Software Architecture by Example

Using C# and .NET

Paul Michaels

Foreword by Mark Richards

Apress®

Software Architecture by Example

Using C# and .NET

Paul Michaels

Foreword by Mark Richards

Apress®

Software Architecture by Example: Using C# and .NET

Paul Michaels
Derbyshire, UK

ISBN-13 (pbk): 978-1-4842-7989-2 ISBN-13 (electronic): 978-1-4842-7990-8
https://doi.org/10.1007/978-1-4842-7990-8

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Smriti Srivastava
Development Editor: Laura Berendson
Coordinating Editor: Shrikant Vishwakarma

Cover designed by eStudioCalamar

Cover image designed by Pexels

To my wife, Claire, who has always been on, and at, my side,
even when I have not.

And to my dad, who has understood and supported me in everything
that I've done from the minute I was born.

Table of Contents

About the Author

Paul Michaels is the Head of Development at musicMagpie. He started his career as a professional software engineer in 1997. Paul is a regular speaker, published author, and Microsoft MVP. He enjoys programming, playing with new technology, and finding neat solutions to problems. When he's not working, you can find him cycling or walking around the Peak District, playing table tennis, or trying to cook for his wife and two children. You can follow him on Twitter at @paul_michaels or find him on LinkedIn. He also writes a blog at `http://pmichaels.net`.

About the Technical Reviewer

Kasam Shaikh is an Azure AI enthusiast, published author, global speaker, community MVP, and Microsoft Docs contributor. He has more than 14 years of experience in the IT industry and is a regular speaker at various meetups, online communities, and international conferences on Azure and AI. He is currently working as Senior Cloud Architect for a multinational firm where he leads multiple programs in the Practice for Microsoft Cloud Platform and Low Code. He is also a founder of the community named Dear Azure-Azure INDIA (az-India) and leads the community for learning Microsoft Azure. He owns a YouTube channel and website and shares his experiences over his website (`www.kasamshaikh.com`).

Acknowledgments

For this book, I have an absolute phone book of people to thank for their help.

Firstly, I have to thank my daughter, Abi, for all the artwork in the book.

Special thanks to Ash Burgess and Kevin Smith for putting up with impromptu requests for a discussion or a review of an idea, and for Kev's painstaking and repeated explanations of event sourcing.

Thanks to Ian Curtis for reviewing my architectural diagram, and to James Little for making me think so hard about architecture that a book seemed a good idea.

The travel industry is very complex, and I don't think I could have landed in a better place, where I had the expertise and advice of both David Hilton and Jagdip Ajimal.

When it came to containers, I found that what I was trying to do kept throwing up roadblocks until Rob Richardson generously offered his time and expertise.

The examples in this book are based on my time working across industries and trying to solve problems. I've been fortunate in my career to work with a number of very clever and talented people. There are probably too many of these to mention, and while they may not have directly helped with this book, I'd like to acknowledge their contribution.

Finally, I'd like to thank the team at Apress – especially Shrikant for dealing with the various twists and turns that the book, and my life, has taken during its creation, and Smriti for bringing me onto Apress in the first place.

Introduction

All the code in this book is available from the following GitHub repo:

`https://github.com/Apress/software-architecture-by-example`

Should you choose to follow along and create the solution for each chapter, it may be helpful to have a clone of the code available for reference.

Note As I'll be working on a Windows machine, what I do will only be tested on Windows; however, again, this is software architecture, not software architecture for Windows, so everything should also work on MacOS, Linux, or any other modern OS: .Net is a cross-platform framework.

Technology

Let's quickly discuss the specific technologies that we're going to choose and why. Firstly, I will use Visual Studio to write all of the code samples in this book. You can download the community edition of this here:

`https://visualstudio.microsoft.com/vs/community/`

You may also wish to use VS Code, which can be found here:

`https://code.visualstudio.com/download`

All of the applications that we create will be in .Net and written in C#. This choice I made simply because it's the language that I'm most familiar with; however, I don't believe there's anything in here that couldn't be translated to any other modern OO development language; after all, architecture should be language agnostic. Most of the principles are broader than a specific language and could apply to any language capable of making HTTP calls.

Setup

In this section, we'll cover the basic setup that you'll need to follow along with the code samples. However, since this is predominantly a book on architecture, you should be able to translate the concepts to any language.

Let's cover a basic setup for those readers that wish to follow along.

Terminal

If you choose to use VS Code, you have a terminal built in; however, there are other options.

Since you're likely to be using Git, you can easily use git bash for the terminal commands; you can download **git for windows** here:

`https://gitforwindows.org/`

Another possibility if you're on Windows is the new (at least at the time of writing) Microsoft Terminal. This can be found here:

`https://aka.ms/terminal`

Note This is, in fact, an open source product; you can find the source code for it here:

`https://github.com/Microsoft/Terminal`

Examples

The title of this book is *Software Architecture by Example*, so it will not surprise you to learn that there are examples in each chapter. The purpose is to propose a problem, suggest one or more solutions, and provide an example of how that solution might work in reality.

What this doesn't mean is that contained within the pages of this book are full, complete, solutions to each problem. To illustrate my point by example, for the first chapter, we address the problem of a business that sells tickets for concerts and festivals.

In that chapter, there are code samples that will compile and run, but those samples are for illustration purposes; there's no website there, but I've made sure that each element of the system is there by proxy – so the website will be simulated by a console app.

How to Use This Book

There are a number of ways that you may choose to use this book. Each chapter has an explanation of an architectural principle, driven by the typical requirement that it satisfies; once this has been explained, there is an example in each chapter.

All of the code for every chapter can be found here:

https://github.com/Apress/software-architecture-by-example

You can choose to follow along and recreate the examples, or you can clone the repo and simply view the code, or you may decide that you're not interested in a specific implementation, in which case, you can simply skip the examples altogether.

Foreword

The topic of software architecture is hard to describe, teach, and learn, mostly because no one really knows what it is. Some say it's the structural aspect of a system, similar to an architectural blueprint of a large office building or skyscraper. Some say it's how different parts of a system interconnect or interact with one another. Others say it's the foundational aspects of a system that meets certain business goals and needs, irrespective of the system functionality. So who's right? Well, in fact, they all are, which is why it's so hard to explain and teach software architecture.

Decades ago, a software architect primarily focused on the technical aspects of a system – how the various parts, or components, of a system interacted with one another through various interfaces, contracts, and protocols. Today, however, software architecture impacts and influences so much more, including business alignment, data, deployment environments, methodologies, platforms, and so on. These intersections and necessary corresponding alignments have significantly expanded the role of a software architect. In addition to technical skills, a software architect must also possess exceptional people skills to be able to collaborate and negotiate with numerous business and technical stakeholders to ensure that the architecture is aligned with all these factors. With all this responsibility in an ever-expanding role, it's no wonder why software architecture is so difficult to understand.

Back in 2010, a well-known architect named Ted Neward came up with the notion of an "architectural kata" – a way of being able to practice software architecture, much in the same way different moves, or forms, are practiced in martial arts. These small, targeted exercises provide a context to practice some of the core skills an architect must hone to become effective – identifying important driving characteristics ("-ilities") that the architecture must support, identifying possible solutions, analyzing the trade-offs of these solutions, and making architecture decisions.

Through the years, I have found that *teaching by example* helps bring students (and readers) from the abstract to the concrete, allowing them to better understand not only what software architecture is but also why it's so vital to the success of any system, which brings me to this book. Through the use of concrete examples, Paul Michaels helps the reader understand some of the core abstract concepts of software architecture. This form of teaching not only makes the connection between the abstract and concrete but also gives the reader a chance to practice these concepts. As Paul states in this book, “In software, as in life, everything has a price.” Leveraging practical examples is one way of being able to learn how to perform the trade-off analysis necessary to arrive at the most appropriate architectural solution. After all, as we all know, “…it depends.”

—Mark Richards
Founder, DeveloperToArchitect.com
Author of *Fundamentals of Software Architecture* and
Software Architecture: The Hard Parts

CHAPTER 1

The Ticket Sales Problem

When I first started out in IT, the industry was still quite niche. Although many people used computers in their day job, access to the Internet or even a personal computer at home was still a way off for most people. This was around the time that the industry was gearing up to fix the millennium bug: an issue that was prevalent in many systems, because 20 or 30 years earlier, when these systems were written, the programmers had assumed they would have been replaced before the year 2000.

At this time, if you wanted to go and see a band play, you typically had two choices: you physically visited a record shop or ticket sales venue, for which the queues often stretched out of the door, or you used the telephone. These days, you wouldn't have to go far to find someone young enough to not remember such times, but it's worth bearing in mind how far we have come in 20 years!

P. Michaels, *Software Architecture by Example*, https://doi.org/10.1007/978-1-4842-7990-8_1

The queues stretching out of the door of the ticket booths have now been replaced by millions of people accessing a website at the same time to try and buy tickets. In this chapter, we'll be discussing one of the most prevalent issues in the IT industry today: how to cope with massive spikes in traffic.

Background

Our new client, ***123 Tickets***, has asked us to replace their existing system. The existing system that they run works fine for most of the year, but the company makes most of their revenue from just three dates, when they are contracted as the main reseller for premier music festivals. Their existing system simply can't cope with the huge spike in sales and frequently crashes just minutes after the tickets go on sale.

The venues are unhappy with ***123 Tickets***' ability to cope with the sales, and the contracts are in danger of not being renewed.

Let's have a look at last year's usage. Figure 1-1 shows a graph with the usage and error statistics from last year.

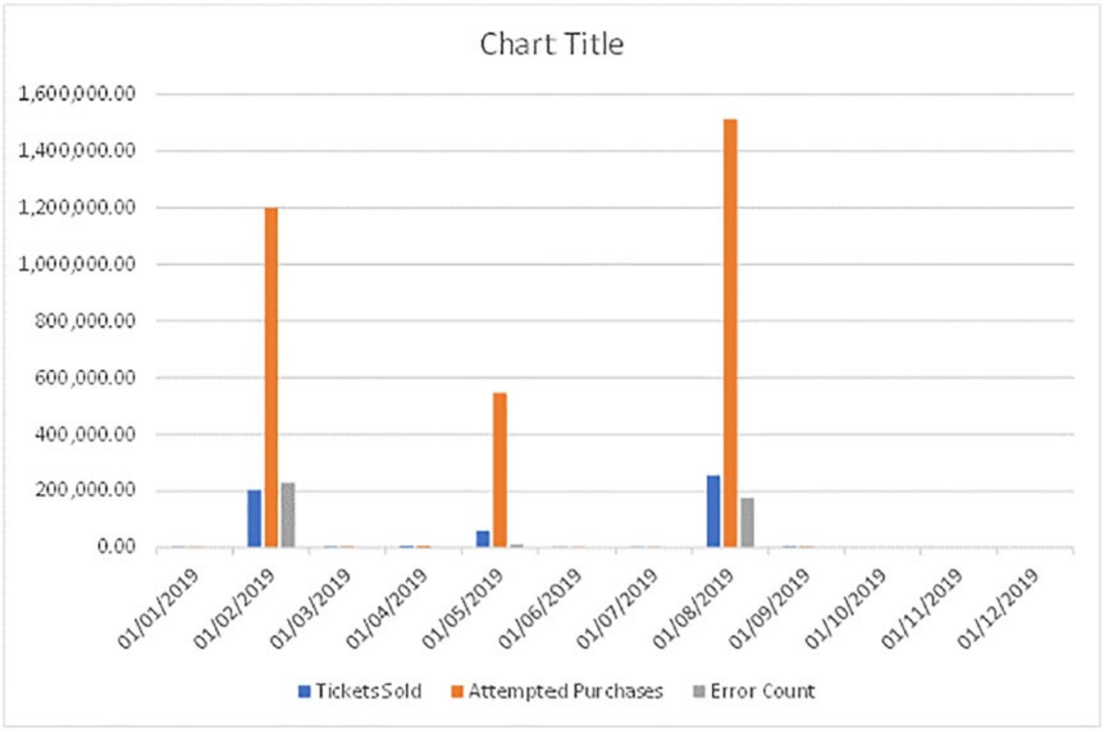

Figure 1-1. *System usage graph*

In Figure 1-1, we can see some very useful information. Firstly, we can see that almost all the business that this company has conducted is during three months of the year; and we can see that when the system is busy, the system errors spike. We can also see that the demand for the tickets far exceeds the supply.

Let's consider exactly what the requirements from ***123 Tickets*** are.

Requirements

Whenever a system, of any type, is designed, a target should be established. For example, if you're designing a car, your target is a vehicle that transports one or more people between places and is roadworthy (whatever that may mean in your locality). The fact that your car may have three wheels, or two doors, or be painted blue is an optional feature; that is, the car is still a car if it has two, four, or five doors; it's still a car if it is blue or red; however, it is not a car if it has no wheels because it would be unable to fulfill its requirement of transporting people.

When designing a system, it's always worth considering this: what the system needs to do in order to fulfill its basic function. For example, our ticket ordering system presumably needs to allow people to purchase tickets – if it did not, we could not sensibly call it a "ticket ordering system"; but does it need to allow people to purchase ice cream when they arrive at the venue? Probably not, as without that, it's still a "ticket ordering system."

We should, therefore, discuss with the client the list of things that the system needs to do in order to *be* the system; and all the time, we should challenge whether that thing is necessary. To clarify, I'm not saying that anyone should sit in front of a client and argue them into submission about features that they are requesting and willing to pay for; however, we may decide that what is being described is not a single system, but two, or three. Why this is useful is something we'll revisit later in this chapter.

I'm very purposely staying away from any reference to software at this stage, and the reason will become clearer later on.

Let's lay out exactly what we need the system for ***123 Tickets*** to do. This list is a high-level list of features that the current system provides and which the client has identified we would need to provide:

- Maintain a list of registered users.
- Provide a list of upcoming events for which there are tickets available.

- Allow a user to purchase up to ten tickets for any single event.
- Maintain a count of available tickets.
- Allow users to pick a seat where applicable – not all events are seated (and none of the big festivals are seated).

Now that we've identified what's required, we can discuss the options for providing that.

Options

All too often, software developers and architects reach for the tools that they know best. I'm no exception; any code samples that you'll see in this book are written in .Net. However, exploring other possibilities is not only a useful exercise but also solidifies the requirements in our minds. In each chapter, I'll make the case for solving the problem *without* using technology.

It may seem like a strange thing for a book on software architecture; however, all over the world, people are solving problems without technology; in some cases, that's the best solution. Software design and development costs money; in some cases, it costs a considerable amount of money, and it is not without risk. According to a 2017 report from the Project Management Institute, between 6 and 24% of projects end in failure. These are not only software projects; however, if we accept that as a rough guide, it means that we can reasonably expect around one in ten software projects to fail (source: `www.pmi.org/learning/thought-leadership/pulse/pulse-of-the-profession-2017`).

In our case, ***123 Tickets*** has an existing system, but let's imagine that our advice to the client is to remove that system and replace it with a manual process. What would that look like?

Manual Process

First of all, we would need to maintain a list of valid users for the system; we could keep this in an address book. Each time someone wished to be added to the system, we would write their name and address into our address book; the maintenance of this book would represent all or part of somebody's job.

Secondly, we would keep a list of events; presumably, we'd use something like a yearly diary to do so; each event would be marked in on the day it was to happen. Somebody would then go through every event for the following two or three months and write on a sheet of paper what, and when, these events were.

Our next step would be to order the tickets from the supplier – when they arrive, our ticket count would simply be that somebody would simply count the remaining tickets for each event.

When a customer phoned up, the operator would go through the following process:

1. Ask for a name, and look them up in the phone book; if they are not already in there, then add them.
2. Check the event that they wished to book a ticket for and ensure that there were sufficient tickets.
3. If the venue is seated, talk through the options for seating with the customer, and establish which tickets would be best.
4. Put the tickets in an envelope (so that they cannot be sold to another person) and take payment details.
5. If the payment fails, or the customer changes their mind before payment is made, the tickets are returned to the pile for that event; otherwise, they are posted to the customer.

In fact, when we consider this, we realize that the manual process is actually quite neat; maybe this is the right approach. Of course, there's a minor snag; even during the smallest festival, over 500,000 attempts were made to purchase tickets; however, before we abandon our manual approach, let's just continue this thought experiment for another few paragraphs.

Let's say that we did need to implement this manually and we had a single operator. What would happen if 500,000 or more people tried to phone in to buy tickets at the same time? Well, the way most basic phone systems work is that the first person would be connected, and until that sale had finished, everyone would get an engaged tone.

So how could we structure this so that, given enough time, we could deal with all these requests? One possible solution may be to divert the calls to an answering machine service (for the purpose of this, we'll assume that the answering machine can take multiple calls at any one time without the caller getting an engaged tone), asking the customer to leave details of the venue and ticket requirements; the operator could then phone each person back as they became free.