, this could be achieved by typing this command:

```
dotnet tool install --global bookmarkr --version 1.0.3
```

Or, they can simply type the following command to get the latest version:

```
dotnet tool install --global bookmarkr
```

Once this command is executed, the previous version of the tool will be replaced by the new one.

By running the new version of the application, we can now see that the error is resolved:

```
PS C:\apps> dotnet tool install --global bookmarkr
Tool 'bookmarkr' was reinstalled with the stable version (version '1.0.3').
PS C:\apps> bookmarkr -h
Description:
  Bookmarkr is a bookmark manager provided as a CLI application.

Usage:
  bookmarkr [command] [options]

Options:
  --version       Show version information
  -?, -h, --help  Show help and usage information

Commands:
  export       Exports all bookmarks to a file
  import       Imports all bookmarks from a file
  interactive  Manage bookmarks interactively
  link         Manage bookmarks links
  sync         sync local and remote bookmark stores

Performing shutdown tasks...
PS C:\apps>
```

Figure 11.16 – Bookmarkr as a .NET tool running successfully

And that's it! We now know how to manage versions of a .NET tool.

Let's see how we manage versions of a Docker container.

Managing versions of a Docker container

As you may have noticed, when we pushed the Docker image to Docker Hub, we tagged it with a version number. Hence, if we want to distribute a new version, we can tag the new image with a different version number.

However, if you are familiar with Docker, you may know about working with containers without providing their version number or using the `latest` tag.

When distributing multiple versions of a container, it is important to indicate which one of these is the latest version, by tagging that version using the `latest` tag.

So, let's assume that we are distributing a new version of Bookmarkr and that we want to indicate that this new version is the latest one. We can achieve this as follows:

```
docker build -t theazurian/bookmarkr:2.0.0 .
docker tag theazurian/bookmarkr:2.0.0 theazurian/bookmarkr:latest
docker push theazurian/bookmarkr:2.0.0
docker push theazurian/bookmarkr:latest
```

Let's explain these commands:

- The first one creates a new Docker image that is tagged with version 2.0.0
- The second one tags version 2.0.0 as the latest version
- The third command pushes the image tagged with version 2.0.0 to Docker Hub
- The fourth one pushes the image tagged with the latest version to Docker Hub

The fact that we push the same image using two different tags allows our users to get it (using the `docker pull` command) with or without specifying its version number. Hence, as we keep updating the application and pushing new Docker images, we will tag the newest version with the `latest` tag. Should that version contain a bug, we can direct our users to a previous version by tagging it with the `latest` tag.

If we head to the Docker Hub portal, we will see that the new version has been successfully pushed. Notice that there are two versions of the same image: one with version number 2.0.0 as a tag, and the other with the `latest` tag.

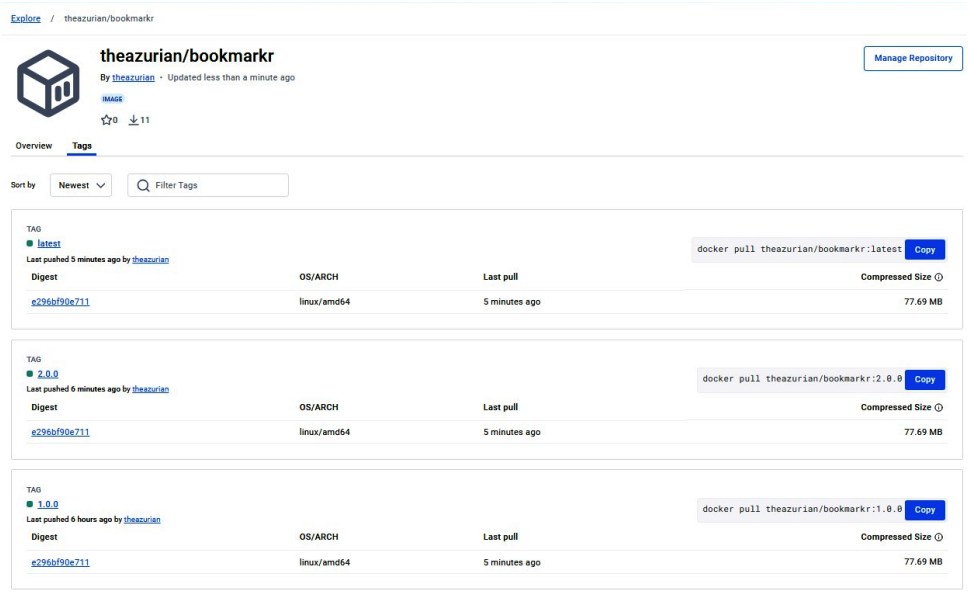


Figure 11.17 – New version of the Docker image pushed to Docker Hub

On the user's side, they can get a specific version by explicitly mentioning its tag, as follows:

```
docker pull theazurian/bookmarkr:2.0.0
```

Or, they can get the latest version (i.e., the version tagged as `latest`) by omitting the tag, as follows:

```
docker pull theazurian/bookmarkr
```

The user will then see this:

```
PS C:\Users\tidjani> docker pull theazurian/bookmarkr
Using default tag: latest
latest: Pulling from theazurian/bookmarkr
Digest: sha256:e296bf90e711f7a0ae071bf0ec242d7f278b5f5a1aaa5d369fda4e29e161f95d
Status: Image is up to date for theazurian/bookmarkr:latest
docker.io/theazurian/bookmarkr:latest
PS C:\Users\tidjani> docker run theazurian/bookmarkr -h
Description:
  Bookmarkr is a bookmark manager provided as a CLI application.

Usage:
  bookmarkr [command] [options]

Options:
  --version       Show version information
  -?, -h, --help  Show help and usage information

Commands:
  export       Exports all bookmarks to a file
  import       Imports all bookmarks from a file
  interactive  Manage bookmarks interactively
  link         Manage bookmarks links
  sync         sync local and remote bookmark stores

Performing shutdown tasks...
PS C:\Users\tidjani>
```

Figure 11.18 – Running the new version of the Docker container

And that's it! We now know how to manage versions of a Docker image.

Let's see how we manage versions of a WinGet package.

Managing versions of a WinGet package

In order to submit a new version of the application, following an update to the application's code or functionalities, we first need to update the version number in the `.csproj` file (the `<Version>` element).

Next, we need to publish the application again using the same command we saw earlier:

```
dotnet publish -c Release -r win-x64 -p:selfcontained=true
-p:IncludeNativeLibrariesForSelfExtract=true -p:PublishSingleFile=true
```

We then need to upload the resulting binaries to our distribution location, which is our Azure Storage account, keeping in mind that we should create a new directory for the new version. The path will then look like this for version 1.0.3:

```
https://bookmarkr.blob.core.windows.net/releases/1.0.3/bookmarkr.exe
```

The next step is to update the manifest using this command:

```
wingetcreate update theAzurian.Bookmarkr --version 1.0.3 https://
bookmarkr.blob.core.windows.net/releases/1.0.3/bookmarkr.exe
```

The new manifest is then generated and is ready for submission.

However, as we previously learned, it is always good practice to test the new version locally before submitting it. To do that, we will execute the same command as we did previously:

```
winget install --manifest "C:\code\Chap11\bookmarkr\manifests\
Bookmarkr\1.0.3\"
```

Once the tests are successful, we submit the new version using this command:

```
wingetcreate submit "C:\code\Chap11\bookmarkr\manifests\
Bookmarkr\1.0.3\"
```

The remaining steps are similar to those we followed when submitting the initial version of the application.

Once the new version has been approved and added to the WinGet package repository, users can find it and install it. They can install the latest version using this command:

```
winget install --id theAzurian.Bookmarkr
```

Or, they can install a specific version by passing the desired version number as a parameter to the command, as follows:

```
winget install --id theAzurian.Bookmarkr --version 1.0.3
```

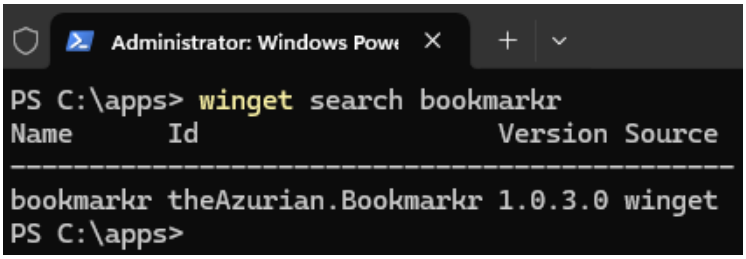


Figure 11.19 – The updated version of Bookmarkr is available in WinGet

And that's it! This is how we manage multiple versions of a WinGet package.

Summary

In this chapter, we learned how to package and deploy Bookmarkr onto different platforms in order to distribute it to users all around the world, no matter their platform of choice, be it Windows, Linux, or macOS.

This is quite a milestone we achieved, from the inception of the idea of our CLI application all the way to getting it into the hands of millions of users worldwide. Let's take a moment to celebrate this achievement and be proud of ourselves. Congratulations! 🎉

However, some users are telling us that the application is sometimes slow. We haven't experienced these performance issues since we are running on fast and powerful computers, but that is not the case with all our users. Although we could simply specify the minimum requirements to run Bookmarkr, we don't want to limit the number of users that can benefit from and be able to use it. So, we have decided to see whether there is something we can do.

In the next chapter, we will explore different techniques that will allow us to optimize the performance of our application.

Your turn!

Following along with the provided code is a great way to learn through practice.

An even better way is by challenging yourself to complete tasks. Hence, I challenge you to improve the Bookmarkr application by adding the following features.

Task #1 – allowing Linux users to install Bookmarkr using apt-get

Currently, Bookmarkr can be deployed to Windows using WinGet. However, this doesn't work on Linux, where Linux users typically use `apt-get` for deploying applications. You are thus challenged to distribute Bookmarkr as an `apt-get` package so Linux users can also enjoy using it.

Task #2 – allowing macOS users to install Bookmarkr using Homebrew

The same applies to macOS users: they typically install their applications using the `brew` command. You are thus challenged to distribute Bookmarkr as a Homebrew formula.

Part 5: Advanced Techniques and Best Practices

In this part, you will explore crucial aspects of CLI application development that enhance performance, security, and functionality. You'll delve into performance optimization and tuning techniques, learning how to profile your CLI applications, identify bottlenecks, and implement efficient algorithms and data structures. This includes strategies like caching, load balancing, and code refactoring to improve execution speed and resource utilization. Next, you'll focus on security considerations specific to CLI applications, covering best practices and protection against common vulnerabilities. You'll learn how to implement strong authentication mechanisms, use encryption for sensitive data, and follow the principle of least privilege in your CLI tools. Finally, you'll explore additional resources and libraries that will allow you to dive deeper into the various concepts and techniques presented in this book.

This part has the following chapters:

- *Chapter 12, Performance Optimization and Tuning*
- *Chapter 13, Security Considerations for CLI Applications*
- *Chapter 14, Additional Resources and Libraries*

12
Performance Optimization and Tuning

Performance separates applications that are used and loved from those that are uninstalled and forever forgotten.

It is not enough to have an application that responds to users' needs. To be used frequently (and likely daily), an application needs to bootstrap and perform tasks quickly.

This speed and responsiveness directly impact user satisfaction, as people have increasingly high expectations for digital experiences. Studies have shown that even small delays in load times or task completion can significantly reduce user engagement and overall satisfaction.

In this chapter, we will discuss different areas where performance can be improved and what techniques we can use to achieve this. More specifically, we will cover the following:

- The different areas to be considered to improve application performance
- How to instrument an application to identify performance problems
- How to improve your application performance

Technical requirements

The code for this chapter can be found in the GitHub repository accompanying this book, `https://github.com/PacktPublishing/Building-CLI-Applications-with-C-Sharp-and-.NET/tree/main/Chapter12`.

Performance optimization areas

The performance of an application is not only a matter of code. It is a series of fine tuning, at different levels, that helps achieve performance efficiency.

Performance optimization therefore occurs in areas such as the following:

- **Application design and architecture**: The longer the path you must walk, the longer it takes you to get to your destination. As I always tell my customers, you may run twice as fast as me, but if your path is twice as long as mine, we will arrive at our destination at the same time. The idea here is that using performant frameworks and libraries is of little use if your architecture is not efficient. Too often, I see architectures that are over-decoupled, with too many hops and context switching, leading to applications that are not performant and slow. The key is to build an architecture that balances performance with the optimal level of decoupling. From a design perspective, I often see designs that can be improved (leading to more efficient and performant applications) by finding a shorter path to achieve a goal. Of course, you won't do that for every piece of code, but you will want to focus your attention on hot paths, namely paths that are often used by your users. There may be no point in optimizing the performance of a functionality that is used by one user once a year. You must find a balance between the cost of the optimization effort (it takes time, so it has a cost) and the benefit you are expecting from it.
- **Infrastructure**: If we host the application on an infrastructure, we must ensure that this infrastructure is efficient and has been optimized to maximize the throughput of the application while minimizing its latency. However, in the context of CLI applications, the application runs on the user's computer, so we might be tempted to say that there is nothing to do here, but we would be wrong! There are tuning tasks we can perform that will positively impact performance. For example, we can reduce resource utilization so that running the application on the user's computer will consume the least amount of resources and therefore will be executed efficiently, even if the computer is running other applications side by side or has little horsepower.
- **Frameworks and libraries**: Of course, using efficient and performant frameworks and libraries helps to improve the application's performance. For example, every new release of .NET promises better performance. Hence, upgrading the .NET version can be an easy way to improve the performance of our application. The same goes for the libraries that we use: some are known to have better performance than others.
- **Coding practices**: The last piece of the puzzle is the coding practices. We have already mentioned hot spots and hot paths, but coding practices also include using the most appropriate data structures.

Before we start optimizing our application's performance, we need to instrument it and identify its hot spots and hot paths.

Instrumenting .NET applications

Multiple tools exist to help us instrument.NET applications. The main difference between these tools is their scope of action.

Nevertheless, a key benefit of instrumentation is the ability to detect memory leaks and identify slow code paths.

Instrumentation can be achieved both during the development phase and continuously, while the application is running in production.

Development-time profiling	Visual Studio Diagnostic Tools, `BenchmarkDotNet`, `dotTrace`, `dotMemory`, and `PerfView` are great for profiling CPU, memory leaks and allocation, and application performance.
Production-time monitoring	Azure Application Insights, AppDynamics, and New Relic help monitor and diagnose performance issues in real time in production environments.

Table 12.1 – Some popular instrumentation tools

You may have noticed the terms *"profiling"* and *"monitoring."* There are some key differences between them:

- Profiling provides a detailed, granular view of an application's performance, often focusing on specific code sections or methods. This includes CPU usage per functionality or method, memory allocation, execution time, and method call frequency and duration.
- Monitoring is usually done in production and provides an overview of the application's health, looking at broader performance trends and operational data over time rather than focusing on individual code paths. This includes CPU and memory usage across the entire application, error rates (exceptions, failures), response times and throughput (e.g., how long requests take, how many requests per second), and the application's resource usage (disk I/O, network usage, etc.).

Because a CLI application runs on the user's computer, it may be harder to monitor it. It requires permission from the user to collect the necessary data, usually at frequent intervals. We may then expect the user to refuse to share telemetry data, and therefore monitoring may not be possible.

While it is important to know the tools that help us instrument our applications, it is equally important to understand where to use them, in other words, how to identify these areas that may be good candidates for performance optimization. In that regard, it is important to be able to identify hot spots and hot paths.

Hot spots versus hot paths

This is not the first time in this chapter that I have mentioned hot spots and hot paths. However, I haven't taken the time to explain them. Let's fix this right away!

A **hot spot** is an area of intense activity in the code, typically referring to frequently executed methods that consume a significant amount of execution time. Therefore, a hot spot represents a potential optimization target for improving the overall performance of the application.

A **hot path** refers to an execution path through the code that is frequently taken and therefore contributes *significantly* to the application's runtime. Hot paths can help locate inefficiently used resources, such as memory usage and allocation.

The question that may arise here is *"what process can we follow to identify the application's hot spots and hot paths?"*

Identifying the application's hot spots and hot paths

Fortunately, identifying an application's hot spots and hot paths does not have to be done by shots in the dark. Instead, we can follow a structured process that consists of three steps: profiling, analysis, and optimization. If monitoring is implemented, it will serve as an input for that process, since this process should be performed periodically to ensure optimal performance of the application.

The process is described in the following table:

Step	What to do
1. Profiling and data collection	• Use performance profilers to gather data about your application's execution. A library such as `BenchmarkDotNet` can collect detailed information about CPU usage, memory consumption, and execution times. • Collect metrics on method execution times, resource usage, and frequency of calls to identify performance bottlenecks.
2. Analysis and identification	• Analyze the profiler output and find: ▪ Methods with high execution times ▪ Frequently called methods ▪ Areas of high CPU or memory usage ▪ Long-running database queries or I/O operations • Look for patterns in the data that indicate potential hot spots or hot paths: ▪ Methods that consume a disproportionate amount of resources ▪ Execution paths that are frequently taken and contribute significantly to the overall runtime

Step	What to do
3. Optimization	• Once hot spots and hot paths are identified, implement optimizations targeting these areas. • Use benchmarking tools such as `BenchmarkDotNet` to measure and compare the performance of the code before and after optimization to assess the gain in performance. You may also measure and compare different implementations to identify the most optimal one.

Table 12.2 – Identifying hot spots and hot paths

We mentioned that `BenchmarkDotNet` can help us profile our application. Then, it is time to learn how to use it.

Profiling Bookmarkr with BenchmarkDotNet

Although `BenchmarkDotNet` is considered to be a benchmarking library (that is, it is used to compare different implementation alternatives against a baseline to identify which one is the most performant), when used strategically, it can also identify hot spots and hot paths in our code.

Let's see how we can leverage this library to profile our CLI application.

The first thing we need to do is to reference the `BenchmarkDotNet` library. This can be achieved by executing the following command:

```
dotnet add package BenchmarkDotNet
```

The next step is to configure benchmark collection and reporting. For that matter, let's add the following block of code at the very beginning of the `Main` method:

```
if(args.Length > 0 && args[0].ToLower() == "benchmark")
{
    BenchmarkRunner.Run<Benchmarks>();
    return 0;
}
```

This allows us to run the benchmarks if we execute the application and pass `benchmark` as a parameter.

What this block of code does is ask `BenchmarkDotNet` (via the `BenchmarkRunner` class) to run all the benchmarks that will be found in the `Benchmarks` class.

Let's create that `Benchmarks` class!

Following the folder structure convention that we defined in previous chapters, we will create a `Benchmarks` folder within which we will create a `Benchmarks.cs` file.

We can either have a single class where all benchmarks are located, or we can create one benchmark class for every command or service to be benchmarked. We will take the first approach in this chapter as we will only benchmark the `export` command.

Let's add our first benchmark method. Its code looks like this:

```
public async Task ExportBookmarks()
{
    var exportCmd = new ExportCommand(_service!, "export", "Exports
    all bookmarks to a file");
    var exportArgs = new string[] { "--file", "bookmarksbench.json" };
    await exportCmd.InvokeAsync(exportArgs);
}
```

This method creates an instance of the `ExportCommand` class and executes it by calling its `InvokeAsync` method, passing in the required parameters for the command.

Right now, this method is not yet considered as a benchmark by the `BenchmarkRunner` class. The reason is that for a method to be considered as a benchmark, it needs to be decorated with the `[Benchmark]` attribute. Let's fix this!

```
[Benchmark]
public async Task ExportBookmarks()
{
    var exportCmd = new ExportCommand(_service!, "export", "Exports
    all bookmarks to a file");
    var exportArgs = new string[] { "--file", "bookmarksbench.json" };
    await exportCmd.InvokeAsync(exportArgs);
}
```

Awesome! But we are not ready to run it yet…

See what's missing?

You got it! The `ExportCommand` class takes an instance of type `IBookmarkService` as a parameter, but we haven't so far provided such an instance of an object.

Since we already have such an instance defined in the `Program` class, you may expect that we can pass it to the `Benchmarks` class through its constructor, and this would be a perfectly reasonable assumption. However, the `BenchmarkRunner` class does not allow us to do so (at least with the current version of `BenchmarkDotNet`).

What we will do instead is to instantiate this object in the `Benchmarks` class directly. The code will then look like this:

```
#region Properties
private IBookmarkService? _service;
#endregion

#region GlobalSetup
[GlobalSetup]
public void BenchmarksGlobalSetup()
{
    _service = new BookmarkService();
}
#endregion
```

Notice that the instantiation of the service is not performed in the class constructor but rather in a method decorated with the `[GlobalSetup]` attribute. This special attribute instructs `BenchmarkDotNet` to call this method once before executing each benchmark method. This is to have a clean instance of the service for each benchmark method, hence preventing side effects from previous benchmarks.

GlobalSetup versus class constructor

The execution time of the `[GlobalSetup]` method is not taken into account in calculating the benchmarked method execution time, as opposed to the execution time of the constructor. While this might seem negligible, it will not be if the method is meant to be executed a significant number of times.

We are now ready to execute our benchmark.

To do this, we first need to build the application, but this time we need to build it in `Release` mode. Otherwise, `BenchmarkDotNet` will generate an error. The reason is that running a program in `Debug` mode is not optimal and has a significant performance cost compared to running the program in `Release` mode, which is the mode the application should be run on in production. Therefore, when benchmarking our application, we should do it in its optimal performance mode.

Debug vs Release modes

Building the code in `Debug` mode produces unoptimized code with full symbolic `debug` information, enabling easier debugging and breakpoint setting. In contrast, `Release` mode generates optimized code for better performance and smaller file sizes. `Release` builds typically omit `debug` symbols, inline methods, and apply various optimizations that can make debugging more challenging but result in faster execution. While `Debug` builds are ideal for development and troubleshooting, `Release` builds are used when deploying to production.

Building the application in `Release` mode can be achieved by typing the following:

```
dotnet build -c Release
```

We then run the benchmarks by typing the following:

```
dotnet C:\code\Chap12\bookmarkr\bin\Release\net8.0\bookmarkr.dll
benchmark
```

`C:\code\Chap12\bookmarkr\bin\Release\net8.0` is the location of the generated DLL of the Bookmarkr application.

The result is as follows:

```
Intel Xeon Platinum 8171M CPU 2.60GHz, 1 CPU, 2 logical cores and 1 physical core
.NET SDK 8.0.204
  [Host]     : .NET 8.0.4 (8.0.424.16909), X64 RyuJIT AVX-512F+CD+BW+DQ+VL
  DefaultJob : .NET 8.0.4 (8.0.424.16909), X64 RyuJIT AVX-512F+CD+BW+DQ+VL


| Method          | Mean     | Error     | StdDev    |
|---------------- |---------:|----------:|---------:|
| ExportBookmarks | 6.356 ms | 0.7840 ms | 2.287 ms |
```

Figure 12.1 – Benchmarking the export command

The benchmark method has run 98 times and, on average, it takes 6.356 milliseconds to run the `export` command, which is not bad at all, is it?

You can see the table in the middle of the screen. This table compiles the metrics per benchmark method. Let's explain what each of its columns represents:

- `Mean`: This represents the average duration of the benchmarked method over all its executions (98 in our example).
- `Error`: Simply stated, this value represents the precision of the mean value's measurement. The smaller the error, the more precise the measurement of the mean value. As an example, since our mean value is 6.356 ms and the error is 0.7840 ms, all measurements fall within the range of 6.356 ms ± 0.7840 ms, which means between 5.572 ms and 7.140 ms.
- `StdDev`: This value represents the standard deviation of all measurements. It quantifies the amount of variation or dispersion in the execution times. In other words, a lower value of `StdDev` indicates that the execution times are clustered closely around the mean.

Benchmarking is not only for commands!

Although we are benchmarking a command here, it is important to note that benchmarking does not only apply to commands but rather to all code artifacts that may have an impact on the application's performance, which also includes services. Therefore, by benchmarking commands *and* the services they use, we can determine the percentage of the execution time and memory consumption that is attributable to the service and the command.

Great! There is, however, one measurement that we haven't seen here, which is the measurement of memory consumption. Let's fix that!

To collect data about memory consumption, we simply need to add the `[MemoryDiagnoser]` tag on top of the `Benchmarks` class, as follows:

```
[MemoryDiagnoser]
public class Benchmarks
{
    // …
}
```

Now, if we run the code in exactly the same way as before, we get the following results:

```
.NET SDK 8.0.204
  [Host]     : .NET 8.0.4 (8.0.424.16909), X64 RyuJIT AVX-512F+CD+BW+DQ+VL
  DefaultJob : .NET 8.0.4 (8.0.424.16909), X64 RyuJIT AVX-512F+CD+BW+DQ+VL


| Method          | Mean     | Error     | StdDev    | Allocated |
|---------------- |---------:|----------:|---------:|----------:|
| ExportBookmarks | 6.297 ms | 0.9739 ms | 2.872 ms |  18.63 KB |
```

Figure 12.2 – Benchmarking memory consumption

Notice that now we have a new column called `Allocated`, which represents the amount of allocated memory for every execution of the benchmarked method, in kilobytes. This column is interesting for two reasons:

- It allows us to see if the benchmarked method is using way too much memory than it should (or way more than it is expected to use). This can indicate memory leaks in our code that require deeper investigation.
- When we optimize our code, we can see if the new implementation has an impact on memory consumption. For example, we could come up with an implementation that speeds up the execution time at the expense of significant memory consumption.

Execution time versus memory consumption optimization

You may be wondering whether we should concentrate on optimizing memory consumption or execution time. The decision about where to focus our attention and energy depends on what we value the most, memory consumption or execution time. It is interesting to note that, in some situations, we may even be able to optimize both at the same time! To do that, we have to come up with a creative implementation that addresses both concerns by leveraging advanced features of the frameworks and libraries that we use, combined with advanced and creative algorithms.

While `BenchmarkDotNet` helps us identify optimization opportunities during the development phase, it is important to implement monitoring so that we can continuously check the application's performance while it is being used in production.

Monitoring BookmarkrSyncr with Azure Application Insights

We mentioned earlier that a CLI application runs locally on the user's computer and that the user may refuse to allow us to collect telemetry data that is absolutely essential for monitoring. That is why we won't implement monitoring in Bookmarkr but rather in **BookmarkrSyncr**, the external web service invoked by Bookmarkr. Since this is a web service hosted and managed by us, we can implement monitoring and ensure that telemetry data will be collected, therefore ensuring that monitoring can take place.

Since this web service is deployed to the **Microsoft Azure** cloud platform, we will rely on Azure Application Insights, the **application performance monitoring** (**APM**) solution provided natively by the Microsoft Azure cloud platform.

When we deployed BookmarkrSyncr to Microsoft Azure, we created an infrastructure for hosting it. More specifically, we created an **Azure App Service** instance. As part of the process of creating this service, we are offered the opportunity to create an instance of the **Azure Application Insights** service. This service is a monitoring solution that is provided and managed for us by Microsoft.

Azure Application Insights is a fantastic service that allows us to monitor performance, availability, failed requests, exceptions, page views, traces, browser timings, usage (including **user flows**, which allow us to identify hot paths in the application), and even access live metrics so we can monitor in real time. Another great feature of **Azure Application Insights** is the ability to configure alerts to be triggered if a certain metric reaches a certain threshold, for example, if the server response time (which measures the duration between receiving the HTTP request and sending the response to the client) is above the maximum allowed value as defined by our organization's standards. When an alert is raised, we can then trigger an automated processing or a notification (such as an email to a specific group of people).

To see what monitoring with **Azure Application Insights** may look like, check out (this article on Microsoft Learn, which can be found at `https://learn.microsoft.com/en-us/azure/azure-monitor/app/overview-dashboard`.

Okay. Now that we know how to identify the areas of our application that require performance tuning (using profiling and monitoring), let's discuss the most common techniques that we can use to enhance the performance of our application.

Common performance optimization techniques

It is worth mentioning that the techniques we will be discussing here do not only apply to CLI applications but can rather be applied to any kind of application. Let's break these techniques down according to the categories we presented earlier. For every category, I will give you a list of techniques commonly used for it.

Application design and architecture:

- Establish the shortest path to achieve a goal, removing all unnecessary intermediaries.
- This can be achieved by using efficient algorithms.
- Find the optimal balance between decoupling and low latency.
- Use lazy loading for resources that aren't immediately needed.
- Implement efficient error handling and logging mechanisms.
- Design for scalability from the start.

Infrastructure:

- When packaging and distributing your application, compile it in `Release` mode. While `Debug` mode is great during the development phase, it may add a significant performance overhead.
- Also, when packaging and distributing your application, compile it as platform-specific if the target platform is known ahead of time or if the packaging and distribution mechanism is not cross-platform. For example, distributing our application as a **Winget** package means that it will exclusively be used on the Windows platform. The same goes with an apt-get package (where the application will run exclusively on **Linux**) and with **Homebrew** (where the application will run exclusively on **macOS**). It is therefore easy to know what platform-specific compilation should be used and will make .NET apply all the possible optimizations, which is something it wouldn't do if the target platform is not known ahead of time (an example of that is file handling, which is different on Windows, Linux, and macOS). This will result in a version of the application that runs in the most efficient manner on that target platform.

- You might also choose to use **AOT** (**Ahead-Of-Time**) compilation to precompile your code to native code (instead of relying on **JIT**) for faster startup times or to reduce the dependency on runtime compilation. This could be particularly useful if you're targeting environments like mobile (iOS/Android) or WebAssembly, where JIT might not be feasible. Note that platform targeting and AOT can be combined for even better performance optimization.

Frameworks and libraries:

- Avoid using libraries that rely on reflection, unless absolutely necessary.
- Choose lightweight frameworks and libraries that align with your specific needs. Beware of libraries that pull off tenth of other libraries when you reference them.
- Keep dependencies up to date to benefit from performance improvements.
- Consider using micro-frameworks for smaller, focused tasks.

Coding practices:

- Rely on asynchronous operations whenever possible. This will avoid blocking the main thread and increases the feeling of responsiveness of the application.
- Choose the most optimized data types or data structures for the pursued purpose. This will ensure we have the minimal footprint on the computer's resources.
- Whenever possible, try to achieve a task with as little memory allocation as possible. For example, at the time of this writing, .NET 9 was released and introduces split operations with no memory allocation by calling `AsSpan().Split(…)`.
- Implement a caching mechanism to avoid unnecessary calls to external dependencies (such as web services or databases).
- Optimize database queries and implement proper indexing.
- Speaking about databases, if you are using an **ORM** (**Object/Relational Mapper**) such as **Entity Framework Core**, you may want to call `AsNoTracking()` to significantly improve query performance and reduce memory usage, especially when dealing with large datasets or read-only operations. This method tells the ORM not to track changes to the retrieved entities, bypassing the change tracking mechanism and resulting in faster queries with lower memory overhead.
- Use connection pooling, which consists in reusing established database connections instead of creating a new one for every request. This is because it can be expensive to establish a connection to a database, therefore connection pooling reduces connection latency and enables high database throughput (transactions per second) on the server.
- Implement proper memory management and dispose of unused resources.

We have seen a bunch of techniques that are commonly used for optimizing the performance of any kind of application that is built with any technology stacks, including CLI applications built with .NET.

Let's now apply some of these techniques to enhance Bookmarkr's performance.

Optimizing Bookmarkr's performance

We cannot optimize what is already perfect, can we?

Just kidding. Of course we can! There is always room for improvement.

Let's see some of the quick wins that we can apply to enhance the performance of our beloved CLI application.

Looking at the handler method of the `ExportCommand` class (namely, `OnExportCommand`), we can see that it already leverages async operations. This is a great start and is actually one of the techniques we described earlier.

However, the handler method can be optimized. To illustrate this, let's create a copy of the `ExportCommand` class and name it `ExportCommandOptimized`. Let's copy the code from the `ExportCommand` as is, and we will optimize it in a moment.

The reason we are creating a copy of the original class rather than directly optimizing it is so that we can add a benchmark method for the optimized version and compare it with the original one.

In the handler method of the `ExportCommandOptimized` class, let's change these two lines of code:

```
string json = JsonSerializer.Serialize(bookmarks, new
JsonSerializerOptions { WriteIndented = true });
await File.WriteAllTextAsync(outputfile.FullName, json, token);
```

Replace them with the following two lines:

```
using var fileStream = new FileStream(outputfile.FullName, FileMode.
Create, FileAccess.Write, FileShare.None, 4096, true);
await JsonSerializer.SerializeAsync(fileStream, bookmarks, new
JsonSerializerOptions { WriteIndented = true }, token);
```

Let's see what we have done:

- Using `JsonSerializer.SerializeAsync` is more efficient for large datasets as it streams the JSON directly to the file without keeping the entire serialized string in memory
- Using `FileStream` with async operations allows better control over file I/O operations and can improve performance, especially for large files

Okay. Let's compare this new implementation with the original one.

To do this, let's add the following benchmark method to the `Benchmarks` class:

```
[Benchmark]
public async Task ExportBookmarksOptimized()
{
    var exportCmd = new ExportCommandOptimized(_service!, "export",
    "Exports all bookmarks to a file");
    var exportArgs = new string[] { "--file", "bookmarksbench.json" };
    await exportCmd.InvokeAsync(exportArgs);
}
```

This `benchmark` method is identical to the previous one. Well, almost identical… The only difference is that we are instantiating (and invoking) the `ExportCommandOptimized` class rather than the `ExportCommand` class.

Since we want to compare the new, optimized, implementation against the original one, we will modify the `[Benchmark]` attribute of the original method to look like this.

This instructs `BenchmarkDotNet` to use this method as a baseline for the comparison:

```
[Benchmark(Baseline = true)]
```

Let's rebuild the application (in `Release` mode, of course) and execute the benchmarks.

The results are the following:

```
BenchmarkDotNet v0.14.0, Windows 11 (10.0.22621.4317/22H2/2022Update/SunValley2) (Hyper-V)
Intel Xeon Platinum 8171M CPU 2.60GHz, 1 CPU, 2 logical cores and 1 physical core
.NET SDK 8.0.204
  [Host]     : .NET 8.0.4 (8.0.424.16909), X64 RyuJIT AVX-512F+CD+BW+DQ+VL
  DefaultJob : .NET 8.0.4 (8.0.424.16909), X64 RyuJIT AVX-512F+CD+BW+DQ+VL


| Method                  | Mean     | Error     | StdDev   | Ratio | RatioSD |
|------------------------ |---------:|----------:|---------:|------:|--------:|
| ExportBookmarks         | 6.761 ms | 0.7801 ms | 2.288 ms |  1.30 |    1.51 |
| ExportBookmarksOptimized | 4.704 ms | 0.9471 ms | 2.609 ms |  0.91 |    1.19 |

// * Warnings *
MultimodalDistribution
  Benchmarks.ExportBookmarks: Default         -> It seems that the distribution can have several modes (mValue = 2.88)
  Benchmarks.ExportBookmarksOptimized: Default -> It seems that the distribution can have several modes (mValue = 2.91)
MinIterationTime
  Benchmarks.ExportBookmarks: Default         -> The minimum observed iteration time is 87.848ms which is very small. It's recommended to increase it to at least 100ms using more operations.
  Benchmarks.ExportBookmarksOptimized: Default -> The minimum observed iteration time is 37.346ms which is very small. It's recommended to increase it to at least 100ms using more operations.
Environment
  Summary -> Benchmark was executed on the virtual machine with Hyper-V hypervisor. Virtualization can affect the measurement result.

// * Hints *
Outliers
  Benchmarks.ExportBookmarks: Default         -> 1 outlier  was  removed, 2 outliers were detected (686.31 us, 13.23 ms)
  Benchmarks.ExportBookmarksOptimized: Default -> 12 outliers were removed (12.00 ms..23.10 ms)

// * Legends *
  Mean    : Arithmetic mean of all measurements
  Error   : Half of 99.9% confidence interval
  StdDev  : Standard deviation of all measurements
  Ratio   : Mean of the ratio distribution ([Current]/[Baseline])
  RatioSD : Standard deviation of the ratio distribution ([Current]/[Baseline])
  1 ms    : 1 Millisecond (0.001 sec)

// ***** BenchmarkRunner: End *****
Run time: 00:02:22 (142.46 sec), executed benchmarks: 2

Global total time: 00:02:41 (161.54 sec), executed benchmarks: 2
// * Artifacts cleanup *
Artifacts cleanup is finished
PS C:\code\Chap12\bookmarkr>
```

Figure 12.3 – Benchmarking the new implementation against the original one

Notice the appearance of two new columns:

- `Ratio`: This indicates the average measure of the performance relative to the baseline benchmark method
- `RatioSD`: This indicates the average standard deviation relative to the standard deviation of the baseline benchmark method

The value of `0.91` in the `Ratio` column indicates that the optimized implementation (`ExportCommandOptimized`) is on average 9% faster than the baseline implementation (`ExportCommand`). We mentioned earlier that the implementation we made in `ExportCommandOptimized` is especially more performant when dealing with large files. Therefore, we can expect it to be even faster than the baseline implementation as the output file becomes larger.

Awesome! We now know how to improve the performance of our beloved CLI application and we have made our users happy.

Summary

In this chapter, we explored the various areas of performance optimization, we learned techniques to identify performance hot spots and hot paths, and we saw how to improve their performance, with the ultimate goal of offering our users a great and efficient application that they will love to use.

Hopefully, you have understood that there is not one single area or action that leads to better performance, but rather a series of fine-tuning here and there that do the trick.

Awesome! So, we have an application that efficiently provides great functionality.

There is, however, one key area that we have not yet covered when it comes to building CLI applications (and, for that matter, any kind of application). That key area is **security**, and this is the topic of the next chapter.

Your turn!

Following along with the provided code is a great way to learn through practice.

A better way is by challenging yourself to achieve tasks. Hence, I challenge you to improve the Bookmarkr application by adding the following features.

Task #1 – Write more benchmarks

In this chapter, we have illustrated writing benchmark methods by only writing a benchmark for the `export` command. However, as we mentioned earlier, benchmarks do not only apply to commands, but they can also apply to services.

That's why you are tasked with writing additional benchmark methods for each command and for the services used by the Bookmarkr application.

Task #2 – Fine-tune Bookmarkr for optimal performance

Throughout this chapter, we haven't implemented every performance optimization opportunity, and we have probably missed some (was that intentional? *wink wink*). Therefore, you are tasked with identifying other potential performance optimizations in Bookmarkr and implementing them.

13
Security Considerations for CLI Applications

Security is one of the most critical concerns in any application development project. It's interesting to see that, in many projects, development teams tend to think that because they implemented measures to prevent their application's code from being subject to SQL injection, XSS attacks, or similar, their application is secure.

However, it is important to keep in mind that security takes different forms and spans different areas, which means that it does not only concern the security of the application's code or its usage but also extends to the security of the whole development lifecycle.

In this chapter, we will discuss these different areas and how they are related to securing CLI applications, and we will cover the key areas that you need to consider in order to enhance the security of your CLI application and the security of your development lifecycle. More specifically, we will do the following:

- Discuss the different areas in which security should be considered
- Learn how to assess the security posture of a CLI application
- Learn how to implement authentication in order to secure access to user data

Technical requirements

The code for this chapter can be found in the GitHub repository accompanying this book, `https://github.com/PacktPublishing/Building-CLI-Applications-with-C-Sharp-and-.NET/tree/main/Chapter13`

Security areas

As mentioned earlier, security is not only limited to the application's code, nor can it only be achieved by implementing authentication, although these areas are very important.

Let's first start by highlighting the key areas involved in securing an application throughout its lifecycle. It is important for me to make this point (or reminder) here as I still meet with customers who introduce security too late in the lifecycle of an application (typically after the application has been developed and released to production), expecting security professionals to do miracles and secure the application with minimal or no modification to the application, which is, obviously, unrealistic.

Always keep in mind that security should span the entire lifecycle of the application. In concrete terms, security should be a concern from the early stages of designing the application:

- **In the design phase**: It is important to define security requirements and objectives and to conduct threat modeling using methodologies such as STRIDE to identify potential security risks so we can incorporate the appropriate security controls into the initial design. STRIDE helps categorize threats into six groups: Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, and Elevation of privilege. After identifying threats, we can use the DREAD model to quantitatively assess and prioritize them based on their Damage potential, Reproducibility, Exploitability, Affected users, and Discoverability. During this phase, we should also consider privacy and data protection measures, such as encryption, authentication, and authorization.
- **In the architecture phase**: We are not only talking about the software architecture here but also about the infrastructure and networking architecture. Of course, we need to design a secure architecture that includes defense-in-depth strategies and integrates security mechanisms such as authentication and authorization. But we also need to plan for secure communication channels and data storage, using techniques such as encryption at rest and in transit along with network segmentation to ensure only appropriate routes of communication, through the appropriate network addresses, ports, and domains, and using appropriate protocols are allowed.
- **In the development phase**: It is important to apply secure coding practices by relying on coding standards, such as implementing input validation and sanitization, and to ensure that our application is not vulnerable to common OWASP Top 10 security risks. It is equally important to ensure that secure coding practices are applied during code reviews. But this is not enough! You should use secure libraries and frameworks. In other words, ensure that your frameworks and libraries are still supported and receive security updates. Since we are using .NET 8 here, we know that it is still supported (and will receive security updates) until November 10, 2026 (I'm writing this in November 2024). As a framework or a library becomes unsupported, it is important to plan to migrate to a newer version (not necessarily the latest, by the way 😉).

  Suppose our application communicates with external dependencies, such as web services. In that case, we have to ensure that these communications happen in a secure manner (by leveraging authentication and authorization) and using the appropriate protocols (such as HTTPS).
- **In the infrastructure configuration phase**: All efforts to enhance the application's security could be in vain if the infrastructure on which it is running is unsecured. In the case of CLI applications, which run on the user's computer, this responsibility is delegated to them or their organization's IT department, which usually controls employees' workstations.

- **In the testing phase**: Security-specific testing, including penetration testing and vulnerability assessments, should be performed during the testing phase to ensure user data cannot leak, user accounts cannot be compromised, and the application cannot be diverted from its intended usage toward malicious activities. Performing security testing allows us to validate that security requirements are met. Mature DevSecOps teams perform such testing on a regular basis. However, security tests should at least be performed before releasing an application to production.
- **In the deployment phase**: Every DevSecOps engineer knows that your CI/CD pipelines can be a security threat if not properly secured. Secrets (such as passwords, API keys, service connections, connection strings, and so on) should be kept… well, secret. Every CI/CD tool has its own secret management mechanism, which usually takes the form of a key vault, but we can also rely on external tools that are designed for such purposes, such as Azure Key Vault or HashiCorp Vault. Access to these secrets is usually restricted using roles and permissions.
- **While the application is being used**: It is a common mistake for immature teams and organizations to believe that once the application has been released to production, the job of security is done. Wrong! This is where it all starts. You may argue that we put a lot of effort into securing the application before releasing it to our users, and you would be right. However, by releasing our application into the wild, it will experience a wide range of usage patterns and user environments, way beyond what we could have expected, thought of, and planned for. We will hence have to monitor for security incidents and anomalies, by implementing logging and auditing mechanisms and by regularly updating and patching the application and its dependencies.
- **While logging the application's behavior and data**: We all know that logging is an essential part of monitoring and that it helps understand usage patterns for the application as well as coming in handy when debugging the application. However, it is a good security practice to avoid logging sensitive information, such as passwords, credit card information, or social security numbers, and if we have to do it (for auditing purposes, for example), these logs should be stored in a secure location with restricted and audited access. In *Chapter 6*, we introduced **Serilog** for logging. This tool provides ways to sanitize logs before they are stored. For example, when configuring the logger, we can call the `Destructure.ByMaskingProperties` method and pass a list of properties to be ignored when logging.
- **As new versions or bug fixes are released**: Either when introducing new functionalities or fixing bugs, it is important to conduct a security impact analysis for the introduced changes. This can be done by performing regression testing to ensure existing security controls are still met and no vulnerability has been introduced.

Fortunately, there is a variety of tools to help at every stage. If your team or organization has adopted a DevSecOps culture, you will already be aware of many of these tools.

DevSecOps being way beyond the scope of this book, we will not be discussing the broad spectrum of tools you may use at every step. However, I do want to cover tools that will help us assess and enhance the security posture of a CLI application.

Assessing the security posture of a CLI application

There are multiple tools that help assess the security posture of an application (including a CLI application). Among the most widely used ones, we find the following:

- **SonarQube** (and its cloud edition, **SonarCloud**): This provides powerful static analysis for code quality and security vulnerabilities, including dependency checks. They can be integrated into CI/CD pipelines and scan `.csproj` files to detect vulnerabilities in third-party libraries.
- **Snyk**: This popular tool offers vulnerability reporting and scanning for .NET projects. It can be integrated into local development environments as well as into CI/CD pipelines for continuous monitoring.
- **Mend Bolt**: Previously known as **WhiteSource Bolt**, this security scanning tool integrates with **Azure DevOps** or **GitHub** pipelines, scanning .NET projects for open source vulnerabilities and generating detailed reports.
- **OWASP Dependency-Check**: This tool is effective for scanning third-party libraries and dependencies, which is crucial for .NET applications that often rely on external packages.
- **GitHub Advanced Security**: This tool integrates security features directly into GitHub or Azure DevOps workflows, performs code scanning (it uses static analysis to detect potential security vulnerabilities and coding errors) and secret scanning (it recognizes patterns for passwords, API keys, and other secrets and detects whether they are stored in clear text in the repository), and performs dependency reviews and highlights vulnerable ones.

These tools have in common that they inform us about the vulnerable library, and give us detailed information about the vulnerability itself and the recommended fix for it.

These tools are either free (such as Mend Bolt and OWASP Dependency-Check) or offer a free plan but with limited functionality (such as SonarQube, Snyk, and GitHub Advanced Security).

They also vary in the level of complexity and effort required to set up and configure them. Hence, you will find SonarQube to be the most complex to set up, Snyk to be moderately easy to set up, and Mend Bolt, OWASP Dependency-Checker, and GitHub Advanced Security to be the easiest ones to set up.

I would recommend Mend Bolt as a good starting point since it comes as an Azure DevOps or GitHub free extension that can be obtained from their respective marketplaces. For these reasons, these tools are usually intended for organizations or larger teams.

However, the good news is that .NET already provides us with an out-of-the-box tool to serve this very purpose of assessing and enhancing the security posture of our application.

This tool, commonly known as dotnet-audit, focuses on detecting vulnerabilities in .NET project dependencies (namely, `NuGet` packages) by relying on the `GitHub Advisory Database`. For .NET-only projects, such as **Bookmarkr**, this tool is the perfect starting point!

To execute this tool, we simply need to type the following command:

```
dotnet list package --vulnerable
```

The results will be shown in the terminal window, and will look like this:

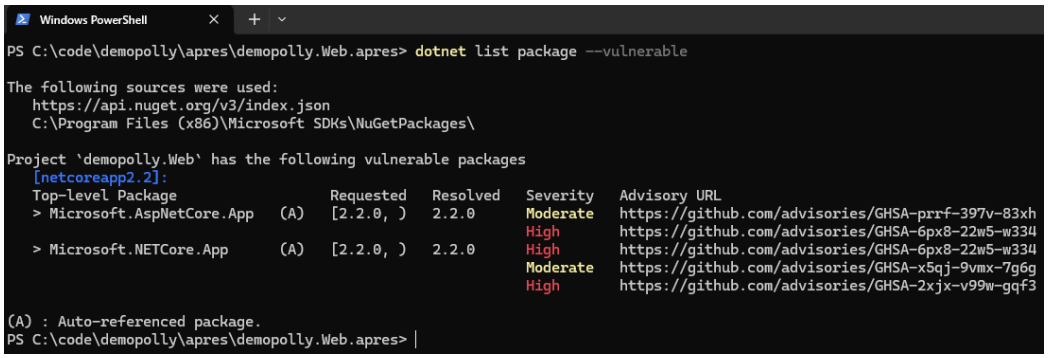


Figure 13.1 – Listing vulnerable packages

As you may have noticed in this figure, the command has not been executed on Bookmarkr's code but rather for another application. As it turns out, we are glad that no vulnerability has been detected for Bookmarkr (yeah!), but this might change over time as vulnerabilities in referenced libraries are detected.

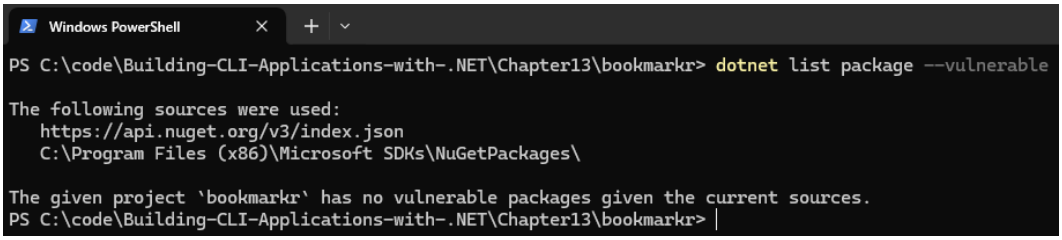


Figure 13.2 – No vulnerable packages for Bookmarkr

Keep in mind that this command may ask you to run `dotnet restore` before you can execute it. This is especially true if this is the first time you have cloned the Git repository.

There is another command that does not directly scan for vulnerabilities, but which I recommend. This command lists outdated packages. While these packages might not have known vulnerabilities, being outdated means that they will no longer receive security updates. My recommendation is that you consider upgrading these packages to newer supported versions (once again, not necessarily the latest version if this introduces breaking changes).

To run this command, simply type the following:

```
dotnet list package --outdated
```

As you can see from the result of executing this command, although no vulnerability has been detected in Bookmarkr's dependencies, some of them are outdated:

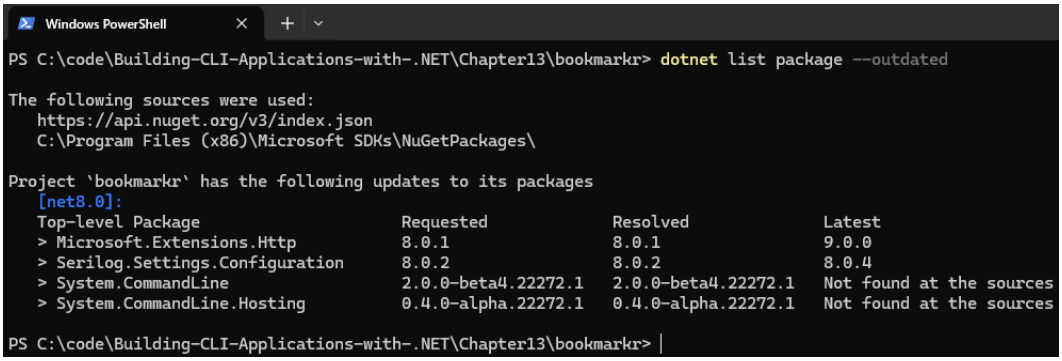


Figure 13.3 – Listing outdated packages

The appropriate way to upgrade these packages is to create a new branch, update the packages, test the application (both using manual and automated testing, as described in *Chapter 10*) to ensure that the application still works as expected and that we did not introduce regressions, and finally, issue a pull request to merge the modifications into the `main` branch.

We now have the necessary knowledge and tools to assess, and ultimately enhance, the security posture of our CLI application. Let's focus our attention on securing communication between our CLI application and the external services it interacts with, to protect against unauthorized access and ensure proper user management. This can be achieved through authentication.

Securing remote communications using authentication

In *Chapter 9*, we introduced the `sync` command, which allows Bookmarkr to back up local bookmarks to a remote location and retrieve them when needed. When doing so, the command also synchronizes local and remote bookmarks.

Until now, the communication between the local CLI application and the remote external service has happened in an insecure manner. This means that anyone who invokes the `sync` command may retrieve your personal bookmarks, which you clearly don't want, do you?

To address this issue, we need to implement authentication.

Why is authentication important?

You may be wondering why authentication should be required in the context of CLI applications. After all, a CLI application runs on the user's computer, which already requires that user to authenticate to their session.

In the context of CLI applications, authentication is usually required when communicating with external services (i.e., sending data to and retrieving data from these services). This ensures that users have access to their data at the remote location by proving who they are.

How to perform authentication

Authentication can be achieved in many ways. One of the most common ones is for the external service provider to provide you with a **Personal Access Token** (**PAT**). You can usually get or generate such tokens by visiting the service provider's website and logging in to your account. From there, on your account's settings page, you should be able to get that token or to generate a new one. Such tokens are usually valid for a given period of time and expire after that.

Once you have this token, you can pass it as a parameter to a command that performs the call to the external service. An example of this would be the `sync` command, which could be invoked as follows:

```
bookmarkr sync --pat YOUR_PAT
```

Here, `YOUR_PAT` is most commonly a GUID value.

This call would then authenticate the user using the received PAT value before performing the sync operation.

As you may have noticed, this approach could quickly become tedious as it is not easy to remember the value of a PAT. For this reason, CLI applications usually store such values either in a local configuration file, in an environment variable, or in the operating system's key vault. This allows the user to invoke the command without always having to pass the PAT as a parameter. The command will be smart enough to look for it in the local configuration file or in the environment variable and use it if it is present. If it is not present, the command should display an error message asking the user to provide it. If the PAT is invalid or expired, the command should also be able to inform the user about that.

This is the approach adopted by GitHub – for example: you create a PAT within your GitHub account, which you can use to authenticate and access GitHub resources from external services or applications. In our use case, the external service is the one that provides and manages the tokens, whereas the CLI application is only responsible for sending these tokens to the external service.

Another commonly used approach is to authenticate the user against an **identity provider** (**IdP**) such as Google, Facebook, or Microsoft. With this approach, a CLI application will usually provide a specific command for authentication (such as `auth`). Invoking this command usually triggers the default web browser and redirects the user to the application provider's login page. After successful authentication, the IdP provides access and an ID token that the CLI can use for subsequent requests to the external service. These tokens are, as for the PAT, usually included in the HTTP headers when performing the requests to the external service. This approach leverages the `OAuth 2.0` and the `OpenID Connect` protocols.

In the remainder of this chapter, we will explore how to implement authentication for Bookmarkr to secure communications with the **BookmarkrSyncr** external service.

Implementing authentication

Here, we will leverage the PAT approach as it is more convenient and can work even in environments where no web browser can be launched, such as CI/CD pipelines.

To illustrate how this could be achieved, we will need to implement functionalities at two levels:

- **BookmarkrSyncr**: The external service will receive the PAT, validate it, and authenticate the user if it is valid
- **Bookmarkr**: The responsibility of the CLI application is to pass the token to the external service and act according to the response it gets from the sent request

Let's start by adding the required functionality into BookmarkrSyncr.

Authenticating external services using a PAT

To make things simple, we will assume that BookmarkrSyncr (the code for which can be found in the `AppendixB` folder) holds two PAT tokens: a valid one and an expired one. All other values that the user passes will be considered invalid and will be rejected for this reason.

The token validation service will also be a very basic one.

While most of the code remains as it was, we had to make the following changes to the parameters we pass to the `MapPost` method:

```
app.MapPost("/sync", async ([FromHeader(Name = "X-PAT")] string pat,
List<Bookmark> bookmarks, ITokenValidator tokenValidator, HttpContext
context) =>
{
     …
}
```

Let's explain these changes:

1. We indicate that the value of the PAT token comes from an HTTP header named `X-PAT` and that this value will be stored in the input parameter named `pat`.
2. We pass a parameter of type `ITokenValidator`, which is a service we have created to retrieve and validate PAT tokens.
3. We pass the current HTTP context that we will need to set HTTP headers for the HTTP response, especially to notify the client that the received PAT token is either invalid or expired. We use a different response header to ensure that the client knows exactly why the request was unauthorized, as this is good programming practice.

We then invoke methods of the `TokenValidator` service to check whether the PAT token is invalid or expired, and if so, we set the appropriate HTTP header in the response object. The code looks like this:

```
// Ensure the Personal Access Token (PAT) is valid
if (!tokenValidator.IsValid(pat))
{
    context!.Response.Headers["X-Invalid-PAT"] = pat;
    return Results.Unauthorized();
}

// Ensure the Personal Access Token (PAT) is not expired
if (tokenValidator.IsExpired(pat))
{
    context!.Response.Headers["X-Expired-PAT"] = pat;
    return Results.Unauthorized();
}
```

Notice how we set the HTTP header. We use the indexer syntax. This will set the header. If the header already exists, it will replace the existing value with the new one.

The last thing we need to do is to register the `TokenValidator` service. This can be done for any service, using this widely known syntax:

```
builder.Services.AddScoped<ITokenValidator, TokenValidator>();
```

The `TokenValidator` service is very basic and its code looks as follows:

```
public class TokenValidator : ITokenValidator
{
    private readonly List<PatToken> _tokens = new();

    public TokenValidator()
    {
        // we are simulating a token store here...
        _tokens.Add(new PatToken { Value = "4de3b2b9-afaf-406c-ab0d-
        d59ac534411d", IsExpired = false });
        _tokens.Add(new PatToken { Value = "16652977-c654-431e-8f84-
        bd53b4ccd47d", IsExpired = true });
    }

    public bool IsExpired(string token)
    {
        var retrievedToken = _tokens.FirstOrDefault(t => t.Value.
        Equals(token, StringComparison.OrdinalIgnoreCase));
```

```
        if(retrievedToken == null) return false;
        return retrievedToken.IsExpired;
    }

    public bool IsValid(string token)
    {
        var retrievedToken = _tokens.FirstOrDefault(t => t.Value.
        Equals(token, StringComparison.OrdinalIgnoreCase));
        return retrievedToken != null;
    }
}
```

The only thing worth mentioning here is that we are simulating a token store (in the class's constructor). In a real-world scenario, we would have a persistent token store (such as a database). But for the purposes of this demonstration, storing these PAT tokens in memory makes the code easier to understand. I also want you to notice that, for the purpose of the demonstration, we have two tokens: one is a valid token, and the other represents an expired token. Any other value will be considered invalid since it cannot be found in the token store (which is realistic, by the way 😉). The client (the CLI application) will hence behave differently depending on the validity of the token.

Excellent! Now that we have our external service ready, it's time to update the client so that it can pass the PAT.

Passing the PAT from the CLI application to the external service

For this to happen, we will need to modify the code of both the `sync` command and the service agent.

More specifically, we will need to do the following:

1. Add a `--pat` parameter to the `sync` command.
2. Modify the request that the HTTP client makes to the external service so that it sends the PAT token.
3. Modify the code to store the PAT token in an environment variable and retrieve it from there if the `--pat` parameter is not specified when invoking the `sync` command.

Let's start with the first step.

Let's go to the `SyncCommand.cs` file and, in the `Options` region, let's add an `Option` for the PAT as follows:

```
private Option<string> patOption = new Option<string>(
    ["--pat", "-p"],
    "The PAT used to authenticate to BookmarkrSyncr"
);
```

This `Option` is optional, as we mentioned earlier.

Next, we need to make the command use this `Option`. For that matter, we need to add this instruction to the class's constructor:

```
AddOption(patOption);
```

Now, let's modify the command's handler method so that the HTTP client can send the PAT to the BookmarkrSyncr service. For that matter, we need to modify the handler method's signature to pass the PAT as a parameter, as follows:

```
private async Task OnSyncCommand(string patValue)
{
    ...
}
```

Back to the class's constructor again, we need to modify the call to the `SetHandler` method to pass the PAT Option as a parameter. The modified method call looks like this:

```
this.SetHandler(OnSyncCommand, patOption);
```

Next, we need to pass this token to the service agent. For that matter, we first need to modify the call to the service agent in the handler method, as follows:

```
var mergedBookmarks = await _serviceAgent.Sync(patValue,
retrievedBookmarks);
```

Then, we need to update the code of the `BookmarkrSyncrServiceAgent` class to ensure the presence of the token before calling the external service.

The first thing we need to do is, obviously, pass the PAT value as a parameter to its `Sync` method, as follows:

```
public async Task<List<Bookmark>> Sync(string pat, List<Bookmark>
localBookmarks)
```

After that, the first thing the method should do is to ensure the presence of the PAT (i.e., ensure that we are not sending a null or an empty value to the external service). We do this by adding the following code at the beginning of the method:

```
// ensure that the pat is present
if(string.IsNullOrWhiteSpace(pat))
{
    string? value = Environment.GetEnvironmentVariable("BOOKMARKR_
    PAT");
```

```
    if(value == null) throw new PatNotFoundException(pat);
    pat = value;
}
```

A few things to note here:

- We do not validate the token, nor do we check whether it is expired. This is the role of the BookmarkrSyncr external service.
- The environment variable that is used to hold the PAT token is named `BOOKMARKR_PAT`. This name ensures that this variable does not conflict with any other variable set on the user's computer.
- We have created a custom exception class, `PatNotFoundException`, to inform the client in case the PAT token has not been found.

Now, since the PAT token is retrieved, we need to send it to the external service. We hence need to modify the HTTP client to pass the token in the HTTP headers of the request, as follows:

```
var client = _clientFactory.CreateClient("bookmarkrSyncr");
// Add the PAT to the request header
client.DefaultRequestHeaders.Add("X-PAT", pat);
var response = await client.PostAsync("sync", content);
```

If the request to the external service is successful, we know that the PAT token is valid. We will then save it to the environment variable if it is not already present there. The updated code is therefore as follows:

```
if (response.IsSuccessStatusCode)
{
    // saving the PAT to the environment variable, if not already
    string? value = Environment.GetEnvironmentVariable("BOOKMARKR_
    PAT");
    if(value == null || !value.Equals(pat)) Environment.
    SetEnvironmentVariable("BOOKMARKR_PAT", pat);

    // remaining of the code
}
```

However, if the PAT token is invalid or expired, we need to inform the client so that the proper error message is displayed to the user. We therefore need to modify the code block that handles the `Unauthorized` status code, as follows:

```
switch(response.StatusCode)
{
    case HttpStatusCode.Unauthorized:
        if (response.Headers.TryGetValues("X-Invalid-PAT", out var
        headerValues))
```

```
            throw new PatInvalidException(pat);
        if (response.Headers.TryGetValues("X-Expired-PAT", out var
        headerValues2))
            throw new PatExpiredException(pat);
        throw new HttpRequestException($"Unauthorized access:
        {response.StatusCode}");

    // remaining of the code
}
```

This code is straightforward and easy to understand. One thing worth mentioning though is that here, again, we created custom exception classes, namely `PatInvalidException` and `PatExpiredException`, to clearly express the intent of an error when raising them. These exception classes can be found in the `Exceptions.cs` file located in the `BookmarkrSyncrServiceAgent` folder.

The last step is to go back to the `SyncCommand` class, handle the responses from the service agent, and display the appropriate error messages to the user. For that matter, we need to modify the code of the `OnSyncCommand` handler method to catch the custom exceptions described previoulsy. The `catch` blocks are as follows:

```
catch(PatNotFoundException ex)
{
    Helper.ShowErrorMessage([$"The provided PAT value ({ex.Pat}) was
    not found."]);
}
catch(PatInvalidException ex)
{
    Helper.ShowErrorMessage([$"The provided PAT value ({ex.Pat}) is
    invalid."]);
}
catch(PatExpiredException ex)
{
    Helper.ShowErrorMessage([$"The provided PAT value ({ex.Pat}) is
    expired."]);
}
```

And that's it – we have completed all the required code modifications for Bookmarkr to ensure secure communication with the external service, BookmarkrSyncr, by using a PAT (it was about time, right? 😉).

Let's see if this works!

The first test is to invoke the `sync` command without passing a PAT token. Knowing that it will not be found in the `BOOKMARKR_PAT` environment variable, the following error is expected:

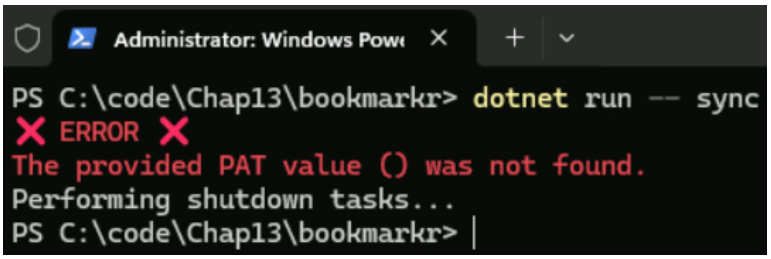


Figure 13.4 – The PAT token was not found

The second test is to invoke the `sync` command with an invalid PAT token. Here too, an error message is expected:

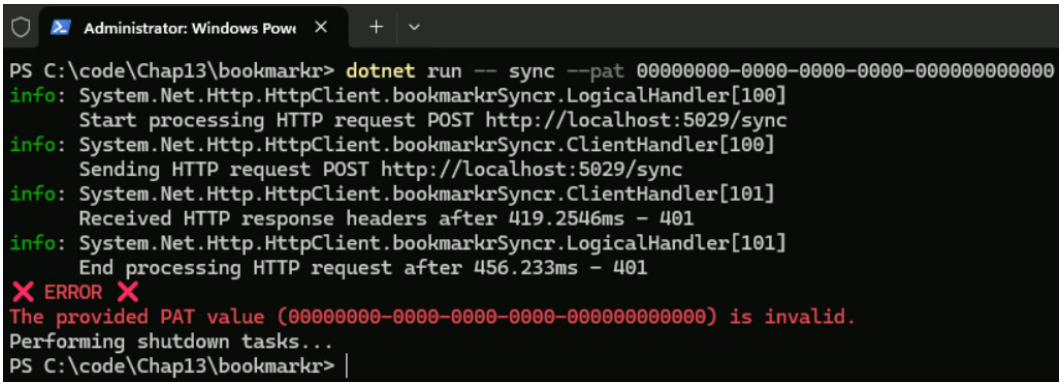


Figure 13.5 – The PAT token is invalid

The next test is to invoke the `sync` command with an expired PAT token. Here too, an error message is expected:

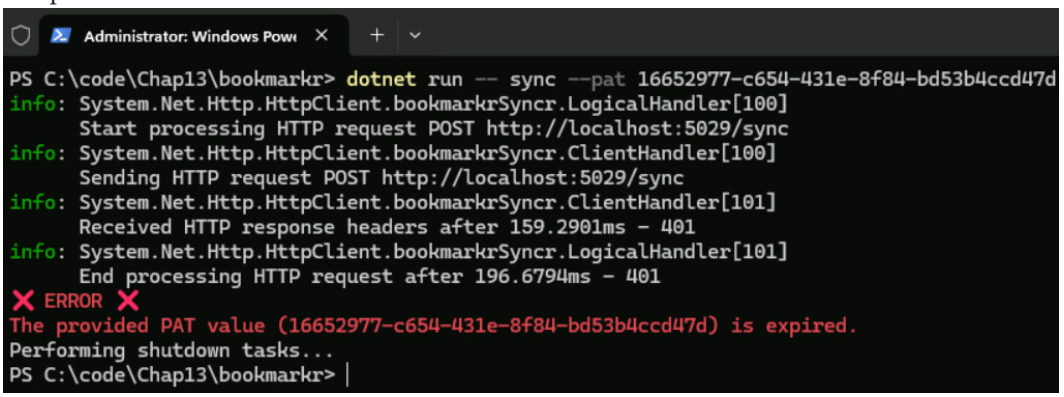


Figure 13.6 – The PAT token is expired

The final test is to invoke the `sync` command with a valid PAT token. We can see that the bookmarks have been synchronized and that the PAT token has been saved to the `BOOKMARKR_PAT` environment variable:

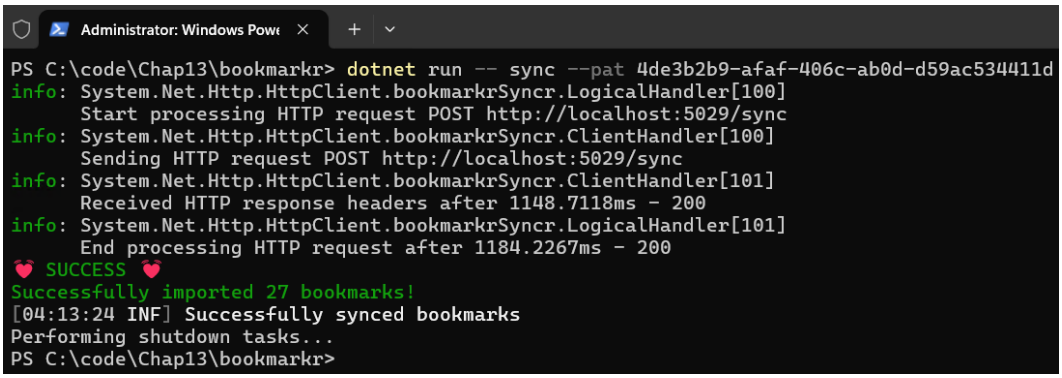


Figure 13.7 – The PAT token is valid

Wonderful! We now know all we need to know to assess and enhance the security of our CLI applications.

Summary

In this chapter, we explored the various forms security can take, and we learned that it is not limited to the security of the application's code but rather covers a broad spectrum of areas that span its entire lifecycle. We learned what tools and techniques we can use to assess and enhance the security posture of our CLI application. We also learned how to secure access to user data, especially when dealing with external services, by implementing authentication and authorization.

Congratulations! You have now all the necessary knowledge and skills to build, secure, test, package, and release your very own CLI applications to the world. Look at you – what a milestone you have achieved. Take a moment to be proud of yourself and to celebrate!

My journey with you, throughout the pages of this book, is almost over. But before you turn the last page and gently close the cover of this book, there is one last thing I wanted to share with you, one final chapter whose purpose is to provide you with additional learning material to help you continue your journey as well as pointing you to some useful tools and libraries.

Your turn!

Following along with the provided code is a great way to learn through practice.

A better way is by challenging yourself to achieve tasks. Hence, I challenge you to improve the Bookmarkr application by adding the following features.

Task #1 – Update dependency versions

We have seen how to list outdated packages that have reached their end of support and talked about the importance of updating these. However, I haven't updated these.

Your task is to clone the code into your GitHub or Azure DevOps account and to update it.

Once the dependencies have been updated, you will need to validate that the behavior of the application was not impacted, in other words, you need to ensure that the application still behaves as expected. This can be done by running the tests you developed earlier.

Task #2 – Use Mend Bolt to scan the code for vulnerabilities

Your task is to clone the code into your GitHub or Azure DevOps account, enable Mend Bolt, and run a security scan. If any vulnerability is to be found, fix it!

Task #3 – Allow BookmarkrSyncr to manage multiple users

Although we updated BookmarkrSyncr to receive and validate a PAT, it does not use this token to retrieve and update the appropriate user's data.

You are tasked to update the code to make this happen. The easiest way to achieve this is to have a separate JSON file for every user whose name matches the value of the PAT. Hence, the bookmarks for every user can be stored and retrieved from there.

14
Additional Resources and Libraries

At this point, you have everything that you need to start building outstanding CLI applications. In the pages of this book, we have covered every step of the way: setting up your development environment, understanding the foundations of CLI applications, developing a CLI application – from the basic concepts to the most advanced ones, testing, security, performance optimization, and finally packaging and deployment.

However, while developing your CLI applications, you may face situations where you need to deepen your skills in one or more of these areas. You may also face situations where you will need to design a complex feature or code. These are situations every developer faces, no matter what kind of applications they build, and you will too.

That's why I designed this last chapter: to give you some guidance in these situations so you don't get lost. Use it as a reference whenever you need guidance.

Further reading and resources

While there are a lot of very interesting things to read on the web, I will mention some books from the Packt Publishing collection that I've found particularly useful.

C# 12 and .NET 8, by Mark J. Price

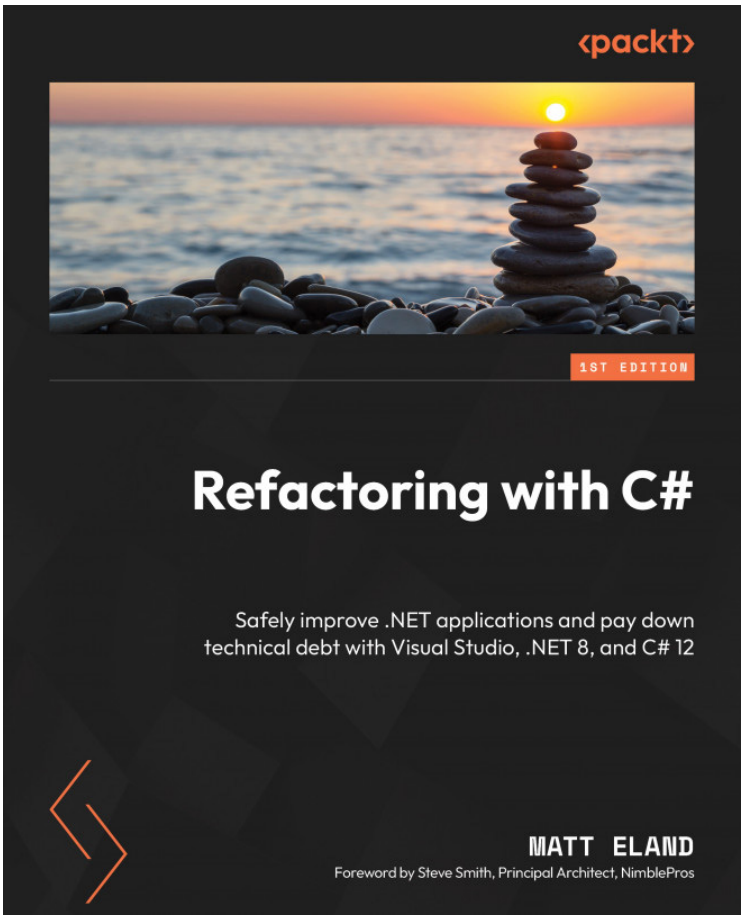
If you are learning .NET, it is impossible not to recommend Mark J. Price's book. This book covers everything you need to know about .NET development. Also, Mark updates his book with every new version of .NET. The latest edition, at the time of writing, is the one covering .NET 8.

To learn more about this book, please visit `https://www.packtpub.com/en-ca/product/c-12-and-net-8-modern-cross-platform-development-fundamentals-9781837635870`.

Refactoring with C#, by Matt Eland

When building your CLI application, you will most likely have to deal with existing code, and you may need to refactor it in order to make it safer and improve its performance. When this happens, you will be happy to have this book on hand.

By reading this book, you will learn many refactoring techniques that take advantage of the latest C# features. You will also learn how to leverage AI assistants, such as GitHub Copilot Chat, for your refactoring, testing, and documentation.

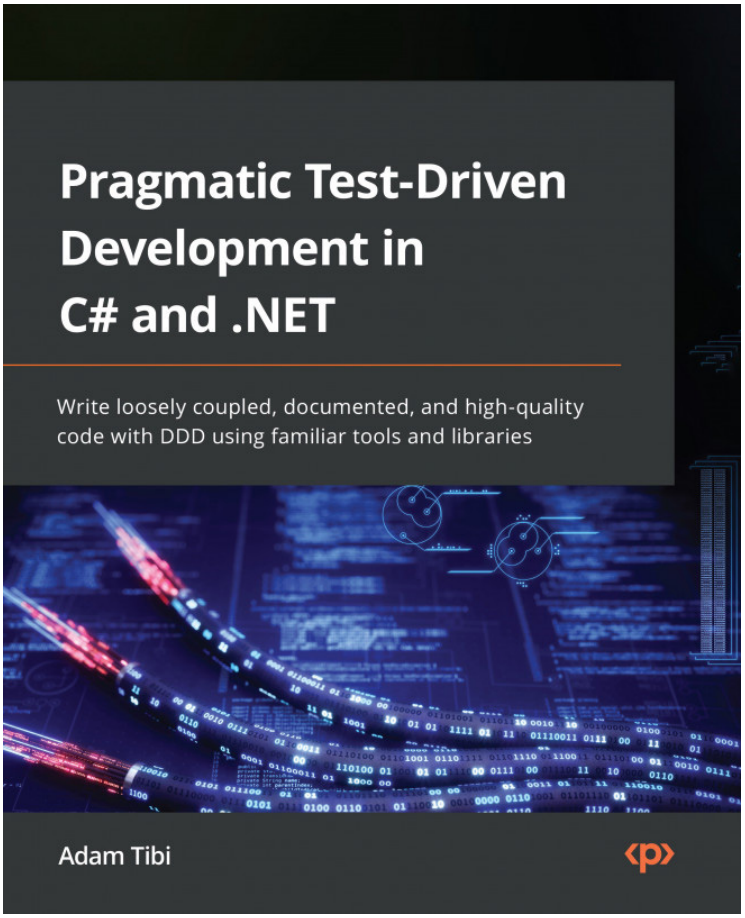


To learn more about this book, please visit `https://www.packtpub.com/en-ca/product/refactoring-with-c-9781835089989`.

Pragmatic Test-Driven Development in C# and .NET, by Adam Tibi

You simply cannot guarantee the quality of your application if you don't implement automated testing. Although you can implement automated testing without relying on **Test-Driven Development** (**TDD**), as I used to say in every class where I teach this topic: "*Automated testing ensures that your code performs the task it has been designed for, while TDD ensures that your code performs the task stated by your business requirements.*" This summarizes the difference between *simple* automated tests and the practice of TDD.

This book covers what you need to know not only to implement unit testing in your application (including mocking external dependencies) but also to integrate a TDD practice.

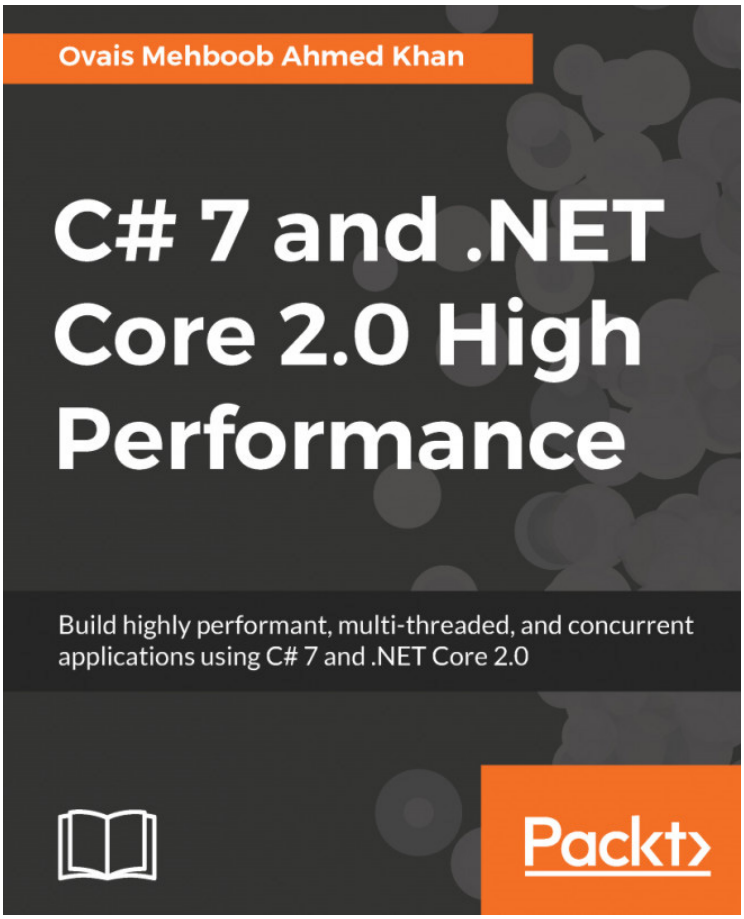


To learn more about this book, please visit `https://www.packtpub.com/en-ca/product/pragmatic-test-driven-development-in-c-and-net-9781803230191`.

C# 7 and .NET Core 2.0 High Performance, by Ovais Mehboob Ahmed Khan

Although this book relies on an older version of .NET, the concepts and principles it teaches are still valuable and applicable.

By reading this book, you will understand the most common pitfalls that negatively impact your application's performance. You will also learn how to measure the performance of your application (as a whole, but more importantly at its hot paths). You will then learn design and memory management techniques (including async programming, multithreading, and optimized data structures) that will help you improve the performance of your application.



To learn more about this book, please visit `https://www.packtpub.com/en-ca/product/c-7-and-net-core-20-high-performance-9781788470049`.

These readings will give you a deeper knowledge and understanding of various topics covered throughout this book.

However, you will find that, in many situations, a library already exists for what you are trying to achieve. You *should* consider using such libraries.

I'm highlighting "should" here on purpose. My point is that you don't have to use any library, but you have to, at least, disregard a library consciously after analyzing it and deciding that it doesn't suit your needs.

By relying on an existing library, you avoid having to write extra code, maintain it, and test it. You simply use it as a means to achieve your goal more quickly.

If you have been developing for a long enough period of time, you will most likely have encountered libraries that you found useful and that you reused over and over in your projects. In the next section, I will share with you some of the libraries that I have discovered (and used) along my journey.

Useful libraries for CLI application development

There are plenty of useful libraries and this section doesn't intend to list them all (we would need an encyclopedia for that 😉). Also, the ones listed here are by no means the best ones. They are the ones I've discovered along the way and use on a regular basis. These libraries add up to the ones we have already used throughout the pages of this book.

Polly

`Polly` is a resilience and transient fault-handling library. For CLI applications, it can help manage network failures, timeouts, and other transient errors when interacting with external services or databases. This ensures your CLI application remains robust and responsive even in unreliable environments.

To learn more about `Polly`, please visit `https://github.com/App-vNext/Polly`.

HangFire

`HangFire` is a background job processing library that does not rely on Windows services or a separate process. In CLI applications, it can be useful for scheduling and executing long-running tasks asynchronously. This allows your CLI application to handle time-consuming operations without blocking the main execution thread.

To learn more about `HangFire`, please visit `https://github.com/HangfireIO/Hangfire`.

StackExchange.Redis

`StackExchange.Redis` is a high-performance client for Redis. While primarily used in web applications, it can be beneficial in CLI applications that require caching or need to interact with Redis databases for data storage or retrieval. This allows your CLI application to retrieve data from an external service and cache it locally, reducing the dependency on a network connection while improving the overall performance of your application.

To learn more about `StackExchange.Redis`, please visit `https://github.com/StackExchange/StackExchange.Redis`.

MediatR

`MediatR` implements the mediator pattern, which can help simplify and decouple the communication between components in your CLI application. It's particularly useful for organizing command handling and implementing complex workflows.

To learn more about `MediatR`, please visit `https://github.com/jbogard/MediatR`.

MassTransit

`MassTransit` is a distributed application framework. It can be very useful for CLI applications that need to interact with message queues or implement event-driven architectures. For that matter, `MassTransit` makes it easy to create applications and services that leverage message-based, loosely coupled, asynchronous communication for higher availability, reliability, and scalability.

To learn more about `MassTransit`, please visit `https://github.com/MassTransit/MassTransit`.

BenchmarkDotNet

`BenchmarkDotNet` is a powerful benchmarking library. It's extremely useful for CLI applications that need to measure and optimize performance, allowing you to identify the slowest paths in your application, compare different implementations, and make data-driven decisions.

To learn more about `BenchmarkDotNet`, please visit `https://github.com/dotnet/BenchmarkDotNet`.

Portable.BouncyCastle

`BouncyCastle` is a cryptography library. It allows you to generate random passwords with different settings to meet the OWASP requirements. For CLI applications, this can be useful when you want to generate either a default password for a user account or a one-time passcode (for accessing an encrypted file, for example).

To learn more about `BouncyCastle`, please visit `https://github.com/novotnyllc/bc-csharp`.

NSubstitute

`NSubstitute` is a mocking framework for unit tests. It is useful for mocking external dependencies when performing unit tests.

To learn more about `NSubstitute`, please visit `https://github.com/nsubstitute/NSubstitute`.

AutoFixture

`AutoFixture` is an open source library for .NET that helps developers write maintainable unit tests by streamlining the process of writing and maintaining unit tests through the automation of test data generation. It integrates with popular testing frameworks (such as MS Test, NUnit, and xUnit) and with popular mocking frameworks (such as NSubstitute and Moq).

To learn more about `AutoFixture`, please visit `https://github.com/AutoFixture/AutoFixture`.

RichardSzalay.MockHttp

`MockHttp` is a testing layer for Microsoft's `HttpClient` class. It allows stubbed responses to be configured for matched HTTP requests and can be used to test your application's service layer without actually performing network calls. It is useful during the testing phase for mocking calls to the external APIs and services that your CLI application may rely upon.

To learn more about `MockHttp`, please visit `https://github.com/richardszalay/mockhttp`.

Summary

With the help of these readings and libraries, you can take your CLI applications to the next level and build some quite impressive and useful ones. I am eager to see what you will create. Do not hesitate to let me know.

One more thing: this list is not frozen in time. It is alive and you should keep it alive. How? By continually revisiting it and updating it as technology and trends evolve. I am passing this mission on to you.

Your turn!

Following along with the provided code is a great way to learn through practice.

A better way is by challenging yourself to achieve tasks. Hence, I challenge you to improve the **Bookmarkr** application by adding the following features.

Task #1 – List additional features for Bookmarkr

This is where I hand you **Bookmarkr** so you can make it your own.

I am challenging you to list some additional features you would like to have in **Bookmarkr**. You can add them as feature requests in GitHub so either you or someone else can implement them.

Task #2 – List the skills and libraries you need to implement a feature

Once you have decided what features you would like to implement, start by listing the skills and libraries you need to do so. The purpose of this step is to avoid having to reinvent the wheel but rather take advantage of what already exists and use it to achieve new horizons.

Remember: your users aren't interested in your coding skills (they won't even see the code). They are interested in the value your (CLI) application brings them.

Index

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X

packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

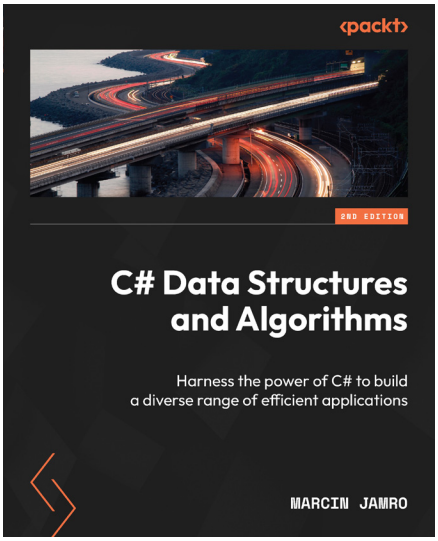
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at packtpub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

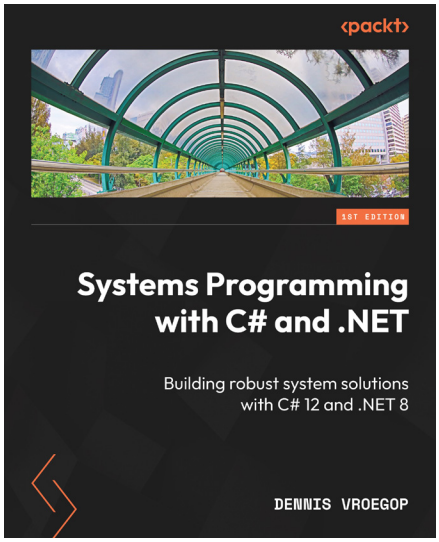


C# Data Structures and Algorithms, Second Edition

Marcin Jamro

ISBN: 978-1-80324-827-1

- Understand the fundamentals of algorithms and their classification
- Store data using arrays and lists, and explore various ways to sort arrays
- Build enhanced applications with stacks, queues, hashtables, dictionaries, and sets
- Create efficient applications with tree-related algorithms, such as for searching in a binary search tree
- Boost solution efficiency with graphs, including finding the shortest path in the graph
- Implement algorithms solving Tower of Hanoi and Sudoku games, generating fractals, and even guessing the title of this book



Systems Programming with C# and .NET

Dennis Vroegop

ISBN: 978-1-83508-268-3

- Explore low-level APIs for enhanced control and performance
- Optimize applications with memory management strategies
- Develop secure, efficient networking applications using C# and .NET
- Implement effective logging, monitoring, and metrics for system health
- Navigate Linux environments for cross-platform proficiency
- Interact with hardware devices, GPIO pins, and embedded systems
- Deploy and distribute apps securely with continuous integration and continuous deployment (CI/CD) pipelines
- Debug and profile efficiently, addressing multithreaded challenges

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share Your Thoughts

Now you've finished *Building CLI Applications with C# and.NET*, we'd love to hear your thoughts! If you purchased the book from Amazon, please `click here to go straight to the Amazon review page` for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

`https://packt.link/free-ebook/978-1-83588-274-0`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly