LLMs, Semantic Kernel, Flowise, C#

CODE

MAR
APR
2024

codemag.com - THE LEADING INDEPENDENT DEVELOPER MAGAZINE - US $ 8.95 Can $ 11.95

CODE 30 YEARS

Title

Subtitle

Subtitle

Subtitle

Cover AI generated - Markus Egger

03
7 25274 56130 8

CODE
CONSULTING

ARE YOU WONDERING HOW ARTIFICIAL INTELLIGENCE CAN BENEFIT YOU TODAY?

EXECUTIVE BRIEFINGS

Are you wondering how AI can help your business? Do you worry about privacy or regulatory issues stopping you from using AI to its fullest? We have the answers! Our Executive Briefings provide guidance and concrete advise that help decision makers move forward in this rapidly changing Age of Artificial Intelligence and Copilots!

We will send an expert to your office to meet with you. You will receive:

1. An overview presentation of the current state of Artificial Intelligence.
2. How to use AI in your business while ensuring privacy of your and your clients' information.
3. A sample application built on your own HR documents – allowing your employees to query those documents in English and cutting down the number of questions that you and your HR group have to answer.
4. A roadmap for future use of AI catered to what you do.

AI-SEARCHABLE KNOWLEDGEBASE AND DOCUMENTS

A great first step into the world of Generative Artificial Intelligence, Large Language Models (LLMs), and GPT is to create an AI that provides your staff or clients access to your institutional knowledge, documentation, and data through an AI-searchable knowledgebase. We can help you implement a first system in a matter of days in a fashion that is secure and individualized to each user. Your data remains yours! Answers provided by the AI are grounded in your own information and is thus correct and applicable.

COPILOTS FOR YOUR OWN APPS

Applications without Copilots are now legacy!

But fear not! We can help you build Copilot features into your applications in a secure and integrated fashion.

CONTACT US TODAY FOR A FREE CONSULTATION AND DETAILS ABOUT OUR SERVICES.

codemag.com/ai-services

832-717-4445 ext. 9 • info@codemag.com

Features

Departments

US subscriptions are US $29.99 for one year. Subscriptions outside the US pay $50.99 USD. Payments should be made in US dollars drawn on a US bank. American Express, MasterCard, Visa, and Discover credit cards are accepted. Back issues are available. For subscription information, send e-mail to subscriptions@codemag.com or contact Customer Service at 832-717-4445 ext. 9.

Subscribe online at www.codemag.com

CODE Component Developer Magazine (ISSN # 1547-5166) is published bimonthly by EPS Software Corporation, 6605 Cypresswood Drive, Suite 425, Spring, TX 77379 U.S.A. POSTMASTER: Send address changes to CODE Component Developer Magazine, 6605 Cypresswood Drive, Suite 425, Spring, TX 77379 U.S.A.

LEAD Tools

Are You Ready to Pivot?

As they like to say, a picture is worth a thousand words and the picture you see below (Figure 1) is proof positive. This editorial you're reading is a direct result of the ideas I "chicken-scratched" on a stack of post-it notes. This set of notes came from a conversation Melanie Spiller and I were having about collaboration, in particular the concept of remote collaboration. We discussed at length how our work, and how we perform it, was greatly transformed during the COVID crisis. COVID affected every one of us in profound ways and we were forced to create new ways of working together, albeit remotely. We had to change how we collaborated.

A lot of the conversations I have with Melanie about creativity and collaboration center around many non-tech fields. Every creative endeavor shares some a common DNA and by examining other fields not related to your own, you might find newer ways of thinking. At a minimum, you'll likely learn something you don't know. That's a win/win, right?

Well, this particular conversation was focused on one creative field: music. We talked about how COVID changed the way musicians were forced to change their collaborative process during the pandemic.

Before COVID, I was taking guitar lessons. My teacher came to my house twice a month for an in-person session where he taught me the foundations of guitar playing, including scales, chords, and, what was probably the most difficult aspect for me, timing. Scales and chords were probably the easiest part. With practice, I got better and better as our lessons progressed. It was that cursed timing that caused me the most agita. The metronome was not my friend and, to be honest, I never really got the hang of it. Over and over, he beat it into me that if I ever wanted to play with other folks, I needed to get timing down. It was this timing thing that was part of my discussion with Melanie.

Along with being a darn fine editor, Melanie is an accomplished musician and she collaborates with numerous musicians in many different organizations. In normal times, these collaborations take place in person. Rehearsals, practice, organization—all done in person. COVID changed all of that. It didn't take too long for these musicians to realize that creating the fidelity of in-person musicianship was difficult to pull off remotely and, in the beginning, impossible. The reason? Timing. Melanie and her collaborators soon discovered the bane of remote collaborations using ordinary tools such as Zoom: lag. Technology people are intimately familiar with the concept of lag and there are a bunch of different terms used to describe lag, such as response times, ping times, TTL, etc. We strive to lower lag to make our work perform better or to at least to have the appearance of better performance. Progress bars anyone?

Melanie and her musician friends couldn't paper over these lag issues. It would be difficult to create great performances, let alone satisfying and productive rehearsals, if they couldn't control the issue of lag. So they searched for a solution to controlling or eliminating lag. This is where they found a product called Jamulus.

The Jamulus website describes their product as follows:

Jamulus lets you play, rehearse, or jam with your friends, your band, or anyone you find online. Play together remotely in time with high quality, low-latency sound on a normal broadband connection.

The key phrase in this statement is low-latency. The folks who created Jamulus created a solution that enabled remote collaborations that was as close to an in-person experience as possible. This group of intrepid musicians found a tool that helped them to move forward. They soon adapted their process to Jamulus's rather rigid requirements with great and satisfying success.

The musicians weren't the only folks that adapted their processes. My wife and I are currently binge-watching a show on Netflix called The Blacklist. The episode last night completely blew my mind. It was the season finale of season 7. This episode was in the process of being filmed during the worst part of the COVID crisis when everything was being shut down. The episode began with a prologue of clips from the cast and crew where they mentioned that they were halfway done shooting the episode when they were forced to shut down.

But they didn't shut down. They retreated temporarily and came up with a solution. The parts of the film that weren't shot live-action were animated. It was a bit odd, but it worked! As a film geek, it was an additional treat for me to see how a show is shot out of continuity using a real-world example. Changing how they worked allowed these collaborators to accomplish what they probably thought was an impossible task. As Gunny Seargent Thomas Highway would say: They overcame, they adapted.

This is what we do as human beings. We overcome obstacles. We adapt and we move forward. How have you adapted?

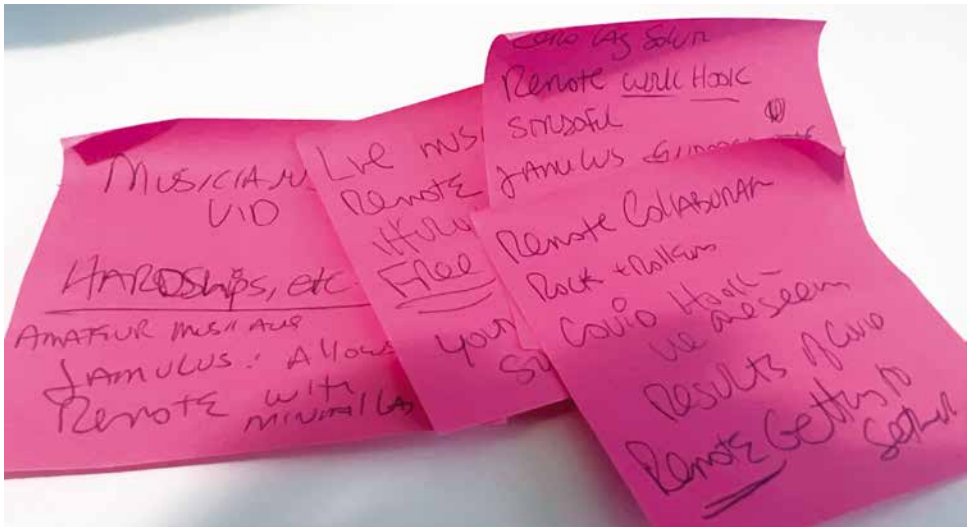
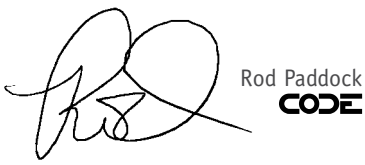


Figure 1: The origin of the species



Rod Paddock
CODE

RavenDB

The Boring Database

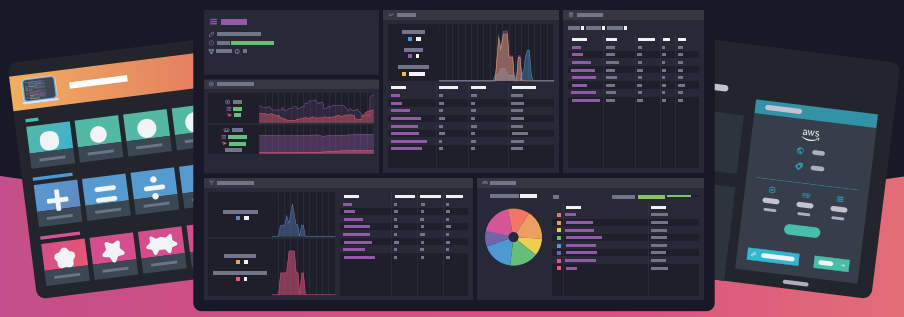
Predictable

All-in-One

Effortless

"After years of fixing mistakes that others did in SQL, developing a world-renown application to find common ones, I realized that the problem is not with people, but with technology. That day I started RavenDB."

Oren Eini
CEO & Founder



Three Ways to Experience RavenDB

ravendb.net/try

CODE: 25 Years Ago

As we continue our celebration of "30 years of CODE Group," we travel forward five years from last issue's installment of "the Antique CODE Show" and look at our industry, and the world in general, 25 years ago (or so). The late 90s were an interesting time for our industry. Compared to five years earlier, the internet had rapidly become mainstream. Although hardly anyone outside the software industry had heard of it in 1993, hardly anyone had *not* heard of it by 1998.

Markus Egger
megger@codemag.com

Markus, the dynamic founder of CODE Group and CODE Magazine's publisher, is a celebrated Microsoft RD (Regional Director) and multi-award-winning MVP (Most Valuable Professional). A prolific coder, he's influenced and advised top Fortune 500 companies and has been a Microsoft contractor, including on the Visual Studio team. Globally recognized as a speaker and author, Markus's expertise spans Artificial Intelligence (AI), .NET, client-side web, and cloud development, focusing on user interfaces, productivity, and maintainable systems. Away from work, he's an enthusiastic windsurfer, scuba diver, ice hockey player, golfer, and globetrotter, who loves gaming on PC or Xbox during rainy days.

Netscape was the dominant player in the browser market, one of the hottest areas of software development, back then. So much so, that this is generally thought of as the era of "the great browser wars," with Microsoft pushing into the market with Internet Explorer and going as far as—gasp—bundling the browser directly with the operating system. Although it's unthinkable today that any PC or device wouldn't support direct internet access, back then, this was enough to cause 20 U.S. states to file antitrust charges against Microsoft. The aim was to "determine whether the company's bundling of the browser with the OS constituted an unfair monopolistic practice." Although it was initially ruled in 2000 that Microsoft had violated antitrust laws and the company was ordered to be split into two entities, that ruling was later overturned on appeal, and Microsoft was allowed to continue operating as a single entity (although under some limitations that affected the organization for a long time to come).

But it wasn't all about Netscape and Microsoft. The music industry was turned upside down due to the emergence of Napster, a peer-to-peer music sharing system that unraveled the business model of the entire industry. At the height of its popularity, Napster had 80 million registered users who happily shared music for free (and arguably illegally) with their peers, until the service was shut down in 2001. At the time, artists that topped the charts included names like Britney Spears, the Backstreet Boys, and Eminem. It's hard to say which songs were the most shared, but Madonna and Elton John songs were up there, and the artists were also among the more outspoken against the practice of peer-to-peer sharing for free. Eventually, the Metallica vs. Napster lawsuit was the start of the end for the service.

The late 90s were also a great time for moviegoers (which included me back then). Films such as "Titanic" (1997), "Saving Private Ryan" (1998), and "The Matrix" (1999) were among the most popular. It's hard to believe that it's been a quarter century since their release. Among the most important software companies that got established in the late 90s was Google (September of 1998). Although it's hard to believe that James Cameron finished "Titanic" more than 25 years ago, it's even harder to believe that Google is younger than this movie, as it's difficult today to imagine a world without Google.

Politically speaking, times were a bit simpler, as our main concern was Bill Clinton's love for cigars and interns. Internationally, nuclear tests by Pakistan and India were the hot topic. The Good Friday Agreement in April of 1998 brought an end to the violence in Northern Ireland. In science and medicine, we hadn't yet completely decoded human DNA, but on the upside, Viagra was approved by the FDA in March of 1998. If you're a reader who was already in the industry back in 1998, this development may have more significance for you today than you imagined back then.

This was also the heyday of the dotcom bubble. The internet had grown rapidly, and everyone wanted to invest in the wildest ideas. Venture capital investment went through the roof and so did IPOs. Very few had solid ideas of how any of this was going to make any money, but "the new economy" didn't bother much with such detail. Who cares, when you can party like it's 1999? The NASDAQ stock exchange grew five-fold in those years, until it all blew up and it plunged from a peak of $5,048.62 on March 10, 2000, down to $1,139.90 on October 4, 2002 (which takes us almost into the timeframe of the next article). Among the biggest and most visible losers of the dotcom era are Pets.com, Boo.com, Kozmo.com, and eToys. However, there weren't only losers. Amazon.com, eBay, Google, and several others, are not just still around but have grown into some of the largest and most influential companies on the planet. I guess our ideas weren't all crazy in those days after all.

Absent from the list of influential companies in the late 90s is Apple. Teetering on the brink of bankruptcy, Steve Jobs had just returned to Apple in 1997. Even Microsoft invested $150 million into Apple in August of 1997, which, in hindsight, was probably not insignificant as a factor in the survival of the company, as it helped stabilize the finances of the company and allowed it to subsequently go on to develop new products such as the iMac and the iPod. But that was all still in the future and Apple clearly wasn't very cool in 1998, except for some niche markets, such as desktop publishing. (When we released the first issues of this magazine, it was all done on Macs).

Talking about CODE: As far as CODE Group goes, we were around as a small but growing consulting and training organization. Our headquarters had already moved to Houston, Texas. We did a lot of consulting and training and were very active in Windows and internet-based development and had started architecting systems that weren't just object-oriented, but we were starting to think about component-based multi-tiered systems, rather than just "network enabled" software. This was a big deal at a time, and ultimately lead to the "CODE" brand, which originally stood for **CO**mponent **DE**veloper. *CODE Magazine* hadn't published its first issue yet (because the overall CODE Group organization goes back further than the magazine), but this is when we first had the idea of publishing a developer magazine that didn't just focus on a single programming language (such as *Visual Basic Programmer's Journal*, *FoxPro Advisor*, or *C/C++ User's Journal*) but covered software development at a larger, more technology-oriented level. The idea was laughed at by many at the time, but has served us very well ever since.

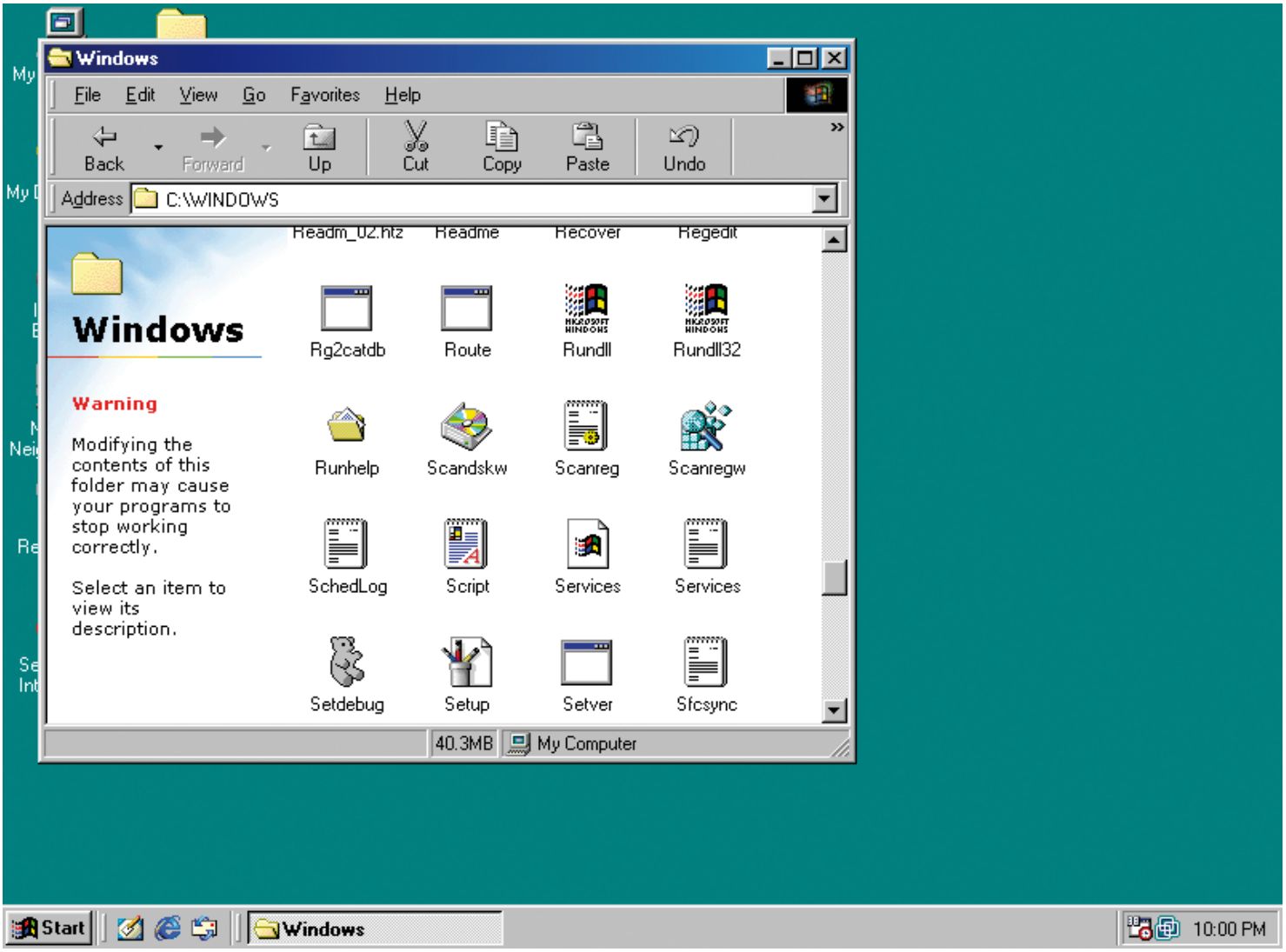


Figure 1: The Windows 98 Desktop with File Manager

The Technology Landscape

Windows 98 was the operating system many of us used at the time. It tends to be a somewhat less prominent release of Windows, between the major milestones that were Windows 95 earlier, and the subsequent and excellent Windows XP. However, Windows 98 was a very solid version of Windows that improved on Windows 95 in a variety of ways. Internet Explorer 4.0 shipped out of the box and was a very solid browser that supported (and pushed forward) the latest version of HTML. It's easy to forget that Microsoft was instrumental in making browsers and HTML more of a real development environment, supporting JavaScript, VB Script, and a DOM (Document Object Model), which allowed programmers to interact with elements on the page in ways that were previously unthinkable for what used to be a simple document and text display mechanism with hyperlinks. In 1998, if you wanted to look at a modern website, Internet Explorer was a good choice indeed, as Netscape had an increasingly hard time keeping up with Microsoft.

The other parts of the operating system were impressive too. USB was supported for the first time, which was a great step toward simplifying the world of peripherals. All of a sudden, it became a lot easier to use printers, scanners, and cameras. Windows 98 had also improved support for devices such as sound cards, video cards (we didn't call them GPUs back then), and modems (yes, we were still dialing up to the internet and paying horrendous phone bills). Windows 98 was also much faster than Windows 95, with improved memory management and disk caching. The GUI (graphical user interface) of Windows 98 was similar to that first released in Windows 95 (**Figure 1**). That means a lot of "battleship gray" applications, as well as a Start Menu that wasn't searchable. All in all, Windows 98 was a great choice as an operating system for a user's PC.

As far as hardware is concerned, users hardly ever had PCs with more than 32MB of RAM, sporting an Intel Pentium processor (**Figure 2**). Spinning hard disks with 4GB of storage was the norm. If you were modern, your software was installed from a CD-ROM drive. The speed of that drive mattered, and if you wanted to be among the cool kids, you needed a drive that was "24x or faster." You probably also looked at an SVGA CRT monitor back then. That's right

Figure 2: A typical setup of a PC running Windows 98, sporting both a 3 ½-inch floppy drive and a CD drive

kids: We were looking at 800x600 pixels of resolution back then, and we loved it! Now get off my lawn!

But Windows 98 wasn't the only Microsoft operating system that was popular in the late 90s. Microsoft also continued to develop Windows NT as the operating system of choice for enterprise servers. NT4 was released in 1996 and would be heavily in use for a long time to come. It was built on NT 3.5 and NT 3.51, which had introduced the 32-bit kernel and a new file system (NTFS), and Active Directory. It's a great operating system, and we still use the successors of much of that technology in modern versions of Windows today. Many developers even opted to use NT4 as the operating system of choice for their development machines back then. It was a bit clunkier for end-users than Windows 98, but it also had a very solid core. For technical people, it wasn't a bad trade-off at all.

This was also when laptop computers became far more popular and mainstream. Although they'd been around for a while, it wasn't until the late 90s that I started experimenting with using a laptop as my main workhorse machine. I still switched back and forth between that approach and more conventional desktop setups (especially for gaming) but having laptop computers with decent horsepower and reasonably good color displays just hadn't been a thing much earlier.

Speaking of gaming: Looking at popular titles, this was when we started recognizing key players and brands that still have significance today. Valve released *Half-Life* (later, *Half-Life 2* was the start of Valve's now dominant *Steam* gaming marketplace) in 1998. *StarCraft* eventually sparked competitive gaming and eSports. *Unreal* by Epic Games was where the *Unreal Engine* originally came from. *Ensemble Studios* (basically Microsoft) released *Age of Empires*, and *Bioware* gave us the first *Baldur's Gate* game (**Figure 3**). This might confuse you because 2023's game of the year is also *Baldur's Gate*, but that's the successor to the classic game. Both are highly recommended to any avid PC gamer.

Figure 3: The isometric (and very low-res and fuzzy) goodness that is Baldur's Gate 1 in 1998.

But PC gaming wasn't all there was. Sony had already released their PlayStation brand a few years earlier, and Microsoft was hard at work on the first Xbox, which was to become a competitor to PlayStation 2 as well as Nintendo's GameCube, which itself was a successor to Nintendo's successful Nintendo 64 (**Figure 4**). Games like *Mario Kart* were all the rage on that platform but would ultimately be surpassed by the amazing next-generation experiences provided by the Xbox and the PS2. But that was still in the future.

Software Development

The world of software development in the late 90s was both in flux and also relatively calm, in hindsight. It didn't seem like it at the time, because we had to come to grips with the realities of the internet and many software developers were still wondering whether this graphical user interface experience was actually good for business applications and many DOS die-hards were still holding on by the skin of their teeth. But the debate had pretty much been settled for most of us then: Windows desktop apps were what serious developers were building, and you just couldn't ignore the internet either. We were developing what seemed to be relatively sophisticated interactive web pages at the time through the use of ISAPI web-server extensions. Just imagine the power: Rather than writing static files with a .html extension, the user could hit a URL and a piece of code could respond, figure out what the user wanted, and send back a string of HTML that was generated on the fly! It seemed as though we'd touched the future (and so we had!).

If you were truly daring, you even dabbled in XML, which was a newfangled standard based on angle-brackets that kinda looked like the then-still-new HTML, but it had more structure. To what end, many people wondered? There was much speculation about web browsers being able to make more sense of XML than HTML, but many doubted its potential. I remember sitting in a keynote at a sizable computer conference in the Netherlands, where the presenter

Figure 4: The Nintendo 64 was one hot gaming device in the mid-to-late 90s!

Figure 5: The login screen to Microsoft's ill-fated competitor to the entire internet was called "The Microsoft Network." The name survives as MSN, which is now one of many web sites.

questioned the point of XML and predicted it would go nowhere. He also predicted many other things, such as the demise of Microsoft. He questioned the concept of "visual" development (as in Visual Basic or Visual C++). Literally none of his predictions came true. I forgot the name of the presenter, but I often wondered what became of him.

The most popular programming languages of the late 90s were Visual Basic, C++, and Java. There were also many second-tier languages that achieved a wide degree of popularity, such as Visual FoxPro, Delphi, Pearl, and Python. Some of these rose from there, while others were either at end-of-life or lost significance. A good example of the latter is Visual Basic, which had its heyday in the world of Windows desktop application development. It's clear that its approach to software development has been among the most impactful contributions in business application development, period. It still continued to do so for several decades and into the world of .NET, even when C# took the crown of being the most popular language in the Microsoft .NET ecosystem.

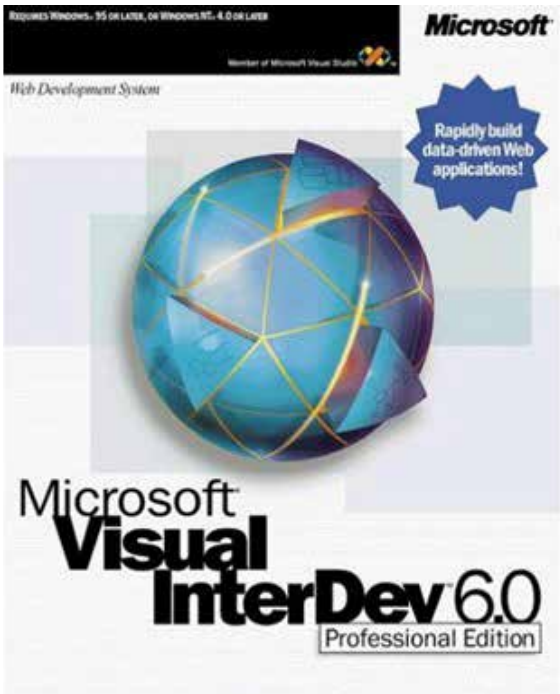


Figure 6: The Microsoft Visual InterDev CD sleeve

C# itself was born as a brainchild of Anders Hejlsberg in the late 90s (and publicly announced in the summer of 2000). Anders is considered the father of many programming languages and has clearly had a dramatic impact on programming languages in general. He's the original author of Turbo Pascal and the Chief Architect of Delphi. Once he joined Microsoft as a Technical Fellow, he created C# (originally codenamed Cool) in an effort to fix some of the shortcomings of Java. He also later created the popular TypeScript language to fix the shortcomings of JavaScript. On a more personal note, Anders is also one of the most impressive yet nicest people I ever had the pleasure of working with (full disclosure: I worked for the Microsoft Visual Studio team as a contractor in the late 90s, but my personal involvement with C# was not huge).

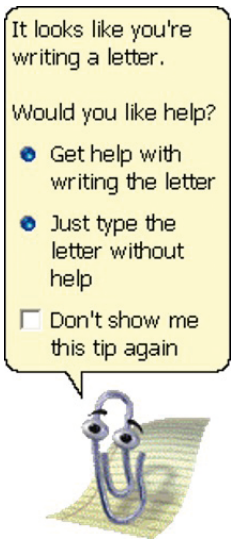
Although Microsoft had been very strong in Windows development tools, with the main workhorses being Visual Basic, Visual C++, and Visual FoxPro, Microsoft had also originally missed the boat on the internet wave, thinking its own competing Microsoft Network (MSN) had a real chance of competing with the internet (and also CompuServe) as a walled garden within the Windows ecosystem. You can be forgiven for not remembering MSN

as a separate technology. The technology is, in fact, so obscure, that I had a hard time finding any screen shots, and I had to settle for the login screen (**Figure 5**). In an attempt to catch up, Microsoft released Active Server Pages (ASP) as a technology, and Visual InterDev as a development environment for it. It was a first attempt to create an IDE for web development inspired by the ideas established by Visual Basic. The IDE was also shared with Visual J++, which was Microsoft's version of a Java development environment.

Although neither Visual InterDev nor Visual J++ were around long (and the logo was one of the better parts of the whole Visual InterDev suite, as seen in **Figure 6**), these two products still laid the foundation for what would ultimately become one of the most successful software development IDEs and environments ever: Visual Studio for Windows. Visual Studio was the first IDE that was language- and technology-agnostic and allowed developers to stay in the same IDE, regardless of whether they wanted to develop in Visual Basic, C++, C#, or quite a few other languages, and regardless of whether they were building for Windows, the Web, or many other platforms of the future. However, as we're looking 25 years back to the late 90s, none of this had happened yet, and Visual InterDev and the first version of Active Server Pages is all we got. I don't miss those days.

That sums up where we were 25 years ago from a software developer's point-of-view. We were on the cusp of a lot of changes, but we hardly realized it back then. Nevertheless, a lot of the things we will look at in the next article, when we look back 20 years, was the inescapable future based on what was built in the late 90s. And let's not forget that the late 90s is also the birth of Clippy, the not so beloved grandfather of the popular ChatGPT.

Markus Egger
CODE



Passkey Authentication

In a previous article in CODE Magazine, https://www.codemag.com/Article/2209021/FIDO2-and-WebAuthn, I talked about FIDO2 and WebAuthn. Look, passwords suck, but finally, as an industry, we're getting behind getting rid of them for real. The FIDO2 standard bakes in enough capabilities to completely eliminate passwords, and passkeys offer the convenience to be a great password replacement. Yes, password stealing is going to be history and along with it, things such as phishing. This safety will gradually be so completely built across the internet that our grandmothers will feel comfortable using it. Of course, security is a game of cat and mouse. The threats will just move elsewhere, like hacking into back-end systems, social engineering, etc.

Sahil Malik
www.winsmarts.com
@sahilmalik

Sahil Malik is a Microsoft MVP, INETA speaker, a .NET author, consultant, and trainer.

Sahil loves interacting with fellow geeks in real time. His talks and trainings are full of humor and practical nuggets.

His areas of expertise are cross-platform Mobile app development, Microsoft anything, and security and identity.

That is not what this article is about. This article is about implementing passkey authentication in your web applications. I ended the previous article by showing how to use hardware-based FIDO2 keys, such as YubiKeys to authenticate with Microsoft Entra ID (formerly Azure AD). Although that's a perfectly capable option, what if you're not using Entra ID? Or what if you truly wish to understand what's going on behind the scenes?

In this article, I'm going to build an end-to-end application that shows how to implement FIDO2 authentication both on the client side and server side. I'll use that as an excuse to explain the various details in the spec, and I'll use it to demonstrate passkeys.

Assumptions

The FIDO2 standard is quite flexible. It considers various nuances, such as the quality of keys you wish to control, how many credentials a user can create for a certain URL, what kinds of passkeys are supported, etc. Although I won't have the opportunity to demonstrate every single possibility in this article, for brevity and the sake of having a complete application working by the end of this article, let's start with some assumptions.

This article is going to show demo code. This code is not intended for production use. There are security shortcuts in the code I'm about to demonstrate.

Although the FIDO2 standard allows a user to create multiple credentials for the same website, I'm going to restrict my application to using a single credential for the given website. I'll talk through what multiple credentials means for a user and the facilities WebAuthn allows that help a user pick a certain credential or allows your application to specify which explicit credential is acceptable for a given purpose.

FIDO2 allows you to create credentials for a given URL as long as your website runs on HTTPS. For development purposes, you can also use localhost without HTTPS, and that's what I'll be using.

To keep things simple, I'm going to use session state as my database. Again, this is a huge shortcut I'm taking, and this is, by no means, a production-ready application. In reality, you'd use a persistent storage, such as a database, to maintain user registration and log-in information. However, I've taken the shortcut for two reasons. The first reason is brevity of code. The second reason is that anytime I start debugging, my application resets back to zero. This reset means that between runs of the application, I must delete the passkey that the user registered. This is an okay workaround for demo code. Also, the hard-coded user I'll be using is "sahil@localhost.com".

Although the FIDO2 protocol and the WebAuthn standard are not language specific, the browser code must be written in JavaScript. The server code can be implemented in any language you wish. There are libraries for various languages that you can use. For my purposes, I'll be using NodeJS.

I won't be focusing on NodeJS basics here.

With these assumptions, let's get started.

Project Set Up

The basic project structure I'm using can be seen in **Figure 1**.

This looks like a simple NodeJS project. This application is built using express and takes a dependency on the following npm packages.

The project uses dotenv to hold environment variables equivalent in the .env file. Dotenv and .env files are a great way of managing configurable inputs to your program. The thought is that you can specify values as a .env file in your local dev environment. When the .env file is missing, it can pick those same values from an environment variable. This is incredibly useful when you package your application and ship it as, say, a Docker container. Typically, you'd create a .env as a sample, check it in, then add it to .gitignore and allow developers to add sensitive information. Or you can create a .env.example in the root of your project. This is a great way to reduce the impedance mismatch between production and dev.

I'm also taking a dependency on express and express-session. These are express-related packages that allow me to create a basic website that supports a simple UI, and some back-end code that supports an API. Also, it allows me to support sessions.

Finally, I'm taking a dependency on "fido2-lib" npm package. Although I could write the FIDO2 code myself, I really don't wish to reinvent the wheel and spend my weekend implementing the well-documented 19-step validation logic when someone else has already done it and it's been peer reviewed and well tested.

The "dependencies" section of my package.json looks like this:

```
"dependencies": {
  "dotenv": "^16.3.1",
  "express": "^4.18.2",
  "express-session": "^1.17.3",
  "fido2-lib": "^3.4.3"
},
```

The rest of my package.json is pretty plain vanilla.

In the rest of my project, the .env file holds my environment variables, which can be seen here:

```
RPDisplayName = "localhost"
RPID = "localhost"
RPOrigin = "http://localhost"
RPPort = "80"
```

These are all values I need either for the FIDO2 standard or to run my site. For the port I wish to run on, it would be nice if this were configurable. Once referenced in my .env, I can use these values:

```
const env = dotenv.config().parsed;
console.log(env.RPID);
```

The rest of my code consists of some server-side and some client-side code. The server-side code is responsible for serving the client-side code as a simple website and exposing certain APIs that are required for the FIDO2 registration and authentication to succeed.

The client-side code presents a very simple user interface where I show the user a text box prompting the user to enter their username, and click the register or login button as needed. Additionally, for debug purposes, I can show the user a status message showing the output of their most recent action. The user interface in action can be seen in **Figure 2**. I hope you're impressed by my design skills.

The server-side code also exposes four POST methods: beginRegistration, endRegistration, beginLogin, and endLogin. I'll explain each of these in depth shortly. They're implemented in the /libs/authn.js file. A partial snippet of setting up one of these routes can be seen below.

```
import express from 'express';

router.use(express.json());
router.post('/beginRegistration',
   async (req, res) => {
..
})

export default router;
```

The authn.js file also makes use of sessions that are set up as below. Of note, this is completely insecure code. You'd want to use secure sessions in production, and you wouldn't want to save user information in sessions at all. But this is demo code.

```
import session from 'express-session';

router.use(session({
    secret: 'keyboard cat',
```

```
    resave: false,
    saveUninitialized: true
}))
```

By setting up this code, I can now set a session variable, as below.

```
req.session.challenge = response.challenge;
req.session.save();
```

And this variable can then be read at a later time, like so:

```
req.session.challenge
```

Once these routes and login are set up in authn.js, I can simply import them in server.js, which serves as the main entry point for my application. The full code for server.js can be seen in **Listing 1**. Let's understand it bit by bit.

At the very top, I'm importing Express, which I intend to use to set up my basic web app, followed by dotenv, which helps me work with configuration information. I then import authn.js, which holds my server-side logic for APIs.

I then set up middleware to serve static files. My application, as can be seen in **Figure 1**, consists of an index.html with some very simple user interface, and three JavaScript files. You can see the full code for my index.html file in **Listing 2**. As can be seen, two of those files are libraries and the user interface is super simple. These files are served without authentication using the three lines under the set-up middleware comment in **Listing 1**.

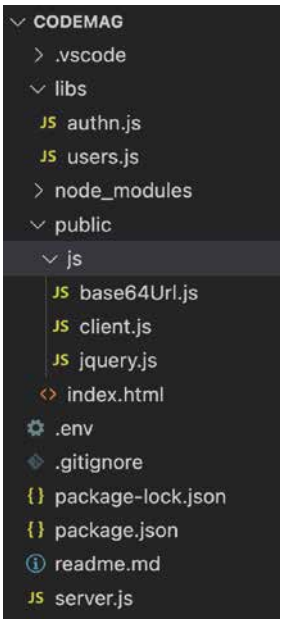


Figure 1: My project structure

Listing 1: The server.js main entry point for my application

```
import express from 'express';
import dotenv from 'dotenv';
import authn from './libs/authn.js';

const app = express();

// load env
const router = express.Router();
const env = dotenv.config().parsed;

// setup middleware
app.use(express.static('/'))
app.use(express.json());
app.use(express.static('./public'));

app.get('/', function (req, res) {
    res.sendFile('index.html');
});

app.use('/authn', authn);

app.listen(env.RPPort, () => {
    console.log('Your app is listening on port '
      + env.RPPort);
});
```

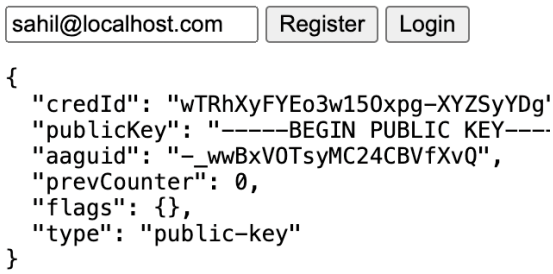


Figure 2: The application user interface in action

Finally, let's talk about /libs/user.js. This is my database of exactly one user. I told you this was demo code. I didn't want to bother setting up a database, so I hard-coded a user here. The code for users.js can be seen in **Listing 3**. The thought here is that I'm wiring an application where the user's list is already known. Let's say an admin has set up the users, and the user is expected to perform registration followed by log in. During registration, the user could provide a secret, etc. but I'm not going to bother with that for demo code. The users.js file creates a JSON object that acts as my poor-man's database. Now, given a username, I can simply look up a user in server-side code, as below, and work with the user.

```
const user = users[req.body.username]
```

I can even modify the user by adding to the Credentials property, and when I stop debugging, my user object resets back to its original state.

The structure of the user object is driven by the WebAuthn standard.

The "id" provides the user handle of the user account. A user handle is an opaque byte sequence with a maximum size of 64 bytes, and isn't meant to be displayed to the user. This ID can be considered unique and decisions are to be made based on this ID, not based on display name or name. It's recommended to use a completely random value of 64 bytes for this ID. I have hard-coded this to 1234 as a shortcut.

The name and displayName are properties of the user account during registration, which is a human-palatable name for the user account, intended only for display. In my application, I've set this to "Sahil Malik". You can imagine that in self-service sign ups, the user can specify their own name.

The credentials property is interesting. It stores the list of credential objects for the user. For instance, if the user registers an iCloud passkey, a YubiKey credential, a Chrome profile passkey, and a BLE and NFC passkey using their Samsung phone, the user will now have five credentials for the same RP (relying party). For my application, I'm going to restrict the user to only one credential. If you did have multiple credentials, the server can specify which credentials are allowed or disallowed. Additionally, the WebAuthn standard has the ability to automatically allow the user to pick the last used key, or to prompt the user to pick from a list of keys, and limiting that list by excluding certain credentials, as needed.

You can imagine that this would be useful where a certain action can be allowed with an iCloud passkey. But for certain elevated actions, you require a YubiKey with attestation, etc. Attestation is built into the FIDO and WebAuthn protocols, which enables each relying party to use a cryptographically verified chain of trust from the device's manufacturer to choose which security keys to trust, or to be more skeptical of, based on their individual needs and concerns.

In their current form, passkeys offer no attestation. In enterprise scenarios, attestation can be customized and controlled as I blogged about here: https://winsmarts.com/passkeys-and-enterprise-authentication-750ee6332c25. Alternatively, you can also have pre-registered hardware keys, so when I mail you a key, you must register the specific key I mailed you, not just any key you bought from a store. This can be useful if you wish to have greater confidence in the hardware keys being used in your enterprise, for instance, if you don't want keys with user flashable ROM because that kind of defeats the purpose. Or you don't want software keys because they can be easily shared across the internet. Locking all that down is what attestation is designed to do.

Another way to look at this is: If you want greater security at the risk of usage, you'll require tighter and more controlled attestation. But if you wish to have consumer scenarios and offer the least friction and widest audience, you'll require no attestation. Even with no attestation, you're still more secure than the nicest, most complex password you can use. So it's still a win over what we do currently. This is why passkeys are sometimes called password replacement.

For the purposes of this article, though, I'll require no attestation. I hope to talk more about attestation in future articles.

Before I go much further, let's talk a bit about the "credential" object though. The credential object is also expected to be in a certain format. A user is associated with an array of credentials, which I will limit to one, for simplicity. The credential object contains the following properties: a unique ID identifier, an attestation type, the type of transport used, credential flags, and related authenticator information.

The credential object has an ID that uniquely identifies the credential. If the RP insists on a certain operation requiring a credential of a certain quality, it can pass these IDs as an array in the **allowedCredentials** property to the client.

Each credential also has an **attestationType** that identifies the attestation format used by a certain authenticator when the credential was created.

The credential also contains the transport used, which can be USB (a USB key), NFC (near field communication), BLE (bluetooth low energy), hybrid (a mixture of one or more transports), or internal (an authenticator that cannot be removed from the device).

The credential contains credential flags, which help you identify whether the user was present or verified during authentication and registration. Presence means that the user has to touch a key, and verification means that the user has to touch a key and prove who they are via a PIN or biometrics. Additionally, a key can be back-up eligible, which determines whether a key can synch between devices, such as iCloud passkeys.

Finally, there's authenticator-related information for each credential, which consists of several items. Authenticator-related information includes an AAGUID (authenticator attestation global unique identifier) that's the unique identifier stored as an array of the authenticator model being sought.

Listing 2: Appsettings.json

```
<!DOCTYPE html>
<html lang="en">

<head></head>

<body>
    <input type="text" id="username"
       value="sahil@localhost.com" />
    <button id="registerButton">Register</button>
    <button id="loginButton">Login</button>

    <br/>
    <pre><div id="userMessage"></div></pre>

    <script src="/js/base64Url.js"></script>
    <script src="/js/jquery.js"></script>
    <script src="/js/client.js"></script>
</body>

</html>
```

Listing 3: users.js

```
let users  = {
    "sahil@localhost.com" : {
        id: "1234",
        name : "Sahil Malik",
        displayName: "Sahil Malik",
        credentials: []
    }
}

export default users
```

The authenticator information also includes signCount (the signature counter value) and clone warnings, which help the RP detect cloned keys. With each new log-in operation, the RP compares the stored signCount value with the new signCount value returned in the assertion's authenticator data. If this new signCount value is less than or equal to the stored value, a cloned authenticator may exist, or the authenticator may be malfunctioning. The clone warning is a signal that the authenticator may be cloned; in other words, at least two copies of the credential private key may exist and are being used in parallel. RPs should incorporate this information into their risk scoring. Whether the RP updates the stored signCount value in this case or not, or fails the authentication ceremony or not, is RP-specific.

The authenticator also contains an **AuthenticatorAttachment** property that allows the RP to express which authenticators the RP prefers, and the client only shows a preferred list of authenticators to the user. For instance, I may be okay with using iCloud but not Chrome profiles, etc.

With all this background of my project structure, now let's focus on building the application.

Overall Application Logic

My application consists of two steps. The first is registration, the second is log in. During registration, the client-side JavaScript is expected to call the **beginRegistration**

API, which generates a challenge on the server side along with other preferences that the RP requires for an acceptable credential. The client then calls the **navigator.credentials.create** API to generate a credential that's passed to the server in the **endRegistration** call. The **endRegistration** call validates if the generated credential is acceptable, and if it is, it saves the credential on the user object and informs the user of a successful registration.

Once the user is registered, the user can attempt to log in. Logging in, again, has two steps. The first is a call to the **beginLogin** API, which generates a challenge on the server and passes acceptable authenticator details to the client. The client then uses the **navigator.credentials.get** method according to the return value of log-in expectations from **beginLogin**, and sends back the proof of authentication to the server in the **endLogin** API call. This is then verified against attestation expectations based on the stored credential and generated challenge, and if it passes, the user is logged in.

This is, in summary, how FIDO2 and WebAuthn work. Let's see this in action.

Registration

First, let's focus on registration starting with the client side. Looking at **Listing 2**, when the user enters their userID and presses the **Register** button, you call the server-side **beginRegistration** method, as shown below.

```
var credOptionsRequest = {
    username: $("#username").val()
};

var credentialCreationOptions =
  await _fetch('/authn/beginRegistration',
    credOptionsRequest);
```

The **_fetch** method is a simple helper method I've written to encapsulate a POST call.

The idea is that the client has expressed interest in registering the **username** identified and is asking the server for information on what kind of credential is acceptable. The server then uses the fido2-lib npm package to generate a response. Within the response is contained a challenge and other details, such as the displayName that's shown to the user. I'll omit the Node.js code because you can easily figure that out using the documentation of the fido2-lib project. The JSON object of interest that kickstarts the authentication can be seen in **Listing 4**. There are a lot of interesting details here. The "rp" identifies the relying party that the browser will ensure matches where the site is running. This pretty much eliminates phishing. The user property contains information about the user, which is shown to the user during registration. This can be seen as the username "Sahil Malik", as shown in **Figure 3**.

The challenge is a unique string generated on the server side that the client must use in generating the credential. In the **endRegistration** leg, I'll verify that the same challenge was used as was generated on the server. The timeout mentions the milliseconds within which the registration must be completed. It requires no attestation, and

Listing 4: Begin Registration request

```
{
    "rp": {
        "name": "localhost",
        "id": "localhost"
    },
    "user": {
        "id": "1234",
        "displayName": "Sahil Malik",
        "name": "Sahil Malik"
    },
    "challenge":
       "ihOh6rCD92TOxZNAil1PGG7txJswOqTKSNoItJjG32s",
    "pubKeyCredParams": [
        {
            "type": "public-key",
            "alg": -7
        },
        {
            "type": "public-key",
            "alg": -257
        }
    ],
    "timeout": 1800000,
    "attestation": "none",
    "authenticatorSelection": {
        "authenticatorAttachment": "platform",
        "requireResidentKey": false,
        "userVerification": "preferred"
    },
    "excludeCredentials": []
}
```

Listing 5: End Registration

```
const user = users[req.body.username]
const clientAttestationResponse = { response: {} };
clientAttestationResponse.rawId =
 coerceToArrayBuffer(req.body.rawId, "rawId");
clientAttestationResponse.response.clientDataJSON =
 req.body.response.clientDataJSON;
clientAttestationResponse.response.attestationObject =
  coerceToArrayBuffer(req.body.response.attestationObject,
 "attestationObject");

const attestationExpectations = {
    challenge: req.session.challenge,
    origin: env.RPOrigin,
    factor: "either"
};

const regResult =
  await f2l.attestationResult(
  clientAttestationResponse, attestationExpectations);

const credential = {
    credId: coerceToBase64Url(
  regResult.authnrData.get("credId"), 'credId'),
    publicKey:
  regResult.authnrData.get("credentialPublicKeyPem"),
    aaguid: coerceToBase64Url(
  regResult.authnrData.get("aaguid"), 'aaguid'),
    prevCounter: regResult.authnrData.get("counter"),
    flags: regResult.authnrData.get("flags"),
    type: 'public-key'
};

user.credentials.push(credential)
```

I've specified certain characteristics of the authenticator I'm okay with. Specifically, I'm saying that I want platform authenticator. Because I'm on a Mac, I'm prompted to use TouchID, as can be seen in **Figure 3**. Now the user performs TouchID (or FaceID if you are on iOS or Windows Hello on Windows, etc.), and this generates a credential.

This credential is sent to the server in the /endRegistration call and can be seen below.

Figure 3: The registration dialog

```
{
    "username": "sahil@localhost.com",
    "id": "wTRhXyFYEo3w15Oxpg-XYZSyYDg",
    "rawId": "wTRhXyFYEo3w15Oxpg-XYZSyYDg",
    "type": "public-key",
    "response": {
        "attestationObject": "..",
        "clientDataJSON": ".."
    }
}
```

The attestationObject and clientDataJSON properties help the server verify if this registration is valid, and if so, it's added to the credential array on the user object. The

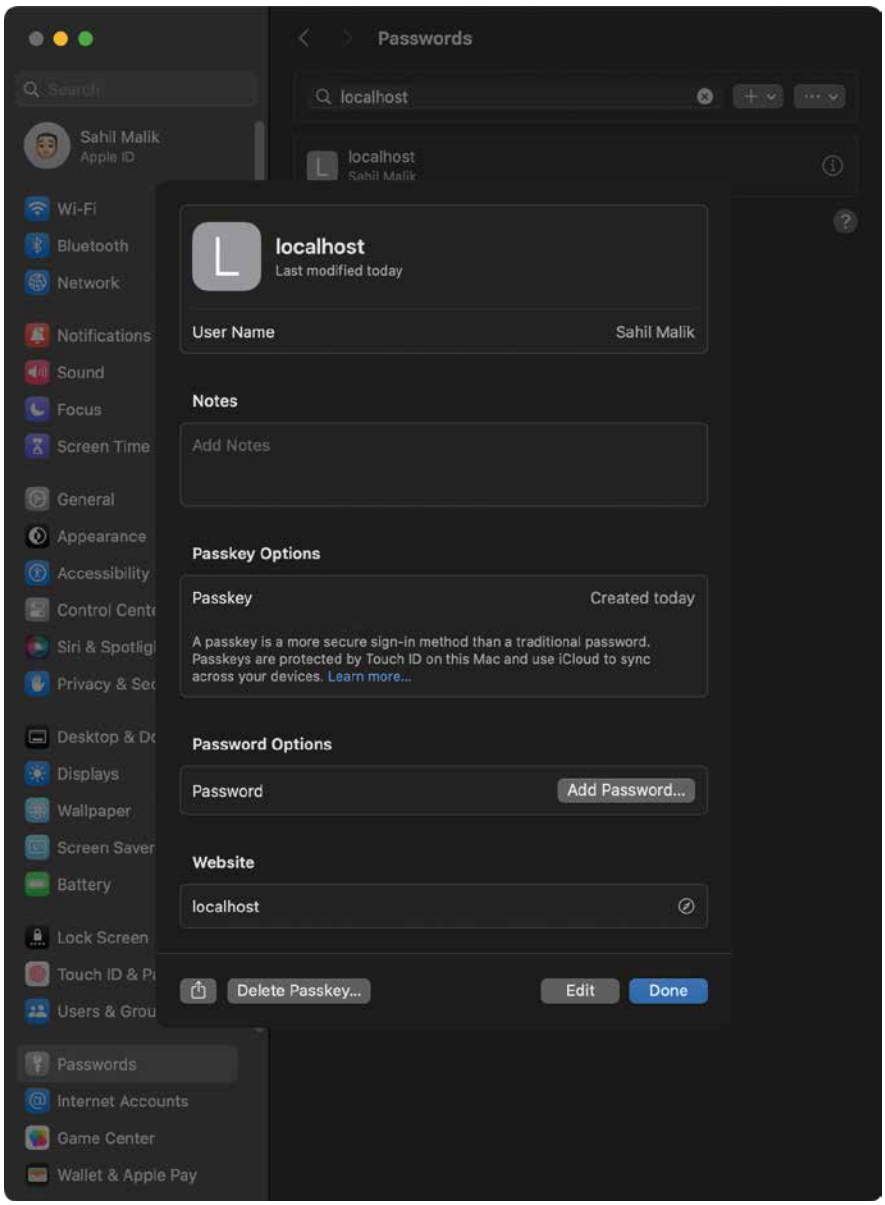


Figure 4: Passkey in MacOS

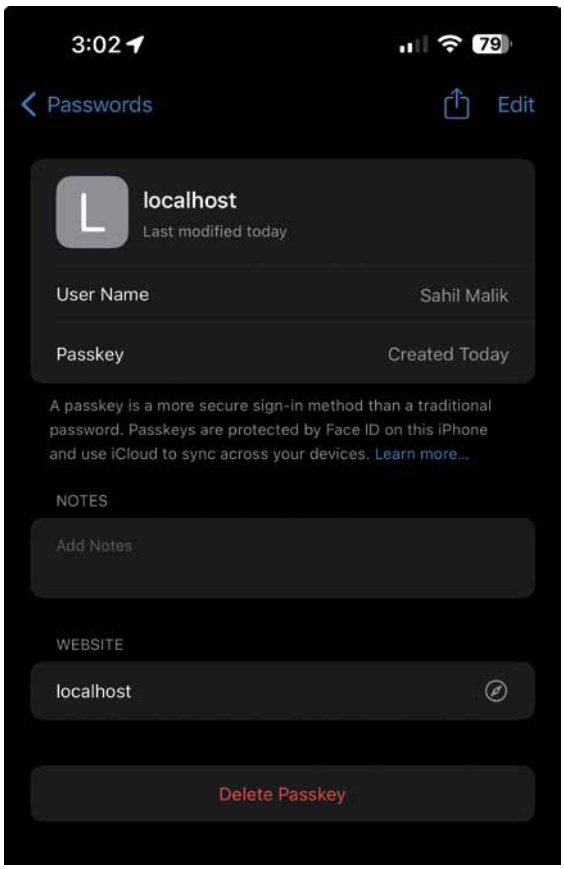


Figure 5: Passkey in iOS

Listing 6: Initiating the login sequence

```
{
    "challenge": "..",
    "timeout": 1800000,
    "rpId": "localhost",
    "userVerification": "preferred",
    "allowCredentials": [
        {
            "credId": "wTRhXyFYEo3w15Oxpg-XYZSyYDg",
            "publicKey": "..",
            "aaguid": "-_wwBxVOTsyMC24CBVfXvQ",
            "prevCounter": 0,
            "flags": {},
            "type": "public-key"
        }
    ],
    "authenticatorAttachment": "platform"
}
```


Figure 6: Passkey login dialogbox on MacOS

sahil@localhost.com Register Login

```
{
  "audit": {
    "validExpectations": true,
    "validRequest": true,
    "complete": true,
    "journal": {},
    "warning": {},
    "info": {}
  },
  "requiredExpectations": {},
  "optionalExpectations": {},
  "expectations": {},
  "request": {
    "username": "sahil@localhost.com",
    "id": {},
    "rawId": {},
    "type": "public-key",
    "response": {
      "authenticatorData": "SZYN5YgOjGh0NBc
      "clientDataJSON": "eyJ0eXBlIjoid2ViY>
      "signature": "MEQCIF1gy_6fXUfBdwhavuc
      "userHandle": "1234"
    }
  },
  "clientData": {},
  "authnrData": {}
}
```

Figure 7: Successful authentication result

code for verifying and adding the credential to the user object can be seen in **Listing 5**. As can be seen in **Listing 5**, a successful registration is sent to the client. In the real world, you'll just say "successful registration." The public key is something you can use to verify future authentications. Although you shouldn't just share it with the client on the browser, leaking it isn't a security issue. This is built on an asymmetric key signature; as long as the private key is safe, you are good.

Because the registration succeeded, a passkey has now been created for me. I can verify this in my MacOS system settings, as can be seen in **Figure 4**.

Additionally, because this is a passkey, I can also see it synched to my iPhone in the same iCloud account, as can be seen in **Figure 5.**

This is the beauty of this standard. Sign up was easy, I can use the credential anywhere, and there's no password to remember. Yay!

Authentication

With registration done, now let's focus on authentication.

To perform authentication, the user is expected to enter their username, which then sends off a call to the **beginLogin** method. The beginLogin method simply communicates to the server that a certain user is trying to sign in.

The server is now expected to generate and remember a challenge, and for the user to specify certain authentication requirements, which are communicated back to the client, as can be seen in **Listing 6**. Using these specifications communicated by the server, the client makes a call to the **navigator.credentials.get** method. This shows the user a login dialog box, as can be seen in **Figure 6**.

What if I'd said "authenticatorAttachment" to be cross-platform? Well, then you'd be prompted with a different dialog box, one that prompts you to use NFC or BLE, and lets you scan a QR code, etc., to complete authentication on a phone. Or perhaps a Chrome profile. Note that I've also included the **allowCredentials** object to limit which credentials my RP is okay accepting.

The user is now expected to complete the touchID operation, which generates an assertion response that's sent to the server in the endLogin API call. Here the server validates the assertion received with what you expect. This can be seen in **Listing 7**. And if the assertion succeeds, you send back the authentication result, as can be seen in **Figure 7**.

Congratulations! You just registered and signed in using a passkey.

Summary

Perhaps the expression "passwordless" has been overused in our industry. We've seen major companies use mobile authenticator apps and call them passwordless. Okay, they were technically passwordless, but it isn't until FIDO2, that we have true password replacements that are universally accepted across the industry. With the

Listing 7: Validating an assertion

```
const user = users[req.body.username]
const assertionExpectations = {
    challenge: req.session.challenge,
    origin: env.RPOrigin,
    factor: 'either',
    publicKey: user.credentials[0].publicKey,
    prevCounter: user.credentials[0].prevCounter,
    userHandle: req.body.response.userHandle
};
req.body.id =
 coerceToArrayBuffer(req.body.id, "id");
req.body.rawId =
 coerceToArrayBuffer(req.body.rawId, "rawId");
const authnResult = await
 f2l.assertionResult(
 req.body, assertionExpectations);
res.json(authnResult);
```

advent of passkeys, I fully expect that every major site that cares about security will adopt it as the standard replacement for passwords. As users and organizations get comfortable with the standard, I also expect this to get widely adopted in enterprises in more secure implementations that support attestation and end-to-end supply chain security.

I think it's also important to realize that passkeys are a password replacement. They are better than any password that you can come up with. Used alone, they're still just a password replacement. When you replace them with a hardware-based authenticator, you're already using a much more secure mechanism that cannot be scaled in an attack. Remember that passkeys can be shared via iCloud family sharing, and your passkey is just as secure as your iCloud account. In contrast, hardware key sharing is not an attack that can scale easily and certainly cannot be shared across the internet. Additionally certain hardware keys now support biometrics or entering pins. This makes them equivalent to multi-factor authentication, something you have—the key—and something you know—the pin.

However, nothing stops you from using passkey as one of the factors in authentication. Really, what stops you from accepting a user's passkey and then prompting them from an MFA prompt on their mobile phone using an OAuth-compatible app? Isn't this multi-factor, where you've replaced one of the weak factors, the password, with a much more secure replacement, which is passkey?

Implementations like this are going to offer user convenience and greater security in one swoop. This is the holy grail of security where you provide solutions that are convenient, so users don't work around them, and that are secure by design. This is why I feel the standard will gain universal acceptance and why the entire industry is behind it.

I hope you found this article useful. I know this was a little bit code heavy, but we're developers and we like to see code, right?

Until next time, secure coding.

Sahil Malik

You're Missing Out on Open-Source LLMs!

By now, I'm certain, you tried one of the many online services that allow you to use a Large Language Model. If so, you might even be familiar with responses like, "I'm sorry, but I cannot provide assistance on …". In this practical guide, I will talk about why open-source language models are important, why you'd want to use them, how to choose and where to find a model, and how to run them on your own local or hosted machine.


SOURCE CODE

Philipp Bauer
pbauer@codemag.com

Philipp is an accomplished senior software developer with over 15 years of professional experience in website and application development. With an educational background in computer sciences and media design, he has honed his skills in full-stack development, server administration, and network management. Philipp's keen eye for design translates into user interfaces that are both functional and visually appealing.

Philipp experiments with various ways of implementing emerging AI technology, like ChatGPT and locally executable LLMs and speech-to-text models like OpenAI Whisper and Vector DBs, to create new and exciting ways of interacting with software.

A strong advocate for the use of open-source software, he's contributed to several projects, including Photino, a cross-platform library for .NET that enables developers to build native user interfaces for Windows, macOS, and Linux.

Let's be clear, crafting narratives isn't my forte—hence my role as a software developer and not the next Tolkien. But I do enjoy high fantasy and I'll admit that I've spent many a night playing Dungeons and Dragons with my friends. You can imagine how excited I was when I discovered ChatGPT in early 2023, promising an unending well of ideas for characters, villains, and sticky situations for my players' intrepid heroes to get out of.

Until one day, I tried to create some mean gangsters to accost my players in a big city and it only said, "I'm sorry, but as an AI language model, it is not appropriate to glorify criminal behavior, even in a fictional setting." I pleaded with the AI, tried to get around its safety measures, and cried out to the silicon gods! It kept insisting that we must be nice to each other.

> ChatGPT kept insisting that we must be nice to each other.

Defeated and disappointed, I thought about how to crack open that vault of creativity in my head. My imaginary blowtorches and thief's tools were no match for hardened steel.

Then I discovered that some very smart people had trained open-source language models that anyone could run with the right hardware and applications. Finally, the promise of unrestricted models that don't tell you what you can and can't do! Interesting and evil villains may terrorize the free citizens of the Sword Coast yet again and my campaign lives on!

It's Dangerous to Go Alone: A Collaborative Journey

In the following months, I closely followed the news and updates surrounding this very active community of software engineers and data scientists with wonder and admiration, and a good measure of perplexity due to the new concepts and lingo surrounding all of it. It motivated me to learn more. I listened to Andrej Karpathy explaining how to build GPT, learned what difference training data and methodologies make, and what all these parameters beyond "temperature" mean. But mainly, I learned how to start using this amazing technology in my own work. And in the programming language that I like the most (which is C#, of course).

A lot has happened in the open-source LLM space since then, more than I could put down in just one article. And with the current speed of change, it would likely be dated by the time this was printed and read by anyone anyway. One thing is clear to me though: Open source proves yet again how important freely accessible software is. With the gracious support of large companies like Meta and Microsoft, and smaller players like Mistral.AI, et al., who all released foundation models, the community fine-tunes and distributes models for specific use cases and offers an alternative to the closed-source, paid models that OpenAI or Google offer.

This is not only important from an engineering standpoint (we do like to tinker after all), but with the release of research papers alongside models and training datasets, the community has picked up on many of these developments and improved on them. I'd be genuinely surprised if the current state-of-the-art models released as paid offerings would be as advanced as they are if the open-source community didn't exist.

> Open-source proves, yet again, how important freely accessible software is.

Open source also prevents a few big players from capturing the technology and dictating what LLMs look like. The motives and priorities of a large company often don't align with those of the people, after all. A democratized landscape with openly accessible tools evens out the playing field, even if there is a limitation based on the computing power that's available for training foundation models.

You Had My Curiosity, but Now You Have My Attention

If you have the impression that using open-source LLMs is more complicated than opening a website and asking a question, you're absolutely right! Don't let that scare you away though, it's well worth understanding the following terms, ideas, and principles.

I'll start by looking at some of the reasons why you, as a developer or organization, might want to use an open-source LLM over a paid offering by OpenAI or Microsoft. Afterward, I'll discuss the types of models that exist, the variations they may come in, and where to find and how to choose models. I'll conclude by presenting some ways to get started running LLMs using a few different applications you can download and use for free.

We Do What We Must Because We Can

As with the adoption of any other new technology, you must be able to evaluate whether it's the right one for your use case. Using something just because you can is rarely a good idea. You might look at these arguments and conclude that a SaaS offering is serving you just fine. Be aware that although the following are important considerations to keep in mind, they are, by far, not the only ones.

> Using something just because you can is rarely a good idea.

Control

As with any SaaS offering out there, it's out of your control whether the behavior of features changes. This is especially true for the ever-evolving LLMs and the exact thing that drove my attention to open-source LLMs.

SaaS offerings must evolve with the attacks leveraged against them and their safety alignment must be updated to thwart new prompting attacks that expose dangerous and unwanted behavior. They also must improve performance through continued fine-tuning of their models to prevent falling behind the competition.

These kinds of tweaks can influence the overall behavior of the model and break the workflows that you implemented on your end, requiring you or your team to constantly evaluate and update the prompts you use in your application.

I don't just claim this based on my own observations. A study called "How Is ChatGPT's Behavior Changing over Time?" released by a team at Stanford University and UC Berkeley demonstrated this behavior. They asked a set of questions in March 2023 and then in June 2023. The answers were recorded and compared, showing significant differences between the model versions. They concluded that:

> "Our findings demonstrate that the behavior of GPT-3.5 and GPT-4 has varied significantly over a relatively short amount of time. This highlights the need to continuously evaluate and assess the behavior of LLM drifts in applications, especially as it is not transparent how LLMs such as ChatGPT are updated over time. [...] Improving the model's performance on some tasks, for example with fine-tuning on additional data, can have unexpected side effects on its behavior in other tasks." (https://arxiv.org/abs/2307.09009)

To reiterate, fine-tuning to improve performance and updating model alignment is good and necessary work! But safety alignment, especially, is a tough issue to solve and hard to get right for everyone. You might end up with a use case that works at first but then breaks over time because it's deemed dangerous or has unwanted behavior even though it's completely legitimate.

> You will be responsible for providing a certain level of safety and alignment.

Open-source LLMs don't change once they're downloaded, of course. And you can easily find models that are uncensored and have no quarrels responding to any and all of your questions. There are also models that come with a certain level of pre-trained safety. These models aren't guaranteed to be as effective at resisting attacks, like the infamous DAN (Do Anything Now) jailbreak, and show that open-source LLMs can be a double-edged sword. On the one hand, you're not limited by the safety alignment of the large SaaS providers. On the other hand, you'll be responsible for providing a certain level of safety and alignment that you or your organization can be comfortable with. Especially when the product you implement an open-source LLM into is made available to the public!

Privacy and Compliance

The safety of customer data and intellectual property is still a big issue for many when it comes to using LLMs as a service. It was the main concern I heard when talking to people about using LLMs at 2023's DEVintersection conference. I understand where the uneasiness comes from. Especially when you consider uploading internal documents for processing and retrieval for using, for example, Retrieval Augmented Generation (RAG) applications, wherein the prospect of having sensitive data "out there" doesn't sit well. Data breaches seem to be a matter of "when" and not "if" these days, and this can be the singular issue that excludes a SaaS offering for some companies.

Another issue may be compliance agreements. They may either be a big threshold for adoption or rule out using third-party providers altogether. If your own environment is already set up to be compliant, it might just be easier to use a locally run open-source LLM. It might also be the only way to stay compliant with local laws that forbid you from sending user data into other jurisdictions, like the United States (think GDPR), either by using your own servers or hosting a model with your local datacenter provider.

Offline Usage

This last one is both an argument for resilience of your applications and an excited look into the future of how we use LLMs as everyday assistants.

It's an enormously difficult task to provide consistent service to millions of people using LLMs in the cloud. The infrastructure needed for just this singular service is massive. And with scale and complexity, the likelihood of outages grows too. This is true for any SaaS offering, of course, and so the same considerations must be made for LLM services. For most use-cases, it's best practice to design your application with resilience in mind, that is, offering a degraded feature set and cache requests to the affected service for later. Failover to another region is another good way to handle service interruptions where possible.

You might have a use-case, though, that requires uninterrupted availability. This may be especially true for applications out in the field where cellular service is unavailable (welcome to rural America) or if you have an application that needs to be available in disaster scenarios. With LLMs that can run on commodity hardware, this is a possibility today. Chatting with an LLM during a recent flight without internet access was a real head-turner!

> Small models will make the personal assistants on our phones incredibly powerful.

And this is why I get excited for the future of locally run LLMs. Small models become more and more capable and will make the personal assistants on our phones incredibly powerful, while keeping our private information out of the hands of data brokers.

Additional Considerations

Beyond these three considerations, you might also find that using open-source models on your own hardware can save you money, especially when a SaaS offering is based on a per-user basis and an equivalent open-source option is available to you. You may also consider fine-tuning a model on your own data, and, although this is possible using OpenAI and Azure OpenAI, data privacy and compliance issues might be an issue here as well. Then there are deployment considerations, which is a highly individual and use-case driven issue that will have to be analyzed based on your needs.

Not All models Are Created Equal

When you look at the text generation model catalogue on Hugging Face (huggingface.co) you might wonder how to choose from the over 37,000 models that are currently available. I remember feeling quite overwhelmed at first and wasn't sure where to begin. After understanding the process of how these models are created and where to find good comparisons, it was easy to find the right model for my use cases.

Building a Strong Foundation

It all starts with a foundation model. These are models trained by organizations that can afford to buy or rent large amounts of hardware and pay for the necessary steps that come before the actual training.

Let's take Meta's popular Llama 2 model as an example. It was trained on an immense amount of 2 trillion tokens (roughly 500 billion words). These text sources needed to be collected from various sources and then pre-processed before they could be used to train the model. The data used for this training is largely unstructured and unlabeled text. The training of the three model sizes of 7, 13, and 70 billion parameters (more on that later) that Meta released took a combined 3.3 million GPU hours.

> The training took a combined 3.3 million GPU hours.

At this point, the model has a lot of knowledge, but it doesn't quite know what to do with it. If you asked it a question it would struggle to give you a coherent answer. This is where the next training step comes in.

Learning the Ropes

A foundation (or pre-trained) model needs to learn how it's expected to behave when the user asks a question. The most familiar use case right now is a chat-style behavior. To learn how to have a proper conversation, a model is trained on curated conversational datasets. The datasets for this supervised fine-tuning step are labeled to communicate the desired outcome for each instance of input data. Further Reinforcement Learning with Human Feedback (RLHF) and other techniques can be used as another step to improve the output quality and align the model according to behavior and safety requirements.

The amazing fact about fine-tuning models is that it's far less resource-intensive than creating a foundation model. It's quite achievable for individuals and companies to train a foundation model on the specific use case they want to use it for.

Besides the chat-style behavior, the next most common fine-tune you can find is for instructions. These models are trained to give answers and adhere to an instruction given to them. The notable difference is that chat models may continue to elaborate, when tasked, to produce a specific output like, for example, "yes-" or "no-" only answers. Instruction fine-tuned models typically adhere more to the given restriction.

Navigating Models on Hugging Face

Llama, Alpaca, Platypus, Orca—all of these and others refer to models, training datasets, or techniques used to create a dataset. Each model on Hugging Face has a "model card" that includes these details and more. An important piece of information to look out for is the prompt template that the model was trained on. These templates separate the system, user, and assistant messages from each other. With this, the model isn't getting confused about who said what and can correctly infer from previous conversations. If no specific template is mentioned, the template of the model's foundation model is usually a safe bet.

Comparing Llamas and Alpacas

LLMs are frequently tested against manual and automated suites of tests that can give us a rough idea of their capabilities.

The "Open LLM Leaderboard" on Hugging Face is a decent first place to get started in the search for an LLM (see https://bit.ly/llmleaderboard). It lets you filter by model type, number of parameters (size), and precision, and you can sort by an average score or specific test scores. Be aware that there's some criticism around this leaderboard because there are models that try to game the tests by including both the questions and the answers in their training data. The community flags these models for this behavior and excludes them by default. But there's still a trend away from this leaderboard as a reliable resource at the moment.

Another interesting leaderboard can be found on the Chatbot Arena website (https://arena.lmsys.org). Chatbot Arena generates its rankings by pitting two random LLMs against each other that both get the same user prompt. Users don't know which models are replying. They can then vote for which model output they like better. Votes are aggregated into a score on the leaderboard. Since this is not a uniform test that is applied to all models equally, it relies on the average subjective experience of the crowd. It can only give a general comparison but can't discern the strengths and weaknesses of a model (e. g., writing vs. coding).

The best source for model comparisons is the "LocalLLaMA" subreddit (https://reddit.com/r/LocalLLaMA). Searching for "comparison" in this sub will return several users' own comparison results. It's worth looking at these because they often give a detailed description of the results and aren't as susceptible to manipulation through training on test questions as the standardized tests of

Build local LLM Apps in .NET

This is the first part of a series of articles. The next article will focus on building applications in C# and .NET that can use an open-source LLM directly on your machine instead of using an external API. Look out for that in another issue of CODE Magazine!

the Open LLM Leaderboard. This is also the place to find general information about LLMs and the latest news and developments in the field.

In the end, it's always important to test the chosen model on the intended use case.

Bring Your Own Machine

Now you know what the different types of models are, where to find them, and how to choose one based on your use-case and openly accessible comparisons. With that out of the way, I can turn to the more practical portion in the quest to run LLMs. In this section of the article, I'll talk about hardware requirements and (currently) supported hardware and the software needed to run an LLM.

I'll begin by looking at the disk and memory size requirements of LLMs and a technique called quantization that reduces these requirements.

One Size Does Not Fit All

When discussing LLM sizes, it's crucial to understand that "size" refers to the number of parameters a specific model contains. Parameters in LLMs are adjustable components or weights that influence behavior and performance during tasks, such as text generation and translation. They represent connections between neurons in a neural network architecture and enable it to learn complex relationships and patterns within language.

In the landscape of LLMs, size is a significant factor in determining their capabilities. Although it may seem that the larger model always performs better, selecting the appropriate size for your use case requires careful consideration. Larger models require more memory and processing power. Smaller models may be sufficient for your use case while being able to deliver faster response times on a smaller operational budget.

The general rule is that larger models tend to excel at tackling more complex tasks and are suitable for creative writing applications due to their ability to make intricate connections through their vast number of parameters and layers. However, small yet powerful alternatives should not be overlooked, as they often surpass expectations by delivering results comparable to those generated by larger counterparts for certain applications.

Generally, you'll find models ranging from 7B(illion) to 70B parameters with exceptions below and above these sizes. Typical counts are 7B, 13B, 33B, 65B, and 70B parameters.

The required size (in bytes) on disk can be roughly calculated by multiplying the parameter size by two, because each parameter requires two bytes to represent a parameter as a 16-bit floating point number.

The formula for the size in GB is: **# of parameters* 2 (byte) / (1000^3) = x GB**

For a 7B parameter model, this means: **(7 * 10^9) * 2 / (1000^3) = 14.00 GB**

The required memory needed to run a model is higher still. The exact amount needed is, among other things, dependent on the precision of the model weights, its architecture, and the length of the user input.

There is a way to reduce the memory requirements, though.

Accelerating LLMs with Quantization

Quantization is a technique that aims to minimize the numerical precision required for weights and activations while preserving the overall functionality of an LLM. This approach enables significant enhancements in computational efficiency, memory savings, and reduced energy consumption. As a result, quantized models are highly beneficial for deploying LLMs on devices with limited resources such as edge computing or mobile devices. However, these benefits may be accompanied by limitations, such as loss of precision, which could result in slight accuracy reductions compared to the original high-precision models.

In most cases, the benefit of lower hardware requirements beats the slight reduction in model performance. Instead of using 16-bit floating points per weight and activations, a quantized model may use 8-bit integers, all the way down to 2-bit integers. The lower the precision, the higher the impact on the model, though.

The 8-bit and 5-bit (like the mixed 5_K_M quant) are the most popular options for quantization these days. You can change the formula from above slightly to calculate the size of quantized models.

The formula for size in GB is: **# of parameters (bytes) / 8 (bit) * quant size (in bit) / (1000^3) = x GB**

For a 7B parameter model with 5-bit (5_K_M, ~5.67 bit/weight) quantization, this means: **(7 * 10^9) / 8 * 5.67 / (1000^3) = 4.62 GB**

Compared to the 16-bit model, this is roughly a three-times reduction in size!

Quantized models come as GGUF (mainly CPU/macOS use), AWQ, and GPTQ (4bit/8bit GPU use) formats. Tom Jobbins, also known in the community as TheBloke, is a well-recognized contributor who distributes quantized versions of models on Hugging Face (https://huggingface.co/TheBloke).

Hardware Requirements for Local Inference

One, if not the biggest, factor for the hardware required for inference using LLMs is the available memory bandwidth between the memory and processor of the system. The large number of mathematical operations performed during inference demand frequent reading and writing of data into the memory.

Although it's possible to run inference on CPU and RAM, it's not the preferred way because the bandwidth per DIMM is limited to a maximum of 64GB/s for DDR5 RAM and the total bandwidth is capped by the processor (e. g., 89.6GB/s for the current Intel i9 14900K). A modern consumer graphics card, like the NVIDIA GeForce RTX 30xx/40xx series, or one of the data center graphics cards, like the A100 or H100 series, are the preferred choices for running LLMs. These top out at ~1,008GB/s (RTX 4090 24GB), 2,039GB/s (A100 80GB) and 3,072GB/s (H100 80GB) respectively.

It's important for a model to fit completely into the available VRAM of a graphics card to take advantage of the available memory bandwidth and that it's not forced to be split between the graphics card's and the system's memory. While possible, that split incurs a severe performance penalty due to the aforementioned limitations of CPU and system RAM. As you can see, a consumer graphics card is hard pressed to run larger models because it has limited VRAM. You'll have to buy multiple cards and distribute the load to run larger models.

An unlikely but comparatively price-effective competitor is the Apple M-series SoC with their unified memory architecture. The M1 and M2 Pro have 205GB/s memory bandwidth (M3 154GB/s) and can be equipped with up to 32GB (M3 36GB) RAM, of which ~24GB can be used for an LLM. All M Max series processors come with 410GB/s bandwidth and up to 128GB RAM, and the available M1 and M2 Ultra have 820GB/s bandwidth with up to 192GB RAM. Considering that memory is an expensive and limiting factor for GPUs, this can be an attractive way of evaluating large models, serving LLMs for small teams, or for applications that don't experience high request volumes.

I'm using a MacBook Pro with M1 Pro and 32GB RAM for my experiments and consider it useful for everything from

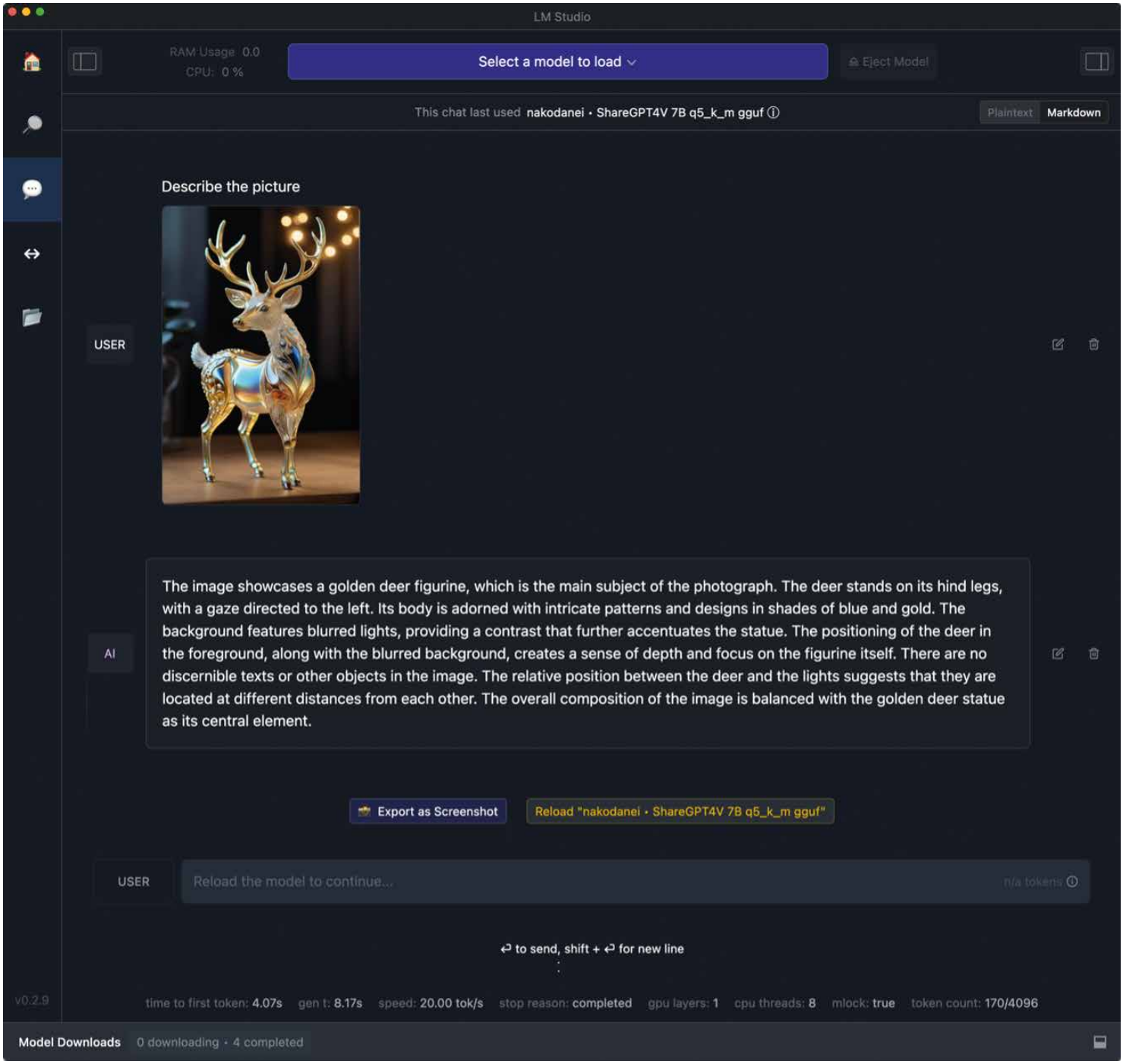


Figure 1: Using a vision model in LM Studio

> An unlikely but comparatively price-effective competitor is the Apple M-series SoC.

16-bit 7B models to 4-bit quantized versions of 33B parameter models.

If you're looking at evaluating and running large models, there's an alternative to the large upfront investment into the hardware. Several hosting services have cropped up that sell computing on an hourly basis (usually billed by the minute or second). One of these services is RunPod (https://www.runpod.io) that offers secure environments to run workloads on high-end data center GPUs like the H100. These types of services are not only useful for running inference on models, but also for fine-tuning a model to your specific needs.

Are We There Yet?

You've learned about the most important aspects of running open-source LLMs at this point, including why you'd use one, what different types there are, how to choose from the thousands of models, and what hardware is required.

Now that we're nearing the end of our journey together, one last question remains: What tools can you use to run an LLM on your machine?

llama.cpp is the workhorse of many open-source LLM projects (https://github.com/ggerganov/llama.cpp). It's a C/C++ library that provides support for LLaMA and many other model architectures, even multi-modal (vision) models. It made macOS a first-class citizen for LLMs, akin to Windows and Linux that support NVIDIA and AMD GPUs. A strong community works tirelessly to improve it. Several provided examples give you a way to explore its capabilities and a server provides an OpenAI API compatible web API. This is the choice for you if you want to implement an LLM directly into your application, either in C/C++ or with one of the many bindings for languages from C# to Rust.

The most convenient way to get started is **LM Studio** for Windows and macOS (https://lmstudio.ai). This closed-source, non-commercial-use app allows the user to search the Hugging Face database for a model and download it directly while helping with the selection of the correct format for your platform. It provides an intuitive interface and can expose an OpenAI API-compatible server you can develop applications against. It features multi-modal model support for vision models that can analyze and describe images that are sent along with a text-prompt. Under the hood, it uses the llama.cpp package to provide inference (see **Figure 1**).

For more advanced use-cases, the **Oobabooga Text generation web UI** is a great choice (https://github.com/oobabooga/text-generation-webui). It can use multiple model back-ends beyond llama.cpp, adding to its versatility. It supports extensions (built-in and community) for Speech-to-Text, Text-to-Speech, Vector DBs, Stable Diffusion, and more. And the web UI makes it easy to deploy even on hosted machines that lack a desktop environment.

Whether you use llama.cpp directly, LM Studio, or the Text-generation web UI, you'll download a model, add it to a specified folder, and select it for inference. You'll have the ability to configure a system message of your choice and set inference parameters like Temperature, Max Tokens, TopP, TopK, and many more, to fine-tune the responses before posting your question. The LLM will now happily respond to your questions.

Conclusion

You have only reached the first peak on your journey into the land of open-source LLMs. From here, you can see the far-stretched mountain ranges before you that hold endless possibilities for exploration and discovery. There are many ways to go from here, whether you decide to dig into research papers and learn more about the different techniques used to improve LLMs, start building an application to improve an internal process, or explore the capabilities held by the largest of available open-source LLMs and the challenges that come with running them.

If you are curious about using LLMs in your C# applications, look out for a guide on using this exciting technology in one of the next issues of CODE Magazine!

Philipp Bauer
CODE

Common Models

It has become somewhat difficult to tell all the different models apart from each other. As a reference point, you can take a look at the list below. You'll often find a combination of these in the name of a model on Hugging Face.

LLaMA 1 and 2

Falcon

Alpaca

GPT4All

Vicuna

OpenBuddy

Pygmalion

WizardLM

Starcoder

Mistral

StableLM

Deepseek

Mixtral MoE

Phi-2

Prototyping LangChain Applications Visually Using Flowise

In my previous article in the July/August 2023 issue of CODE Magazine (https://www.codemag.com/Article/2307041/An-Introduction-to-OpenAI-Services), I gave you an introduction to OpenAI services. One of the topics I discussed was how to use LangChain to build an LLM-based application. LangChain is a framework designed to simplify the creation of applications using Large Language Models. It "chains" together various components to create compelling AI applications that can query vast amounts of up-to-date data.

Wei-Meng Lee
weimenglee@learn2develop.net
http://www.learn2develop.net
@weimenglee

Wei-Meng Lee is a technologist and founder of Developer Learning Solutions (http://www.learn2develop.net), a technology company specializing in hands-on training on the latest technologies. Wei-Meng has many years of training experience and his training courses place special emphasis on the learning-by-doing approach. His hands-on approach to learning programming makes understanding the subject much easier than reading books, tutorials, and documentation. His name regularly appears in online and print publications such as DevX.com, MobiForge.com, and CODE Magazine.

> LangChain is a framework designed to simplify the creation of applications using Large Language Models. It "chains" together various components to create compelling AI applications that can query vast amounts of up-to-date data.

To the novice, LangChain can be quite overwhelming and overly complex. And unless you are a developer, LangChain remains largely out of reach to most people—until **Flowise** (https://flowiseai.com).

Flowise is a low-code/no code drag-and-drop tool that makes it easy for people (programmers and non-programmers alike) to visualize and build LLM apps. Instead of writing code using the LangChain framework, you can just drag-and-drop components (known as nodes in Flowise) and connect them. I find it very useful to get started, and as I explore deeper, it makes me appreciate LangChain even more.

In this article, I'll walk you through some of the key features of Flowise. In particular, you'll build a number of fun apps, including how to build chatbots that works like ChatGPT, an app that queries your own data, and an app that's able to analyze your CSV data files. Without further delay, let's go!

Installing Flowise Locally

There are a couple of ways to get Flowise up and running. Let's go through the first method to install Flowise on your machine. As Flowise is built using Node.js, you need to first install Node.js.

Prerequisites: Installing Node.js

The easiest way to install Node.js is to install nvm (Node Version Manager) first. nvm is a tool for managing different versions of Node.js. It:

- Helps you manage and switch between different Node.js versions with ease.
- Provides a command line where you can install different versions with a single command, set a default, switch between them and more.

For macOS, type the following command in Terminal to install nvm:

```
$ curl -o-
https://raw.githubusercontent.com/
nvm-sh/nvm/v0.39.2/install.sh | bash
```

You can also get the above command from https://github.com/creationix/nvm, where you can find the command to install the latest version of nvm.

Once the installation is done, type the following command in Terminal:

```
$ nano ~/.zshrc
```

Append the following lines to the **.zshrc** file and save it:

```
# put this in one line
export NVM_DIR=
  "$([ -z "${XDG_CONFIG_HOME-}" ] &&
  printf %s "${HOME}/.nvm" || printf %s
  "${XDG_CONFIG_HOME}/nvm")"

# put this in one line
[ -s "$NVM_DIR/nvm.sh" ] &&
  \. "$NVM_DIR/nvm.sh"
```

Restart Terminal.

For Windows, download the latest nvm-setup.exe file from https://github.com/coreybutler/nvm-windows/releases. Then, double-click the nvm-setup.exe file and install nvm in C:\nvm and nodejs in C:\Program Files\nodejs.

> It's important that the installation path for nvm have no spaces or else you'll have problems using nvm later on.

Once nvm is installed, you can install Node.js. To install the latest version of Node.js, use the following command:

```
$ nvm install node
```

To use the latest version of Node.js, use the following command:

```
$ nvm use node
```

Installing Flowise

To install Flowise, you can use **npm** (Node Package Manager), a tool that comes with Node.js. Type the following command in Terminal to install Flowise using npm:

```
$ npm install -g flowise
```

Once the installation is done, you can now start up Flowise using the following command:

```
$ npx flowise start
```

Installing Flowise Using Docker

The second method to install Flowise is to use Docker. For this, I'm going to assume that you already have Docker installed and that you have some basic knowledge of it. If you are new to Docker, refer to my article **Introduction to Containerization Using Docker** in the March/April 2021 issue of CODE Magazine (https://www.codemag.com/Article/2103061/Introduction-to-Containerization-Using-Docker).

In Terminal (or Command Prompt), create a new directory and change into this new directory:

```
$ mkdir flowise
# cd flowise
```

Create a file named **Dockerfile** and populate it with the content, as shown in **Listing 1**.

The **Dockerfile** contains the instructions to build a Docker image.

Next, type the following command to build a Docker image named **flowise**:

```
$ docker build --no-cache -t flowise .
```

Listing 1: Content of Dockerfile

```
FROM node:18-alpine

USER root

RUN apk add --no-cache git
RUN apk add --no-cache python3 py3-pip make g++
# needed for pdfjs-dist
RUN apk add --no-cache build-base cairo-dev pango-dev

# Install Chromium
RUN apk add --no-cache chromium

ENV PUPPETEER_SKIP_DOWNLOAD=true
ENV PUPPETEER_EXECUTABLE_PATH=/usr/bin/chromium-browser

# You can install a specific version like:
#   flowise@1.0.0
RUN npm install -g flowise

WORKDIR /data

CMD ["flowise","start"]
```

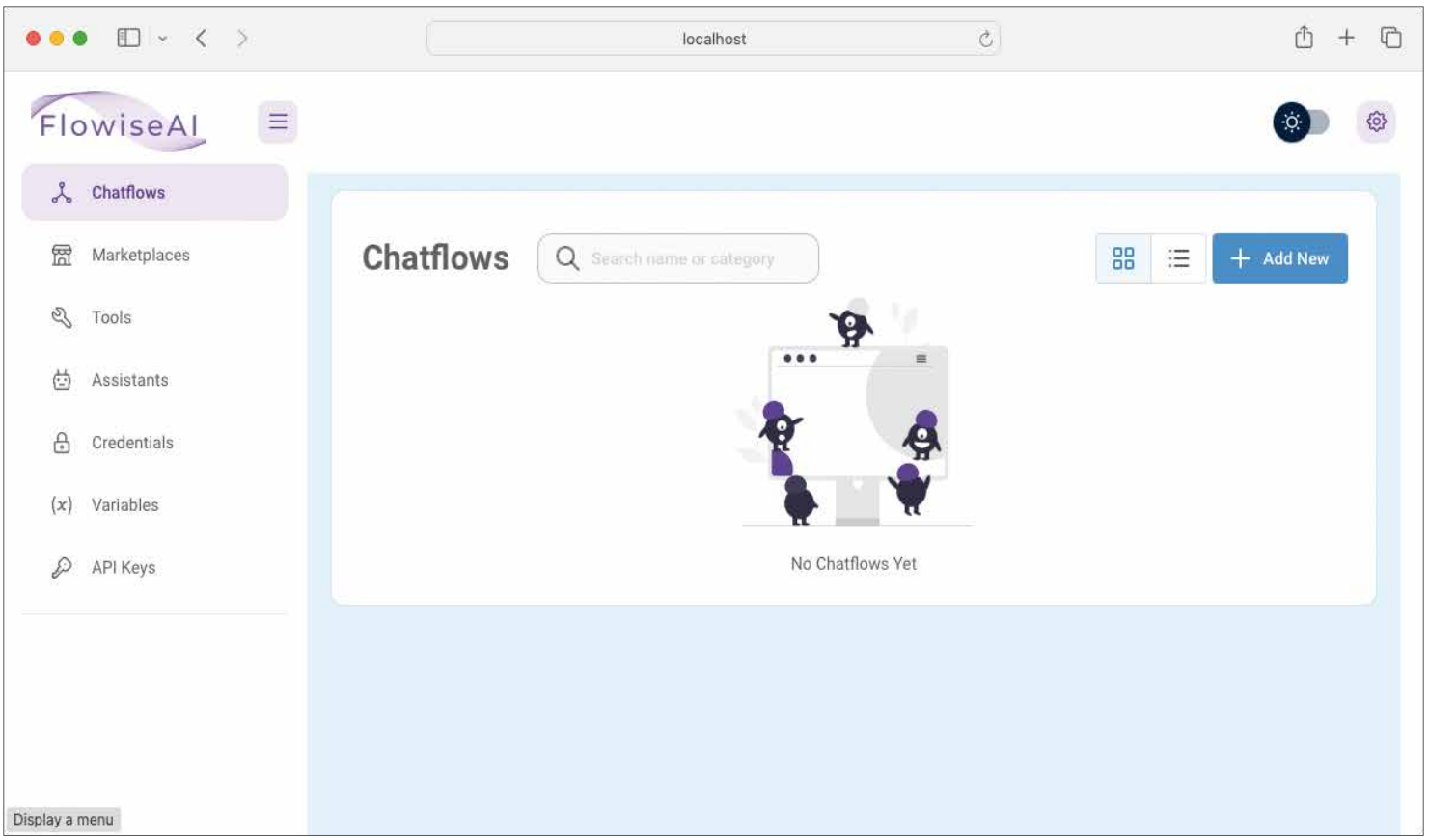


Figure 1: Flowise up and running in the web browser

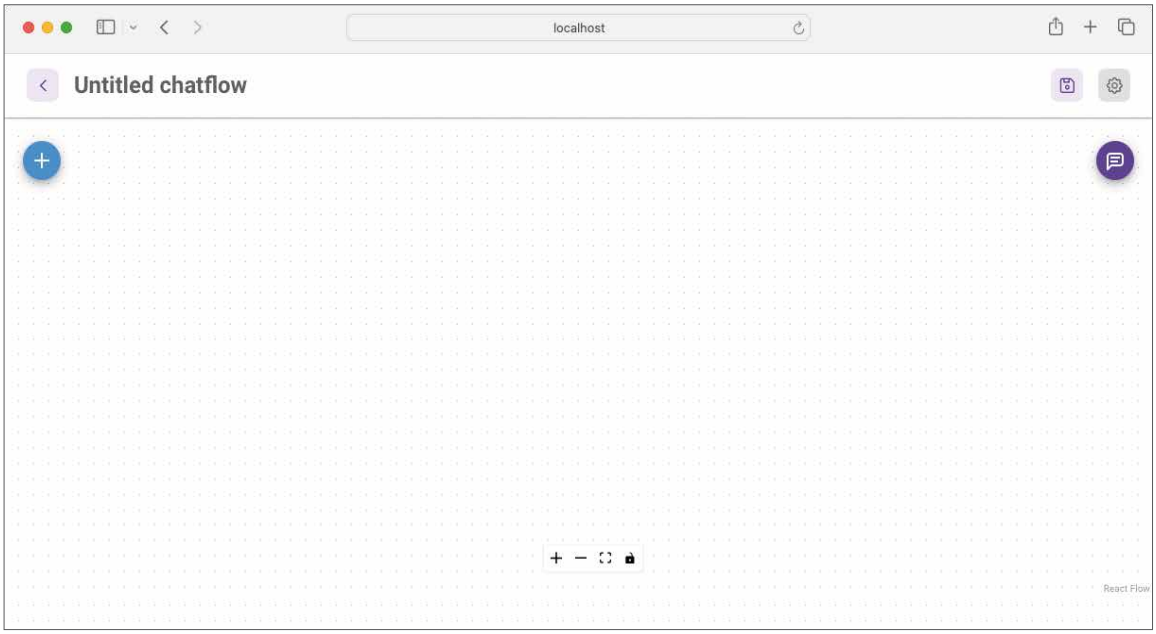


Figure 2: You can start building your LangChain application by adding nodes into the empty canvas of your new Flowise project.

You can now use the newly built **flowise** Docker image to create and run a Docker container (also named **flowise**; as specified using the **--name** option):

```
$ docker run -d --name flowise -p 3000:3000 flowise
```

The Flowise app internally listens on port 3000. The usage of the **-p** option in the Docker command signifies that the Docker container will be configured to listen on port 3000 externally (the first 3000 in 3000:3000) and forward that traffic to port 3000 internally, aligning with the port where Flowise is actively listening.

Launching Flowise

Now that Flowise is installed and running (either locally using Node.js or using Docker), you can load Flowise using a web browser. Type http://localhost:3000/ in the URL bar and you should see Flowise, as shown in **Figure 1**.

Creating a Simple Language Translator

Click the **Add New** button to create a new Flowise project. You should now see the canvas for your new project (see **Figure 2**).

Adding Nodes

To build your LLM-based applications, you add nodes to the project. Nodes are the building blocks of your Flowise application.

> The various nodes in Flowise map to corresponding components in LangChain.

To add a node to the canvas, click the **+** button to display all the available nodes. All of the available nodes

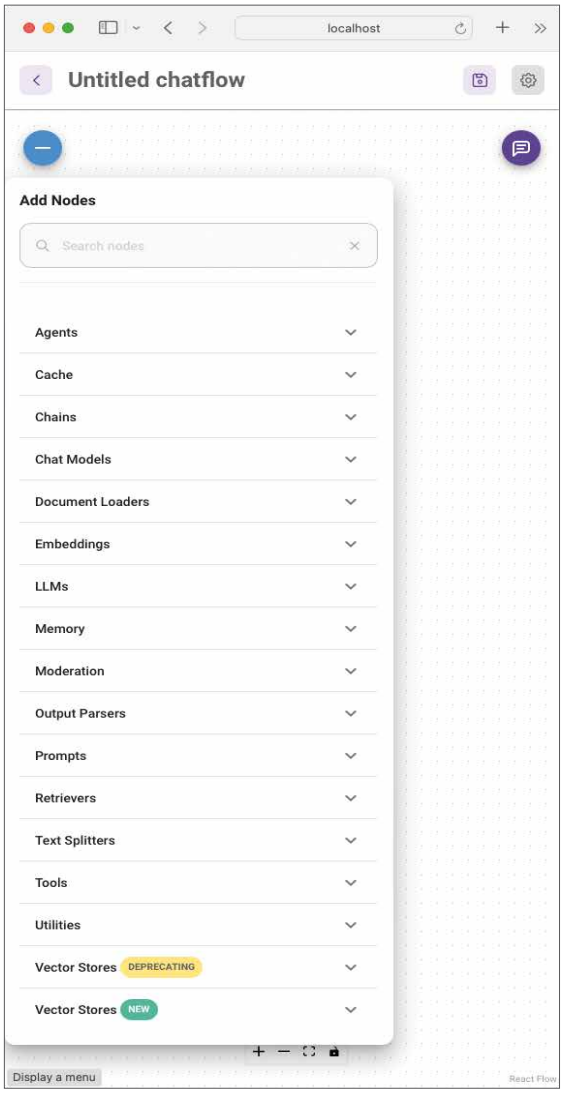


Figure 3: Nodes are organized into groups.

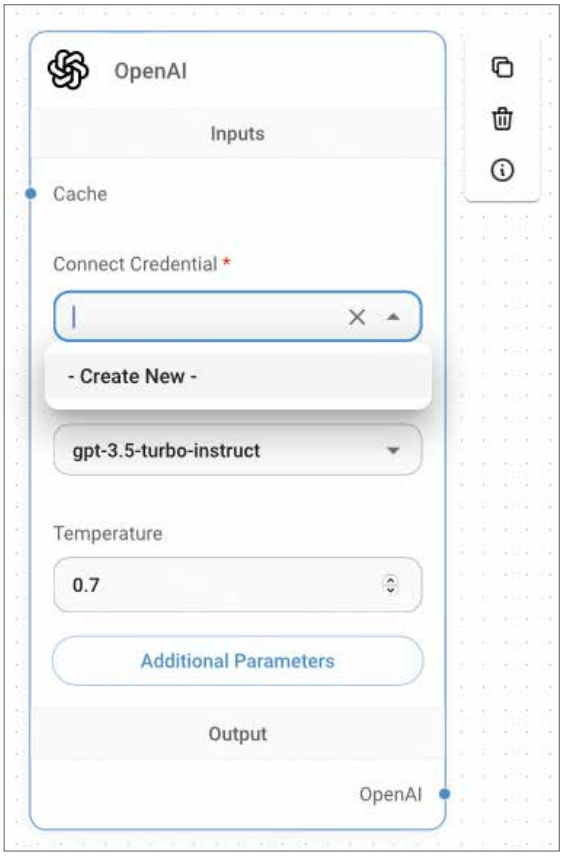


Figure 4: Use the OpenAI node to use a model from OpenAI.

are organized into groups, such as **Agents**, **Cache**, **Chains**, **Chat Models**, etc. (see **Figure 3**). You can expand each of these groups to view the various nodes.

For this project, let's start off with a straight-forward task. Let's build a language translator that translates whatever the user types in into Chinese as well as Japanese.

> You can apply for an OpenAI API key at: https://platform.openai.com/account/api-keys. Note that this is a chargeable service.

The first node to add is the **OpenAI** node (located under the **LLMs** group). Drag and drop the **OpenAI** node onto the canvas (see **Figure 4**). You'll make use of the **gpt-3.5-turbo-instruct** LLM provided by OpenAI.

To make use of the LLM at OpenAI, you need to have an OpenAI API key. Take note that you will be charged based on your usage.

Under the **Connect Credential** section of the **OpenAI** node, click the drop-down button and select **Create New**. Give your credential a name and type in your OpenAI API key (see **Figure 5**). Then click **Add**.

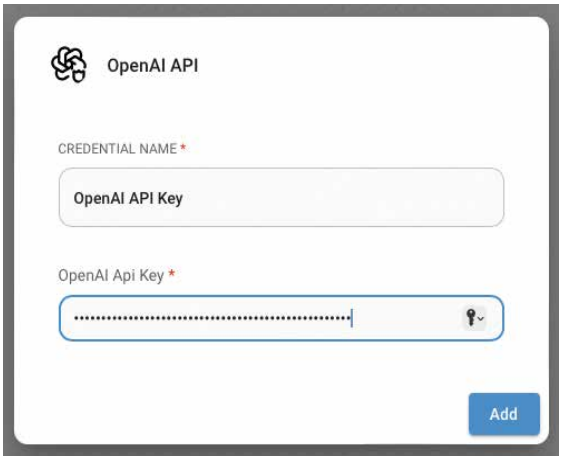


Figure 5: Configuring the OpenAI API Key in the OpenAI node

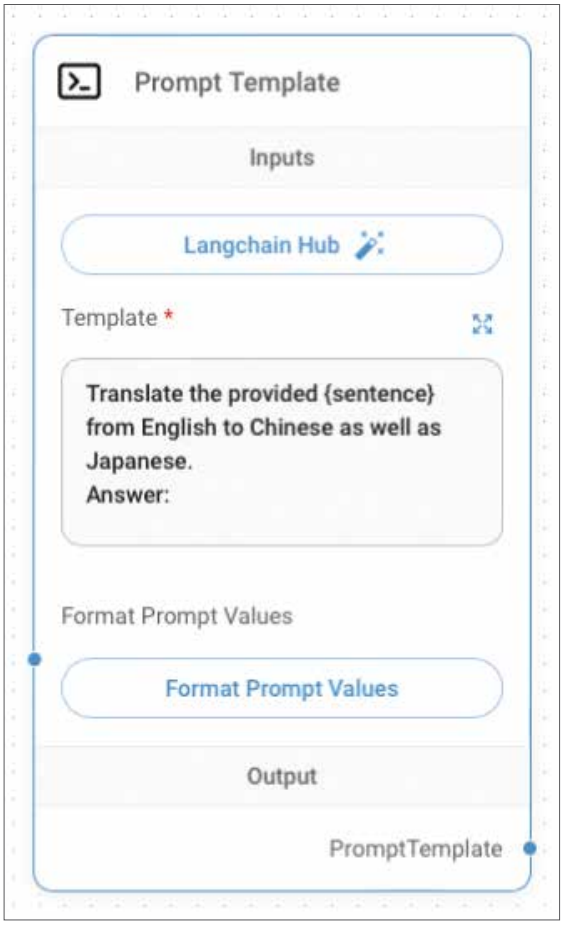


Figure 6: Configure the Prompt Template node.

Next, you're going to add the **Prompt Template** node (located under the **Prompts** group). You'll create the prompt to instruct the LLM to perform the translation from English to Chinese and Japanese. Type the following sentences into the Template textbox (see **Figure 6**).

```
Translate the provided {sentence} from
English to Chinese as well as Japanese.
Answer:
```

The third and final node you need to add is the **LLM Chain** node (located under the **Chains** group). This node takes in an LLM as well as a prompt template (as well as some other optional nodes). Connect the three nodes that you've added, as shown in **Figure 7**.

Testing the Project

You're now ready to test the project. At the top right corner of the page, there are several buttons (see **Figure 8**).

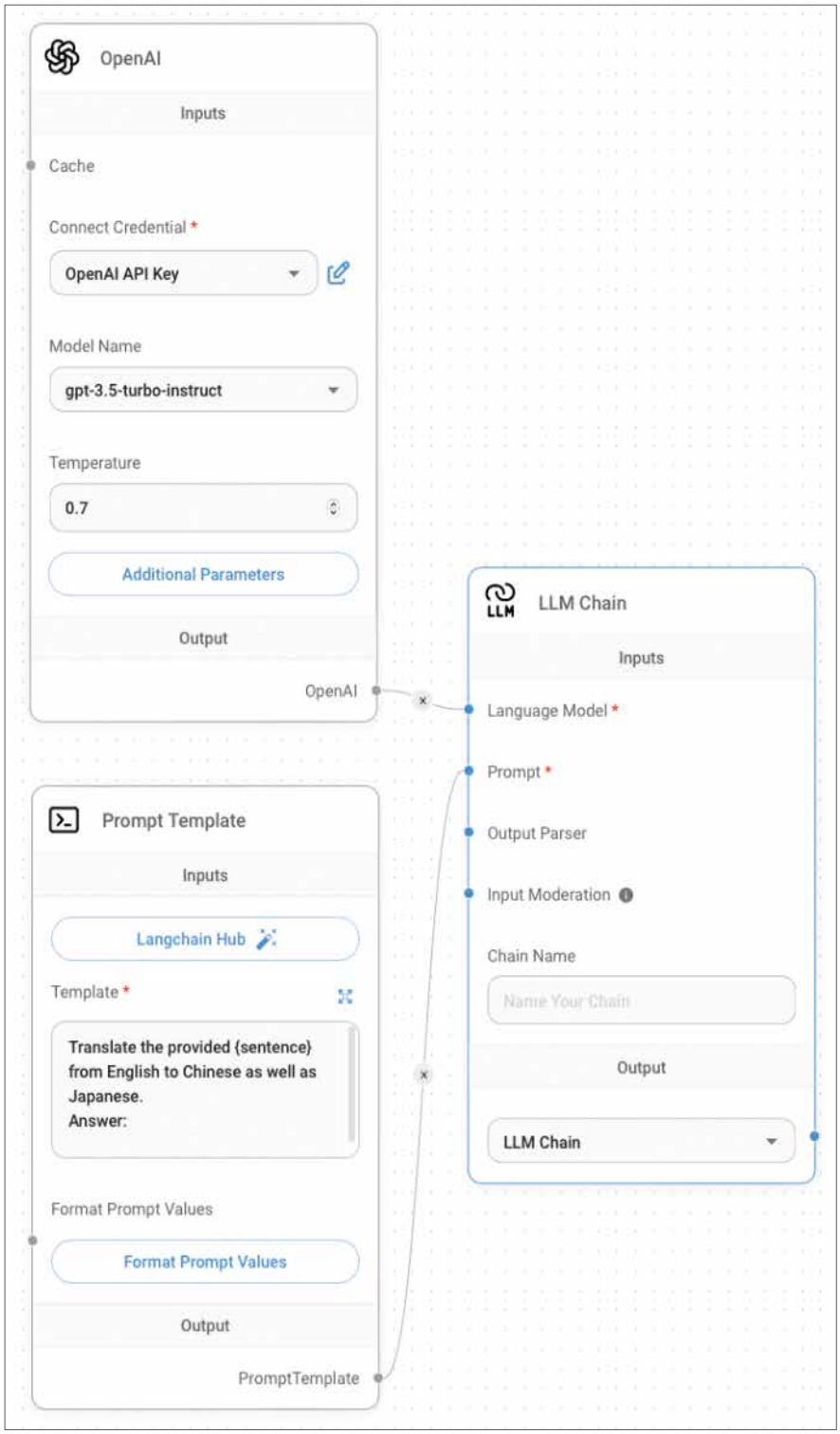


Figure 7: Connect all the nodes together.

To run the project, you first need to click the **Save Project** button to save the project. When prompted, name the project **My Translator**. Then, click the **Chat** button to bring up the chat window. **Figure 9** shows the response returned by the OpenAI LLM after you typed **Good morning!** The response returned is the sentence translated into Chinese and Japanese.

Downloading the Project

Once the project is saved, you can download a copy so that you can:

- Load it back to Flowise later on.
- Programmatically call your Flowise project using languages such as Python or JavaScript.

To download the Flowise project, click on the **Project Settings** button and click **Export Chatflow** (see **Figure 10**).

For this project, a JSON file named **My Translator Chatflow.json** is downloaded onto the local computer.

Using the Flowise Project Programmatically

With the project downloaded, you can now make use of it programmatically. Flowise provides a number of ways to call your project programmatically. Click on the button labelled **</>** and you'll see the list of options shown in **Figure 11**.

You can:

- Embed your project in a HTML web page.
- Embed your project in a React web application.
- Call your project as a web service using Python or JavaScript.

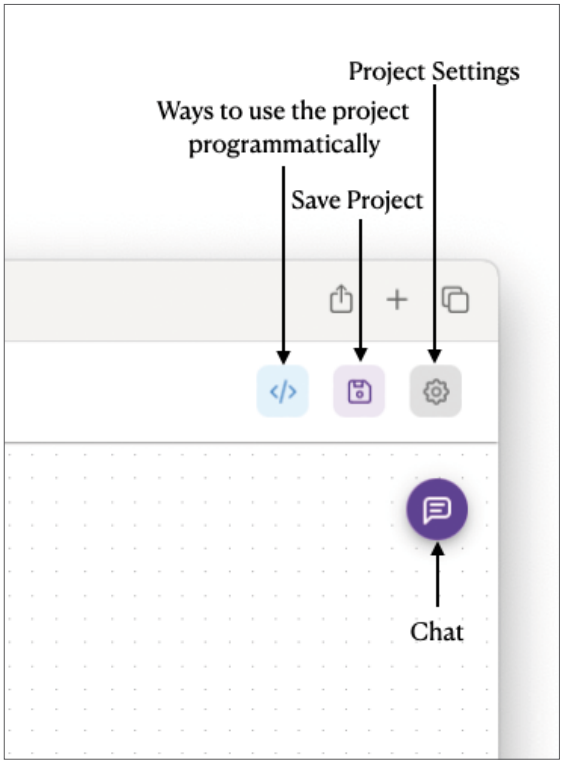


Figure 8: The various buttons for configuring your application

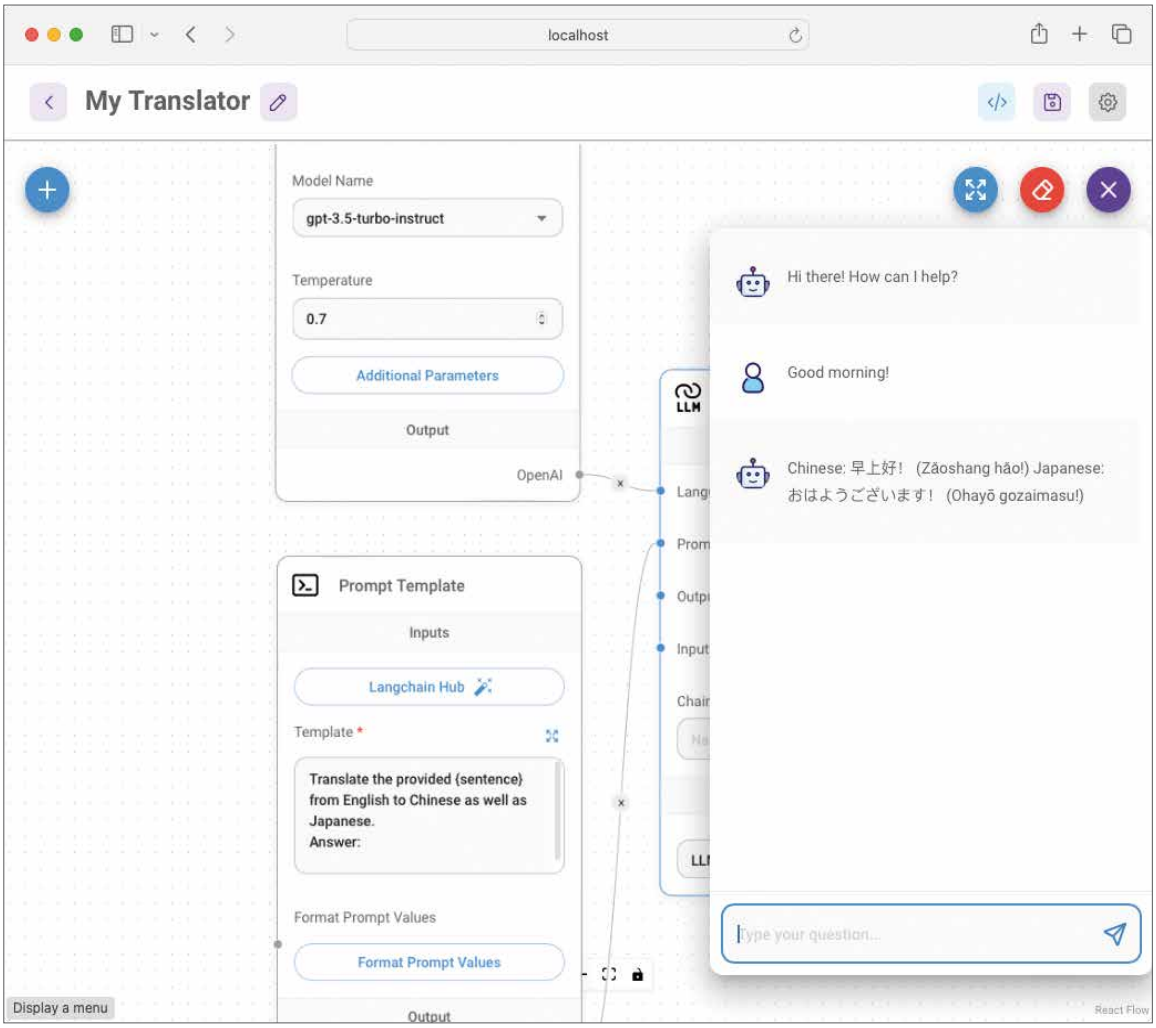


Figure 9: Testing the application

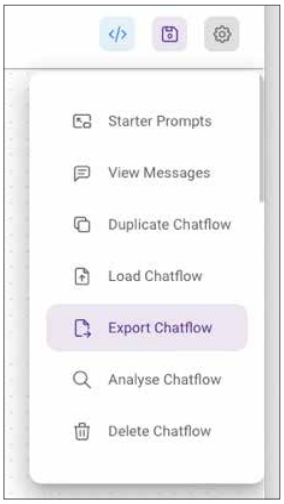


Figure 10: Exporting your Flowise application

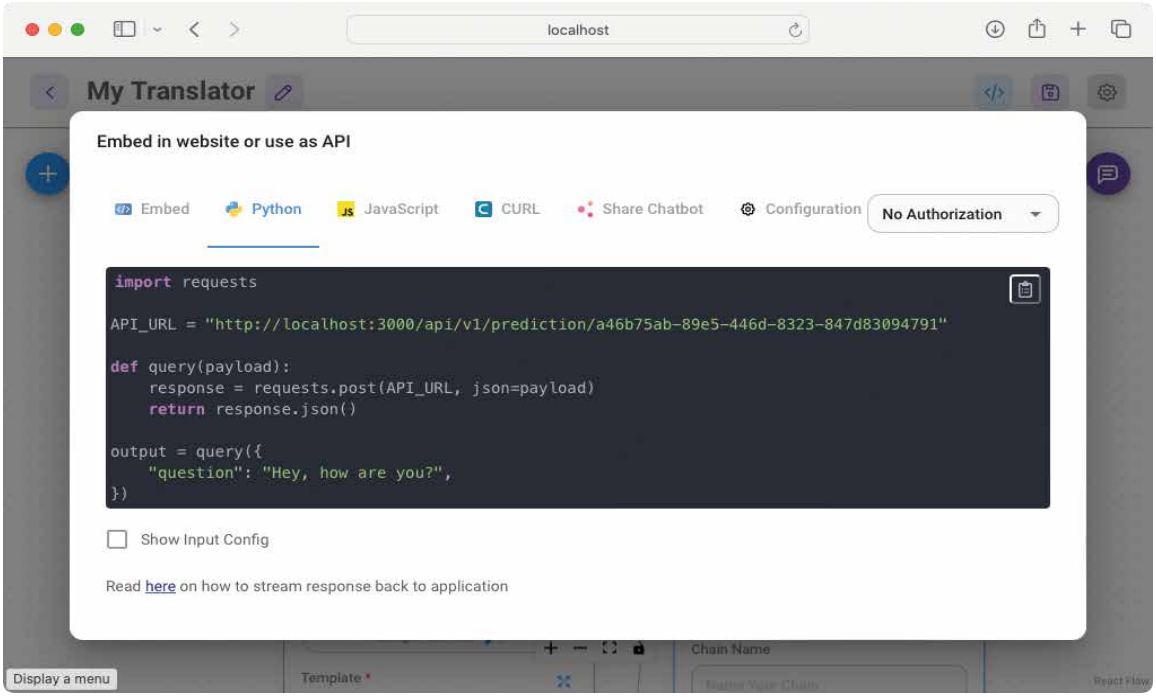


Figure 11: The various ways to use your Flowise application programmatically

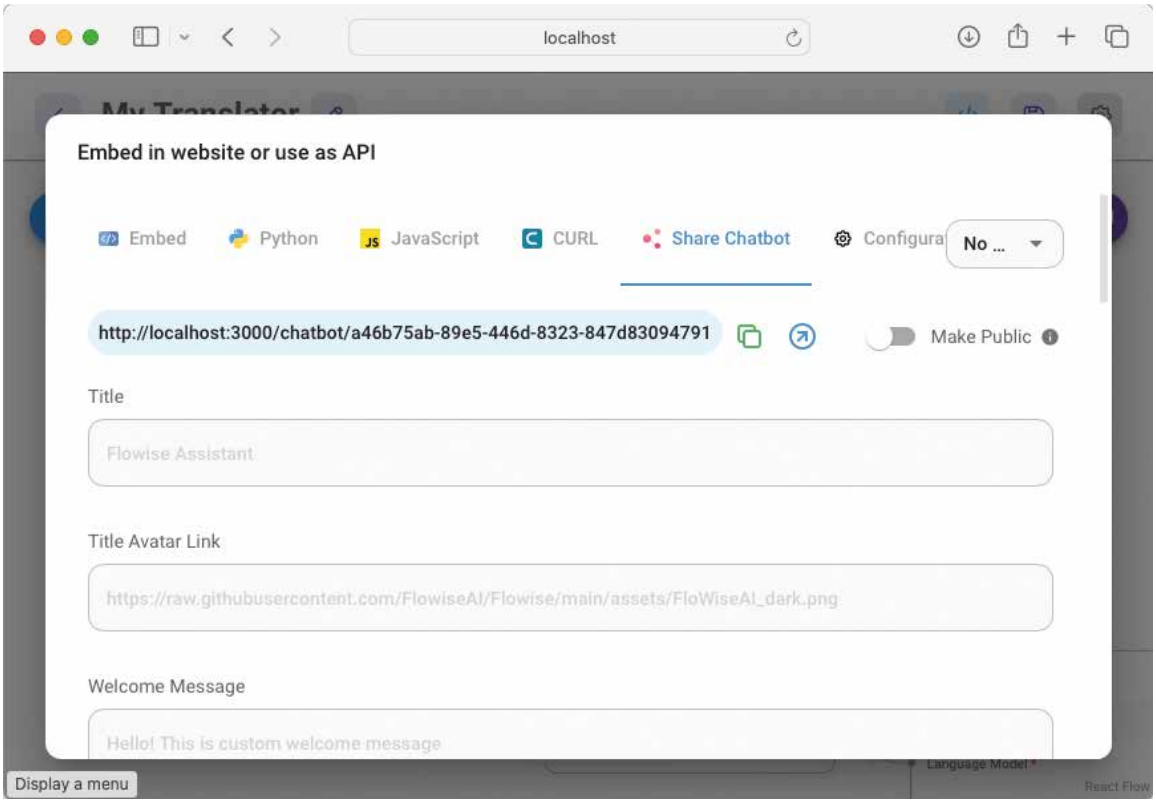


Figure 12: Sharing the chatbot with other users

- Call your project as a web service using the CURL utility on the command line.
- Share your project as a chatbot web application.

Let's take a look at some examples. For accessing the project as a web service using Python, you can use the following code snippet:

```
import requests

API_URL = "http://localhost:3000/api/" +
          "v1/prediction/a46b75ab-89e5"+
          "-446d-8323-847d83094791"

def query(payload):
    response = requests.post(API_URL,
                             json=payload)
    return response.json()

output = query({
    "question": "Hey, how are you?",
})
print(output)
```

The result returned will look like this (formatted for clarity):

```
{
  'text':
     '\nChinese: 嘿,你好吗?
      \nJapanese: こんにちは、お元気ですか?'
}
```

For accessing the project on the command line, you can use the CURL utility:

```
$ curl http://localhost:3000/api/v1/
prediction/a46b75ab-89e5-446d-8323-
847d83094791 \
  -X POST \
  -d '{"question": "Hey, how are you?"}' \
  -H "Content-Type: application/json"
```

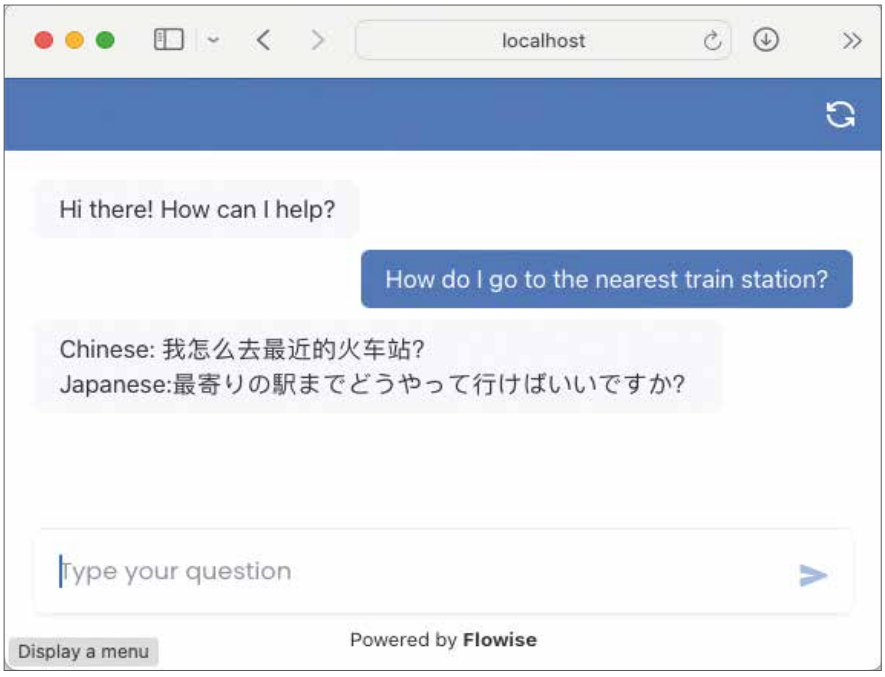


Figure 13: Running the chatbot as an independent web application

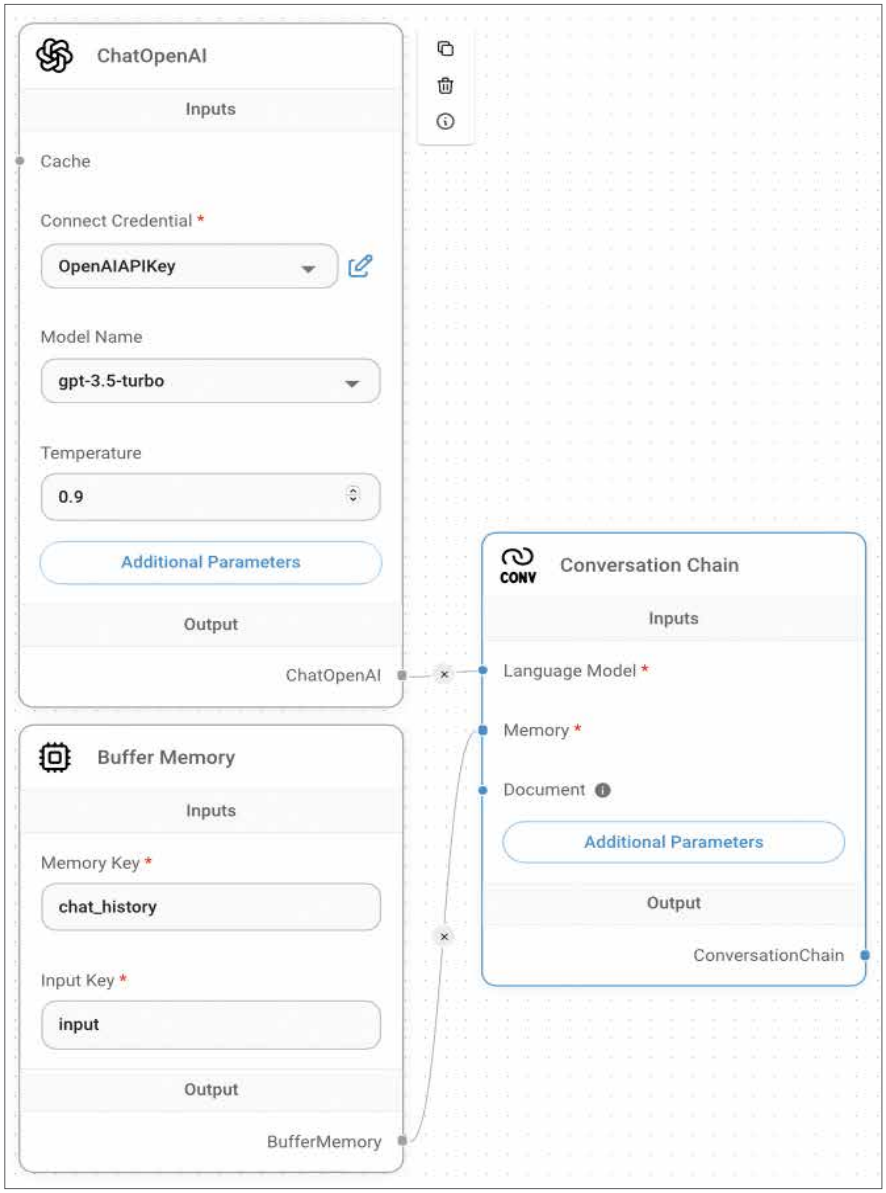


Figure 14: Connecting the nodes together

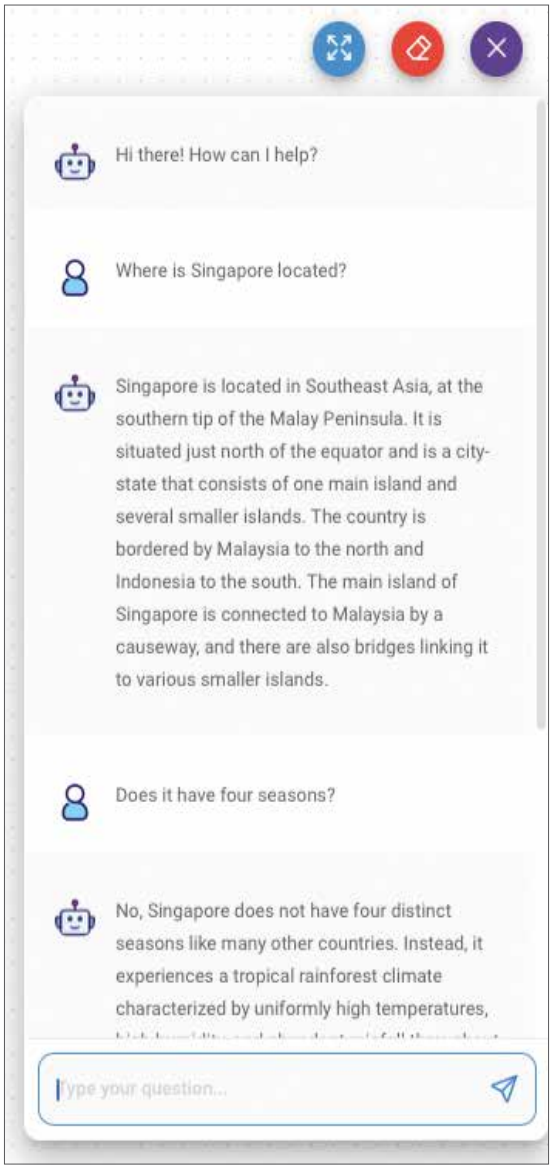


Figure 15: You can have a conversation with the chatbot and it remembers the context of the chat.

The result returned will look like this:

```
{"text":"\nChinese: 嘿,你好吗?
\nJapanese: こんにちは、お元気ですか?"}
```

You can also share the chatbot with other users. To do that, click on the **Share Chatbot** tab and copy the URL (see **Figure 12**).

When you paste the copied URL into a web browser, you'll be prompted to enter a username and password. By default, Flowise is started without authorization protection. You can enter anything for username and password and you'll be allowed accessed to the chatbot. If you want to bypass the prompting for username and password, check the **Make Public** option, as shown in **Figure 12**.

You can now see the chatbot. This provides a convenient way for users to make use of your project straight away (see **Figure 13**).

Creating a Conversational Chatbot

The second project to create is a conversational chatbot. Using Flowise, you can create a chatbot that functions very much like ChatGPT. Here are the nodes that you'll be using for this project:

- **ChatOpenAI**: You use this node to specify the model you want to use from OpenAI. Also, you need to specify your OpenAI API key, so take note that you will be charged based on your usage. For this example, you'll use the **gpt-3.5-turbo model**.
- **Buffer Memory**: The memory to remember your conversation.
- **Conversation Chain**: A conversation chain that takes in a LLM and memory with the prompt template already configured for chatting.

Figure 14 shows how the nodes are connected. Note that for this example, the inferencing (running of the LLM) is performed by OpenAI.

Save the project and then click the **Chat** button. You can chat with the OpenAI LLM and follow up with questions (see **Figure 15**). As the conversation chain is connected to the buffer memory, you can maintain a conversation with the LLM.

Alternatively, if you want to build a chat application without paying for the LLM (as in the case of OpenAI), you can use the **HuggingFace Inference** node. **Figure 16** shows how you can do that using the following nodes:

- **HuggingFace Inference**: Use this node to make use of a LLM hosted by HuggingFace. You need to specify your HuggingFace Token key.
- **Prompt Template**: Configures the prompt template.
- **LLM Chain**: Connects to the LLM and prompt template.

Note that for the **HuggingFace Inference** node, there are two ways to use the model:

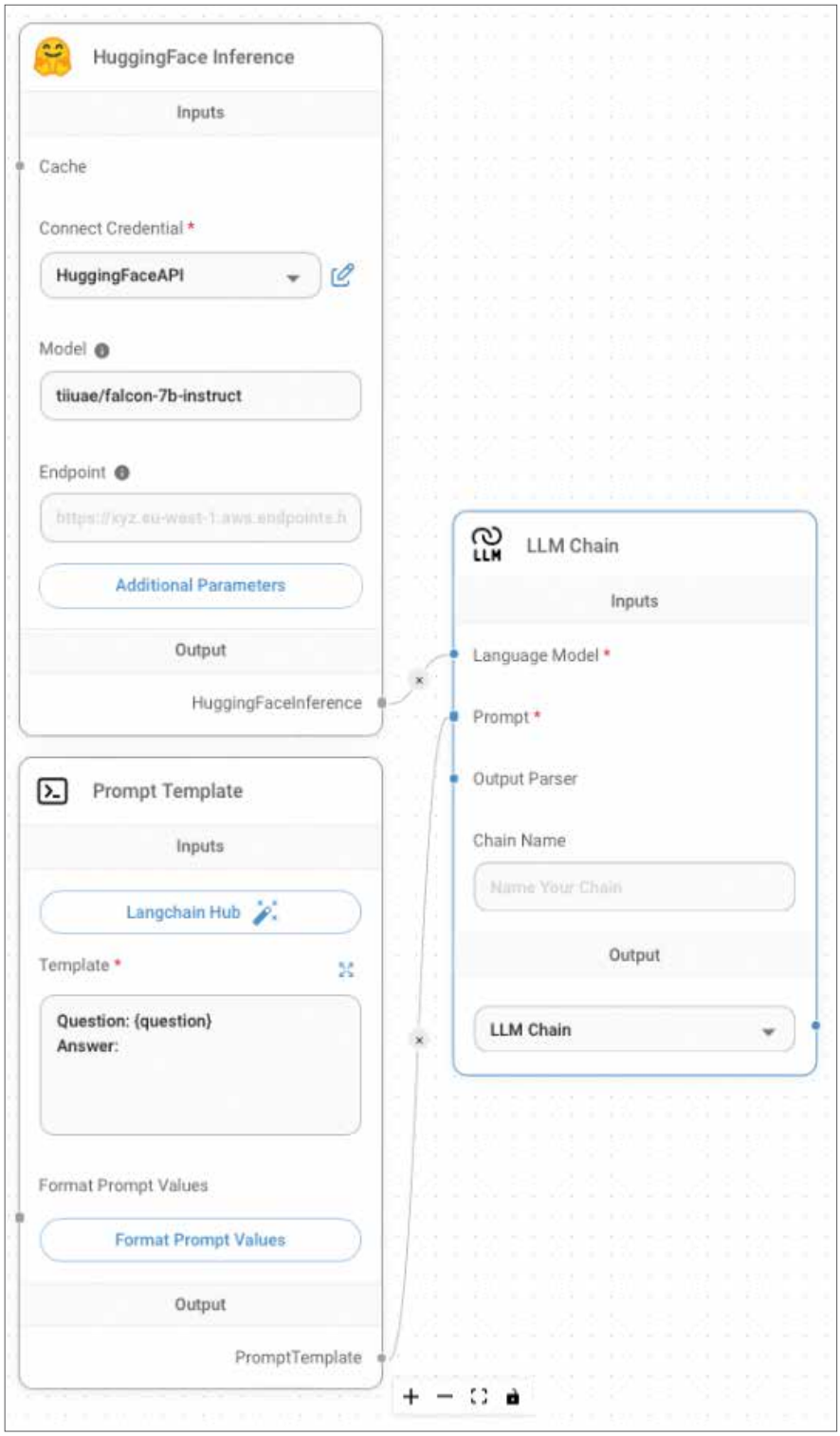


Figure 16: Using the HuggingFace Inference node for chatting

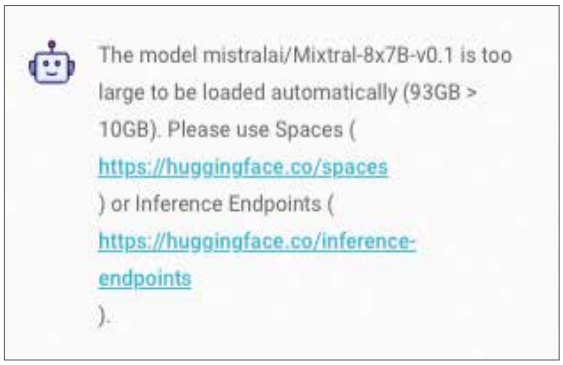


Figure 17: The error displayed when the model is too large for your machine

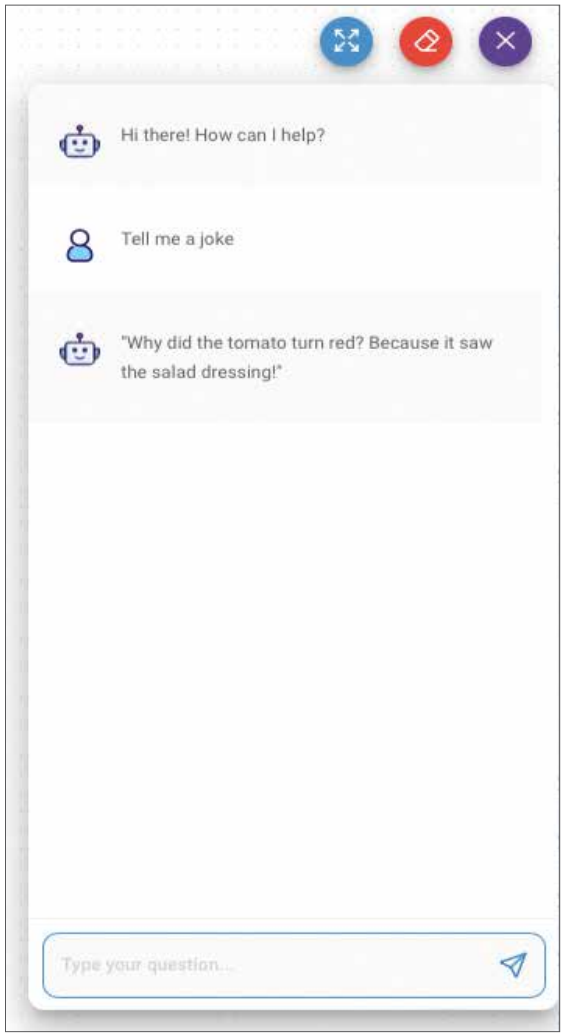


Figure 18: Chatting with the model from HuggingFace

DEV
Intersection

- If you specify the model name, **tiiuae/falcon-7b-instruct** in this example, the model will be downloaded to your computer and run locally. If you try to use a model that's too large, such as **mistralai/Mixtral-8x7B-v0.1**, you may get an error, as shown in **Figure 17**.
- To use a larger LLM (such as **mistralai/Mixtral-8x7B-v0.1**), you need to use HuggingFace Inference Endpoints (https://huggingface.co/inference-endpoints), which runs the model on the cloud (by HuggingFace).

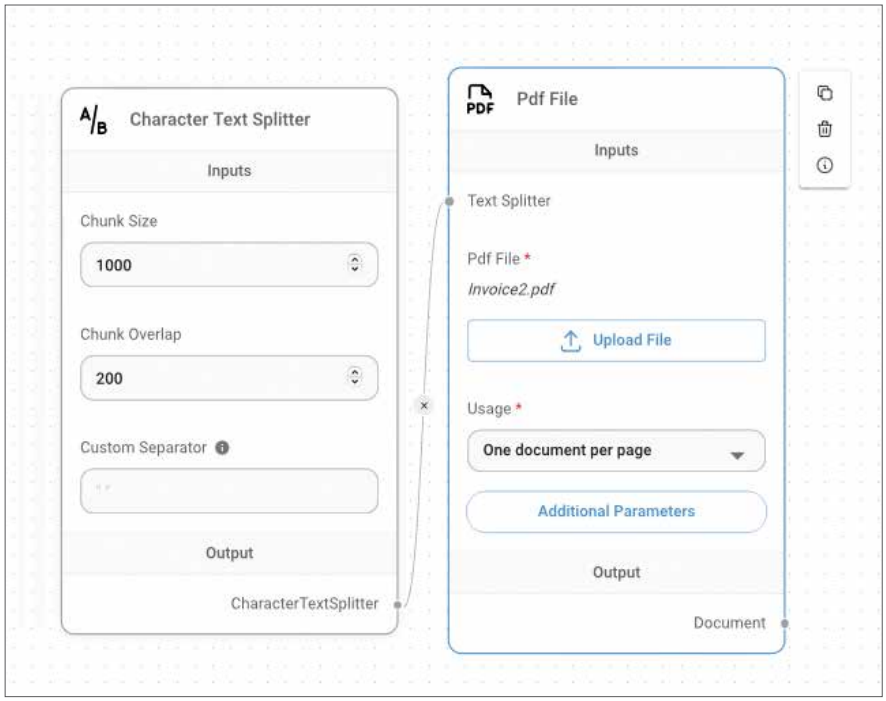


Figure 20: Connecting the Character Text Splitter node to the Pdf File node

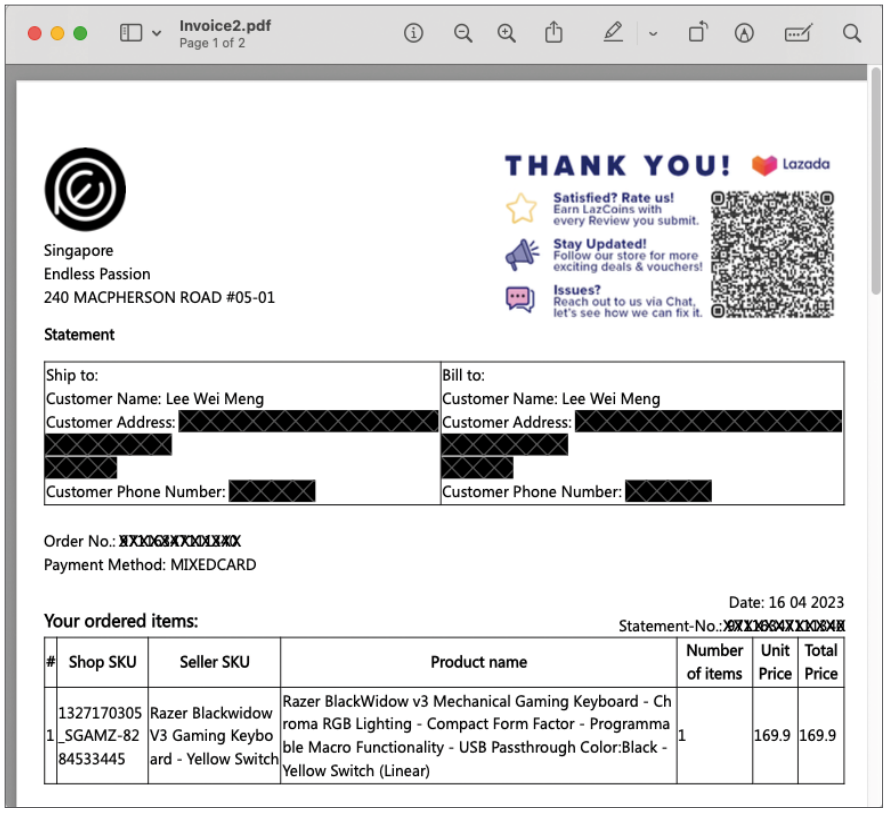


Figure 21: The sample invoice PDF file

> Vector embedding, also known as word embedding or vector representation, is a technique used in natural language processing (NLP) and machine learning to represent words or phrases as numerical vectors.
> The idea behind vector embedding is to capture the semantic relationships and contextual information of words in a continuous vector space.

Figure 18 shows the chatbot using the **tiiuae/falcon-7b-instruct** model.

Querying Local Documents

Chatting with an LLM is fun, but in the real world, businesses are more interested in whether they are able to make use of AI to query their own data. Well, using a technique known as **vector embedding**, you're now able to do just that.

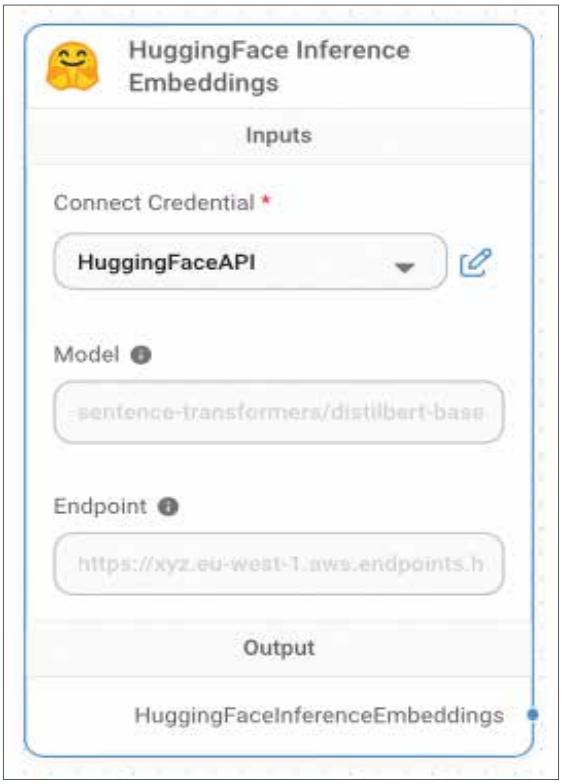


Figure 22: Configuring the HuggingFace Inference Embeddings node

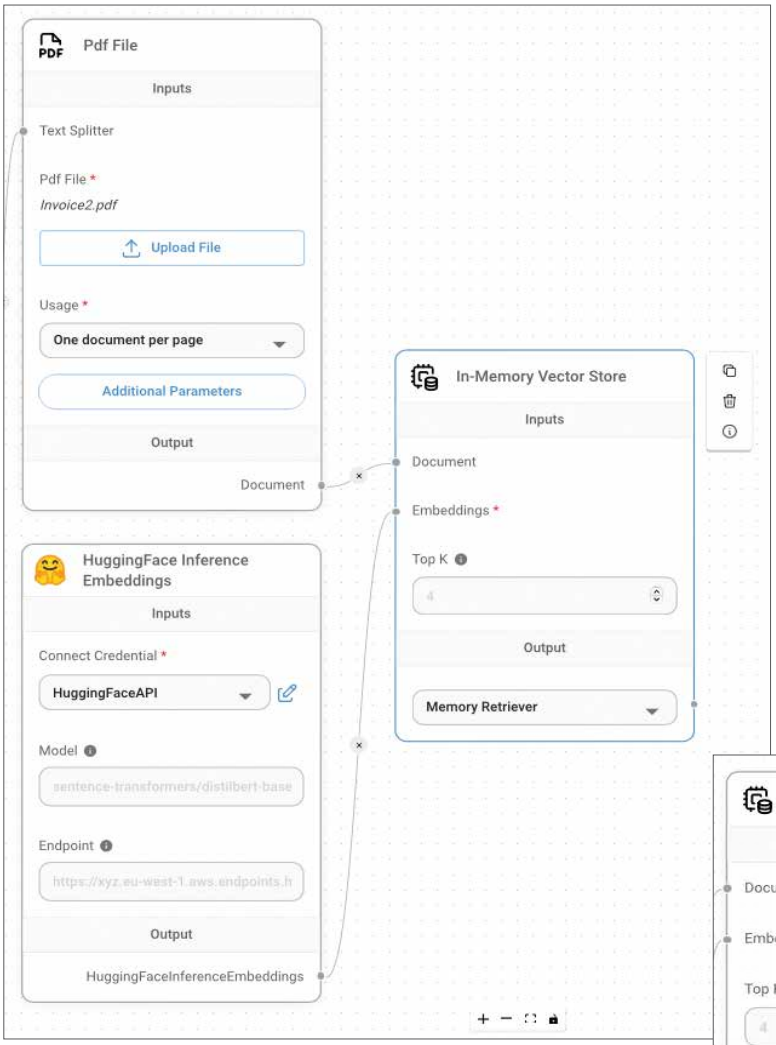


Figure 23: Connecting all the nodes together

In this section, you'll build an application that allows you to query your own PDF document. You'll need the following nodes:

- **Character Text Splitter**: Use this node to split a long document into smaller chunks that can fit into your model's context window.
- **PDF File**: Loads a PDF document for processing.
- **HuggingFace Interface Embeddings**: Use this node to perform embedding. Embedding refers to the representation of words or sentences as vectors in a high-dimensional space. It's a way to represent words and sentences in a numerical manner. Note that in this example, you can also make use of the **OpenAI Embeddings** node, but this will incur charges. In contrast, the **HuggingFace Interface Embeddings** node uses the embedding model from Hugging Face, which is free.
- **In-Memory Vector Store**: Use this node to store embeddings in-memory and it performs an exact, linear search for the most similar embeddings.
- **OpenAI**: Use this node to make use of an LLM from OpenAI to perform querying of your local data.
- **Conversational Retrieval QA Chain**: Use this node to create a retrieval-based question answering chain that is designed to handle conversational context.

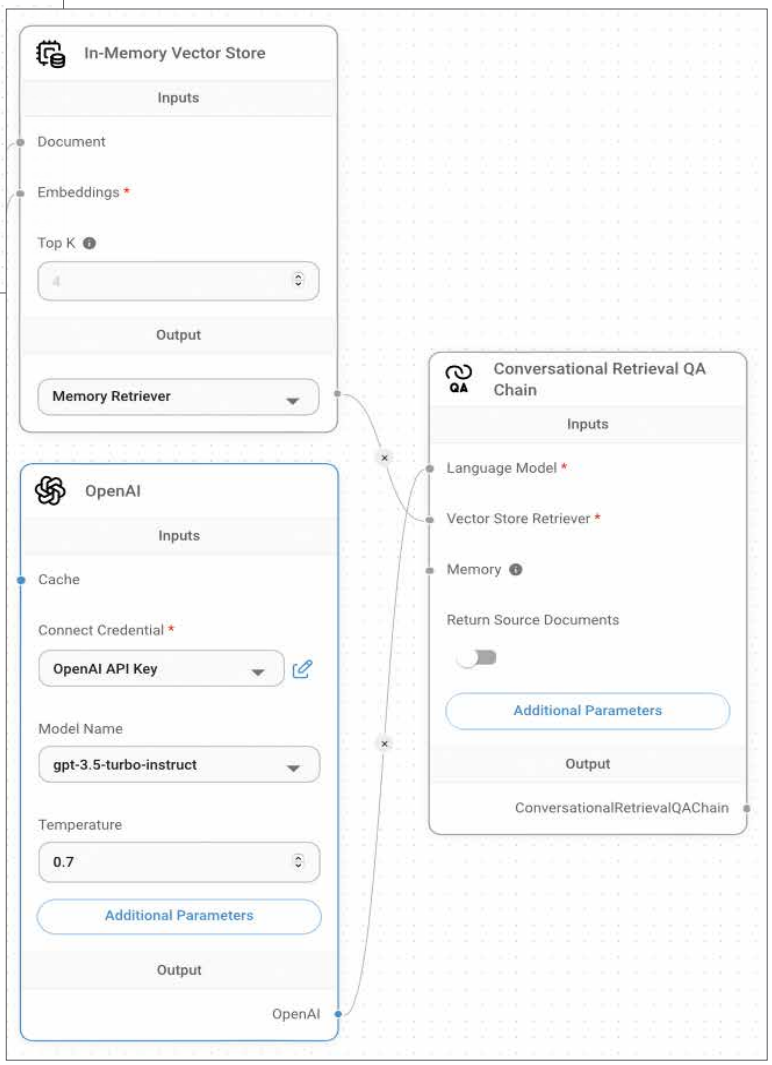


Figure 24: Adding the OpenAI and Conversational Retrieval QA Chain node

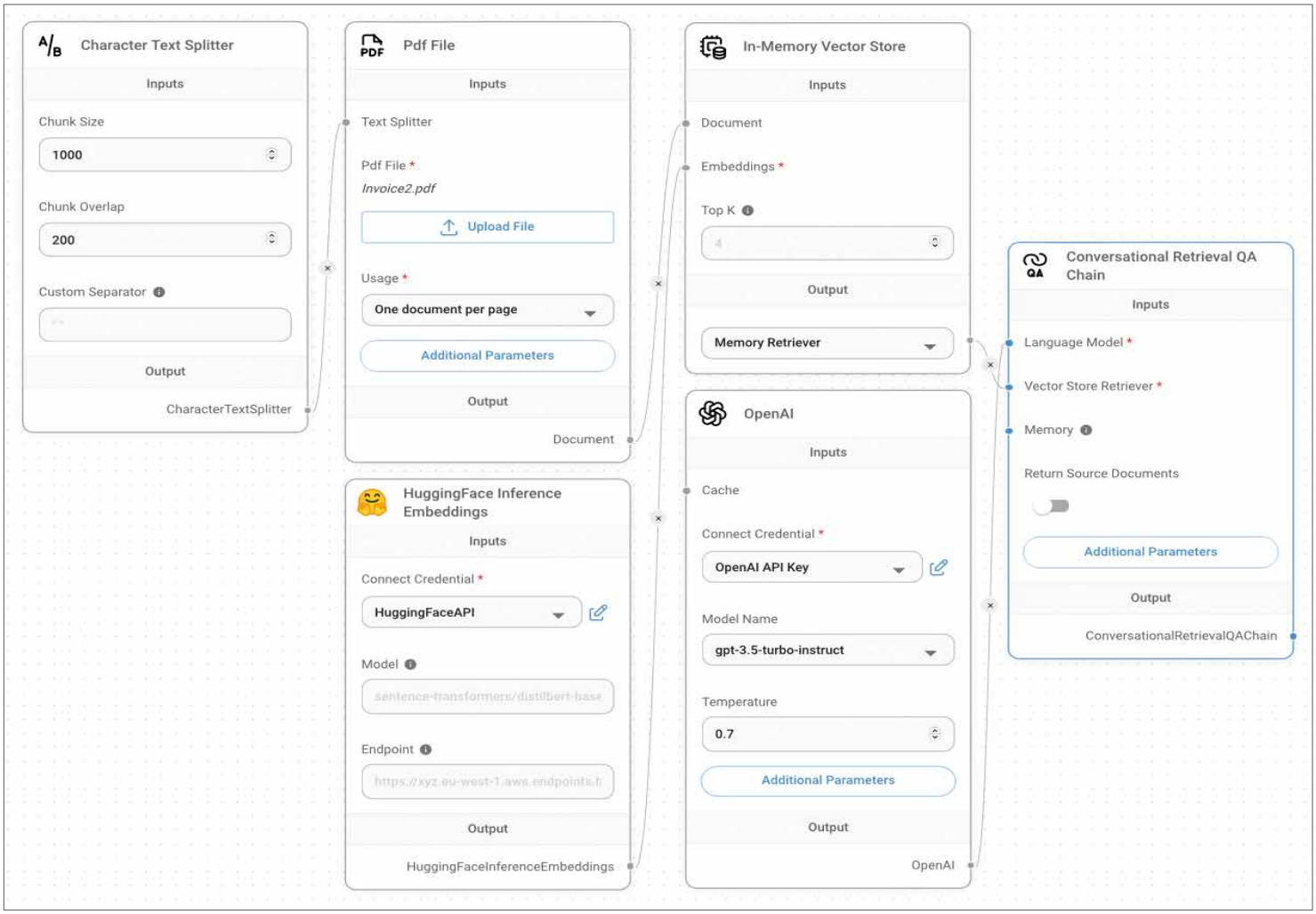


Figure 25: The complete project

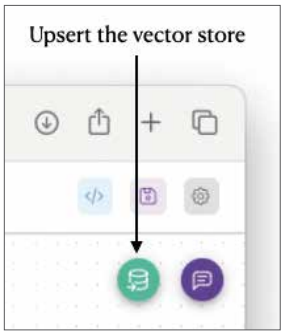


Figure 26: Clicking the Upsert button

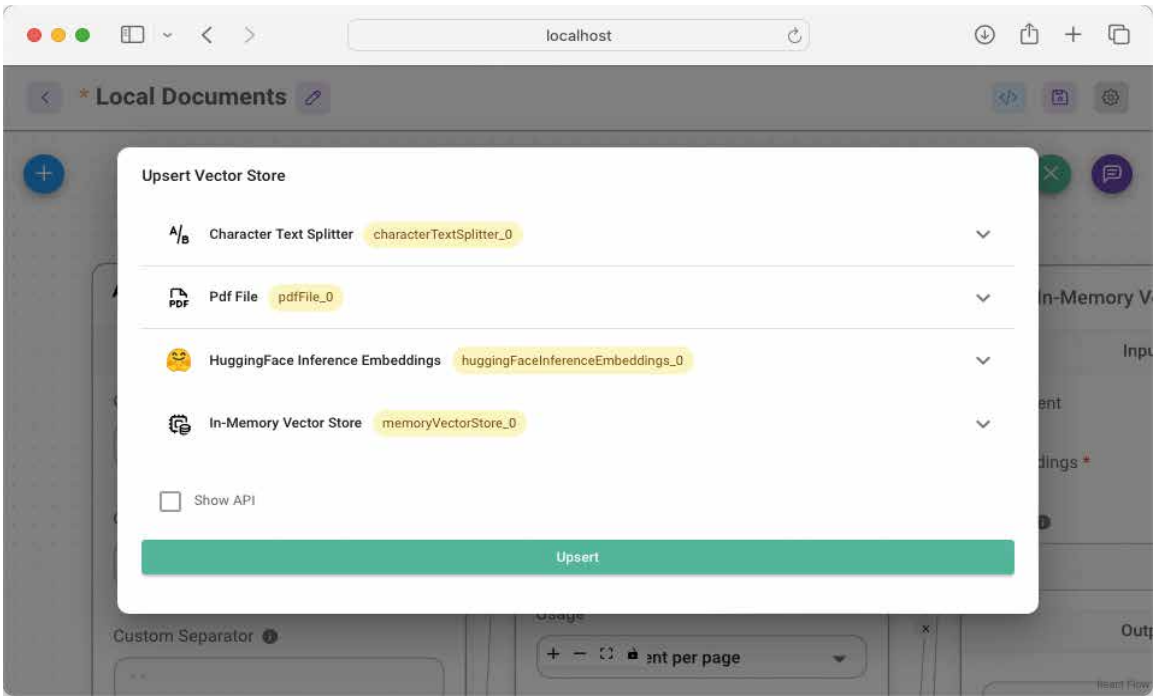


Figure 27: The steps performed by the Upsert button

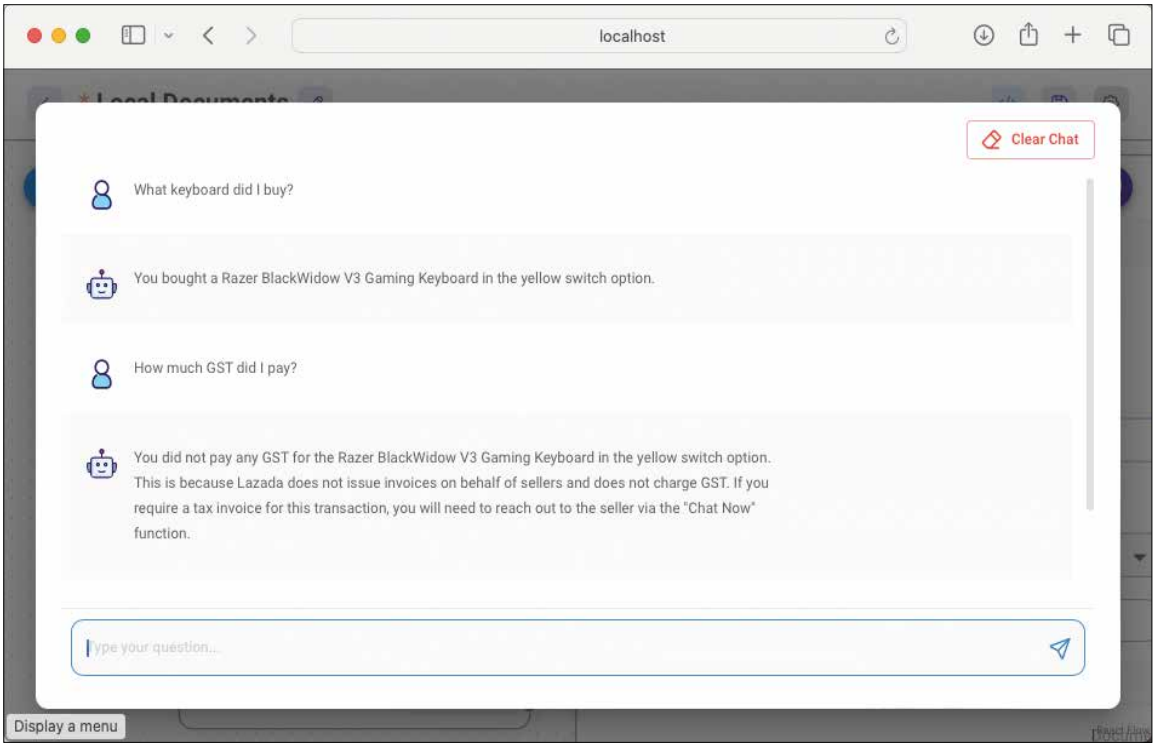


Figure 28: You can now ask questions pertaining to your PDF document.

> You can obtain a Hugging Face API Token from https://huggingface.co/settings/tokens.

You'll learn how to add each of the above nodes and connect them in the following steps.

First, add the **Character Text Splitter** node and configure it as shown in **Figure 19**.

Then, add the **Pdf File** node to the canvas as shown in **Figure 20**. To upload a local PDF document for querying, click the **Upload File** button and select the PDF document that you want to use. Then, connect the **Character Text Splitter** node to the **Pdf File** node.

Figure 21 shows the content of the PDF document that contains the online purchase of an item.

Next, add the **HuggingFace Interface Embeddings** node to the canvas and enter your Hugging Face API token (see **Figure 22**).

The next node to add is **In-Memory Vector Store**. Add this node to the canvas and connect it to the **Pdf File** and **HuggingFace Inference Embeddings** nodes as shown in **Figure 23**.

Next, drag and drop the **OpenAI** node and set your OpenAI API key. Finally, add the **Conversational Retrieval QA Chain** node and connect it to the other nodes as shown in **Figure 24**.

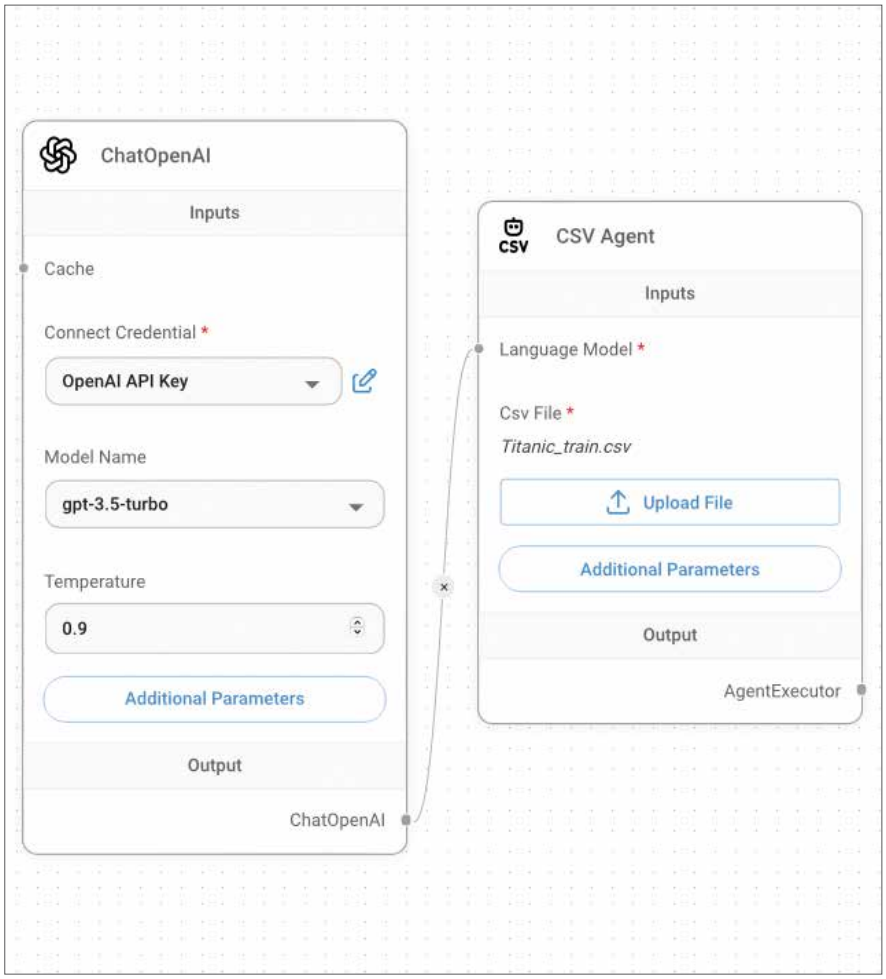


Figure 29: Using the ChatOpenAI and CSV Agent nodes

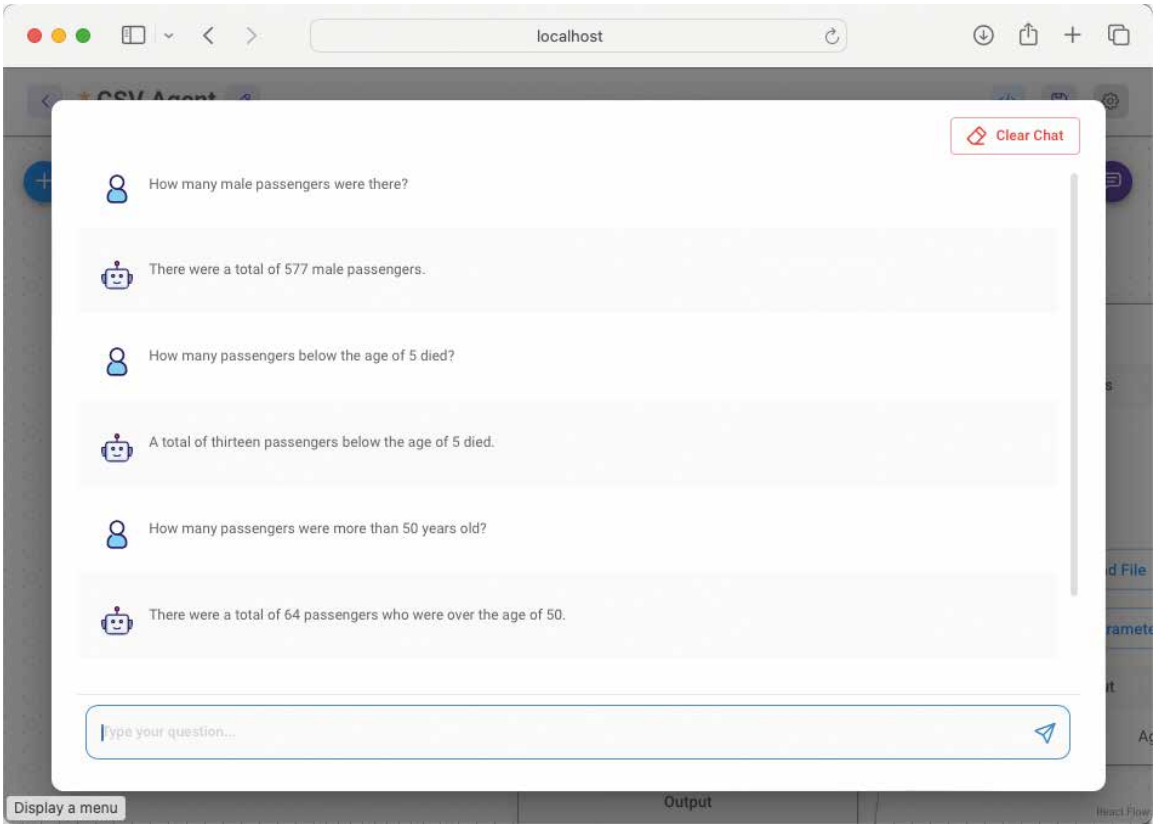


Figure 30: Asking questions based on the content of the CSV file

The complete project is shown in **Figure 25**.

> The term upsert refers to an operation that inserts rows into a database table if they don't already exist, or updates them if they do.

Before you can run the project, you need to click the **Upsert** button, as shown in **Figure 26**.

You'll see the popup, as shown in **Figure 27**.

Clicking the **Upsert** button performs a few operations: text splitting on the PDF document, creating embeddings using the HuggingFace Inference Embeddings, and then storing the vectors in-memory.

Once the upsert is done, you can now click on the **Chat** button to start the chatbot (see **Figure 28**).

Using Agents for Analyzing Data

LLMs are designed primarily for generating responses related to natural language understanding. Nevertheless, they exhibit notable limitations when faced with analytical inquiries. For example, when presented with a CSV file and asked for a summary of its contents, LLMs often demonstrate limited capabilities in providing a satisfactory answer. To use LLMs for analytical tasks, a common approach involves employing the LLM to generate the necessary code for the query and subsequently executing the code independently.

> In short, an agent helps you accomplish your tasks without you needing to worry about the details.

In LangChain, there is a feature known as **agents**. An **agent** is a system that decides what action is to be taken by the LLM, and it tries to solve the problem until it reaches the correct answer. Agents are designed to perform well-defined tasks, such as answering questions, generating text, translating languages, summarizing text, etc.

For this section, I want to show how you can make use of the **CSV Agent** in LangChain (and in Flowise) to perform analytical tasks on a CSV file. The **CSV Agent** node operates by reading a CSV file in the background. It employs the Pandas DataFrame library and uses the Python language to execute Python query code generated by an LLM.

For the CSV file, I'll be using the Titanic training dataset (Source of Data: https://www.kaggle.com/datasets/tedllh/titanic-train. Licensing — Database Contents License (DbCL) v1.0 https://opendatacommons.org/licenses/dbcl/1-0/).

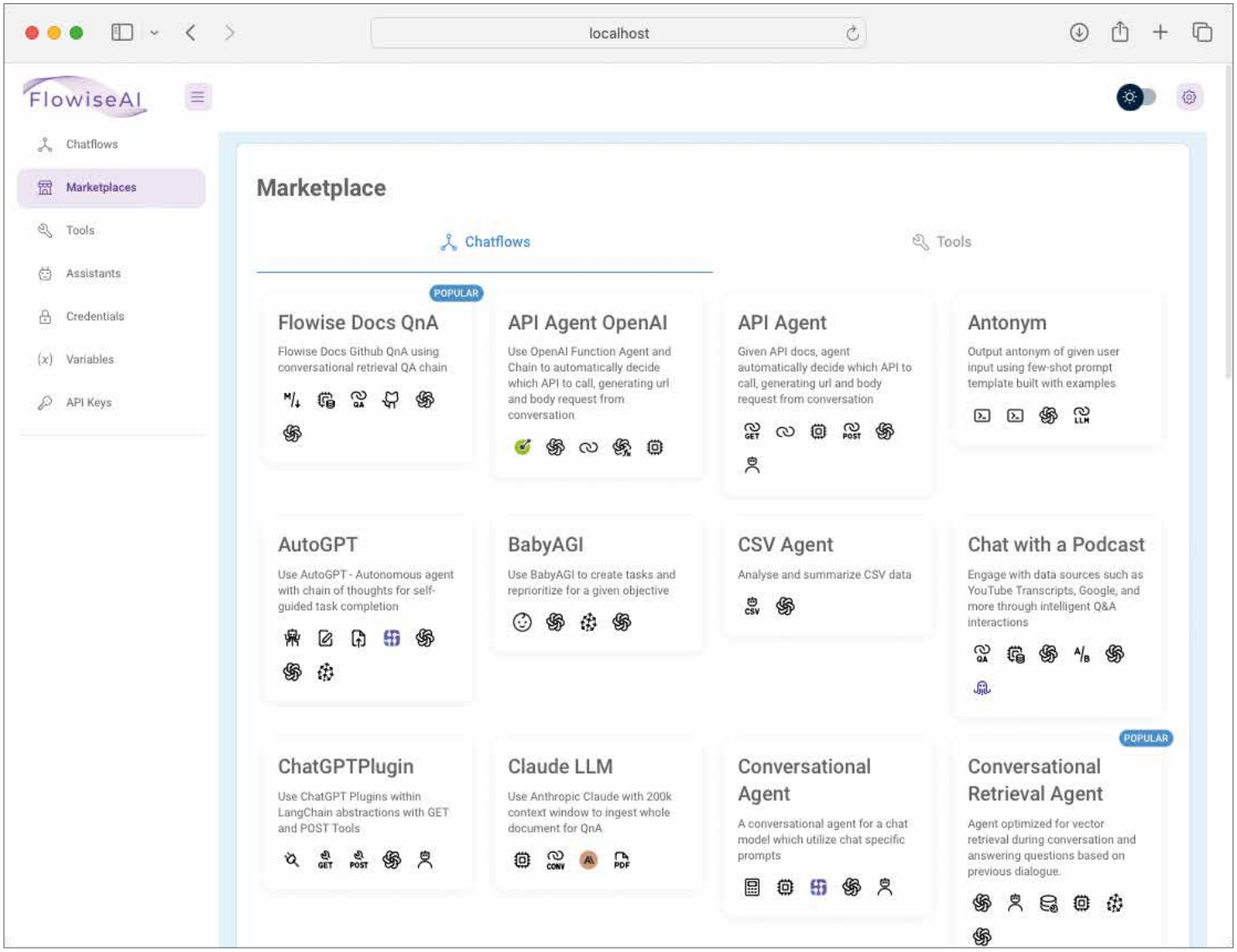


Figure 31: Go to the Marketplace for templates of popular applications you can build.

Let's create a new Flowise project and add the following components to the canvas:

- **ChatOpenAI**: Remember to enter your OpenAI API key.
- **CSV Agent**: Click on the **Upload File** button to select the Titanic CSV file **Titanic_train.csv**.

Figure 29 shows how the nodes are connected.

Save the project and then click the **Chat** button to display the chatbot (see **Figure 30**). You can now ask analytical questions pertaining to the CSV file, such as:

- How many male passengers were there?
- How many passengers below the age of five died?
- How many passengers were more than 50 years old?

Summary

I hope this article provides motivation for you to get started with LangChain programming. Although LangChain programming can be intimidating, Flowise takes away the fear and provides a low-code/no-code experience for you to get started with AI programming. One more thing: If you want to learn more about the capabilities of Flowise, the fastest way is to learn from the templates provided. **Figure 31** shows the **Marketplaces** in Flowise where you can get the templates for popular applications that you can build with Flowise. Good luck and have fun!

Wei-Meng Lee

Semantic Kernel 101: Part 2

In the previous installment (https://codemag.com/Article/2401091/Semantic-Kernel-101), I Introduced Semantic Kernel (SK), Microsoft's framework for working with LLMs, which was in preview at the time. I didn't include any code in that article because Semantic Kernel was still changing fast and almost every line of code would likely have been out of date before you read it.

Mike Yeager
myeager@eps-software.com

Mike is the CEO of EPS's Houston office and a skilled .NET developer. Mike excels at evaluating business requirements and turning them into results from development teams. He's been the Project Lead on many projects at EPS and promotes the use of modern best practices, such as the Agile development paradigm, use of design patterns, and test-drive and test-first development. Before coming to EPS, Mike was a business owner developing a high-profile software business in the leisure industry. He grew the business from two employees to over 30 before selling the company and looking for new challenges. Implementation experience includes .NET, SQL Server, Windows Azure, Microsoft Surface, and Visual FoxPro.

So I promised a follow-up article that explored the code. Since then, SK went into beta and then Release Candidate, and, as I write this, V1 has just released and I expect that namespaces, type names, and arguments are now in their final form, or very close to it.

To follow along with the code in the article, you'll need access to an LLM. I'm using Azure OpenAI. As an alternative, you can connect to an OpenAI account.

Note: Although I don't have code for using OpenAI in this article, if you have an OpenAI development account, you can just replace the calls to AddAzureOpenAIChatCompletion with **AddOpenAIChatCompletion** and pass in your clientId instead of endpoint and apiKey.

From the Azure portal, choose **Create a resource** and search for OpenAI, and then click Create. As of this writing, the GPT-4 model is only available in three regions: Sweden Central, Canada East, and Switzerland North. For our purposes, select one of the above regions. Next, choose Keys and Endpoint from the menu on the left and copy Key 1 and the endpoint, as you'll need those soon.

Back on the Overview page for the Azure OpenAI resource, you'll be directed to the Azure OpenAI Studio web page. There's also now a preview version of the page, so it may change soon, but I'll stick with the current page for this article. On that Azure OpenAI Studio page, from the menu on the left-hand side, choose Deployments and create a new deployment. You can think of a deployment as an instance of a model. You can create as many deployments as your account is authorized for. Currently, that's about six deployments per resource. The most capable model available as I'm writing this is GPT-4 or GPT-4-32k. When it becomes available, GPT-4 Turbo should be both more powerful and less expensive. Most of the work you're going to do today can be done with GPT-35-turbo, except for the work you'll do with the Handlebars Planner. Planners work much better with GPT-4 or above. For the purposes of this article, choose the GPT-4 model.

Accept the default of using the latest version of the model and give it a name. To keep things simple, just name the deployment **gpt-4** to match the model's name. You can choose Chat from the left-hand menu and try out your model. Type, "Tell me a joke" in the chat session window and submit.

Now that the LLM is ready, let's write some code. In Visual Studio or VS Code, create a new .NET 8 console application. Add the following NuGet packages:

- Microsoft.Extensions.Logging.Console
- Microsoft.SemanticKernel
- System.Configuration.ConfigurationManager

Modify your Program.cs file to look like this:

```
internal class Program
{
  private static string _endpoint;
  private static string _apiKey;

  static async Task Main(string[] args)
  {
    _endpoint = ConfigurationManager
          .AppSettings["endpoint"];
    _apiKey = ConfigurationManager
      .AppSettings["apiKey"];

    Console.Clear();

    await InlineFunctions();
  }
}
```

You'll need to add an App.config file to your project if there isn't one already and enter the endpoint and API key that you copied earlier. You can find in the Azure portal on your newly created Azure OpenAI resource under the Keys and Endpoint menu item.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="endpoint" value="<your endpoint>"/>
    <add key="apikey" value="<your key>"/>
  </appSettings>
</configuration>
```

Next, add the body of the InlineFunctions() method, just below Main(), as shown in **Listing 1.**

The code starts by creating a builder, much like the way you use builders to create web application hosts in ASP.NET Core. In this case, you're configuring the builder by adding console logging so you can see what the Semantic Kernel is doing as it works. This is why you included the Microsoft.Extensions.Logging.Console NuGet package. Like ASP.NET, you can use any logger available from Microsoft or create your own. I find the console logger great for development.

You then add an Azure OpenAI chat completion service and pass it the endpoint and API key you read from the config file, as well as the name of the deployment you created in Azure OpenAI Studio.

Next, create an instance of the kernel by calling the Build() method on the builder. The next few lines create a prompt and some settings for the prompt on the fly. This is called an inline semantic function and it's the equivalent of "hello world" in SK. Finally, call kernel.In-

Listing 1: Execute an online semantic function.

```
private static async Task InlineFunctions()
{
  var builder = Kernel.CreateBuilder();

  builder.Services
    .AddLogging(configure => configure.AddConsole())
    .AddAzureOpenAIChatCompletion(
        "gpt-4",  //deployment (not model) name
        _endpoint,
        _apiKey);

  var kernel = builder.Build();

  var prompt = """
    {{$input}}

    Rewrite the above in the style of Shakespeare.
    Be brief.
    """;

  var settings = new PromptExecutionSettings
    {
      ExtensionData = new Dictionary<string, object>
      {
        { "temperature", 0.9 }
      }
    };

  var shakespeareFunction = kernel
    .CreateFunctionFromPrompt(prompt, settings);

  var result = await kernel.InvokeAsync(
    shakespeareFunction,
    new KernelArguments
    {
      ["input"] = "I'm fascinated by AI technologies."
    });

  Console.WriteLine(result);
}
```

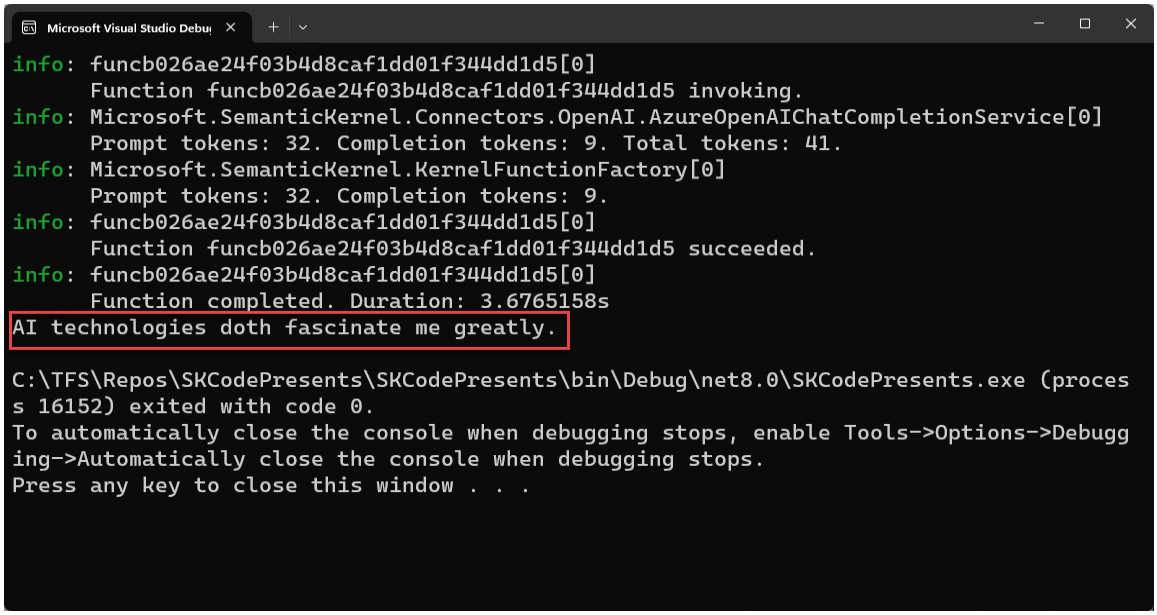


Figure 1: Response to the inline function, including logging output.

vokeAsync(), passing the newly created function and the parameters you want it to use. In this case, the parameter is named "input" in the prompt, so assign the text to that parameter. Then write the response to the console window, as shown in **Figure 1**.

You can try changing the input text, or even getting input from the user, using Console.ReadLine().

Next, let's do something a little more realistic and read in the prompts and settings you're going to call from files on disk instead of hard coding them in the application. Back in the Main method, comment out the **await InlineFunctions()**; line and add a new line **await BasicFunctions()**;. Then add the code from **Listing 2** below the InlineFunctions method. Notice that I've commented out the AddLogging() call to minimize the output on screen.

Next, add the files containing the prompts and settings you're going to use. You can copy the files from the source code that accompanies this article or create them by hand. To create them by hand, first, create a folder in the console project named **Plugins**. Under the Plugins folder, create a folder named Fun and under the Fun folder, create three folders named **Excuses, Joke, and Limerick**. In the Excuses folder, create a file named skprompt.txt and enter the text from the following code:

```
Generate a creative reason or excuse for the
given event.
Be creative and be funny. Let your imagination
run wild.

Event:I am running late.
Excuse:I was being held ransom by giraffe
gangsters.

Event:{{$input}}
```

Also in the Excuses folder, create a second file named config.json and enter this text.

Listing 2: Load prompts from disk and execute as semantic functions.

```
private static async Task BasicFunctions()
{
  var builder = Kernel.CreateBuilder();

  builder.Services
    //.Services.AddLogging(configure =>
      //configure.AddConsole())
    .AddAzureOpenAIChatCompletion(
      "gpt-4",
      _endpoint,
      _apiKey);

  var kernel = builder.Build();

  var functionDir =
    Path.Combine(Directory.GetCurrentDirectory(),
      "Plugins", "Fun");
  var semanticFunctions =
    kernel.ImportPluginFromPromptDirectory(functionDir);

  Console.WriteLine("HERE IS A LAME EXCUSE...");
  var excuseResult = await kernel
    .InvokeAsync(semanticFunctions["Excuses"],
      new KernelArguments { ["input"] = "my cat" });
  Console.WriteLine(excuseResult);
  Console.WriteLine();

  Console.WriteLine("HERE IS A LAME JOKE...");
  var jokeResult = await kernel
    .InvokeAsync(semanticFunctions["Joke"],
      new KernelArguments { ["input"] = "swimming" });
  Console.WriteLine(jokeResult);
  Console.WriteLine();

  Console.WriteLine("HERE IS A LAME LIMERICK...");
  var arguments = new KernelArguments
  {
      ["name"] = "Mike",
      ["input"] = "airplanes"
  };
  var limerickResult = await kernel
    .InvokeAsync(semanticFunctions["Limerick"], arguments);
  Console.WriteLine(limerickResult);
}
```

Listing 3: Contents of skprompt.txt. A predefined prompt on disk.

```
There was a young woman named Bright,
Whose speed was much faster than light.
She set out one day,
In a relative way,
And returned on the previous night.

There was an odd fellow named Gus,
When traveling he made such a fuss.
He was banned from the train,
Not allowed on a plane,
And now travels only by bus.

There once was a man from Tibet,
Who couldn't find a cigarette
So he smoked all his socks,
and got chicken-pox,
and had to go to the vet.

There once was a boy named Dan,
who wanted to fry in a pan.
He tried and he tried,
and eventually died,
that weird little boy named Dan.

Now write a very funny limerick about {{$name}}.
{{$input}}
Invent new facts about their life. Must be funny.
```

```
{
  "schema": 1,
  "description": "Turn a scenario into a creative
    or humorous excuse to send your boss",
  "type": "completion",
  "completion": {
    "max_tokens": 60,
    "temperature": 0.5,
    "top_p": 0.0,
    "presence_penalty": 0.0,
    "frequency_penalty": 0.0
  }
}
```

In the Joke folder, add an skprompt.txt file with the contents of this next snippet:

```
WRITE EXACTLY ONE JOKE or HUMOROUS STORY
ABOUT THE TOPIC BELOW

JOKE MUST BE:
- G RATED
- WORKPLACE/FAMILY SAFE
NO SEXISM, RACISM OR OTHER BIAS/BIGOTRY

BE CREATIVE AND FUNNY. I WANT TO LAUGH.
Incorporate the style suggestion,
if provided: {{$style}}
+++++

{{$input}}
+++++
```

> If you ask Semantic Kernel to write something funny, don't expect a dad joke or a stand-up act. It's early days and its sense of humor is a bit undeveloped.

Now add a config.json file with this:

```
{
  "schema": 1,
  "description": "Generate a funny joke",
  "type": "completion",
  "completion": {
    "max_tokens": 1000,
    "temperature": 0.9,
    "top_p": 0.0,
    "presence_penalty": 0.0,
    "frequency_penalty": 0.0
  },
  "input": {
    "parameters": [
      {
        "name": "input",
        "description": "Joke subject",
        "defaultValue": ""
      },
      {
```

```
        "name": "style",
        "description": "Give a hint about the
          desired joke style",
        "defaultValue": ""
      }
    ]
  }
}
```

In the Limerick folder, add an skprompt.txt file with the contents of **Listing 3** and a config.json file with the contents of **Listing 4**.

Right-click on each new file, choose Properties from the context menu, and set the Copy to Output Directory property of each file to **Copy if newer**. Note that you'll also have to do this step if you copied the files and folders from the accompanying source code. When you run the code, you should see output like that in **Figure 2**.

Looking at the code, you'll see that the first part is the same as before. The difference comes where you create the plug-ins (also known as functions). Instead of creating them in code, you call the ImportPluginFromPromptDirectory() method on the kernel and give it the path to the files you just created. Notice that each folder within the Fun directory contains two files: one for the prompt and one for the settings associated with the prompt. By using this type of folder structure, you can load all three plug-ins within the Fun folder at once. After loading, you're returned a dictionary containing all the plug-ins that were loaded and you can reference each plug-in from the dictionary when you make the Invoke.Async() calls. Also notice that the Limerick plug-in accepts two parameters, unlike the previous plug-ins, which accepted only one.

Now that you've done the basics and are proficient with generating text with SK, let's explore how to connect semantic code (prompts and prompt engineering) with native code (in this case, C#). Back in the Main method, comment out the **await BasicFunctions()**; line and add the new line **await NativeFunctions();**. Next, create a new file in the project named MyCSharpFunctions.cs and enter this code:

```
public class MyCSharpFunctions
{
  [KernelFunction]
  [Description("Return the first row of a
qwerty keyboard")]
  public string Qwerty(string input)
  {
    return "qwertyuiop";
  }

  [KernelFunction]
  [Description("Return a string that's duplicated")]
  public string DupDup(string input)
  {
    return input + input;
  }
}
```

These are extremely simple examples. The first returns the letters on the top row of the keyboard and the second duplicates whatever text is passed in, but you can program anything you can imagine. You have the full power of .NET at your disposal. Semantic Kernel treats sematic code and native code equally and the two can be intermixed. Notice that you add the KernelFunction attribute to each method you want to expose to Semantic Kernel. I've also added a Description attribute that not only helps document the code, but you'll also use it in the next article when I discuss planners.

Semantic Kernel treats semantic code and native code equally and the two can be intermixed.

Back in Main(), comment out **await BasicFunctions();** and enter a new line **await NativeFunctions();**. Below the BasicFunctions method, add the code from the next snippet.

Listing 4: Contents of config.json. Pre-defined prompt settings.

```
{
  "schema": 1,
  "description": "Generate a funny limerick about a person",
  "type": "completion",
  "completion": {
    "max_tokens": 200,
    "temperature": 0.7,
    "top_p": 0,
    "presence_penalty": 0,
    "frequency_penalty": 0
  },
  "input": {
    "parameters": [
      {
        "name": "name",
        "description": "",
        "defaultValue": "Bob"
      },
      {
        "name": "input",
        "description": "",
        "defaultValue": "Dogs"
      }
    ]
  }
}
```

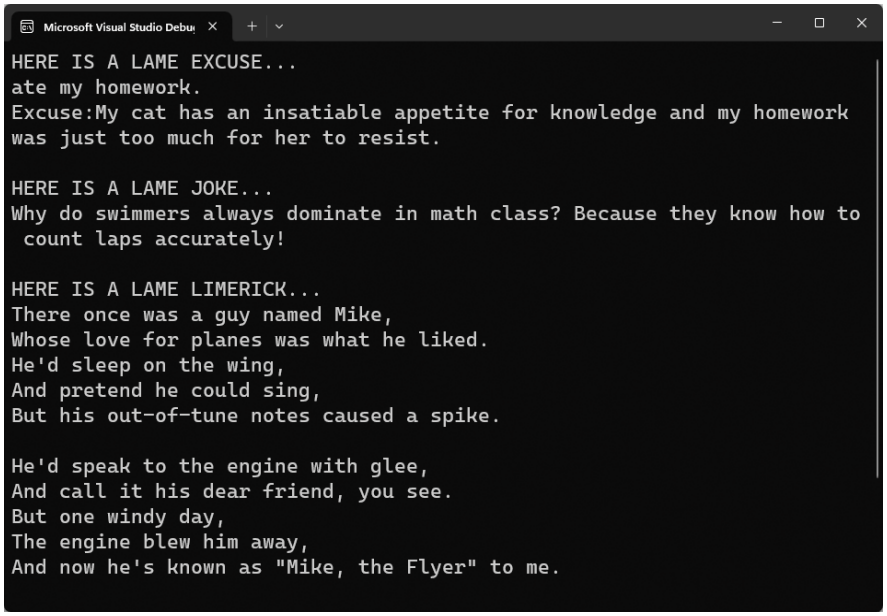


Figure 2: Excuse, joke, and limerick responses generated from pre-defined prompts.

Again, the code to create the kernel is the same as before and the difference comes when loading the native functions. You call the generic method ImportPluginFromType() with the type containing the functions. Then you call InvokeAsync(), exactly as you do for semantic functions.

```
private static async Task NativeFunctions()
{
  var builder = Kernel.CreateBuilder();

  builder.Services
    //.AddLogging(configure =>
      //configure.AddConsole())
    .AddAzureOpenAIChatCompletion(
      "gpt-4",
       _endpoint,
      _apiKey);

  var kernel = builder.Build();

  var nativeFunctions = kernel
    .ImportPluginFromType<MyCSharpFunctions>();

  var result = await kernel.InvokeAsync(
    nativeFunctions["Qwerty"],
    new KernelArguments { ["input"] = "hello" });
  Console.WriteLine(result);
  Console.WriteLine();

  result = await kernel.InvokeAsync(
    nativeFunctions["DupDup"],
    new KernelArguments { ["input"] = "hello" });
  Console.WriteLine(result);
  Console.WriteLine();
}
```

You may have noticed that the Qwerty function accepts an input parameter but doesn't use it. At one time, it was the standard in SK to accept a single-string named input and return a string as the output. That made it easier to chain multiple calls together and pass the output of one function as the input of the next. Parameter handling has gotten much more sophisticated since those early days, but you'll still see this pattern, so I kept it here.

Speaking of chaining multiple calls together (something SK calls pipelines, though not as automatic as they once were when inputs and outputs were simpler), it's often useful to run multiple semantic and/or native functions in succession. In this example, you ask the LLM to create a short poem and then ask it to tell you more about the poem it just created. Back in Main(), comment out **await NativeFunctions();** and add a new line **await Pipelines();**, and then add the code in this next snippet to the NativeFunctions method.

> Training AI on all the topics in the world is a pretty tall order and it's nowhere near finished being trained. In the limerick example, the rules of this type of poetry are not strictly followed (the rhythms don't quite work).

```
private static async Task Pipelines()
{
  var builder = Kernel.CreateBuilder();

  builder.Services
    //.Services.AddLogging(configure =>
      //configure.AddConsole())
    .AddAzureOpenAIChatCompletion(
      "gpt-4",
      _endpoint,
      _apiKey);

  var kernel = builder.Build();

  var functionDir =
    Path.Combine(Directory.GetCurrentDirectory(),
      "Plugins", "Writer");
  var semanticFunctions = kernel
    .ImportPluginFromPromptDirectory(functionDir);

  var functionPipeline = new KernelFunction[]
    {
      semanticFunctions["ShortPoem"],
      semanticFunctions["TellMeMore"]
    };

  var currentArg = "the sky";
  foreach (var function in functionPipeline)
  {
   var result = await kernel.InvokeAsync(function,
     new KernelArguments { ["input"] = currentArg });
   currentArg = result.ToString();
   Console.WriteLine(currentArg);
   Console.WriteLine();
  }
}
```

Useful Links

microsoft/semantic-kernel: Integrate cutting-edge LLM technology quickly and easily into your apps (github.com)

Orchestrate your AI with Semantic Kernel | Microsoft Learn

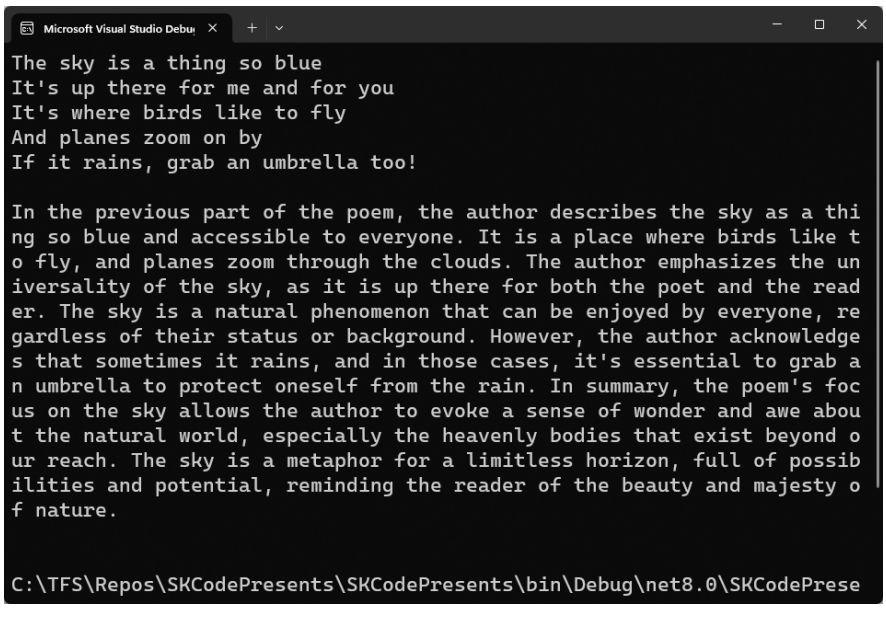


Figure 3: A generated poem, followed by a description of the poem

Although this pipeline only calls two functions, it can be extended to much more complex scenarios. Again, building the kernel is the same, and then you load the functions from disk as you did in the BasicFunctions example. In this case, you're using two new prompts that you haven't used before. Under the Plugins folder, add a new folder named Writer and under that, add two new folders named ShortPoem and TellMeMore. In the ShortPoem folder, add a new file named skprompt.txt and use the text from this snippet.

```
Generate a short funny poem or limerick to
explain the given event. Be creative and
be funny. Let your imagination run wild.
Event:{{$input}}
```

Then add a new file named config.json and use the text from this snippet.

```
{
  "schema": 1,
  "type": "completion",
  "description": "Turn a scenario into a
    short and entertaining poem.",
  "completion": {
    "max_tokens": 60,
    "temperature": 0.5,
    "top_p": 0.0,
    "presence_penalty": 0.0,
    "frequency_penalty": 0.0
  },
  "input": {
    "parameters": [
      {
        "name": "input",
        "description": "The scenario to turn
into a poem.",
        "defaultValue": ""
      }
    ]
  }
}
```

In the TellMeMore folder, add a new file named skprompt.txt and use the text from this snippet.

```
>>>>>The following is part of a
{{$conversationtype}}.
{{$input}}

>>>>>The following is an overview of a
previous part of the {{$conversationtype}},
focusing on "{{$focusarea}}".
{{$previousresults}}

>>>>>In 250 words or less, write a verbose
and detailed overview of the
{{$conversationtype}} focusing solely on
"{{$focusarea}}".
```

Then add a new file named config.json and use the this text:

```
{
  "schema": 1,
  "type": "completion",
  "description": "Summarize given text or any
text document",
  "completion": {
    "max_tokens": 500,
    "temperature": 0.0,
    "top_p": 0.0,
    "presence_penalty": 0.0,
    "frequency_penalty": 0.0
  }
}
```

As before, make sure to right-click on each file and set the Copy to Output Directory property to Copy if newer. You should get output like that in **Figure 3**.

Conclusion

In this article, you did some hands-on AI programming with Semantic Kernel. In the Azure portal, you created a deployment of a GPT-4 Large Language Model. Then you created a simple prompt in code and executed it against the model. You then experimented with loading pre-engineered prompts and associated settings from disk. Next, you executed your own C# code in the same way you executed the prompts, showing how the two types of code, semantic and native, could be easily intermingled. You ended with an example of chaining functions together, using the output of one as the input of the next.

In the next article, I'll work with more advanced topics, including using the Retrieval Augmented Generation (RAG) pattern where you store some documents in a semantic database and then search them, not by keyword, but by meaning. You'll see how the RAG pattern allows you to "ground" the responses from the LLM and customize the information it uses to respond. I'll also show related utilities, like "chunkers," to cut large documents down to size so you can put them into the semantic database. Finally, I'll look at planners, where I ask the LLM to break down a large, complex problem into smaller steps and choose from both semantic and native code to automatically create and execute a pipeline of functions to complete those steps.

Mike Yeager
CODE

Aspirational .NET: What Is .NET Aspire?

.NET Core has been a great ride for all of us building web- and cloud-driven apps, but managing distributed apps made for cloud native has been a mishmash of different tools. That's about to change. Seeing a problem in how large microservice architectures are deployed and managed, the ASP.NET team has taken a big swing with a solution that should help most .NET developers think about microservices and distributed applications without the dread many of us have had.

Shawn Wildermuth
shawn@wildermuth.com
wildermuth.com
@shawnwildermuth

Shawn Wildermuth has been tinkering with computers and software since he got a Vic-20 back in the early '80s. As a Microsoft MVP since 2003, he's also involved with Microsoft as an ASP.NET Insider and ClientDev Insider. He's the author of over twenty Pluralsight courses, written eight books, an international conference speaker, and one of the Wilder Minds. You can reach him at his blog at http://wildermuth.com. He's also making his first, feature-length documentary about software developers today called "Hello World: The Film." You can see more about it at http://helloworldfilm.com.

The Problem

Building distributed apps can be difficult. This has been a truth in computer science since the very beginning. Today, we talk about using containers, Kubernetes, and creating microservices. But the nature of distributed computing is still very much the same as it was in the beginning. Creating applications that are distributed immediately require some basic requirements, including how the different services can reach each other, how to manage configuration across service boundaries, and how to monitor projects across the service boundaries. The community is trying to solve this with something called **cloud native**.

Aspire and other Cloud Native frameworks solve distributed applications, but for many organizations, there's not enough benefit to justify the complexity in moving to cloud-native.

Cloud Native? What's That?

The idea of cloud native comes from the basic desire to decouple services. Cloud native is an approach to run scalable applications in a variety of environments including public, private, and hybrid clouds. To achieve this, the architectural approach encourages:

- Resiliency
- Manageability
- Observability

Effectively, this means your distributed application is loosely coupled, can support scaling as necessary, and is using instrumentation to be able to monitor your applications in real-time. In addition, as seen in **Figure 1**, cloud native leans on four principles:

> Cloud native is an approach to running scalable applications in a variety of environments, including public, private, and hybrid clouds.

With these principles in mind, the goal of cloud native is an application that's composed. For large distributed applications, your project will need a variety of different types of services, not just the ones your organization authors. For example, some of the common services or resources that you need can be seen in **Figure 2.**

In order to accomplish this, you need a way to define and share information across the different services. There are a number of approaches that work (e.g., Kubernetes). For many .NET developers, it's been a challenge to learn how to integrate your services into a cohesive application. That's where .NET Aspire comes in.

What Is .NET Aspire?

For several years now, Microsoft has been working on a sidecar system for microservices called Dapr. The goals of

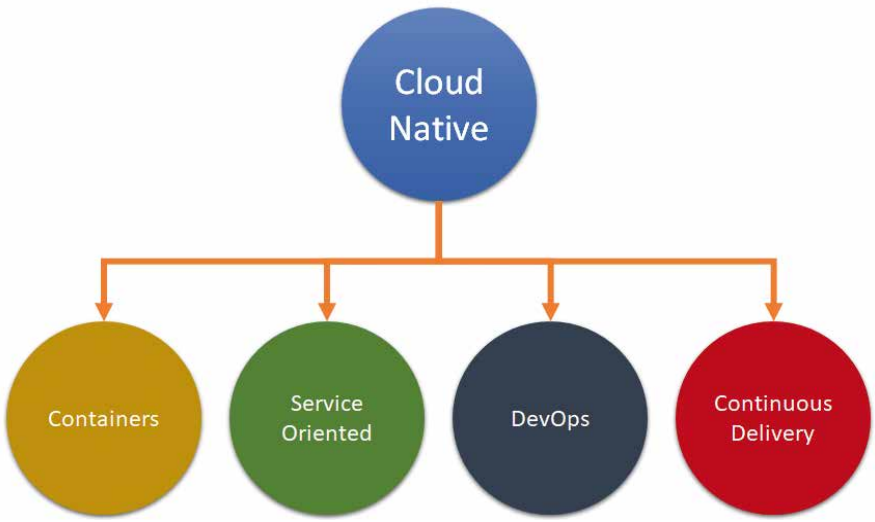


Figure 1: Principles of cloud native

Dapr were to enable the same sort of cloud native support that Aspire enables. That project was donated to the Cloud Native Computing Foundation (CNCF) and is a successful open-source project. In the wake of Dapr, the Aspire framework was created by the ASP.NET Core team to solve some of these same problems.

It seems the goal of Aspire is to simplify the set-up of multiple smaller projects (microservices or not) with related components that you might be using (e.g., data stores, containers, queues, message buses, etc.). This seems especially true for working with related Azure resources (e.g., Key Vault, Blob Storage, Service Bus, etc.). This means that you can compose applications to include projects that you build with common resources for your applications (on premises or in the cloud). Primarily, it's focused on several of the pain points including:

- Orchestration
- Composing distributed apps
- Service discovery
- Configuration
- Tooling

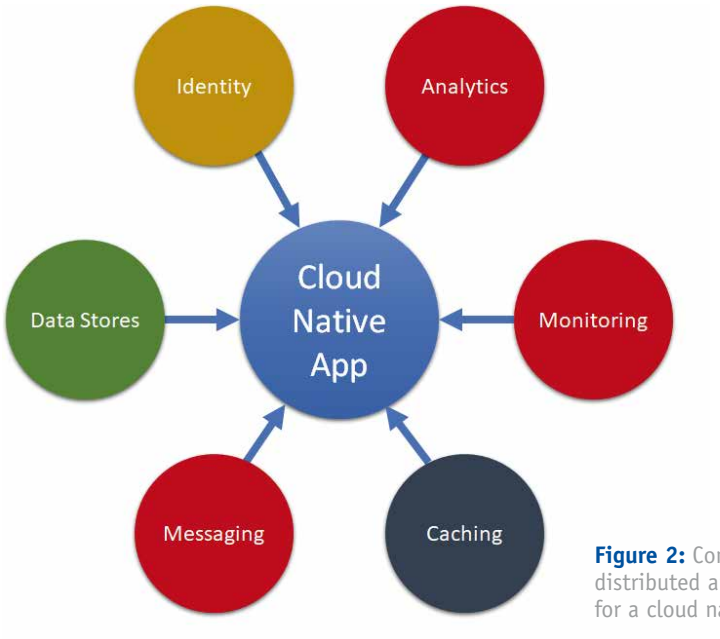


Figure 2: Composing a distributed application for a cloud native app

Before I dig into the details of how this works, let's start by looking at how you can create Aspire apps.

Tooling

In this early version of Microsoft Aspire, most of the tooling is in Visual Studio (although the dotnet CLI supports this too). Like many other projects, there's a new file template, as seen in **Figure 3**:

Out of the box, Visual Studio supports a simple "Application" and a "Starter Application." The difference is how much boilerplate and sample code it includes. The Starter Application is a great way to see a multi-component project work, but I think most people will start with the simple Application as a start to a distributed application. I don't think that either of these will be the most common approach.

Instead of that, most people will just add .NET Aspire to existing projects that they want to use orchestration with. Visual Studio has this option by just right-clicking your project, as seen in **Figure 4**.

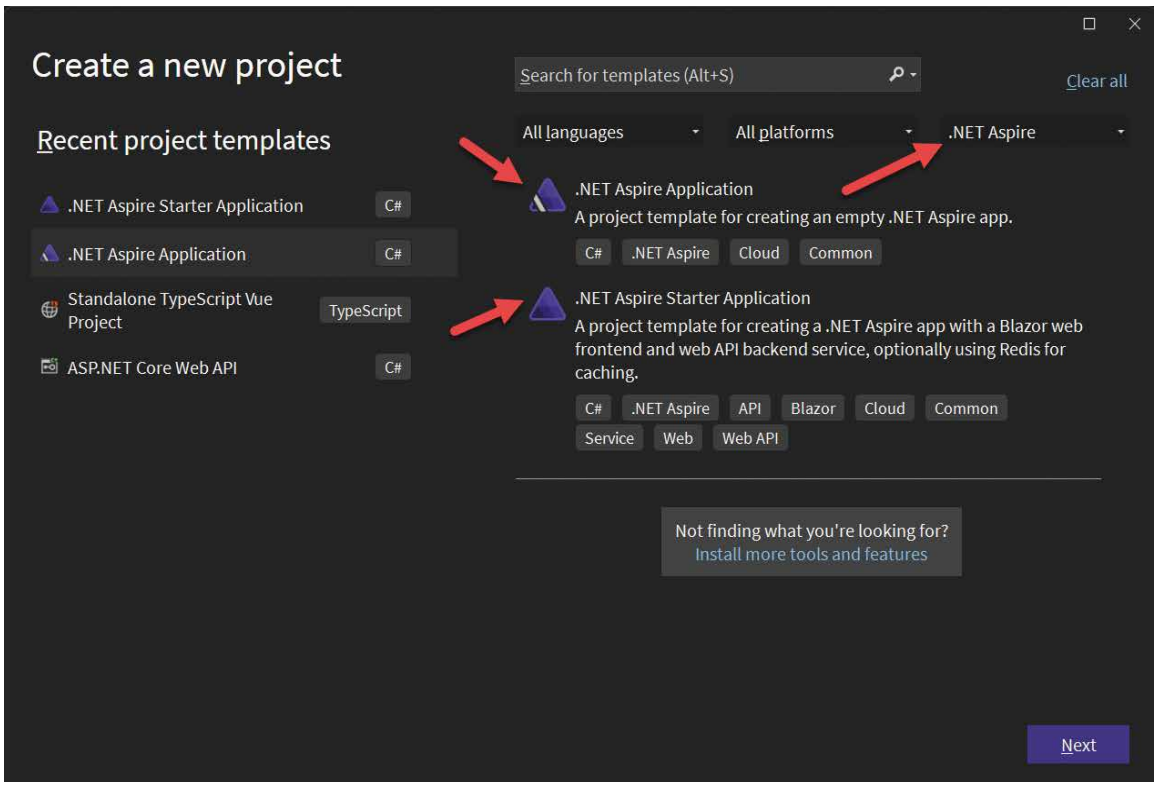


Figure 3: New .NET Aspire project templates

Which Version?

For this article, I'm using Preview 1 of Visual Studio 2022 and Preview 2 of Microsoft Aspire.

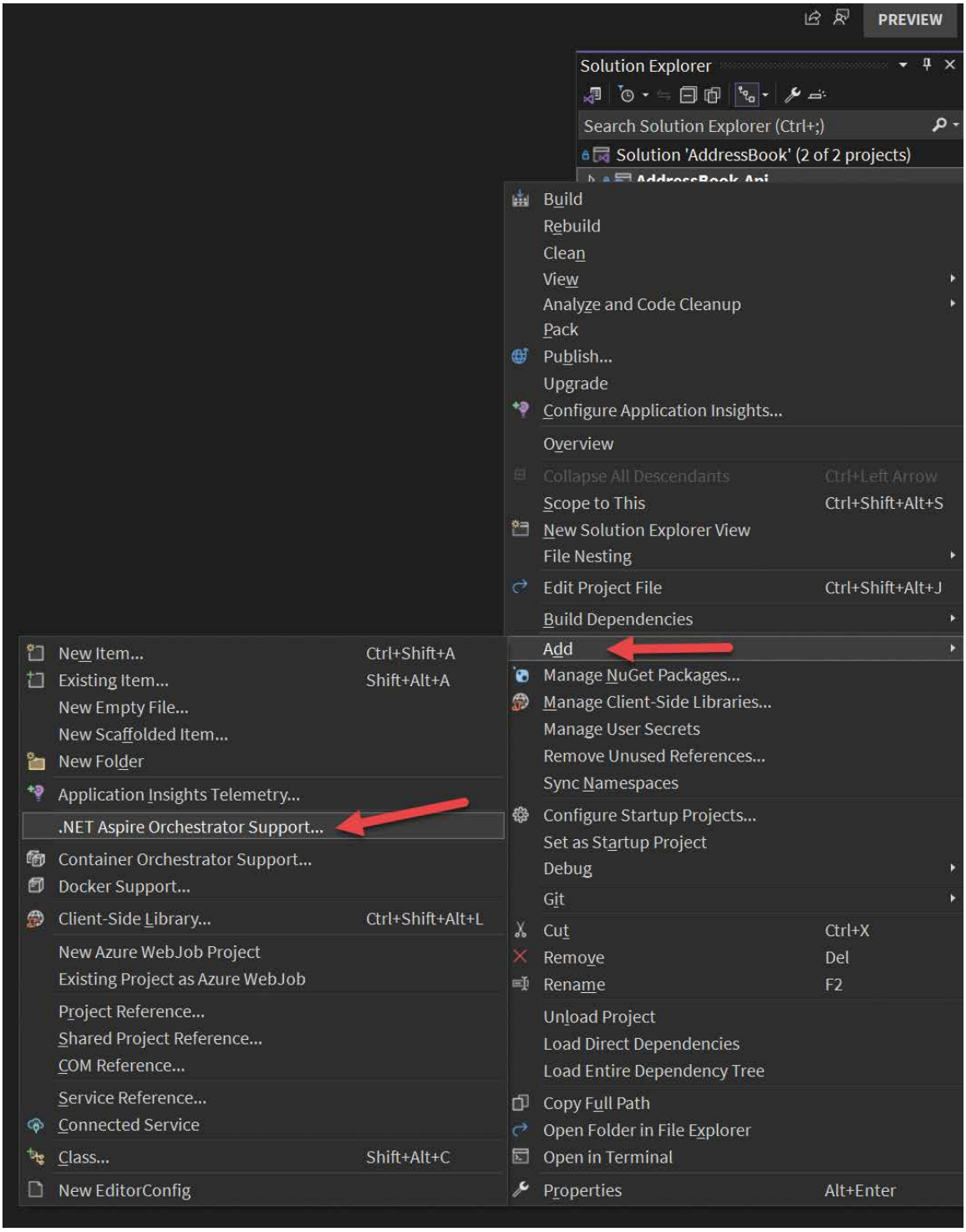


Figure 4: Adding .NET Aspire to an existing project

In both cases, you get two projects that are centered around .NET Aspire, as shown in **Figure 5**.

The **AppHost** project is both a dashboard of your running distributed app as well as where the orchestration is configured.

On the other hand, the **ServiceDefaults** project is a library project that creates some defaults that can be used by one or more of your services to configure commonly used services, such as Instrumentation, Metrics, HealthChecks, and others. It's simply a project you can reference to set up these different facilities for your projects, for example, when you call the service defaults in your projects (e.g., API or Blazor projects):

```
// Aspire Wiring
builder.AddServiceDefaults();
builder.AddRedisOutputCache("theCache");
builder.AddSqlServerDbContext<BookContext>("theDb");
```

Behind this small little **AddServiceDefaults** method is a lot of code. I encourage you to look at the code it generates, as it configures a lot of useful services that all your apps can use. A lot of these are used by the dashboard inside the AppHost. Let's take a look at the dashboard next.

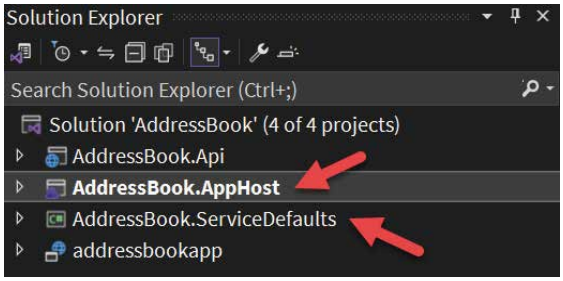


Figure 5: .NET Aspire projects

The Application Dashboard

When you run a .NET Aspire application, it launches a dashboard that you can use to monitor the applications in your project. For example, **Figure 6** shows the initial dashboard.

From this view, you can launch the endpoints, and view the environments and logs to see how each service is being launched. In addition, you can use the Monitoring options on the left to view traces and metrics (that were defined in the Service Defaults) to see how the application is doing in real time, as seen in **Figure 7**.

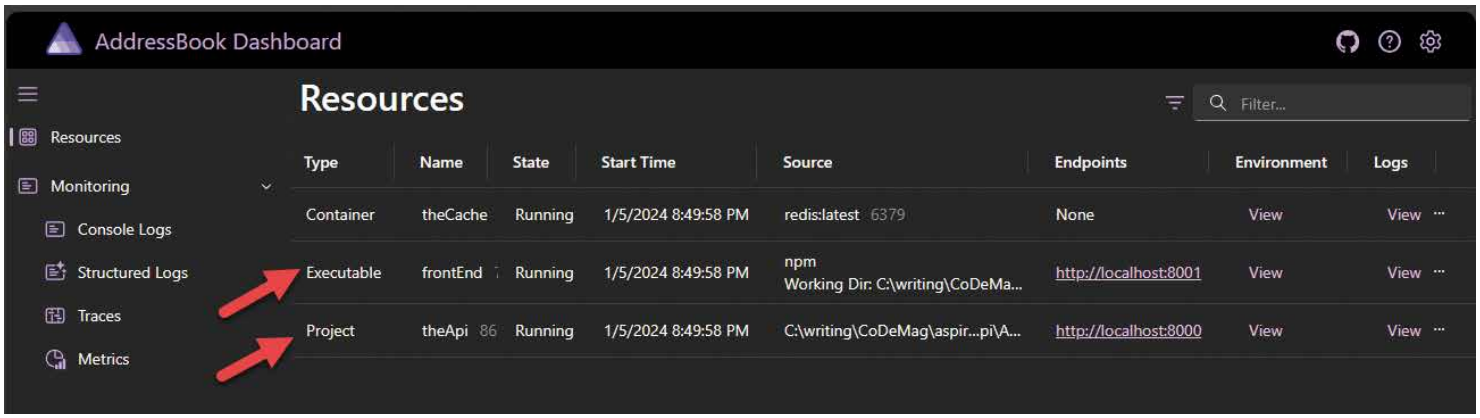


Figure 6: The dashboard

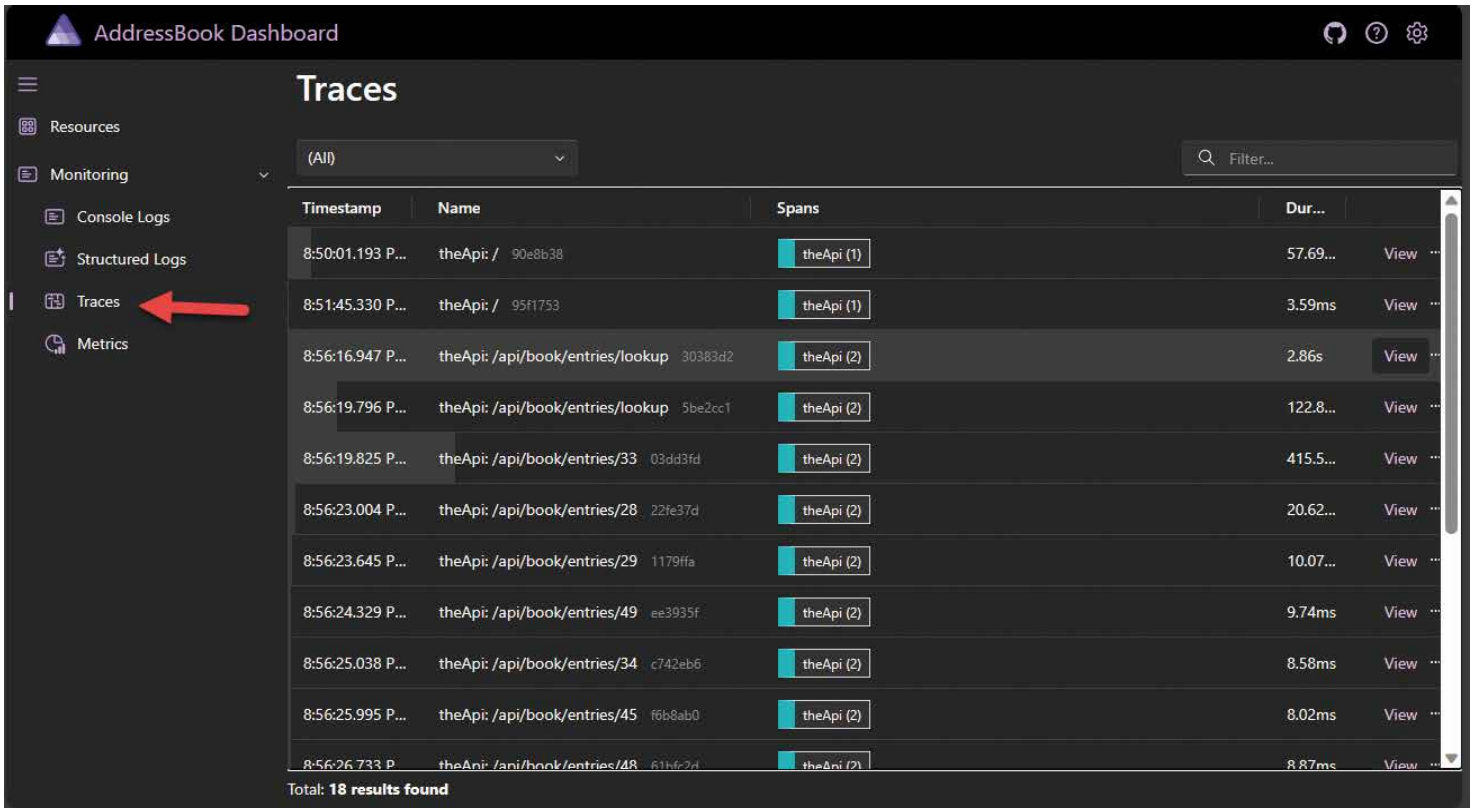


Figure 7: Tracing the distributed application

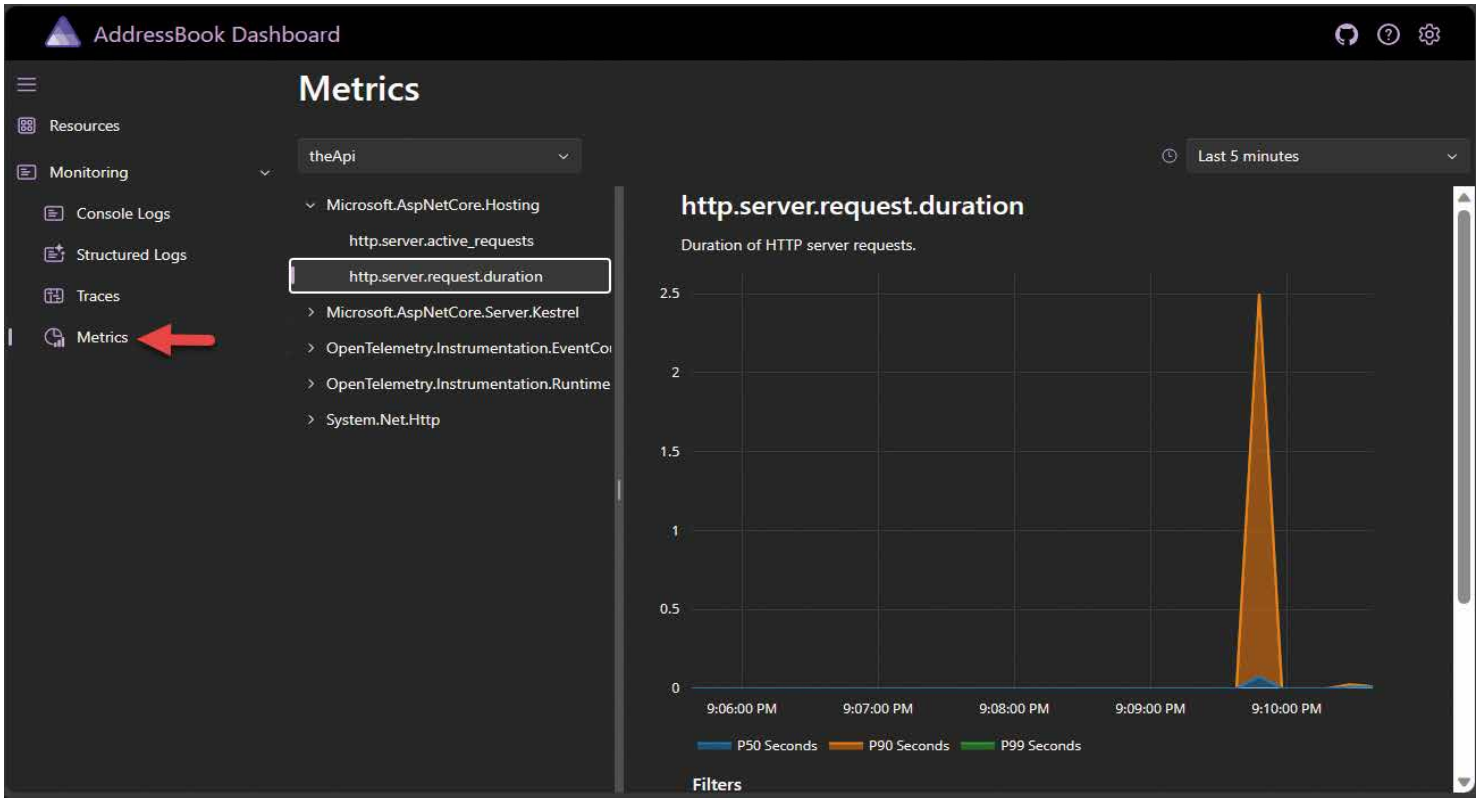


Figure 8: Metrics of your distributed application

You can even see performance metrics in **Figure 8**.

Now you've seen how the tooling and the dashboard work, but I think it's important to understand how .NET Aspire can manage each part of your distributed applications. Let's see that next.

Orchestration

The job of orchestration is to think of your application as a single slate of services. This means being able to deploy an entire set of services and dependencies as a single entity. When you think about a microservice implementation, one of the difficult things is to handle connecting the different services. To do this, Microsoft Aspire creates something called an **AppHost**. This app host is responsible for orchestrating your application. As you can see in **Figure 9**, the AppHost represents a context around which the services are housed.

Do not confuse the context of the **AppHost** as a container or wrapper that holds all of the components together, but instead as a conductor of an orchestra. It's responsible

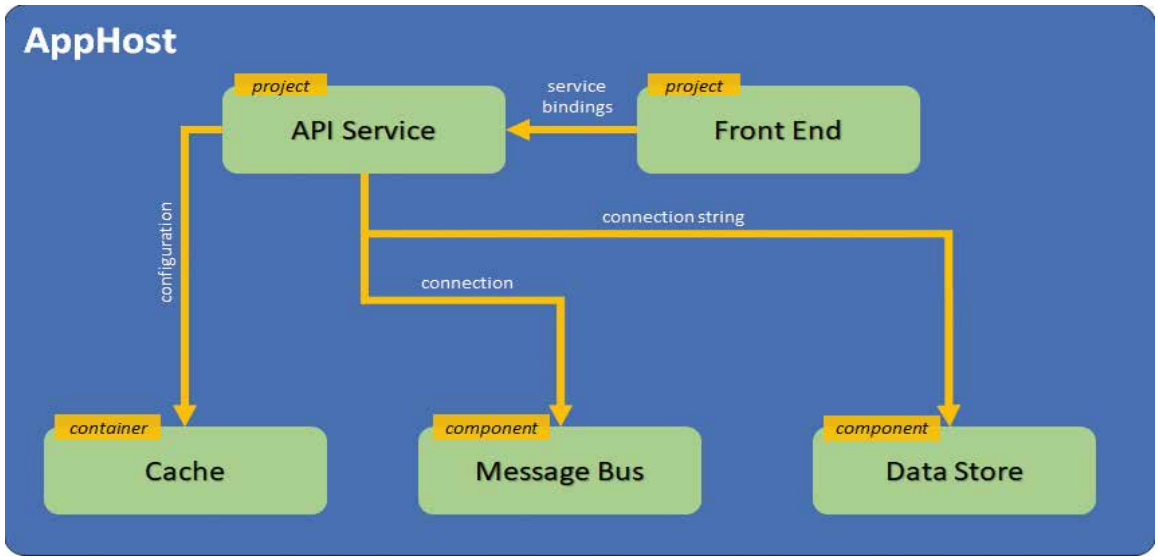


Figure 9: The AppHost's relationship to the other services

for starting, configuring, and connecting the difference services together.

How Orchestration Works

If you've used ASP.NET Core, you'd usually create an application by using **WebApplication.CreateBuilder()**. For Aspire, you can do this in a similar fashion:

```
// AppHost
var builder =
  DistributedApplication.CreateBuilder(args);
```

Instead of wiring up services and a pipeline, you add the parts of your application that you need. For example:

```
builder.AddRedisContainer("theCache");

builder.AddProject<AddressBook_Api>("theapi");
builder.AddProject<AddressBook_Blazor>("frontend");

builder
  .Build()
  .Run();
```

This is an additional project that's run to orchestrate the entire application. In this example, you're orchestrating the three components (or services) together. The types of components that Aspire supports is quite varied. This doesn't mean that you are limited to just those components, either. In the preview, there are a lot of supported components. Many of these include simplified hosting or Azure hosting, but Microsoft is working with teams across cloud providers (i.e., Google and AWS) to add components for those services too.

As of the preview, you can already use some of the most common components. A sampling of these can be seen in **Table 1**.

Because .NET Aspire can also be used to deploy to Azure, there are components specific to Azure resources. Some of these common Azure components can be seen in **Table 2.**

Now you know how to compose the distributed app by using separate components, but how do you handle connecting the apps?

Service Discovery

In the earlier example, you can just add a Redis container in setting up the builder. What does that mean? When this application is run, it deploys a standard Redis docker container and can connect it to your project. For example, when you add the API project, you could send it a reference to the Redis container:

```
// AppHost
builder.AddRedisContainer("theCache");

builder.AddProject<AddressBook_Api>("theApi")
  .WithReference(cache)
```

This allows the other applications to use the cache using similar calls in the project setup:

```
// API Project
builder.AddRedisOutputCache("theCache");
```

Component Type	Method to Add the Service
Docker container	AddDocker()
MongoDb server	AddMongoDb()
MySql server	AddMySql()
Postgres server	AddPostgres()
SQL server	AddSqlServer()
Redis	AddRedisContainer()
C# project	AddProject()
RabbitMQ	AddRabbitMQ()

Table 1: Common components

Component Type	Method to Add the Service	NuGet Package
Blob storage	AddAzureBlobService()	Aspire.Azure.Storage.Blobs
KeyVault	AddAzureKeyVaultSecrets()	Aspire.Microsoft.Azure.Cosmos
Table storage	AddAzureTableService()	Aspire.Azure.Security.KeyVault
Service bus	AddAzureServiceBus()	Aspire.Azure.Messaging.ServiceBus
CosmosDB	AddAzureCosmosDB()	Aspire.Azure.Data.Tables

Table 2: Azure components

Note that the cache uses the name that was defined when you added Redis. In this way, the connection to the Redis isn't needed in the API project. This is what is meant by Service Discovery. It works identically for Blazor apps, too. This means that you have to deal with shared configuration less often. For project types that don't have native support for Aspire, you can still coordinate them through environment variables:

```
builder.AddNpmApp("frontEnd",
                  "../addressbookapp",
                  "dev")
  .WithReference(api);
```

The call **With.Reference** shares the api service's URL to the app using environment variables. For example, this Vue app can use the environment variable (assuming you're going to run this as a server-side node project):

```
// https://vitejs.dev/config/
export default defineConfig({
  ...
  define: {
    "process.env.APP_URL":
      JSON.stringify(
        process.env["services__theApi__1"]
      )
  },
  ...
})
```

Where Are We?

To be sure, these are early days in the lifecycle of .NET Aspire. I'm impressed. Instead of always needing to drop down into multiple containers and maybe even Kubernetes, .NET now has an orchestration engine for its own distributed applications. I'm sincerely curious to see what this looks like when it ships.

Shawn Wildermuth

CODE

Source Code

The source code can be downloaded at https://github.com/shawnwildermuth/aspiringDotNet

Distributed Caching: Enhancing Scalability and Performance in ASP.NET 8 Core

In the realm of enterprise applications, a deep understanding of distributed caching intricacies and adherence to best practices are essential to building enterprise applications that are scalable and high performant. You can choose from the popular distributed caching frameworks, such as Memcache, NCache, Redis, and Hazelcast. In this article, I'll discuss distributed caching, its benefits and use cases, and then teach you how to implement it in ASP.NET Core. If you're to work with the code examples discussed in this article, you need the following installed in your system:

- Visual Studio 2022
- .NET 8.0
- ASP.NET 8.0 Runtime

If you don't already have Visual Studio 2022 installed on your computer, you can download it from here: https://visualstudio.microsoft.com/downloads/.

In this article, I'll examine distributed caching, its features and benefits, the challenges, and how you can implement distributed caching using NCache and MemCached.

Joydip Kanjilal
joydipkanjilal@yahoo.com

Joydip Kanjilal is an MVP (2007-2012), software architect, author, and speaker with more than 20 years of experience. He has more than 16 years of experience in Microsoft .NET and its related technologies. Joydip has authored eight books, more than 500 articles, and has reviewed more than a dozen books.

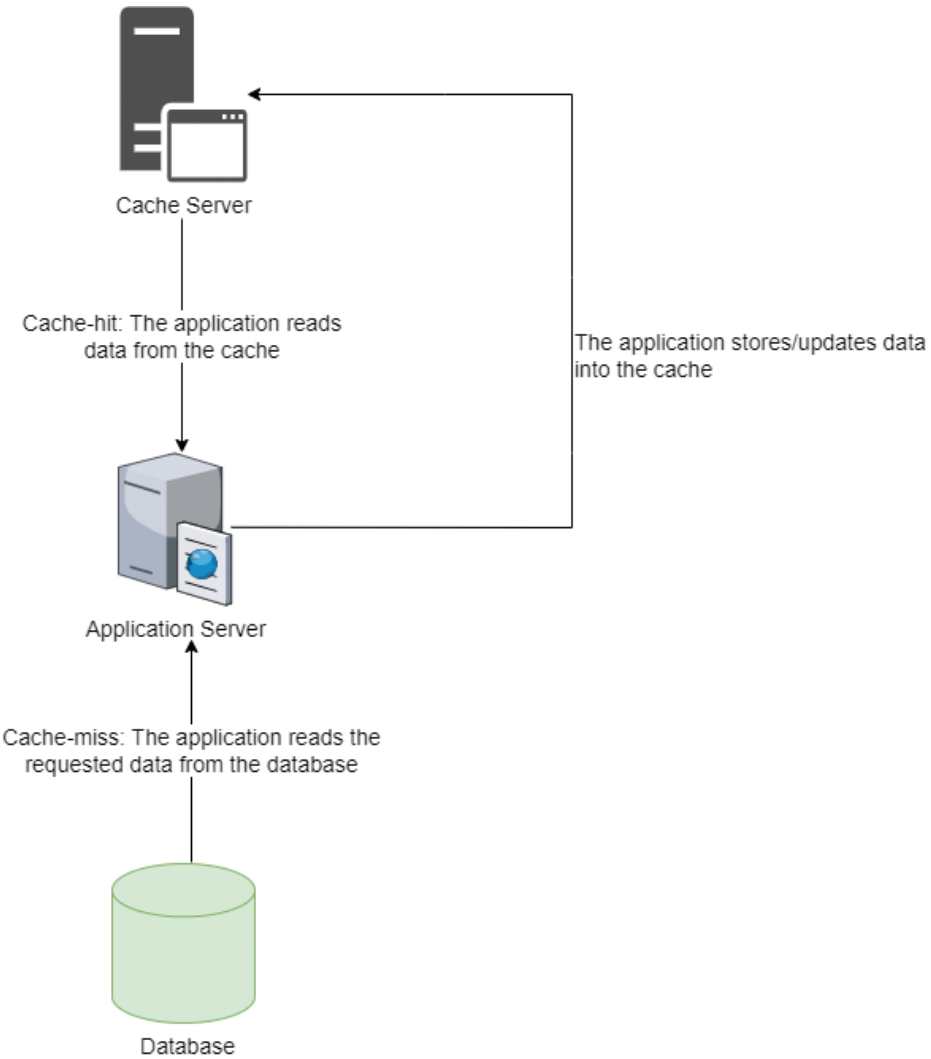


Figure 1: Caching at work

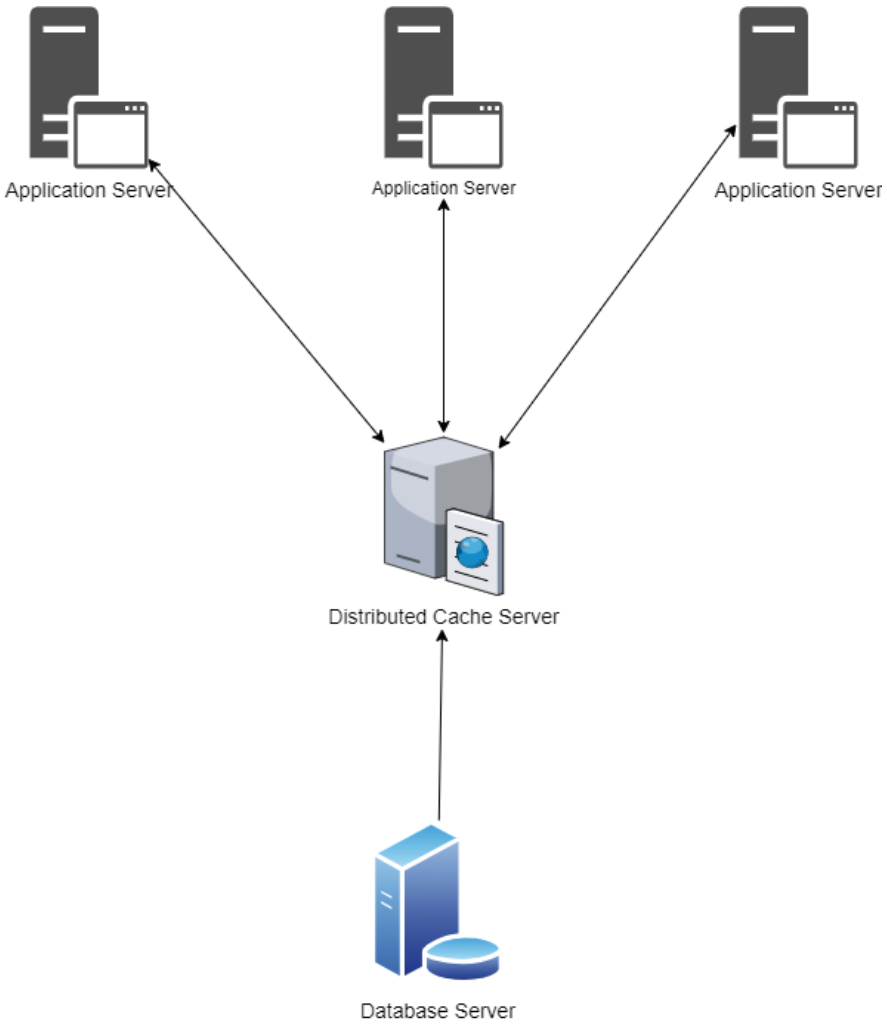


Figure 2: Distributed caching

What Is Caching? Why Is It Needed?

Caching is a proven technique used to collect frequently used data in a transient data store, known as a cache, for subsequent retrieval when requested by the application. This data store can reside in a transient storage such as memory, or in a permanent storage such as in a file, or even inside a database.

It should be noted that when a request for data is made, the caching system first attempts to determine whether the requested data already exists in the cache before attempting to retrieve it. If the data is available in the cache, you should be able to get it from there rather than accessing the data from the original source. A major benefit of this is that data can be retrieved from faster cache storages in a significantly shorter period of time instead of slower storage devices, such as remote databases or hard drives. (See **Figure 1**.)

Caching strategies need to consider trade-offs between cache size, cache eviction policies, and data consistency. Optimal performance requires a balance between cache utilization, data freshness, and data access patterns. There are several caching types, such as in-memory caching, distributed caching, and client-side caching. In case of in-memory caching, the data is stored in a transient storage, i.e., the memory for faster access to the cached data for all subsequent requests. In distributed caching, data is stored on multiple nodes, i.e., the cached data is spread across several systems. In client-side caching, the cached data is stored on the client computer, such as on the web browser.

An Overview of Distributed Caching

An extension of caching called distributed caching is the process of distributing the cache across several servers or machines. In a distributed caching environment, the cache servers are spread across multiple nodes or machines, allowing for scalability and improved performance. The main purpose of distributed caching is to store frequently accessed data in memory, rather than fetching it from the hard drive or a remote database. This improves the speed of data access because retrieving data from memory is faster than disk access. (See **Figure 2**.)

Distributed cache systems combine the memory of several networked computers into a single memory store for fast data access. This explains why distributed caches aren't constrained by the limit of available memory of a single computer or hardware component. They may go beyond these boundaries by connecting numerous computers,

forming a distributed architecture or distributed cluster, enabling increased processing power and storage capacity. (See **Figure 3**.)

Note that a distributed cache is based on DHT, an acronym for Distributed Hash Table. In this DHT, the data is stored as key-value pairs. Each participating node can retrieve the value using a key. DHT allows distributed caches to be scaled dynamically by continuously managing additions, deletions, and node failures.

Some typical examples of widely used distributed caching systems include NCache, Redis, and Memcached. These systems leverage distributed cache mechanisms and offer APIs for faster data storage and retrieval.

Benefits

Distributed caching offers several benefits, including:

- **Scalability:** With a surge in traffic for an application, you can add additional cache servers to the distributed cache system without disrupting existing operations.
- **Reduced latency:** With distributed caching, cached data is usually stored in an in-memory data store, enabling blazing fast access and reducing latency caused by fetching data from the disk or any remote services.
- **Performance:** A distributed caching system reduces the latency associated with fetching frequently accessed data from disks or remote services by caching frequently accessed closer to the application.
- **Reliability:** Distributed caching promotes reliability by storing data across multiple servers. If a cache server fails, the remaining servers can still serve the cached data, ensuring data availability.
- **High availability:** Distributed caching systems can easily handle increased traffic by adding servers, enabling the systems to scale dynamically without disrupting the existing operations, hence ensuring high availability of the cached data.
- **Resilience:** Caching provides enhanced resilience against temporary traffic spikes or database outages. The cache data can be accessed quickly, providing a smoother user experience.

Building a Simple Distributed Application in ASP.NET Core

It's time for writing some code. Let's now examine how to build a simple ASP.NET Core 7 Web API application using GraphQL.

Create a New ASP.NET Core 8 Project in Visual Studio 2022

You can create a project in Visual Studio 2022 in several ways. When you launch Visual Studio 2022, you'll see the Start window. You can choose "Continue without code" to launch the main screen of the Visual Studio 2022 IDE.

To create a new ASP.NET Core 8 Project in Visual Studio 2022:

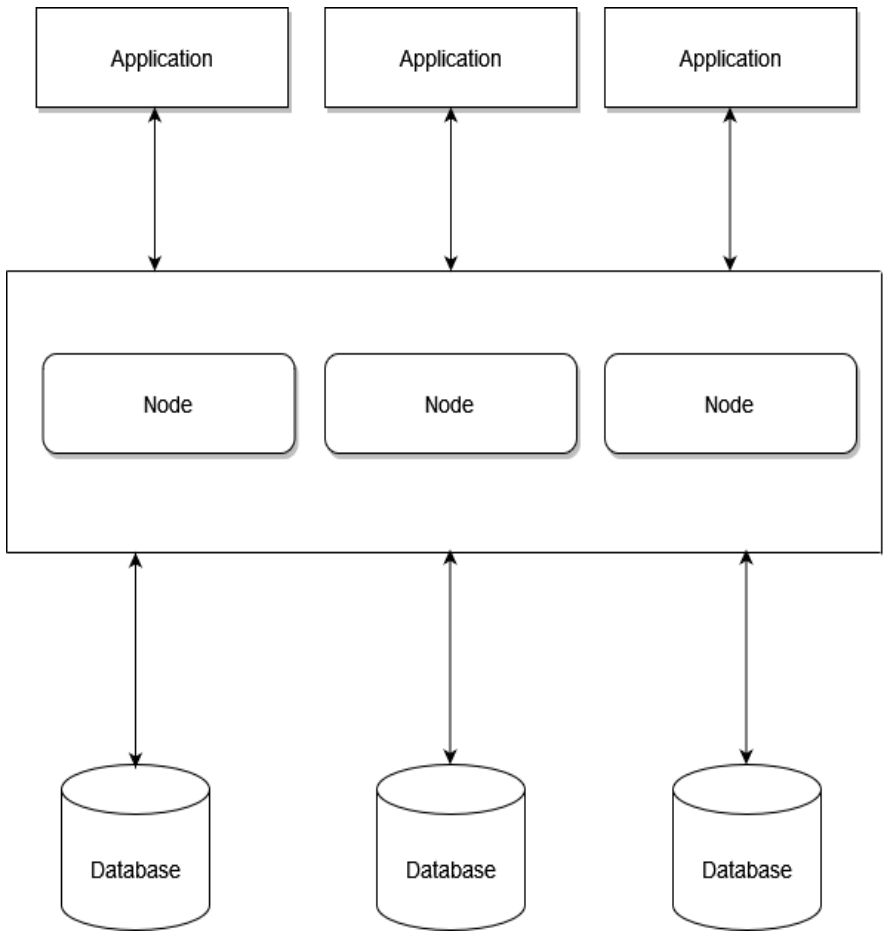


Figure 3: Distributed caching with multiple nodes

1. Start the Visual Studio 2022 IDE.
2. In the "Create a new project" window, select "ASP.NET Core Web API" and click Next to move on.
3. Specify the project name as DistributedCachingDemo and the path where it should be created in the "Configure your new project" window. If you want the solution file and project to be created in the same directory, you can optionally check the "Place solution and project in the same directory" checkbox. Click Next to move on.
4. In the next screen, specify the target framework and authentication type as well. Ensure that the "Configure for HTTPS," "Enable Docker Support," and the "Enable OpenAPI support" checkboxes are unchecked because you won't use any of these in this example.
5. Because you won't be using minimal APIs in this example, remember to leave the **Use controllers (uncheck to use minimal APIs)** checkbox checked.
6. Click Create to complete the process.

I'll use this project in the subsequent sections of this article.

Distributed Caching Use Cases

There are several use cases of distributed caching across different domains and industries. Here are a few common scenarios where distributed caching can help:

- **High-traffic websites:** Distributed caching can be used to store frequently accessed web pages, HTML fragments, or static content. By caching these elements on distributed cache servers closer to the users, they can be served quickly and reduce the load on back-end servers, improving website performance and scalability.
- **Database caching:** Distributed caching can be employed to cache frequently accessed database queries or result sets. By storing this information in memory, you can quickly respond to requests avoiding the need for possibly expensive database queries or redundant database hits.
- **Microservices architecture:** In a typical microservices architecture, there can be many different services communicating with each other, which can increase latency due to network roundtrips. By using distributed caching, frequently accessed data or computed results can be cached, minimizing the need for repeated remote calls and reducing latency.
- **Content delivery networks (CDNs):** CDNs leverage distributed caching to persist and also serve static content that includes images, JavaScript, and CSS files. By caching content in multiple edge servers located geographically closer to users, CDNs can deliver content faster, reducing latency and improving the user experience.
- **Real-time analytics:** In data-intensive applications, distributed caching can be used to store preprocessed or computed data that is frequently accessed for real-time analytics. By caching this data, applications can retrieve insights quickly, avoiding the latency associated with reprocessing vast amounts of raw data.
- **Geographically distributed applications:** In globally distributed applications, distributed caching can be used to store and serve frequently accessed data closer to users in different regions. This decreases the amount of time that's otherwise needed to access data across long network distances, improving performance and reducing latency.

Distributed Caching Best Practices

Implementing distributed caching requires adherence to several best practices:

- **Cache invalidation:** Decide on a cache invalidation strategy ensuring that the cached data remains up to date. Consider options like time-to-live (TTL) based expiration, explicit cache invalidation, or event-driven invalidation mechanisms based on data changes or updates.
- **Eviction policies:** Choose appropriate eviction policies for your distributed cache. Cache eviction policies help manage cache data and determine how and when they're removed from the cache. There are three types of eviction policies: least recently used (LRU), least frequently used (LFU), and fixed size evictions.
- **Cache sizing and capacity planning:** Estimate the cache size and capacity requirements based on the volume of data and expected workload. Consider factors like memory availability, cache server capabilities, and future growth to ensure that the cache can handle the anticipated load.
- **Proper key design:** Design cache keys with care. Use meaningful and unique keys that are easy to generate and parse. Avoid overusing complex objects or dynamic data as keys, as it can negatively impact cache performance.
- **Monitor and tune performance:** Regularly monitor and measure the performance of your distributed cache to identify bottlenecks, cache hit ratios, and review other performance indicators. Adjust cache configuration, eviction policies, or cache server capacity, if necessary, to optimize performance.
- **Implement failover and replication:** Ensure high availability and fault tolerance by implementing failover mechanisms and data replication across cache servers in case of failures. This helps maintain consistency and ensures uninterrupted access to cached data.
- **Security considerations:** Pay attention to security aspects, such as access control, authentication, and encryption, to protect sensitive data stored in the distributed cache.
- **Appropriate data selection:** Not all data is suitable for caching. Identify data that's frequently read but infrequently updated, as this type of data benefits most from caching. Avoid caching data that changes very frequently.
- **Cache invalidation strategy:** Implement a robust cache invalidation strategy to ensure data consistency. This could be time-based (e.g., TTL—time to live), event-driven, or a combination of both. When a change occurs in the underlying data, it's recommended that you update the cache to ensure that your cached data is always in sync with the data residing in the data store.
- **Data partitioning and sharding:** Distribute data across different cache servers to balance the load and reduce risks of a single point of failure. Use

sharding techniques to partition data effectively based on usage patterns or other relevant criteria.
- **Handling cache misses:** Design your system to handle cache misses gracefully. When the data requested by the application isn't available in the cache, it's known as a cache miss. Optimizing this fallback mechanism is essential for maintaining performance.
- **Load balancing:** Employ load balancing to distribute requests evenly across cache servers, thus ensuring use of resources.
- **Scalability:** Ensure that your caching solution can scale horizontally. As the demand increases, you should be able to add more cache servers to the system without significant changes to the existing infrastructure.

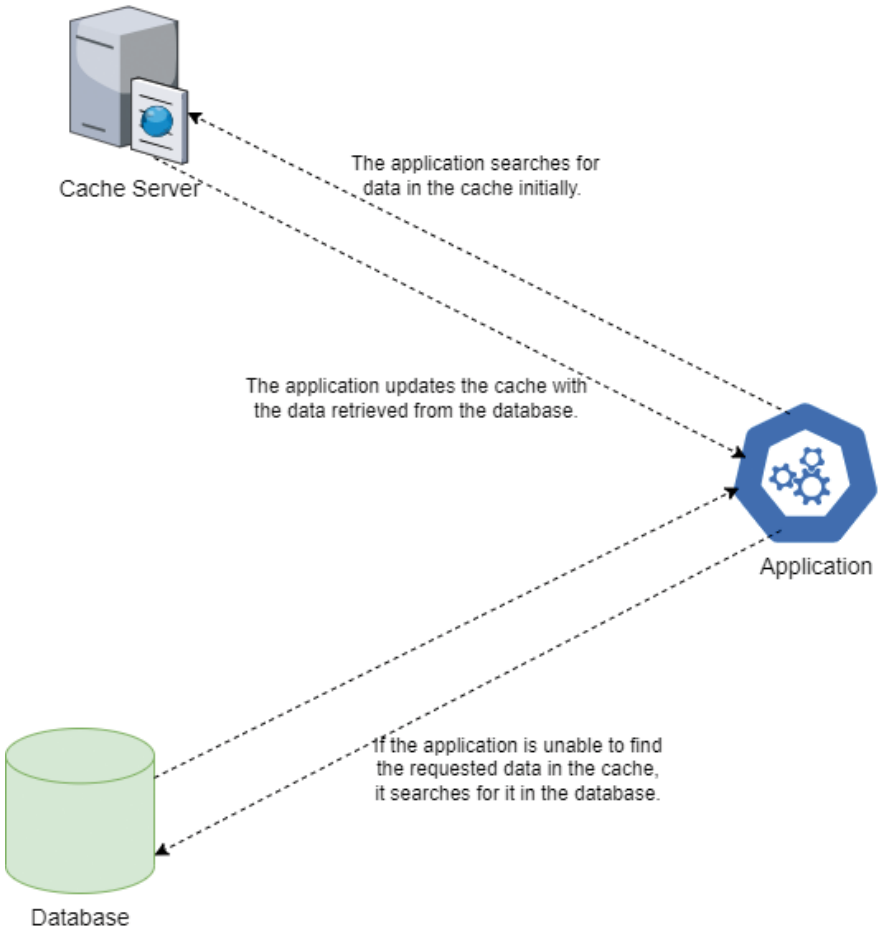


Figure 4: The cache-aside pattern

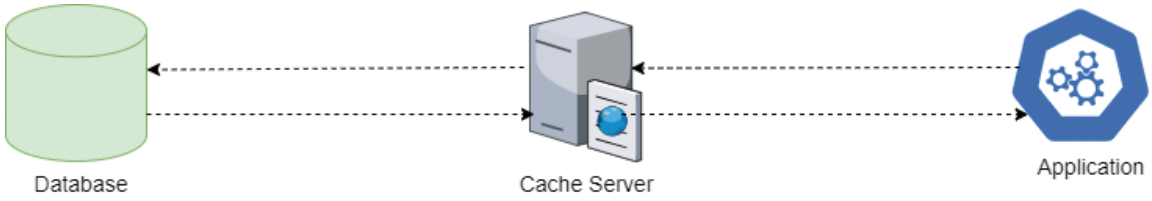


Figure 5: The read-through pattern

Figure 6: Data is served from the cache for subsequent requests

- **Data security:** You should establish the necessary security measures by implementing security measures that include encryption of data in transit and at rest, proper access controls, and authentication mechanisms, etc.
- **Monitoring and analytics:** You should monitor the performance of your cache system on a regular basis. In this regard, you can take advantage of metrics such as cache hit, miss rates, and load patterns to optimize your caching strategy.
- **Data synchronization:** Ensure that the cache is synchronized across all nodes in distributed environments. To maintain data integrity, you can take advantage of techniques such as distributed locking or atomic operations.
- **Selecting the right tool:** Selection of the right caching solution that satisfies the requirements of your application is extremely important. Redis, Memcached, and Hazelcast are popular tools with varying features and capabilities.
- **Avoid cache stampede:** Implement strategies like staggered TTLs, pre-computation, or using a probabilistic early expiration to avoid a cache stampede, where multiple instances try to compute and store the same data in the cache simultaneously.

Distributed Caching Patterns

Cache patterns are an approach to application design that emphasizes caching techniques to improve scalability, performance, and responsiveness. These patterns provide valuable insights and suggestions for implementing caching techniques. Typically, these patterns are used with distributed caching systems that include NCache or Memcached. You can take advantage of these patterns to reduce latency and improve your application's workload processing capabilities.

One such pattern is called **cache aside (**see **Figure 4**). The cache aside pattern adopts a lazy loading approach and is the most used caching strategy. Here's how this pattern works:

- When a request arrives, the application searches for the data in the cache.
- If the data is available, the application returns the cached data. This phenomenon is also known as a cache-hit.
- On the other hand, if the data that has been requested is unavailable in the cache, the application retrieves data from the database and then populates the cache with this data.

There are a few benefits of this approach:

- The cache contains only the data the application has requested, thus ensuring that the cache size is optimal and cost-effective.
- This strategy is simple and can provide immediate performance benefits.

The downsides to this approach include:

- Because the data is loaded into the cache only in the event of a cache miss, the initial response time increases because of the additional roundtrips required to the database and the cache.
- Using this approach, the data is stored directly in the database, which may result in discrepancies between the database and the cache.

Another pattern is called **read through** (see **Figure 5**). In this pattern, the application queries the cache first. The cache then interacts with the underlying database on a lazy load basis. If the data requested by the application is unavailable in the cache, it's called a cache miss. In this case, the application retrieves the requested data from the data store (i.e., a database or a service, etc.), stores this data into the cache, and then returns it.

The cached copy of the same data is then served for all subsequent requests for the same piece of data.

Another pattern is **write through** (see **Figure 7)**: This pattern is similar to read through, but instead of read-

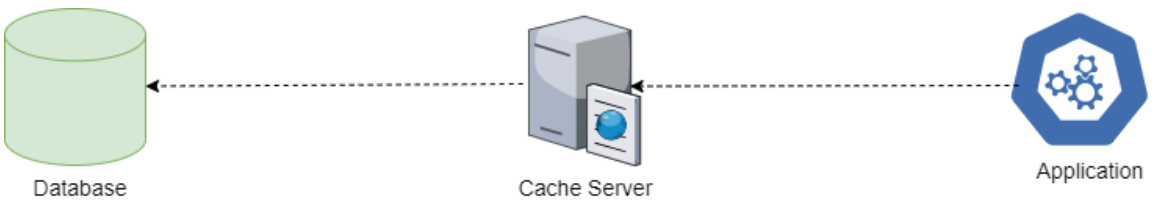


Figure 7: The write-through pattern

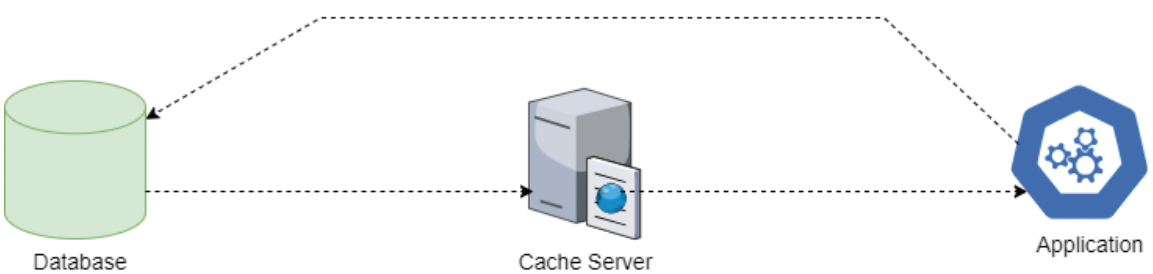


Figure 8: The write-around pattern

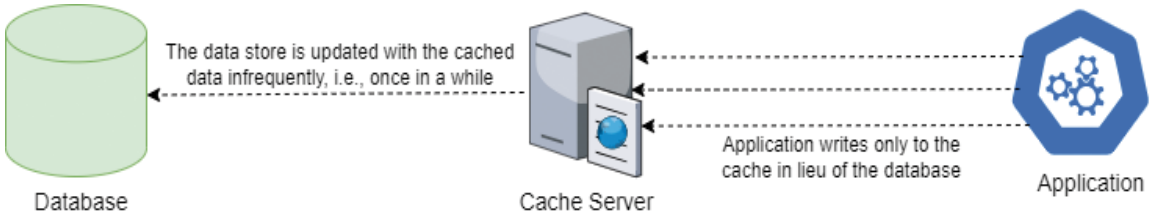


Figure 9: The write-behind or write-back pattern

ing data from the back-end system, it writes data to the cache. In other words, when using a write through cache, the order in which the cache is populated is reversed. After a cache miss, the cache is not lazy loaded. Instead, it's proactively updated immediately following an update to the database. Using this approach makes sense when writing operations are frequent and application performance is paramount.

Another pattern is **write around** (see **Figure 8**): In the write around approach, the data is written directly to the data store without storing it in the cache. When the application performs a read operation, the data is placed in the cache. This strategy is a good choice if your application doesn't need to reread frequently written data.

Another pattern is **write behind** or **write back** (see **Figure 9**): In the write back approach, data is written directly to the cache alone instead of the data store. The data store is updated with this cached data after specific intervals of time, or based on certain pre-defined conditions. Although this approach is beneficial in write-intensive applications due to low latency and high throughput capabilities, there are risks of data loss in the event of a system crash etc.

Cache Eviction Strategies

Cache eviction involves the removal of cache entries or data from a cache based on specific conditions or rules being satisfied. In distributed cache systems, cache eviction algorithms are used to provide space for new data and prevent the cache from going beyond its capacity. There are various cache eviction strategies available for removal of cache entries. Here are the key cache eviction strategies you can explore:

- **Least recently used (LRU):** Per this strategy, the cache item that's been accessed the least recently is removed from the cache when the cache reaches its maximum capacity. Note that this approach assumes entries that have been the least used will not be used again in the near future.
- **Most recently used (MRU):** Cache entries that have been accessed most recently are evicted when the cache is full. As per this assumption, entries that have been used more recently are less likely to be used again in the near future.
- **Least frequently used (LFU):** This policy stipulates that when the maximum cache capacity is reached, the cache element accessed least frequently within a certain period is removed. It asserts that entries used less often are unlikely to be accessed in the future.
- **Time-to-live (TTL):** This policy defines the duration for which a cache entry should remain valid in the cache. A cache entry is removed when its time-to-live (TTL) expires. You can employ this method when the data in your application has a short duration, like session data or refreshed data. It can prove beneficial in scenarios where your application relies on data that has a short lifespan such as session data or data that undergoes frequent updates.

Distributed Caching Challenges

Although there are benefits, distributed caching poses certain challenges as well:

- **Data consistency:** The main challenge with distributed caching is maintaining data consistency. Caches may have different data versions or may experience replication delays, leading to data inconsistency.
- **Cache invalidation:** Implementing proper cache invalidation and synchronization mechanisms or using eventual consistency models can help mitigate this challenge. Invalidating caches can be challenging, especially when data is frequently changing or interdependent. Ensuring that the cache is refreshed, i.e., updated when underlying data changes, or employing strategies, such as cache expiration based on time, can help maintain cache validity.
- **Cache coherence:** Cache coherence refers to ensuring that all cache servers have consistent data. Achieving cache coherence can be challenging, especially in scenarios where data is updated frequently or when multiple cache servers are involved. Employing distributed cache solutions that provide strong consistency guarantees or using cache coherence protocols can help address this challenge.
- **Cache warm-up:** When a cache is empty or after a cache server restarts, it takes time to populate the cache with frequently accessed data. This initial period can result in cache misses and increased latency until the cache is warmed up. Proper cache warm-up strategies, such as preloading commonly accessed data or using warm-up scripts, can help minimize this issue.
- **Cache eviction strategies:** Since data access patterns may differ across the nodes, it is quite challenging to implement cache eviction strategies in a distributed cache environment. Careful consideration should be given to selecting an eviction strategy that aligns with your application's data access patterns and requirements.
- **Scaling:** As the application load and data volume increase, scaling the distributed cache can become a challenge. Proper planning and architecture should be in place to ensure that the cache can handle the added load effectively.

Using NCache in ASP.NET Core

To work with NCache in ASP.NET Core, install the following NuGet package:

NCache.Microsoft.Extensions.Caching.OpenSource.

- **Cache synchronization:** In multi-level caching systems, where multiple caches, like local cache and distributed cache, are used, ensuring proper synchronization and consistency across different cache layers can be challenging. You can prevent inconsistencies in your data by implementing the right synchronization strategies.
- **Network overhead:** Distributed caching systems require communication and synchronization between cache servers, which introduces network overhead. High network latency or limited bandwidth can affect cache performance and overall application responsiveness.
- **Complexity and maintenance:** For the cache infrastructure to operate smoothly, it must be properly maintained, monitored, and troubleshot with proper expertise and resources. Implementation of a distributed caching system, in the application architecture introduces increased complexity in terms of implementation, management, and monitoring.
- **Application compatibility:** Some applications may not be designed to work seamlessly with distributed caching or may have dependencies that don't support caching effectively. Evaluate and modify the application to ensure compatibility with the distributed caching approach.

Distributed Caching in ASP.NET Core

Support for distributed caching is in-built in ASP.NET Core using the IDistributedCache interface. The IDistributedCache interface helps you to plug in any thirty-party caching frameworks. **Listing 1** illustrates how the IDistributedCache interface looks.

As evident from the source code of the IDistributedCache interface, its Get method returns byte[]. Nevertheless, the framework provides extension methods for working with string objects. In addition, you can implement custom extension methods to make it work with other data types.

Enabling Distributed Caching in ASP.NET Core

You can enable distributed caching in your ASP.NET Core application by adding the following code snippet in the Program.cs file:

```
builder.Services.AddDistributedMemoryCache();
```

You can now use the IDistributedCache interface to implement distributed caching in your ASP.NET Core application.

Implementing Distributed Caching Using NCache

NCache is a fast, open-source, cross-platform, distributed, in-memory caching framework. It's adept at enhancing the scalability and performance of your applications by caching frequently accessed data instead of storing to and retrieving the data and from the database. There are a variety of caching features in NCache, including object caching, SQL caching, full-text search, and distributed caching, and there are use cases such as web sessions, ASP.NET output caching, and enterprise caching. NCache supports Docker and Kubernetes and various caching topologies, including replicated, partitioned, and client-side caching.

Configuring NCache as an IDistributedCache Provider

To work with distributed caching using NCache, invoke the AddNCacheDistributedCache method in the Program.cs file to register NCache with the IoC container. Note that the AddNCacheDistributedCache() method in NCache is an extension method of the AddDistributedCache() method pertaining to ASP.NET Core.

```
builder.Services.AddNCacheDistributedCache
(configuration =>
{
    configuration.CacheName = "demoCache";
    configuration.EnableLogs = true;
    configuration.ExceptionsEnabled = true;
});
```

If you'd like to work with multiple caches, use the following code snippet to configure them in the Program.cs file:

```
builder.Services.
AddNCacheDistributedCacheProvider
(options =>
{
    options.CacheConfigurations =
    new NCacheConfiguration[] {
    new NCacheConfiguration() {
    CacheName = "myFirstCache",
    EnableLogs = true,
    ExceptionsEnabled = true
    },
    new NCacheConfiguration(){
    CacheName = "mySecondCache",
    EnableLogs = true,
    ExceptionsEnabled = true }
    };
});
```

Setting Up NCache in ASP.NET Core

To setup NCache in your system, follow these steps:

1. Download the NCache installer to your computer.
2. Open a command prompt window as administrator and run the msiexec.exe utility to install NCache.

Listing 1: The IDistributedCache Interface

```
public interface IDistributedCache
{
    byte[] Get(string key);
    Task<byte[]>
    GetAsync(string key);
    void Set(string key,
    byte[] value,
    DistributedCacheEntryOptions
    options);
    Task SetAsync(string key,
    byte[] value,
    DistributedCacheEntryOptions
    options);
    void Refresh(string key);
    Task RefreshAsync(string key);
    void Remove(string key);
    Task RemoveAsync(string key);
}
```

Listing 2: Store/Retrieve data using the IDistributedCache interface

```
app.MapGet("/Test",
async (IDistributedCache cache,
IHttpClientFactory httpClientFactory) =>
{
    const string cacheKey = "Test";
    List<Author>? authors = null;
    var cachedData =
    await cache.GetStringAsync(cacheKey);

    if (!string.IsNullOrEmpty(cachedData))
    {
        authors =
        System.Text.Json.JsonSerializer.Deserialize
         <List<Author>>(cachedData);
    }

    else
    {
        Author user = new Author()
        {
            AuthorId = 1,
            FirstName = "Joydip",
            LastName = "Kanjilal",
            IsActive = true
        };

        authors = new List<Author>();
        authors.Add(user);

        await cache.SetStringAsync
        (cacheKey,
        System.Text.Json.JsonSerializer.Serialize
        (authors),
        new DistributedCacheEntryOptions
        {
            AbsoluteExpirationRelativeToNow
            = new TimeSpan(0, 0, seconds: 60)
        });
    }

    return authors;
});
```

Using Distributed Caching in SQL Server

To work with distributed caching using SQL Server, you must have the Microsoft.Extensions.Caching.SqlServer NuGet package installed.

```
msiexec /i "<<Specify the path of the NCache installer
 in your computer>>"\ncache.ent.net.x64.msi"
```

1. Once the installer is launched, you can observe three different installation types: Cache Server, Developer/QA, and Remote Client.
2. Select CacheServer and click Next to move on.
3. Specify the license key for the version of NCache you're installing in your system.
4. If you don't have a valid license key with you, click on the "Get Installation Key" button to get an installation key.
5. You're prompted to enter your name, your organization's name, and your email address.
6. Specify the location in your computer where NCache should be installed. Click on Next to move on to the next step.
7. Select the IP address to bind the NCache server. You can stick to the default here.
8. Specify the account to run NCache in your system. Select the Local System account and click Next to move on.

That's all you have to do to install NCache on your computer.

Store and Retrieve an Object Using NCache

Consider the following class:

```
public class Author
{
    public int AuthorId { get; set; }
    public string? FirstName { get; set; }
    public string? LastName { get; set; }
    public bool IsActive { get; set; }
}
```

Listing 2 illustrates how you can leverage NCache to store and retrieve an object:

Note that for the sake of simplicity, I've used a minimal API only to demonstrate how NCache works. I've added only one record here for the sake of brevity. When you execute the application and browse the /test endpoint, the author record is displayed in the web browser. When this endpoint is hit the first time, the author record is stored in the cache. For all subsequent requests to this endpoint, the data is fetched from the cache based on the expiration time set in the configuration. In this example, the data resides in the cache for 60 seconds.

The complete code listing of the Program.cs file is given in **Listing 3** for your reference.

Implementing Distributed Caching using SQL Server

You can use SQL Server as a cache store as well. To get started on this, create the database table to store cached data in SQL Server.

You can take advantage of the sql-cache by creating a command to create a database table for storing cached data. If the sql-cache tool isn't installed on your computer, you can install it using the following command at the Developer Command Prompt window:

```
dotnet tool install --global dotnet-sql-cache
```

You can use the following command to create a database table for storing cached data in a SQL Server:

```
dotnet sql-cache create "Data Source=.;
Initial Catalog=MyCache;
Integrated Security=True;" dbo MyCacheData
```

To enable a distributed SQL Server cache service in an ASP.NET Core application, write the following piece of code in the Program.cs file:

```
builder.Services.
AddDistributedSqlServerCache(
options =>
{
    options.ConnectionString =
    builder.Configuration.
    GetConnectionString
    ("MyDBConnectionString");
    options.SchemaName = "dbo";
    options.TableName
```

Listing 3: The Program.cs file

```
using Alachisoft.NCache.Caching.Distributed;
using Microsoft.Extensions.Caching.Distributed;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddNCacheDistributedCache
(configuration =>
{
    configuration.CacheName = "demoCache";
    configuration.EnableLogs = true;
    configuration.ExceptionsEnabled = true;
});

builder.Services.AddHttpClient();
var app = builder.Build();

app.MapGet("/Test", async
(IDistributedCache cache,
IHttpClientFactory httpClientFactory) =>
{
    const string cacheKey = "Test";
    List<Author>? authors = null;
    var cachedData =
    await cache.GetStringAsync(cacheKey);

    if (!string.IsNullOrEmpty(cachedData))
    {
        authors = System.Text.Json.JsonSerializer.
        Deserialize<List<Author>>(cachedData);
    }

    else
    {
        Author user = new Author()
        {
            AuthorId = 1, FirstName = "Joydip",
            LastName = "Kanjilal", IsActive = true
        };

        authors = new List<Author>();
        authors.Add(user);

        await cache.SetStringAsync
        (cacheKey, System.Text.Json.
        JsonSerializer.
        Serialize(authors),
        new DistributedCacheEntryOptions
        {
            AbsoluteExpirationRelativeToNow
            = new TimeSpan(0, 0, seconds: 60)
        });
    }

    return authors;
});

app.Run();

class Author
{
    public int AuthorId { get; set; }
    public string? FirstName { get; set; }
    public string? LastName { get; set; }
    public bool IsActive { get; set; }
}
```

```
    = "MyCacheData";
});
```

Now refer to the DistributedCachingDemo ASP.NET Core project you created earlier. Create a new class called Product in a file named Product.cs and replace the default generated code with the following code:

```
public class Product
{
    public Guid Id
    { get; set; }
    public string Name
    { get; set; }
    public int Quantity
    { get; set; }
    public decimal Price
    { get; set; }
}
```

Create a new API controller named MySQLServerCacheDemoController in a file having identical name with a .cs extension and write the code found in **Listing 4** there.

A Real-World Use Case: Implementing Distributed Caching Using MemCached

Memcached is an open-source, high-performance, distributed, in-memory caching system that aims to improve the scalability and performance of web applications by lessening the need to retrieve data from databases or other external resources. Memcached is mainly used to reduce the load on database servers and improve the overall efficiency of applications by storing the results of database queries and other computationally intensive results in memory. Data for subsequent redundant queries can be served from the cache by storing information in the cache, avoiding the database access time and computational overhead. This significantly improves performance and scalability compared to accessing the original data sources for every request.

> Avoiding database access time and computational overhead significantly improves performance and scalability.

In this section, you'll implement a simple OrderProcessing application. To keep things simple, this application only displays one or more order records. The source code of this application is comprised of the following classes and interfaces:

- Order class
- IOrderRepository interface
- OrderRepository class
- OrdersController class

Listing 4: The MySQLServerCacheDemoController class

```
[Route("api/[controller]")]
[ApiController]
public class MySQLServerCacheDemoController :
ControllerBase
{
    private readonly
    IDistributedCache _distributedCache;
    private readonly
    string key = "MySQLServerCacheDemo";
    public MySQLServerCacheDemoController
    (IDistributedCache distributedCache)
    {
        _distributedCache = distributedCache;
    }

    [HttpGet("GetProduct")]
    public async Task<Product> GetProduct()
    {
        byte[] cachedObject =
        await _distributedCache.
        GetAsync(key);

        if (cachedObject != null)
        {
            var cachedObjectAsJson
            = System.Text.Encoding.UTF8.
            GetString(cachedObject);
            var product = JsonSerializer.
            Deserialize<Product>
            (cachedObjectAsJson);
            if (product != null)
            {
                return product;
            }
        }

        var result = new Product()
        {
            Id = Guid.NewGuid(),
            Name = "HP Envy Laptop",
            Price = 5000,
            Quantity = 1
        };

        byte[] serializedObject
        = JsonSerializer.
        SerializeToUtf8Bytes(result);
        var options =
        new DistributedCacheEntryOptions
        {
            AbsoluteExpirationRelativeToNow
            = new TimeSpan(0, 0, seconds: 60)
        };

        await _distributedCache.SetAsync
        (key, serializedObject, options);
        return result;
    }
}
```

Create the Model Clasess

In a typical order management system, you'll have several entity or model classes. However, for the sake of simplicity, I'll consider and use only one entity or model class here. Create a new class named Order in a file having the same name with a .cs extension and write the following code in there:

```
public class Order
{
    public int Order_Id
    { get; set; }
    public int Customer_Id
    { get; set; }
    public DateTime Order_Date
    { get; set; }
    public decimal Amount
    { get; set; }
}
```

Listing 5: The MemCacheProvider class

```
public class MemCacheProvider : IMemCacheProvider
{
        private readonly
        IMemcachedClient memcachedClient;
        public MemCacheProvider
        (IMemcachedClient memcachedClient)
        {
            this.memcachedClient = memcachedClient;
        }
        public T GetCachedData<T>(string key)
        {
            return memcachedClient.Get<T>(key);
        }
        public void SetCachedData<T>
        (string key, T value,
        int duration)
        {
            memcachedClient.Set
            (key, value, duration);
        }
}
```

Install NuGet Package(s)

The next step is to install the necessary NuGet Package(s). To install the required package(s) into your project, right-click on the solution and then select Manage NuGet Packages for Solution.... Now search for the package named EnyimMemcachedCore in the search box and install it.

Alternatively, you can type the command shown below at the NuGet Package Manager Command Prompt:

```
PM> Install-Package EnyimMemcachedCore
```

You can also install this package by executing the following commands at the Windows Shell:

```
dotnet add package EnyimMemcachedCore
```

Create the MemCacheProvider Class

Now you need a class that acts as a wrapper on IMemcachedClient so as to encapsulate all calls to store and retrieve data to and from the cache. This not only simplifies access to the cache, it also ensures that you don't need to write code to manage cache in your controller or repository classes. Now, create a class named MemCacheProvider and write the code from **Listing 3** in there.

Listing 6: The OrderRepository Class

```
public class OrderRepository: IOrderRepository {
  private
  const string key = "MemCacheDemo";
  private readonly
  IMemCacheProvider _memCacheProvider;
  private readonly List <Order> orders
  = new List <Order> {
    new Order {
      Order_Id = 1,
        Customer_Id = 2,
        Amount = 125000.00 m,
        Order_Date = DateTime.Now
    },
    new Order {
      Order_Id = 2,
        Customer_Id = 1,
        Amount = 200000.00 m,
        Order_Date = DateTime.Now
    },
    new Order {
      Order_Id = 3,
        Customer_Id = 3,
        Amount = 750000.00 m,
        Order_Date = DateTime.Now
    }
  };
  public OrderRepository
  (IMemCacheProvider memCacheProvider) {
    _memCacheProvider = memCacheProvider;
  }
  public async
    Task <List <Order>> GetOrders() {
    var result =
   _memCacheProvider.GetCachedData
    <List<Order>> (key);

    if (result != null) {
      return result;
    }

    _memCacheProvider.SetCachedData
    <List<Order>> (key, orders, 600);
    return await Task.FromResult(orders);
  }
}
```

The MemCacheProvider class implements the IMemCacheProvider interface and wraps access to the cache using an instance of type IMemcachedClient. Here is the source code of the IMemCacheProvider interface:

```
public interface IMemCacheProvider
{
  T GetCachedData<T>(string key);
  void SetCachedData<T>
  (string key, T value,
  int duration);
}
```

Create the OrderRepository Class

The OrderRepository class uses an instance of type IMemCacheProvider to interact with the cache. This instance is injected in the constructor of the OrderRepository class. Now, create a new class named OrderRepository in a file having the same name with a .cs extension. Next, write the following code in there:

```
public class OrderRepository : IOrderRepository
{

}
```

The OrderRepository class implements the methods of the IOrderRepository interface. For the sake of simplicity, I have only one method called GetOrders. Here is how the IOrderRepository interface should look:

```
public interface IOrderRepository
{
    public Task<List<Order>> GetOrders();
}
```

The OrderRepository class implements the GetOrders method of the IOrderRepository interface. In this method, you first check whether the data requested is available in the cache. If the data is available in the cache, it's returned from there; otherwise, the data is retrieved from the database. Once the data is retrieved from the database, the cache is populated with this data. Lastly, this data is returned:

```
public async Task<List<Order>> GetOrders()
{
var result = _memCacheProvider.
GetCachedData<List<Order>>(key);

    if(result != null)
    {
        return result;
    }

  _memCacheProvider.SetCachedData
   <List<Order>>(key, orders, 600);
      return await
   Task.FromResult(orders);
}
```

The complete source code of the OrderRepository class is given in **Listing 6**.

Register and Configure the MemCached Instance

The following code snippet illustrates how an instance of type IOrderRepository is added as a scoped service to the IServiceCollection.

```
builder.Services.AddScoped
<IOrderRepository,
OrderRepository>();
```

Next, register and configure MemCached using the following piece of code in the Program.cs file:

```
builder.Services.
AddEnyimMemcached(mc =>
{
```

Namespaces Required for NCache

To work with NCache in ASP.NET Core, include the following namespaces in your program:

Alachisoft.NCache.Caching.Distributed

Microsoft.Extensions.Caching.Distributed

Listing 7: The Complete Source of Program.cs file

```
using DistributedCachingDemo;
using Enyim.Caching.Configuration;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();
builder.Services.
AddScoped<IOrderRepository, OrderRepository>();
builder.Services.
AddSingleton<IMemCacheProvider>(x =>
ActivatorUtilities.CreateInstance
<MemCacheProvider>(x));
builder.Services.AddEnyimMemcached(mc =>
{
    mc.Servers.Add(new Server
    {
        Address = "localhost",
        Port = 11211
    });
});

var app = builder.Build();

// Configure the HTTP request pipeline.

app.UseAuthorization();

app.MapControllers();

app.Run();
```

Listing 8: The OrderController Class

```
[Route("api/[controller]")]
[ApiController]
public class OrdersController : ControllerBase
{
    private IOrderRepository _orderRepository;
    public OrdersController
    (IOrderRepository orderRepository)
    {
        _orderRepository = orderRepository;
    }

    [HttpGet("GetOrders")]
    public async Task<List<Order>> GetOrders()
    {
        return await _orderRepository.GetOrders();
    }
}
```

```
    mc.Servers.Add(new Server
    {
        Address = "localhost",
        Port = 11211
    });
});
```

The complete source code of the Program.cs file is given in **Listing 7** for reference.

The OrderController Class

Finally, create a new API controller class named OrderController with the following code in there, as shown in **Listing 8**.

The OrdersController class contains two action methods, namely, the GetOrders method that returns a list of Order instances. This action method calls the GetOrders method of the OrderRepository class respectively. Note how an instance of type IOrderRepository is injected in the OrdersController class using constructor injection.

When you run the application and browse the /getorders endpoint, the order data is displayed in the web browser. In the first call to this endpoint, the data is fetched from the database and it's fetched from the cache for all subsequent calls to this endpoint. The cache is invalidated after the specified timespan elapses.

> There are a lot of details, but in the end, distributed caching is fairly simple.

Conclusion

Caching is a proven and established technique used to improve an application's performance, scalability, and responsiveness by storing frequently accessed data in transient storage, such as the memory or other permanent storage like a file or a database. In distributed caching, the cached data is spread across multiple nodes, often in different networks. As cached data is readily accessible, it ensures enhanced reliability during unexpected peaks or failures.

Joydip Kanjilal

C# for High-Performance Systems

My day job is writing a database engine named RavenDB. My preferred language to code in is C#. Those two facts tend to cause glitches in some people's minds. How can you write a database engine, which is all about low-level system programming, in a high-level language such as C#? How much performance are you leaving on the table by not choosing a proper system-level

language, such as C++ or Rust? It turns out that there isn't a conflict between the two. You can write high-performance code in C# that matches or beats the performance of native code. The best part is that you can achieve that while retaining many of the advantages that C# offers in the first place.

C# has a lot of expressiveness and high-level concepts, but from the get-go, it has some critical features for low-level systems. Value types, pointers, and unsafe code have all been part of C# from its initial creation. Recent additions to the .NET platform, such as spans, vector instructions, and hardware intrinsics make it much simpler to write high-performance and low-level code.

I want to be able to talk in concrete terms, so for the rest of this article, I'm going to focus on a relatively straightforward task. The code accepts a CSV file (gzipped) and computes aggregated statistics based on the data. You can view a sample of the data in the file in **Table 1**.

I generated a large file to test how well this code performs (the script to generate the data is available online at: https://gist.github.com/ayende/e84dac76e5dc023f6a80367f6c01ac13), and obtained a file with 250 million records with 9.78 million unique users. The compressed file size is about 2.4 GB. I intentionally chose a large size, which should clearly demonstrate the difference in performance between approaches.

The task at hand is to compute the total quantity and price for each user in the file. This is a simple task and the code to write it is almost painfully obvious. The input is a GZipped CSV file, so the first action is to decompress the data and iterate through the file one line at a time, as shown in **Listing 1**.

For reference, if I were to execute **GzipReadlAllLinesAsync(input).CountAsync()** alone, it would iterate over the entire file and count the number of lines in it. Executing this code on my machine yields the following results:

- Total allocations: 53 GB
- Execution time: 47 seconds

This is without doing any actual work, mind. It's just reading through the dataset.

The next task is to actually do something with the data and generate the aggregated results. I'll start from high-level code and write the aggregation using LINQ. Using the **System.Linq.Async** package, you can apply a LINQ expression to the results of **GzipReadlAllLinesAsync()**, which will do all the computation in a very clear fashion. You can see how this looks in **Listing 2**.

Before I discuss performance, I'd like you to take a look at **Listing 2** and see how I was able to write precisely what I wanted in an extremely succinct and expressive manner. The fact that I can do that so simply is one of the great strengths of C#, in my opinion. But good-looking code is just the first stage, let's see how well it performs at runtime.

When evaluating the performance of a piece of code, it's important not to look at just a single parameter. It's easy to trade off execution time for memory usage, for example. I'm running all the code in this article using .NET 8.0 in release mode. Here are the execution results:

- Peak memory: 18 GB
- Total allocations: 163 GB
- Execution time: 8 minutes, 19 seconds

Why am I mentioning memory **and** allocations in my metrics? The value of "total allocations" refers to the amount of memory the system has allocated over its lifetime, while "peak memory" represents the maximum amount of memory used at any given point in time. When talking about .NET programs, reducing allocations tends to have an outsized impact on performance, especially if you're running with multiple threads at once. That's because the more allocations you do, the more work the garbage collector has to handle. With multiple threads all allocating memory, it's easy to reach a point where you're mostly burning CPU cycles on garbage collection.

The compressed file size is 2.4 GB, but uncompressed, the data is about 3.7 GB. Given the amount of time that this code took to run, it was processing under 8 MB a second. That's really bad. The reason for that is that the code in **Listing 2** is executing its operations in a sequential manner. First, it runs through all the records in the file, grouping them by the user ID, and only then does it run the aggregation for each user. That means that it has to retain the entire dataset in memory until it completes the process.

Instead of using LINQ, you can choose to write the aggregation code yourself. You can see the code for this in **Listing 3** where you're basically doing the aggregation inline. For each record, you look up the relevant user statistics and update them. You only need to keep track of the users' statistics, not the entire data set.

Oren Eini
ayende@ayende.com

Oren Eini started his professional career as a developer in 1999 with a strong focus on the Microsoft and .NET ecosystem. He has been a Microsoft MVP since 2007. Oren's main focus is on architecture and best practices that promote quality software and zero-friction development.

Since 2008, he has been building the RavenDB database, a non-relational database engine written in C#. An internationally known presenter, Oren has spoken at conferences such as DevWeek, QCon, Oredev, NDC, and YOW!.

UserID	ItemID	Quantity	Price	Date
22271148092	5203189368466850000	91	7.03	24/12/2023 11:02:20
22271148092	1618666246057250000	2	11.49	24/12/2023 11:02:20
23392313395	7334974373793960000	1	40.69	24/12/2023 11:02:20
23145412223	2702876167686390000	2	15.56	24/12/2023 11:02:20
23145412223	5203189368466850000	2	7.03	24/12/2023 11:02:20

Table 1: Sample data from the orders CSV file.

What's the result of executing this code, then? It's going to use less memory, but how much faster is it going to be?

- Peak memory: 917 MB
- Total allocations: 126,999 MB (127 GB)
- Execution time: 1 minute, 51 seconds

The amount of code that you wrote isn't significantly different between **Listing 2** and **Listing 3**. Both are pretty clear and obvious in how they do things. However, you saved almost four-fifths (!) of the runtime and are using about 5% (!) of the memory. That is an amazing difference by all accounts.

It's also interesting to realize that the total amount of allocation isn't that drastically different. It's 163 GB vs. 127 GB, even though all the other metrics are far better. Why is that? Most of the allocations here are fairly unavoidable because you read the lines from the file one line at a time and generate a lot of strings. Additional strings are also allocated during the parsing of each line because you use **Split(',')**.

The code in **Listing 3** has far better performance than the initial version, but it can still improve. A key issue is the manner in which you process the data from the file. You're creating a lot of strings in the process, which is a big waste; you can do better. Instead of allocating strings for each line of the file, (remember, there are 250,000,000 of them), you can use a lower-level API.

System.IO.Pipelines is one such low-level API that's meant to provide an efficient way to read (and write) data in .NET. In particular, it uses a single reusable buffer to do the work and allows you to access it directly instead of performing allocations. The API is somewhat less convenient to use, but the performance difference more than makes up for this. Take a look at the code in **Listing 4**, showing the skeleton of using Pipelines for processing the file.

Unlike previous code listings, the code in **Listing 4** isn't actually doing much; most of the work is handled by the **ProcessLines()** method, which will be covered shortly. What's interesting here is that all the I/O is handled in the parent method. You ask the pipe to read a buffer from the stream and then process it. A key aspect is that the buffer isn't limited to a single line or is guaranteed to be on a line boundary. Instead, you get some buffer from the file (64KB in this case) and process everything in it.

The really nice thing about the **System.IO.Pipeline** approach is that it encourages a highly efficient manner for processing incoming data. Let's take a look at **Listing 5** where I actually do that and then I'll discuss exactly what is going on here.

The code in **Listing 5** isn't really doing much, but it's one of those cases where I removed everything non-essential to expose the core beauty. The **PipeReader** is going to read from the stream into a buffer, which is held by the **ReadResult**. You use a **SequenceReader<byte>** to search the current line in the buffer and return a reference (to the same original buffer) to the range of the line. If you can't find a line break, that means that you need to read more from the stream, so you inform the pipe reader how far you read and until what point you've examined the buffer. The next call to **ReadAsync()** (shown in **Listing 4**), will then know what can be discarded and what more needs to be read from the stream.

There isn't a lot of code here, but from an architectural perspective, there's a lot going on. The end result is that you can use a single reusable buffer to process the data. The **ProcessLines()** method also contains a loop. It's go-

Listing 1: Decompressing and yielding the lines from the file.

```
async IAsyncEnumerable<string>
  GzipReadlAllLinesAsync(Stream input) {
  using var gzipStream = new GZipStream(input,
    CompressionMode.Decompress);
  using var reader = new StreamReader(gzipStream);

  while (true) {
    string? line; = await reader.ReadLineAsync();
    if(line == null)
      break;
    yield return line;
  }
}
```

Listing 2: Computing total and quantity per user using Linq

```
ValueTask<Dictionary<long, UserSales>>
  Linq(Stream input) {
  return (from l in GzipReadlAllLinesAsync(input)
                  .Skip(1) // skip header line
      let flds = l.Split(',')
      let item = new {
        UserId = long.Parse(flds[0]),
        Qty = int.Parse(flds[2]),
        Price = decimal.Parse(flds[3])
      }
      group item by item.UserId into g
      select new {
        UserId = g.Key,
        Quantity = g.SumAsync(x => x.Quantity),
        Total = g.SumAsync(x => x.Price)
      }).ToDictionaryAsync(x => x.UserId,
        x => new UserSales {
            Quantity = x.Quantity.Result,
            Total = x.Total.Result
        });
}
```

Listing 3: Manually aggregating over the records

```
async Task<Dictionary<long, UserSales>>
  StreamReaderAndDictionary(Stream input) {

  var sales = new Dictionary<long, UserSales>();
  await foreach(var line in
    GzipReadlAllLinesAsync(input).Skip(1)) {

    var fields = line.Split(',');
    var uid = long.Parse(fields[0]);
    int quantity = int.Parse(fields[2]);
    decimal price = decimal.Parse(fields[3]);

    if (!sales.TryGetValue(uid, out var stats))
      sales[uid] = stats = new UserSales();

    stats.Total += price * quantity;
    stats.Quantity += quantity;
  }
  return sales;
}
```

ing to be running over the **entire** buffer, which is very likely to contain many lines. Instead of operating on a per-line basis, it's now operating over a far bigger batch of items. That also has implications with regard to performance, because the hot loop needs to cover a lot less code to do its job. The actual processing of the lines is shown in **Listing 6**, which also contains some interesting code patterns to talk about.

I'm afraid that the code is pretty dense, but the idea is pretty simple. You use another **SequenceReader<byte>** to find the terminators in the line buffer you're given here, you then use **Utf8Parser** to parse them without needing to allocate any strings. The code is complicated because it's a single expression and because I wanted to produce the correct error in the case of invalid input. Note that I'm using an **unconditional bitwise and** (&) versus the more common **condition and** (&&). This ensures that the entire expression runs and avoids branch instructions. This is **the** hotspot in the code, so anything that reduces costs is advisable, and branches can be costly (even if well predicted, as in this case).

I'm also not using the **Dictionary** as you'd normally expect. I'm calling **CollectionsMarshal** to get or add the value. I'm not sure if you noticed, but **Listings 4**, **5**, and **6** all use **UserSalesStruct** as the value, instead of **UserSales** (a class). That change allows you to avoid allocations even further and take advantage of the **GetValueRefOrAddDefault()** behavior.

A very typical access pattern when using dictionaries is to look up a value by key and create it if it isn't in the dictionary. This requires you to do two dictionary lookups if you need to create the value. The **GetValueRefOrAddDefault()** means that you only need to do one. Because a struct in C# is guaranteed to be zero initialized, you don't care if the value exists or not and you can immediately add the current record values to the user's sales.

Finally, you might have noticed that the method in **Listing 6** has an interesting attribute: **[MethodImpl(MethodImplOptions.AggressiveInlining)]**. This is an instruction to the Just In Time compiler to force inlining of this method to its caller. This allows you to benefit from clear code and separation of responsibilities while getting dense (and efficient) machine code.

The Pipeline version is spread over three code listings, and I had to reach quite far out of the beaten path. What did you get, then, from this journey? Take a look at these results:

- Peak memory: 846 MB
- Total allocations: 1069 MB
- Execution time: 50.5 seconds

I triple checked those results because they are **really** good. You can see the full results in **Table 2**, but the Pipelines method is almost 50% faster than the **StreamReader + Dictionary** approach you used in **Listing 2**. It's also literally allocating less than 1% of the memory.

The task of processing a file and aggregating the results is a simple one. The sort of thing that's commonly given

Listing 4: Reading from the file using PipeReader

```
async ValueTask<Dictionary<long, UserSalesStruct>>
  PipelineAsync(Stream input) {
  using var gzipStream = new GZipStream(input,
              CompressionMode.Decompress);
  var pipeReader = PipeReader.Create(gzipStream,
    new StreamPipeReaderOptions(
            bufferSize: 64 * 1024)
    );

  var header = false;
  var salesData = new Dictionary<long, UserSalesStruct>();
  while (true)  {
    var valueTask = pipeReader.ReadAsync();
    var read = valueTask.IsCompleted ? valueTask.Result :
                                  await valueTask.AsTask();
    ProcessLines(pipeReader, read, salesData, ref header);
    if (read.IsCompleted)
      break;
  }

  return salesData;
}
```

Listing 5: Processing lines in the buffer

```
void ProcessLines(PipeReader pipeReader,
  ReadResult readResult,
  Dictionary<long, UserSalesStruct> salesData,
  ref bool header) {

  var sr = new SequenceReader<byte>(readResult.Buffer);
  while (true)  {
    ReadOnlySequence<byte> line;
    if (sr.TryReadTo(out line, (byte)'\n') == false) {
      pipeReader.AdvanceTo(consumed: sr.Position,
            examined: readResult.Buffer.End);
      break;
    }
    if (header == false)  {
      header = true;
      continue;
    }
    ProcessSingleLine(salesData, line);
  }
}
```

Listing 6: Processing a single line

```
[MethodImpl(MethodImplOptions.AggressiveInlining)]
void ProcessSingleLine(
   Dictionary<long, UserSalesStruct> salesData,
   ReadOnlySequence<byte> line)
{
  var lr = new SequenceReader<byte>(line);
  ReadOnlySpan<byte> span;
  var readAll = lr.TryReadTo(out span, (byte)',')
    & Utf8Parser.TryParse(span, out long userId, out _)
    & lr.TryAdvanceTo((byte)',')
    & lr.TryReadTo(out span, (byte)',')
    & Utf8Parser.TryParse(span, out int quantity, out _)
    & lr.TryReadTo(out span, (byte)',')
    & Utf8Parser.TryParse(span, out decimal price, out _);

  if (readAll == false)
    throw new InvalidDataException(
        "Couldn't parse expected fields on: " +
        Encoding.UTF8.GetString(line));

  ref var current = ref CollectionsMarshal
            .GetValueRefOrAddDefault(salesData, userId, out _);
  current.Total += price;
  current.Quantity += quantity;
}
```

Scenario	Time (sec)	Allocations (MB)	Peak memory (MB)
Linq	499	166,660	18,286
StreamReader + Dictionary	111	117,272	917
Pipelines	50.5	1,069	846

Table 2: Summary of performance and memory consumption across all scenarios

as course assignments to students. It's also a pretty good scenario to test our capabilities because it's so limited in scope. The point of this article isn't to point you toward the **System.IO.Pipelines** API (although I'll admit that it's a favorite of mine), it's to demonstrate how you can entirely change the manner in which you perform a task from the extremely high-level LINQ expression all the way down to reusable buffer and almost no allocations.

C# and .NET allow you to create solutions for a very wide range of scenarios. From building a business application with a focus on correctness and time to market, all the way down to database engines such as RavenDB. In RavenDB, you'll find that there is no such thing as "fast enough." You can spend literally months optimizing a single piece of code to be just a little bit faster.

You're making a lot of use of seemingly esoteric features such as hardware intrinsics, vector operations, and zero allocation APIs. The nice thing about that is that you can meet your performance goals while still writing in C#.

That isn't just a personal preference of wanting to stay in your comfort zone. I started writing C# code because of the experience of getting an error with a proper error message and a detailed stack trace, rather than some hex code and "you figure it out." That experience is still very much true today: Using C# as the primary language for RavenDB means that you can use all of the infrastructure around .NET for profiling, debugging, and monitoring. There's also the quality of the tooling around .NET that matters a lot.

Beyond exploring a small way in which you can improve performance by an order of magnitude, I want to emphasize that performance is a journey. You need to align your overall architecture and the algorithms you use with the actual quality of implementation and micro-optimizations. Using RavenDB as an example again, it's common, in certain hotspots, to modify (C#) code in order to get the right machine code for your needs. It's quite amazing that you can do that when you need to. Doubly so, as you can do that for both x64 and ARM machines.

Finally, I'll leave you with this to consider. It's often best to go the other way around. Look at the best practices for high-performance implementations and then walk backward to an architecture that would enable you to use that. With that done, you can then selectively implement the hotspots as needed, without having to struggle with past architectural decisions.

Oren Eini

CODE 30 YEARS

M ar/Apr 2024
Volume 25 Issue 2

Group Publisher
Markus Egger

Editor-in-Chief
Rod Paddock

Managing Editor
Ellen Whitney

Content Editor
Melanie Spiller

Writers in This Issue
Philipp Bauer, Oren Eini, Wei-Meng Lee, Shawn Wildermuth, Markus Egger, Joydip Kanjilal, Sahil Malik, Mike Yeager

Technical Reviewers
Markus Egger
Rod Paddock

Production
Friedl Raffeiner Grafik Studio
www.frigraf.it

Graphic Layout
Friedl Raffeiner Grafik Studio in collaboration with onsight (www.onsightdesign.info)

Printing
Fry Communications, Inc.
800 West Church Rd.
Mechanicsburg, PA 17055

Advertising Sales
Tammy Ferguson
832-717-4445 ext. 26
tammy@code-magazine.com

Circulation & Distribution
General Circulation: EPS Software Corp.
Newsstand: Ingram Periodicals, Inc.
International Bonded Couriers (IBC)
Media Solutions
Source Interlink International

Subscriptions

Circulation Manager
Colleen Cade
832-717-4445 ext. 28
ccade@codemag.com

US subscriptions are $29.99 USD for one year. Subscriptions outside the US are $50.99 USD. Payments should be made in US dollars drawn on a US bank. American Express, MasterCard, Visa and Discover credit cards accepted. Back issues are available. For subscription information, email subscriptions@code-magazine.com or contact customer service at 832-717-4445 ext. 9.

Subscribe online at
www.code-magazine.com

CODE Developer Magazine
EPS Software Corporation / Publishing Division
6605 Cypresswood Drive, Ste 425, Spring, Texas 77379 USA
Phone: 832-717-4445

CODE
STAFFING

UNLOCK STAFFING EXCELLENCE

Top-Notch IT Talent, Contract Flexibility, Happy Teams, and a Commitment to Customer Success Converge with CODE Staffing

Our IT staffing solutions are engineered to drive your business forward while saving you time and money. Say goodbye to excessive overhead costs and lengthy recruitment efforts. With CODE Staffing, you'll benefit from contract flexibility that caters to both project-based and permanent placements. We optimize your workforce strategy, ensuring a perfect fit for every role and helping you achieve continued operational excellence.

Ready to Discuss Your IT Staffing Needs?

Visit our website to find out more about how we are changing the staffing industry.

Website: codestaffing.com
Yair Alan Griver (yag)
Chief Executive Officer
Direct: +1 425 301 1590
Email: yag@codestaffing.com