

CODE

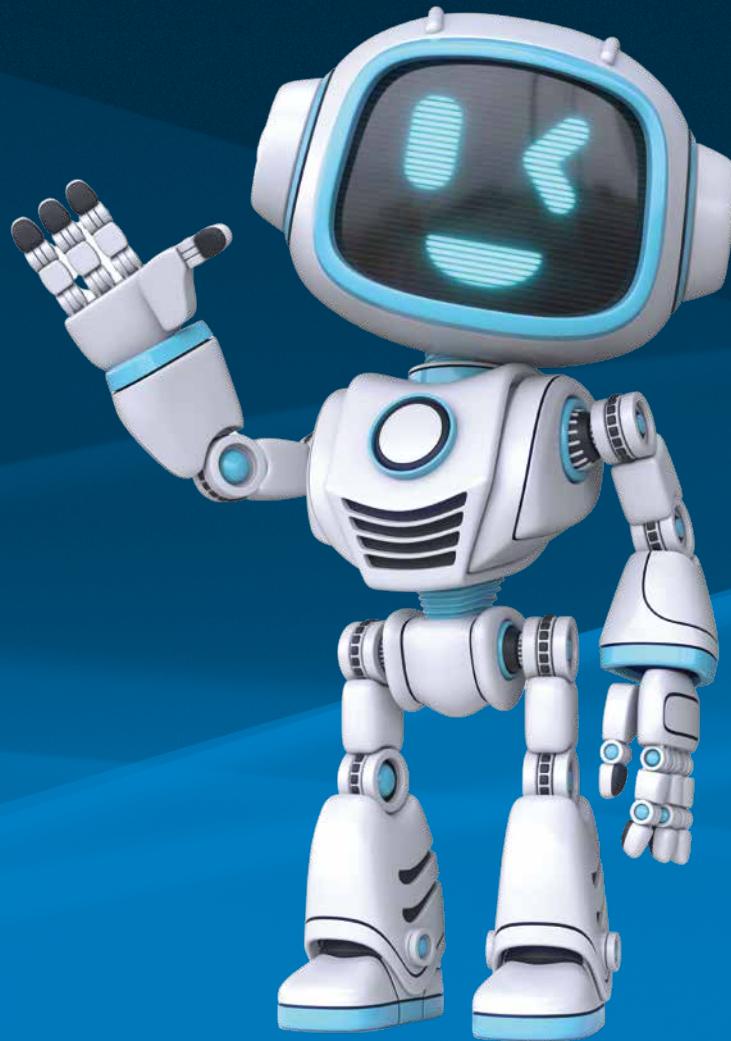
Does Angular End RxJS Boilerplate?

Make Movies
with AI

Embracing JavaScript
Templates

Async Techniques
Explored





**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

8 AI and Developer Productivity

The whole world is talking about artificial intelligence—will it steal jobs? Will it take the artistry out of our lives? Will it make work for developers who have to fix what it breaks? Sahil reveals how to use AI to improve your coding experience by doing the scut work, helping find errors, and taking on repetitive tasks.

Sahil Malik

16 Introduction to JavaScript Templating

Paul takes you along as he explores how to replace string concatenation during client-side manipulation of the DOM by using a templating engine.

Paul Sheriff

30 Wait, What? Async-Await Explained

Just because you've written a scadillion async-await functions doesn't mean you truly understand them. Matthew explains this useful C# language feature.

Matthew Hess

36 Angular Signals: The End of RxJS Boilerplate?

Sonu shows you how to use Angular Signals to model data and UI state where you used to use RxJS. Don't worry, RxJS still has its uses and Sonu explains when to use which tool.

Sonu Kapoor

49 The New Features and Enhancements in .NET 10 (Part 2)

Joydip got his hands on a preview for .NET 10 and, in this second installment, is eager to tell you what he found.

Joydip Kanjilal

66 Veo 3: The Clockwork Hand

Jason wants to make a film. He's got a plan and a vision. Now all he needs is the software to help him get there. Read his analysis of Veo, a text-to-video model developed by Google.

Jason Murphy

Departments

6 Editorial

27 Advertisers Index

74 Code Compilers



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.



OCT 7-9, 2025 | WORKSHOPS OCT 5, 6 & 10
LOEWS SAPPHIRE FALLS RESORT at UNIVERSAL ORLANDO, FL

FEATURED SPEAKERS



SCOTT HANSELMAN
*Vice President,
Developer Community,
Microsoft*



SCOTT HUNTER
*Vice President for
Azure Developer
Experience, Microsoft*



MADDY MONTAQUILA
*Senior Product
Manager, Developer
Tools, Microsoft*



PAULA JANUSZKIEWICZ
*CEO, CQURE Inc. &
CQURE Academy*



LESLIE RICHARDSON
*Senior Program
Manager, Microsoft*



MARKUS EGGER
*President and Chief
Software Architect,
CODE Group*



JOHN PAPA
*Partner GM - Cloud
Advocacy, Microsoft*



**MICHELE LEROUX
BUSTAMANTE**
*CEO & Architect,
Solliance*



CAREY PAYETTE
*Senior Software
Engineer, Trillium
Innovations*



ZOINER TEJADA
*AI Architect,
FoundationalLLM*

REGISTER TODAY!

Scan a QR code to view sessions, workshops, speaker bios, schedule, hotel info, and more.
Save \$200 if you register with discount code CODEMAG using a QR code below.





Discernment and Determinism

When I was in high school, my family owned a car stereo shop. Along with manning the register and sweeping the floors, one of my duties was to help with the car stereo installation process. I spent many of my formative years learning the mystic arts of car stereo installation and after many years of apprenticeship, I was allowed to do the simple installations, primarily in-door speakers. One of the most important things I learned during my apprenticeship was tool selection. There was always a tool for a job no matter how hairy or complex. Sometimes it was tin snips to widen a hole, other times it was a

battery with wires attached to it to test the speaker via its contacts. The most frequent tool used in this process was made using a straightened wire coat hanger and electrical tape. We'd attach speaker wire to the hanger and use it to lace through the nooks and crannies of the customers' car. During my apprenticeship, I learned how to discern which tool was best for each job.

Most of the teams I work on nowadays spend large amounts of time and resources investigating the litany of AI tools and technologies available. The primary goal of this research is to leverage these tools to improve our users' and customers' experiences. One area of research that seems promising is using natural language to build information-retrieval applications capable of gleaning information from company databases, documents, and other sources of information. Basically, internal search engines.

Over the last three-plus decades, I've worked on tons of database systems. Without fail, every system ended up with some form of hand-cobbled query generator in which end users could write their own queries. This hand-cobbled system was usually some type of screen resembling the command console from Starship Enterprise. It takes a PhD just to generate mailing list from these beasts. LOL.

What if we could create systems where users use natural language to ask questions like: "Who were my most valued customers last month?" or

"How have sales broken down by product and by month over the last three years?"

My guess is that the current state of AI tech should be able to handle this scenario.

I began my research looking at available open-source tools and code. Overall, I felt that to create something really useful, the cost-to-value ratio would be prohibitive. I started looking for something that was more self-contained. Luckily for me, Snowflake, my cloud database provider, has added a huge set of AI features, and one of them might work for my needs. After a bit of research, I found Cortex, a natural language-to-SQL tool built into the Snowflake architecture. Cortex works by taking a natural language query, sending it to Snowflake's LLM engine accompanied by metadata (defined with a YAML-based semantic model), and results in a SQL Query. You then take that query and pass it to Snowflake, which returns your query data. **Figure 1** demonstrates this.

After building these different pieces and parts, I was able to start firing queries at the engine and, to be honest, its success rate was impressive. I did a demonstration of this a few months back in front of a live meeting and it generated successful queries around 95% of the time. This begs the question: "Yes, it generated queries that ran without error 95% of the time, but did these queries reflect the actual content of my question?" This was something that I considered nearly immediately. If I want to put this

in users' hands, how do I make sure that the system provides accurate answers?

My team is starting to handle this using two techniques. One is taking advantage of the **Verified Query** feature provided by Snowflake's semantic model specification. Verified queries are ones that developers deem "good" and that can be used to train the model. The more verified queries, the better the model result.

This is augmented by using unit testing to send variants of these queries to the engine and then verifying them against known good queries. These two techniques were very loosely inspired by an article I read about the Cleveland clinic. The article, "Mayo Clinic's secret weapon against AI hallucinations: Reverse RAG in action" (<https://venturebeat.com/ai/mayo-clinic-secret-weapon-against-ai-hallucinations-reverse-rag-in-action/>), discusses using multiple passes of RAG to help prevent hallucinations. The goal is to enable users to ask discerning questions and get deterministic data in return. Although this RAG style of validation isn't perfect, it's moving us closer to putting these systems into user's hands.

As researchers, we need to clarify with our users the true nature of these LLM-based AI systems. The LLMs are discerning systems. They make best guesses from your queries and return answers that may or may not be deterministic. Users of technology are used to getting deterministic answers from technology: We put two-plus-two and we expect to get four. This is why LLMs are really buyer-beware.

That returns this discussion to that apprentice stereo installer, now the AI researcher apprentice. What tools will the researcher decide are the best for the job? Time will tell, but semantic models, RAG-inspired validation, and reinforced model training with verified queries seem like some long-terms tools to be added to the toolbox.

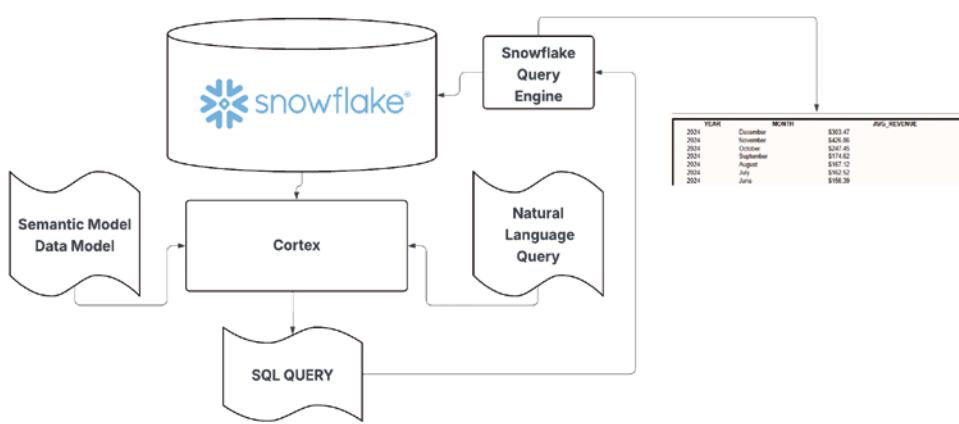
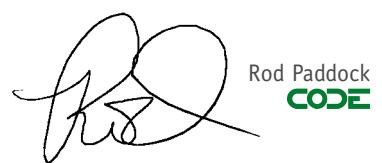
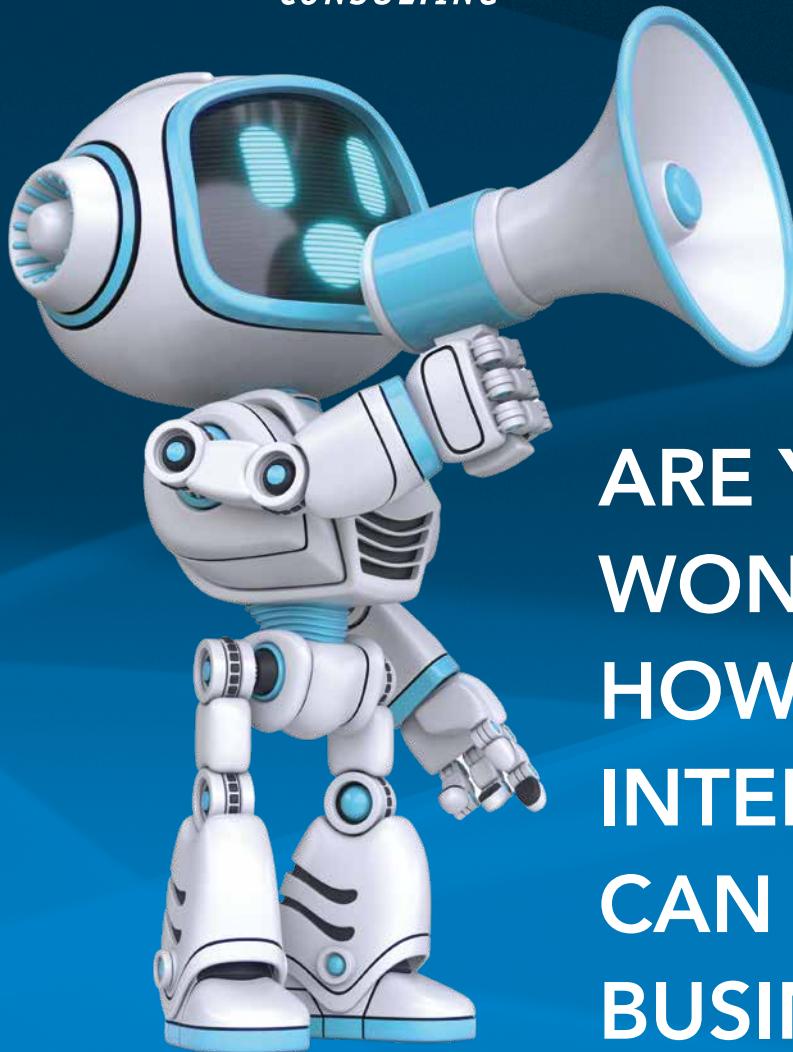


Figure 1: A query's journey using Cortex, Snowflake's natural language tool



Rod Paddock
CODE



ARE YOU WONDERING HOW ARTIFICIAL INTELLIGENCE CAN HELP YOUR BUSINESS?

Do you worry about privacy or regulatory issues stopping you from using AI to its fullest?

We have the answers!

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.

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

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

AI and Developer Productivity

The world of software development is in constant flux, but few forces have driven as profound a shift as artificial intelligence. What once seemed like science fiction is now an everyday reality, with AI tools seamlessly integrating into developer workflows, promising not just incremental gains but a fundamental redefinition of productivity. In 2025, developers are finding themselves



Sahil Malik

[@sahilmalik](http://www.winsmarts.com)

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.



empowered by intelligent assistants, automated guardians of code quality, and even AI “colleagues” capable of tackling complex engineering tasks.

I have written a bunch of articles in CODE Magazine about AI. All of them have focused on learning AI, such as image generation, creating a local chat bot, and more. But what if you’re not an AI developer? Maybe you’re a ReactJS developer writing front-end code all day long. Or maybe you write REST APIs using Python all day long.

Let’s be honest, AI is exciting, but many of us are still working day-in-day-out delivering business functionality code, things your employer needs today. Should you ignore AI? Far from it. This article explores how AI is boosting developer productivity across the software development lifecycle, complete with practical examples to illustrate its transformative power.

The AI-Powered Developer: A New Paradigm

As of today, at its core, AI for developers isn’t about replacing human creativity; it’s about augmenting it. By offloading repetitive, time-consuming, and error-prone tasks, AI frees developers to focus on higher-level problem-solving, architectural design, and innovative solutions. This shift fosters a more engaging and less frustrating development experience.

Let’s dive into the key areas where AI is making a tangible difference.

There are many ways I see that AI can help you as a developer. This is by no means an exhaustive list, if you have ideas do share.

AI as Your Pair Programmer

The most visible and widely adopted application of AI in development is in code generation and intelligent completion. There are many competing tools you can use. Tools like GitHub Copilot, Tabnine, and Amazon CodeWhisperer act as highly intelligent pair programmers, anticipating your next move and suggesting relevant code snippets, entire functions, or even boilerplate structures. In fact, there are VSCode extensions that let you plug into any AI model to get specific help for your scenario. You can even use Ollama to run things locally if you’re in an air-gapped secure environment. Of course, the capabilities of cloud-based models are far ahead of what Ollama on your local machine can do, but Ollama with a local model is still superpowers that you didn’t know you had.

There are many benefits of incorporating AI as your pair programmer.

The first is, of course, speed. Using AI drastically reduces the time spent on writing repetitive code or searching for syntax. How often do you find yourself struggling to find the right syntax for a particular thing you’re trying to do? Or writing repetitive code that you know you can write, but would rather have a helper write for you, and maybe even write it better than you? Like, find username out of a jwt token. I know how to do this, I just wish I didn’t have to do this in every project I land in. Yes you decode the token, which means first convert base 64 to JSON, oh wait, first separate the three parts of the token, validate the signature, blah blah! Dear AI: Just do this for me, please?

The other obvious advantage is accuracy. Using AI minimizes typos and common syntax errors, leading to fewer debugging cycles. When I was a programmer in my teens, I took great pride in my accuracy and typing capabilities. I could type at > 140WPM without errors. Alas, as time has passed, my fingers have too grown older. I do make mistakes now. Unfortunate mistakes that take forever to find the errors they introduce. All because of a stupid typo. If I can have Microsoft Word correct my spelling mistakes, wouldn’t it be nice if AI can fix the errors my IDE cannot catch?

And finally, like any good pair programmer, I learn from my AI buddy. See I’ve never been a fan of pair programming. I know I know, you can put those daggers back in their sheaths. But I learn differently from others. When I’m deep into programming, I don’t want another person interrupting my thought process, or constantly interrupting asking questions. Pair programming may be great for the new person on the team, but as an experienced programmer (sorry for putting myself on a pedestal), I found pair programming was a lot of giveth and not enough taketh. I want to pair program with someone better than me, and those can be hard to find. AI frequently is better than me. Again, please don’t see this as me putting myself on a pedestal, but good developers are very hard to find, and when I do get a chance to learn from one, I take that chance every single time. It’s just that those chances are a few and far in between.

All right, enough talk. Let’s see an example. I went ahead and installed Ollama and pulled down the Code Llama model. If this sounds Greek to you, please see this article where I talk about AI with no internet connection. <https://www.codemag.com/Article/2411031/AI-with-No-Internet-Connection>. I talk about how to use Ollama to create a local chatbot backed by a PDF. Although that article talked about an LLM indexing your content, you can use the same mechanism to work with code, except that you use a different model. There are many models to pick from, I picked Code Llama because it works well with code.

To allow the continue extension to talk my local model, I created a config.yaml, as can be seen in **Figure 1**, in my local profile in the .continue folder.

With this, you're ready to roll. On the left-hand side, you should see an icon for the continue extension. If you click on it, a chat window comes up, as can be seen in **Figure 2**.

Now, you can work with any code base, but I'll just pick the code I showed in my AI with no internet connection article. Some of the concepts were a bit wiry, so I open the index.py from that code example, and, with my mouse, select an arbitrary portion of the code, as can be seen in **Figure 3**.

I see that a couple of interesting tooltips have shown up. I can see that I could use "chat" by pressing CMD_L so I did. And I asked, "What does this code do?" and what happened next blew me away.

Here is the output it gave me.

This code defines a chain of functions that will be called in sequence to retrieve information from the user. The chain consists of three functions: itemgetter, retriever, prompt, model, and parser. Each function takes an input and returns an output. The output of one function becomes the input for the next function in the chain.

Here's a breakdown of what each function does:

itemgetter: This function retrieves a specific item from an object, such as a dictionary or list. In this case, it retrieves the "question" key from the input object.

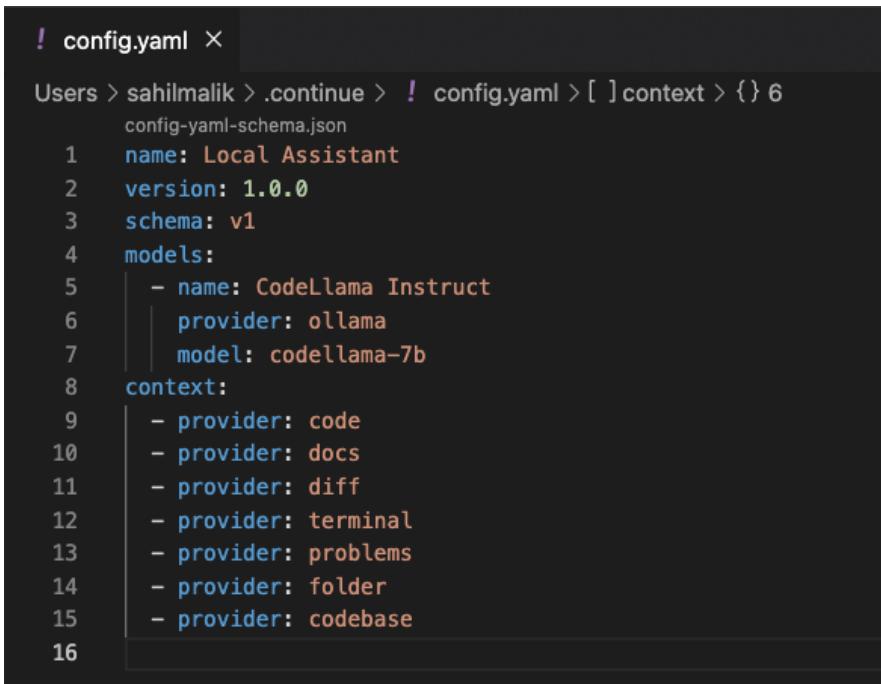
retriever: This function is responsible for retrieving the user's response to a given question. It takes in a prompt as an input and returns the user's response.

prompt: This function is responsible for displaying the prompt to the user. It takes in a prompt as an input and displays it on the screen.

model: This function is responsible for processing the user's response. It takes in a response as an input and returns a processed version of the response.

parser: This function is responsible for parsing the user's response into a format that can be used by the chatbot. It takes in a response as an input and returns a parsed version of the response.

The chain of functions is defined using the | operator, which allows each function to receive the output of the previous function as its input. In this case, the itemgetter function retrieves the question from the input object, and then passes it to the retriever function for processing. The prompt function displays the prompt to the user, and the model function processes the user's re-



```
! config.yaml ×
Users > sahilmalik > .continue > ! config.yaml > [ ] context > {} 6
    config-yaml-schema.json
1   name: Local Assistant
2   version: 1.0.0
3   schema: v1
4   models:
5     - name: CodeLlama Instruct
6       provider: ollama
7       model: codellama-7b
8   context:
9     - provider: code
10    - provider: docs
11    - provider: diff
12    - provider: terminal
13    - provider: problems
14    - provider: folder
15    - provider: codebase
16
```

Figure 1: The config.yaml for the continue extension

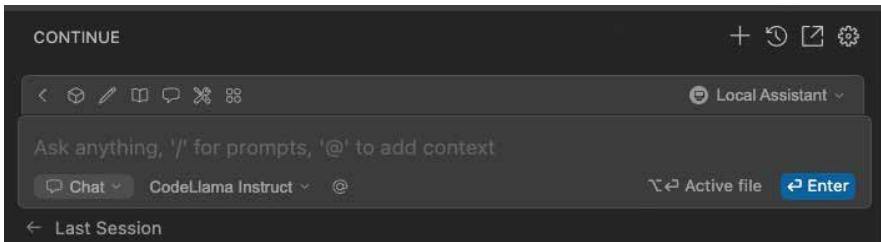
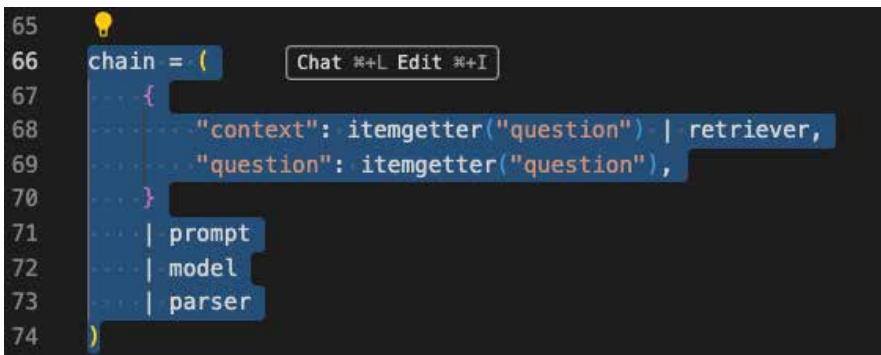


Figure 2: The chat window for the continue extension



```
65
66   chain = (
67     |
68     "context": itemgetter("question") | retriever,
69     "question": itemgetter("question"),
70   |
71   | prompt
72   | model
73   | parser
74 )
```

Figure 3: Some code selected. This code seemed important.

sponse before passing it to the parser function for parsing.

I'm floored! This gave me an output better than I could have described it in the article. Believe me, I'm only getting started here.

There's a lot you can do with "AI as your pair programmer" so let me just go from explaining code, to jumping

10 steps ahead and creating full projects with AI. I went ahead and installed GitHub Copilot extensions in VSCode and pressed **CMD_SHIFT_I** to open the chat window. I also opened VSCode in a blank folder, and I set the mode to “Agent” and the model I’m using to GPT 4.1, as can be seen in **Figure 4**.

GitHub gives us little bit of Copilot for free, and VSCode and GitHub copilot aren’t the only shows in town. You should also check out Cursor AI and Windsurf.

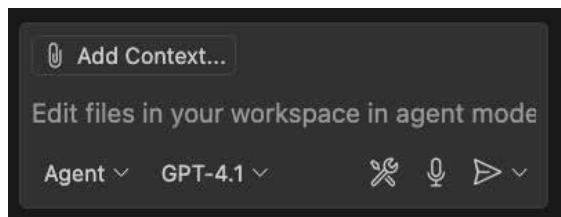


Figure 4: My starting GitHub Copilot settings

SPONSORED SIDEBAR

Adding Copilots
to Your Apps

The future is here now and you don't want to get left behind. Unlock the **true potential** of your software applications by adding **Copilots**.

CODE Consulting can assess your applications and provide you with a roadmap for adding Copilot features and optionally assist you in adding them to your applications.

Reach out to us today to get your application assessment scheduled.
www.codemag.com/ai

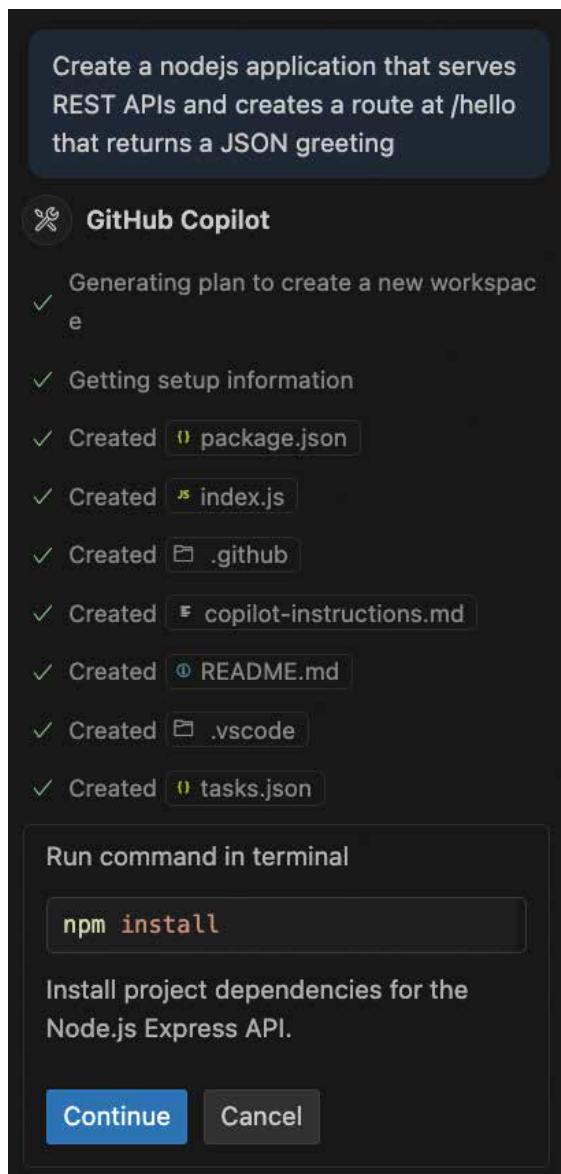


Figure 5: My application written up for me

Back to Copilot. I gave it the following prompt:

>Create a nodejs application that serves REST APIs and creates a route at /hello that returns a JSON greeting.

This set Copilot thinking. More than thinking, it’s showing me exactly what it’s doing and writing code for me. When it’s ready for me to run an action, it prompts me to do so, as can be seen in **Figure 6**.

All I have to do now is hit Continue to run **npm install**. Copilot then instructs me to run **npm start**, which I do, and visit <http://localhost:3000/hello>, and my application is running. This can be seen in **Figure 6**.

I can now glance at the generated code and learn from it. In fact, I can ask the agent to make modifications to it. So, for instance, let’s say:

Modify the "/hello" route to accept a user name as a querystring parameter, and return "Hello <username>" as output.

Now when I visit, "<http://localhost:3000/hello?name=sahil>" the output is:

```
{"greeting": "Hello sahil"}
```

I encourage you to take this much further. Upload a screenshot of a ToDo app and ask it to generate the app for you. I assure you, you’ll be amazed.

As amazed as you’ll be, believe me, we’re just getting started here.

Automated Code Review

Maintaining high code quality, consistency, and security is paramount for any software project. Manual code reviews are essential but can be time-consuming and prone to human oversight. AI-powered tools are transforming this by acting as vigilant guardians of your codebase.

Tools like Qodo (formerly CodiumAI), DeepCode AI (Snyk), and CodeRabbit analyze code for potential bugs, vulnerabilities, and style violations, and they even generate test cases.

There are many benefits of code reviews done by AI.

It’ll help you detect bugs early. AI catches issues before they integrate deeper into the system, reducing costly fixes later. You’ll improve code quality. AI ensures adherence to coding standards and identifies “code smells.” You gain faster feedback cycles. Imagine integrating AI in CircleCI. You’ll get immediate insights, reducing the bottleneck often associated with manual reviews. And my favorite, AI offers enhanced security. You can proactively scan for common security vulnerabilities.

Teamwork is important, and I get requests to review pull requests all the time. I have my work to do, and yet here

I am reviewing other's PRs. Ah well, this is how I stay on top of what others are doing. But there are days when I'm just too busy, or frankly I don't have the intelligence to review a PR. This is where AI can help me.

So for my next trick, I'm going to review a PR, just something real world that exists in the wild. I went ahead and cloned <https://github.com/AzureAD/microsoft-authentication-library-for-python#>, which is the repo for MSAL Python, and I see they have a moderately complex PR going on with some comments on it. Sounds like a perfect candidate to impress my friends with (AI's) skills.

Just to show you that this is absolutely real world, this is the PR I'm reviewing. <https://github.com/AzureAD/microsoft-authentication-library-for-python/pull/759>. I had no coordination with the authors of that PR before this, and it's 3:30 AM on a Sunday night as I write this anyway.

My terminal chops aren't so great, and I keep forgetting how to check out a Git branch. No biggie. In the integrated VSCode terminal, I just press CMD_I, and describe in plain English what I wish to do, as can be seen in **Figure 7**.

This is so easy, it feels like cheating.

Reviewing the author's PR, I notice that the author has changed line 242 in application.py. Lazy person that I am, I select those lines, and ask Copilot to review it.



Figure 6: My application is now running.

Copilot thinks for a while and immediately gives me two possible improvements. I'll show you the first comment for brevity, as can be seen in **Figure 9**.

I could just apply and change the code, but I want to appear smart to the author. So I'll take Copilot's suggestion and leave my review on the PR.

I left my comments on the PR, and my AI based code review is done. All I need to do now is put eight hours on my timesheet and take a long lunch. This programming stuff is too hard.

Jokes aside, my boss will just give me more work. So let's stay focused.

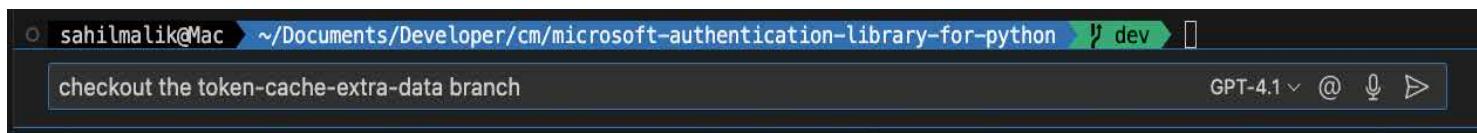


Figure 7: It even helps me with Git stuff.



Figure 8: Reviewing stuff and looking smarter than I really am

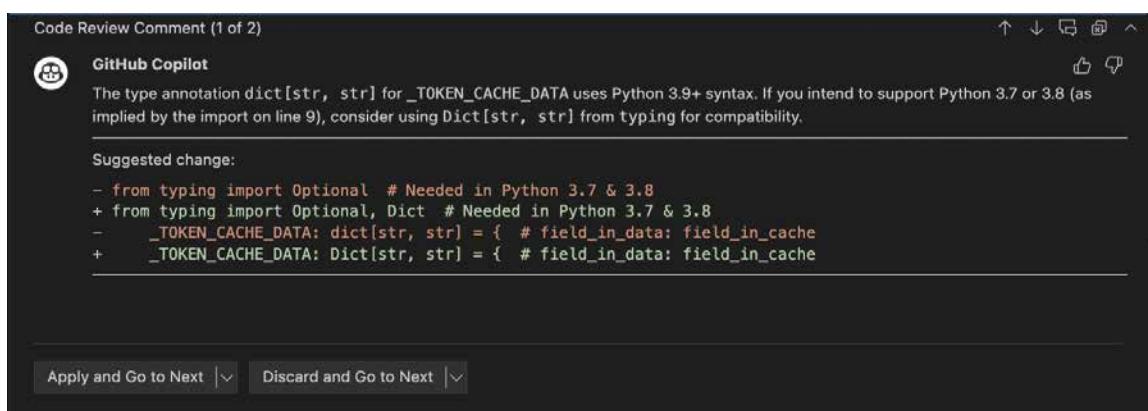


Figure 9: Reviewing PRs the easy way

Automated Documentation

Documentation is crucial for project maintainability and team collaboration, yet it's often the first thing to be deprioritized. Developers love to write code; developers hate writing documentation. AI tools are stepping in to automate this tedious but vital task. Tools like Mintlify can generate comprehensive documentation directly from your codebase, and general AI assistants like Notion AI can summarize meeting notes, create project briefs, or explain complex technical concepts.

This has many benefits.

You reduce manual effort. AI frees developers from the burden of writing and maintaining documentation.

You gain improved accuracy and currency. Documentation stays synchronized with code changes. Imagine a world where a merge of a pull request automatically generates documentation for you.

Key Components

1. Error Handling

- `RedirectUriError` and `TokenTypeError`: Custom exceptions for specific broker errors.
- `_convert_error(error, client_id)`: Converts broker errors to MSAL-style error dicts.

2. Redirect URI

- `_redirect_uri_on_mac`: Special redirect URI for macOS due to platform constraints.

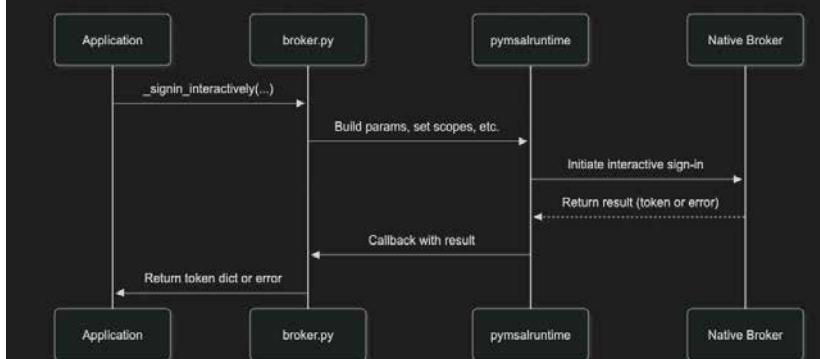
3. Account Handling

- `_read_account_by_id(account_id, correlation_id)`: Reads an account from the broker by ID.

4. Token Acquisition

- `_convert_result(result, client_id, expected_token_type)`: Converts broker results to MSAL-style token responses.
- `_signin_silently(...)`: Attempts silent sign-in using the broker.
- `_signin_interactively(...)`: Performs interactive sign-in.
- `_acquire_token_silently(...)`: Acquires token silently for a given account.
- `_signout_silently(...)`: Signs out an account silently.

Sequence Diagram: Interactive Sign-In



Example: Error Conversion Flow

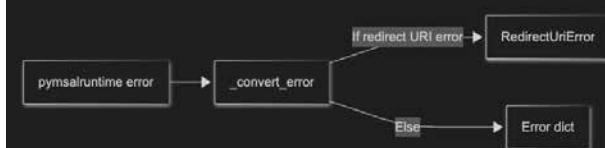


Figure 10: Automated documentation rocks

And you gain faster onboarding. New team members can quickly understand existing codebases. As a new team member myself, I find the ultimate source of truth to be code, not docs. But if I can paste a big function into an LLM and have it document it for me, I'm happy with those superpowers.

Back to the MSAL Python library, or feel free to pick any real codebase. I opened a particularly wavy part of the code, which is `broker.py`. I tried reading it, but it made my head hurt.

I asked GitHub Copilot with the following command in agentic mode.

Please document this code for me with diagrams in a file called `broker.md`.

In a matter of seconds, a new `broker.md` file is created for me. You just have to take my word for it; this is about the best documentation I've seen, complete with diagrams. I've pasted a partial screenshot of the documentation in **Figure 10** for your review.

The diagrams you see are mermaid diagrams rendered visually inside VSCode using the mermaid extension.

Intelligent Troubleshooter

Debugging is an infamous time sink for developers. Debugging is twice as hard as writing code as some famous person once said. AI is increasingly capable of analyzing error messages, tracing code execution, and suggesting potential fixes or even refactoring solutions.

This is again yet another superpower you have with AI. AI will quickly point you to the likely source of errors. This is especially useful in large code bases that you may be unfamiliar with. AI can offer you intelligent suggestions. After it analyzes your code, it offers direct code fixes or alternative approaches. And finally, I don't know about you, but I really dislike taking an error message, searching Google for it, or spending time looking at complex stack traces. AI takes all that work away for me.

For my next code example, I used AI to conjure up an application that decodes a JWT token. And I deliberately introduced an error in it. The typical typo error. I won't tell you what the error is: Let it remain a mystery.

This is a React.JS app I had instructed AI to build for me, so I ran it using `npm`. When I run the application, I'm greeted with an ugly error, as follows.:

```
[plugin:vite:import-analysis]
Failed to resolve import
"./Ap.jsx" from "src/main.jsx".
Does the file exist?
```

Now, those of you who spend your life in React immediately know what the issue is. But humor me here. There are React errors that can be quite cryptic. And, for a moment, let's assume I'm a developer who's not too familiar with React, I lied on my resume, got hired, and now I

The import in `main.jsx` has been fixed to use `@ App.jsx` instead of the incorrect `./Ap.jsx`. This will resolve the "Does the file exist?" error. Your app should now load correctly. Let me know if you need any more help!

Figure 11: Automated bug fixing

need to fix this bug so I can blame it on the last developer and get credit for fixing it.

So off I go to GitHub Copilot, and send the following prompt in agentic mode:

Please fix this bug "Failed to resolve import "./Ap.jsx" from "src/main.jsx". Does the file exist?"

This sets Copilot thinking, and, in a matter of seconds, it fixes my error. The output of Copilot is shown in **Figure 11**.

And it even shows me in a diff format exactly the change it made in the file, as can be seen in **Figure 12**.

And I can verify that my application is back to working!

Using AI to Learn New Stuff

This tip is not specific to developers but is certainly applicable to developers. What's the number one thing developers have to do? Learn new stuff.

In today's fast-paced world, continuous learning isn't just an advantage, it's a necessity. Whether you're aiming to master a new programming language, delve into complex scientific theories, acquire a creative skill, or simply broaden your general knowledge, artificial intelligence has emerged as an incredibly powerful ally. Gone are the days of rote memorization or one-size-fits-all textbooks; AI offers personalized, interactive, and highly efficient ways to learn almost anything.

Because this tip isn't specific to developers, I'll intentionally pick a cumbersome topic. I do a lot of my learning from YouTube. There's a podcast I particularly like called **Adam Taggart: Thoughtful Money**. Let's pick one video as an example: <https://www.youtube.com/watch?v=WSpR770JvXg>

As much as I really like Adam's videos, who has one hour and forty-one minutes to listen to it without falling asleep? Look, this isn't my full-time job. Just give me the key points and skip the chit chat.

```
src > main.jsx
1 import { StrictMode } from 'react'
2 import { createRoot } from 'react-dom/client'
3 import './index.css'
4 import App from './App.jsx'
5 import App from './App.jsx'

6 createRoot(document.getElementById('root')).render(
7   <StrictMode>
8     | <App />
9     | </StrictMode>,
10   )
11 |
```

Figure 12: My bug fixed by AI

This video features an interview with Michael Green, Portfolio Manager and Chief Strategist at Simplify Asset Management. He discusses the current state of the stock market, labor force, and economy, as well as the potential impacts of new administration policies [00:32].

Key points from the discussion include:

- **Passive Capital Flows:** Green observes that passive capital flows are flattening, with combined flows across mutual funds and ETFs potentially turning down [01:21]. This is partly attributed to increased unemployment, though currently offset by the shift from active to passive management [01:35].
- **Multiplier Effect:** Green explains that money invested with an active manager has a multiplier of about two on the aggregate market cap, while passive funds have a much higher multiplier (17 to 20) [02:55].
- **Factors Affecting Passive Flows:** The flattening of passive flows is linked to employment levels, as fewer employed individuals mean less 401k contributions [06:52].
- **Market Events and Passive Investing:** Green references a market event on August 24, 2015, where chaos in US equity markets was initially attributed to Chinese currency devaluation but was actually caused by Vanguard changing allocations in their target date funds [10:10].
- **Future Concerns:** Green expresses concern about the potential for rising continuing claims for unemployment, which historically have exploded higher after the unemployment rate bottoms out [17:23]. He suggests this trend would make the situation with passive capital flows "a lot more dire" [01:39:52].

Figure 13: A crisp summary of a long rambling video

To save me time, I go to gemini.google.com and say "Please summarize <video url>", and Gemini immediately gives me a nice little summary, with time stamps. This can be seen in **Figure 13**.

This is absolutely incredible. If there are points of interest, I can go to specific timestamps and listen to that part. This is a huge time saver. And I can take it a step further.

I went to notebooklm.google.com and created a new notebook and imported this video as source. Note that you can also import multiple sources. Now I can start interacting with that video by chatting with it. Let's give it a try.

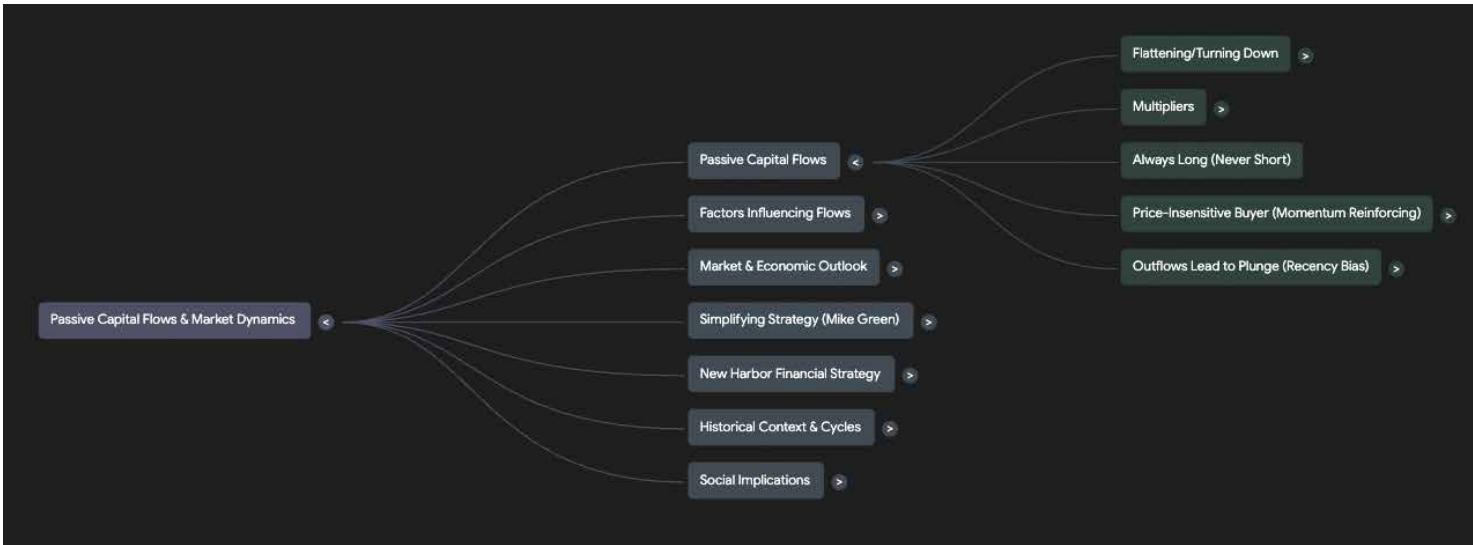


Figure 14: Learning is supercharged with AI.

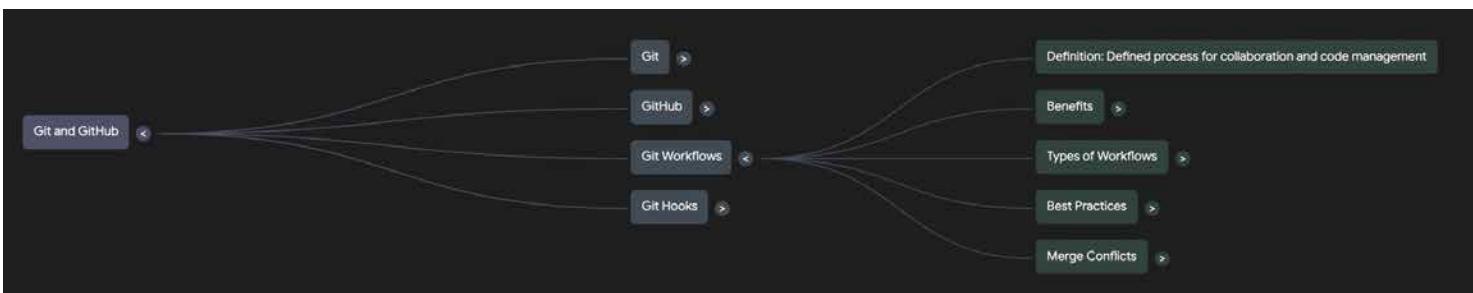


Figure 15: Git mind map

I asked, “How did Vanguard’s changing allocations in their target date funds affect the markets?” and notebooklm immediately gave me a very crisp to-the-point answer with references to sources as tool tips.

This video is full of knowledge and I wish to dive further. There’s a lot you can do here: You can discover new sources, turn this into a briefing, or into a Q&A

for a study guide. One of things I find really helpful is turning this knowledge into a mind map. I went ahead and generated a mind map for the content in this video.

In a matter of seconds, I’m given an interactive mind map that allows me to drill down to topics of interest, as can be seen in **Figure 14**.

When you perform a `git pull` command, it by default performs a merge to integrate changes from the remote repository into your local branch ①. However, you can force `git pull` to integrate the remote branch with a rebase by passing it the `--rebase` option ②.

The command would look like this:

```
* git pull --rebase ①
```

Why choose rebasing instead of merging for `git pull`? The main benefit of rebasing is that it results in a much cleaner and perfectly linear project history ② ③. It moves your entire feature branch to begin on the tip of the main (or develop) branch, effectively incorporating new commits without creating extra “merge commit” entries ②. This can make it easier to navigate your project’s history with commands like `git log` ②.

Important Considerations: The Golden Rule of Rebasing It is crucial to understand when not to use `git rebase` ④. The “golden rule of `git rebase` is to never use it on public branches” ④. If you rebase a branch that other developers are already working from, it will rewrite the project history in your repository, making Git think your branch’s history has diverged from everyone else’s ⑤. This can lead to a confusing situation where synchronizing branches requires extra merge commits and creates duplicate sets of changes ⑤. Therefore, if others are looking at the branch, you should consider non-destructive ways to make changes, such as `git revert` ⑥.

Figure 16: Pretty good help on rebasing with Git

I can keep expanding collapsing, learning things, and going to exact sources and quotes out of the video. In fact, I can combine many such sources together and create a unified knowledge base that I can learn from.

How would you, as a developer, benefit from this? First of all, if I need to learn something out of my area of expertise, like "How to maximize the growth of lettuce in my garden," this is a huge time saver. But even in purely technical terms, if I want to learn about "Git," I go ahead and ask the notebook to discover sources for "Git," and ask it to generate a mind map.

The mind map for Git can be seen in [Figure 15](#).

Or if I wanted to do something complicated with Git, I could simply ask a question in the interactive chat. For example, I asked:

 "How do I rebase when I do a Git pull automatically?"

It gave me a proper answer with exact commands and explanations, along with sources and additional learning I should consider around this command. This can be seen in [Figure 16](#).

Summary

The integration of AI tools into the developer workflow is no longer a futuristic concept but a present-day reality driving unprecedented levels of productivity. From intelligent code completion and automated testing to smarter documentation and even autonomous task execution, AI is transforming how software is built. By embracing these powerful tools judiciously, developers can amplify their capabilities, reduce tedious tasks, and dedicate more energy to the creative and strategic challenges that define truly impactful software engineering. The future of development is undeniably AI-augmented, and those who master its application will be at the forefront of innovation.

This article, although touching many aspects, barely scratches the surface of the possibilities. AI is incredibly powerful and it will supercharge your workflows. I highly encourage you to start integrating these into your daily coding habits.

Or, you know, just get left behind.

Until next time, happy coding!

Sahil Malik
CODE



Instantly Search Terabytes

Enterprise and developer products have:

- over 25 different search features
- credit card search and other forensics-oriented options
- efficient multithreaded indexing and searching

dtSearch's document filters support:

- local and remote "Office" files, PDFs, compression formats, etc.
- emails with multilevel attachments
- a wide variety of databases
- web data

Developers:

- SDKs for Win / Linux / macOS
- APIs for current .NET / C++ / Java
- faceted search, granular data classification and other API options
- deploy on-premises or in cloud

Visit [dtSearch.com](#)

- **for hundreds of reviews and case studies**
- **for fully-functional enterprise and developer evaluations**

The Smart Choice for Text Retrieval® since 1991

dtSearch.com | 1-800-IT-FINDS

Introduction to JavaScript Templating

If you do a lot of client-side manipulation of the DOM using JavaScript or jQuery, you might want to consider using one of the many templating engines available. I'm sure you've built some HTML within a string in your JavaScript code. Using string concatenation to build a set of HTML elements leads to hard-to-read and hard-to-debug code. Instead of using string



Paul D. Sheriff

www.pdsa.com

Paul has been working in the IT industry since 1985. In that time, he has successfully assisted hundreds of companies' architect software applications to solve their toughest business problems. Paul has been a teacher and mentor through various mediums such as video courses, blogs, articles and speaking engagements at user groups and conferences around the world. Paul has multiple courses in the www.pluralsight.com library (<https://bit.ly/3gvXgyj>) and on YouTube.com (<https://www.youtube.com/@pauldsheriff>) on topics ranging from C#, LINQ, JavaScript, Angular, MVC, WPF, XML, jQuery, and Bootstrap. Contact Paul at psheriff@pdsa.com.



concatenation, take advantage of a templating engine. A templating engine separates the HTML to generate from your code and data, which makes your code easier to read and makes it much easier to locate bugs. As every programmer is aware, separation of concerns (SoC) is a best practice. When you employ SoC in your applications it immediately increases the readability and maintainability of your application. In this article, I'll show you several techniques to make your client-side scripting simpler, easier to read, maintain, and debug.

The Problem with String Concatenation

A simple example to illustrate the issue with using string concatenation is if you wish to dynamically generate an HTML `<input>` element with the `value` attribute initialized to a movie name, as shown in **Figure 1**.

Create a file named `index.html` in a folder on your hard drive and add the HTML shown in **Listing 1**. Notice the `<div>` element with the `id` attribute set to "output". In the `<script>` tag, you're going to write some code to dynamically create an `<input>` element with the value from a property inserted into its `value` property.

In the `<script>` tag, add a `window.onload()` function, **Listing 2**, with an object literal called `movie`. This `movie` object contains two properties: `filmId` and `name`. A method named `buildHTML()` is also a part of this object. This method uses string concatenation to create the `<in-`

`put>` element that is insert into the `innerHTML` property of the `<div id="output">` element. Note that I must break the return statement onto multiple lines for purposes of formatting in this article, however, this must all be on a single line when you type it into your editor.

If you were to type in the code exactly as shown in **Listing 2** and display the index page, no `<input>` would appear within the `<div>` element. Why? Because the closing single quote that surrounds the movie name in the `value` attribute is missing. This is a very hard bug to find because you are using string concatenation, and all the single and double quotes are hard to match up, as shown in **Figure 2**.

There are quite a few problems with using string concatenation. A long line cannot be broken into multiple lines, so you are constantly scrolling horizontally to see the entire line. All the text within the string is a single color, which makes it hard to identify what the different parts of the element are. To add variables into the line, you must use the plus sign (+). Strings in JavaScript are immutable, so the string is constantly being moved to a new location to account for the increase in size. While typing into the string, no IntelliSense is available, so you are relying on your typing skills. If double and single quotes are required within the string, you need to escape one or the other. As you can see, this is a lot of issues when working with string concatenation.

Use Template Literals

Template literals (sometimes called template strings) can solve a couple of the issues with string concatenation. Template literals are delimited with a back tick (`) on each side of the string. If single and double quotes are required within this string, they can be used without escaping those characters. Another advantage is you can split a template literal across multiple lines. This eliminates the amount of horizontal scrolling necessary to see the whole code.

In the `buildHTML()` method you wrote previously, modify this code to use a template literal instead of the string concatenation. The return statement now looks like the following code snippet:

```
"buildHTML": function () {
    return `<input type='text'
        id='movieName'
        name='movieName'
        value='${this.name}' />`;
}
```

Using String Concatenation

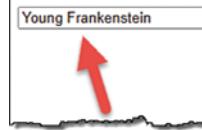


Figure 1: Insert an `<input>` element by building the HTML dynamically.



Figure 2: Missing quotes are very hard to find when performing string concatenation.

This code snippet may be typed in exactly as shown across multiple lines because this is supported when using template literals. Notice the replaceable token \${this.name} within the literal. This syntax allows the value from the **name** property in the **movie** object to be placed into this location within the string. If the closing single quote were to be left out, this error would be much easier to see.

What Is a Templating Engine?

A templating engine is like a template literal in that HTML is combined with some replaceable tokens with a variable name into which data from that variable is replaced. What does a replaceable token look like? Each templating engine can have different tokens, but they all usually start with a few characters, have the name to replace, and then end with a few opposite characters. In a template literal, the token was a dollar sign (\$), followed by an opening curly brace, then the name of a variable, and closed with a closing curly brace.

Many templating engines use two open curly braces to specify the start of a token, as shown in the following snippet. The property name to be replaced, **genre**, is then typed out. The token is closed by using two closing curly braces.

```
 {{genre}}
```

Other examples of tokens from other templating engines look like the following:

```
<%= genre %>
{%
  genre
}%
```

To replace the name **genre** with some real data, use a templating engine and send it a JSON object that contains **genre** as one of its properties, as shown in the following code snippet:

```
{
  "name": "Young Frankenstein",
  "genre": "Comedy"
};
```

The templating engine performs its magic and replaces the **genre** variable in the token with the value of **Comedy** from the JSON object. These tokens are typically inserted directly into some HTML, for example:

```
<input id="genre" name="genre"
      type="text" value="{{genre}}"/>
```

Build an HTML Project

To truly understand the benefits of using a templating engine, let's build a simple token replacement system using JavaScript. Create a new folder named **javascript-Samples**. Under this **javascriptSamples** folder, create a new folder named **scripts**. Add a new file in the **scripts** folder named **common.js** and add the following functions into this file:

```
function getValue(id) {
  return document.getElementById(id).value;
```

Listing 1: Create a web page to dynamically add an HTML <input> element.

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8" />

  <title>Using String Concatenation</title>
</head>

<body>
  <header>
    <h1>Using String Concatenation</h1>
  </header>

  <main>
    <div id="output">
    </div>
  </main>

  <script>
    </script>
</body>

</html>
```

Listing 2: In the onload() function is where you build the <input> element.

```
window.onload = function () {
  let movie = {
    "filmId": 1,
    "name": "Young Frankenstein",
    "buildHTML": function () {
      return "<input type='text' id='movieName' name='movieName' value='" + this.name + "' />";
    }
  };

  document.getElementById("output")
    .innerHTML = movie.buildHTML();
}
```

```
}
```

```
function getElement(id) {
  return document.getElementById(id);
}

function formatCurrency(value) {
  return new Number(value)
    .toLocaleString("en-US",
      { "style": "currency",
        "currency": "USD" });
}
```

These functions are just wrappers around some very common JavaScript code. The first function, **getValue(id)**, retrieves the value from an element based on the **id** you pass in. The second function, **getElement(id)**, retrieves the DOM object based on the **id** you pass in. The third function, **formatCurrency(value)**, takes the value you pass in and returns it formatted as a US currency value.

Create Some Styles

Create a folder named **styles** and add a new file in this styles folder named **site.css**. Add three CSS rules (**Listing 3**) to format the label and input elements you're going to use in the samples for this article. The last CSS rule is

Listing 3: Add some styles to make the web page look nicer.

```
.form-label {
  display: inline-block;
  min-width: 8em;
  margin-bottom: 1em;
}

.form-control {
  font-family: inherit;
  font-size: 100%;
}

.movieInfo {
  display: block;
  margin-top: 2em;
  font-weight: bold;
  font-size: larger;
}
```

used to display a movie name from the data input into the movie name field.

Add a Sample Web Page

Add a file in the javascriptSamples folder named **index.html**. Add the code shown in **Listing 4** into this new file. This code displays a form for the entry of a movie name, and a button to click to display the name entered. This movie name will be displayed in a label that's dynamically added to the DOM within the `<div id="output">` element.

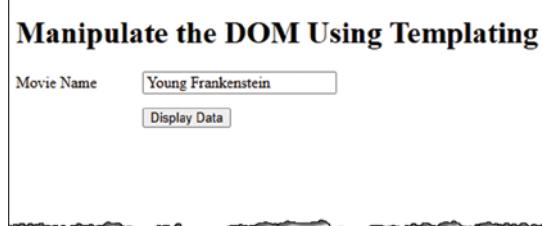


Figure 3: Create a page to help test JavaScript code.

Try It Out

Display the `index.html` file in your browser and it should look like **Figure 3**. The button does not do anything yet; you'll add that code shortly.

Build a Single Label and Add to DOM Using JavaScript

On the `index.html` page, you see a `<div>` with its `id` attribute set to `output`.

```
<div id="output">
</div>
```

The goal of the `displayData()` function is to insert a label into this `<div>` with the movie name input into the text box. The `<label>` is given the CSS class, `movieInfo`, so it stands out on the page, as shown in **Figure 4**. Modify the `displayData()` function so it looks like the following:

```
function displayData() {
  // Create a label with the movie name
  getElement("output").innerHTML =
    '<label class="movieInfo">' +
      ${getValue("name")}
    '</label>';
}
```

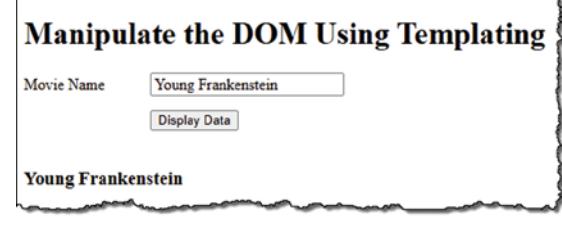


Figure 4: Add a label with the movie name to the `<div>` element.

Listing 4: Create a web page to allow input of a movie name.

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8" />
  <title>
    Manipulate the DOM Using Templating
  </title>
  <link href="styles/site.css" rel="stylesheet" />
</head>

<body>
  <header>
    <h1>
      Manipulate the DOM Using Templating
    </h1>
  </header>

  <main>
    <div>
      <label for="name" class="form-label">
        Movie Name
      </label>
      <input id="name" name="name"
            value="Young Frankenstein"
            class="form-control" />
    </div>
    <div>
      <label class="form-label"></label>
      <button onclick="displayData();">
        Display Data
      </button>
    </div>
    <div id="output">
    </div>
  </main>

  <script src="scripts/common.js">
  </script>
  <script>
    'use strict';

    function displayData() {
    }
  </script>
</body>

</html>
```

The code in the `displayData()` function retrieves the `<div>` element object and replaces its `innerHTML` property with a string. The string is built using a template literal with a `<label>` element, the value from the input element with the `id` attribute set to `name`, and the closing `</label>` element.

Try It Out

Display the `index.html` file in your browser, enter a name of a movie or take the one already defaulted, and click on the **Display Data** button. Clicking the button calls the `displayData()` function, which builds the `<label>` element with the movie name in it. If you have entered everything correctly, your web page should look like **Figure 4**.

Check for No Input and Display Appropriate Label Using JavaScript

You may wish to ensure that the user entered a movie name into the input element. If they do, display the movie name just like you did in the last example. If they don't enter a movie name, display a message to the user that no movie name was input. Modify the `displayData()` function to look **Listing 5**.

In this function, create a variable named `html` into which you build the string to add to the output `<div>`. Retrieve the value entered in the input element and place it into a variable called `name`. Check to see if `name` has a value and build the `<label>` with the value, or if it doesn't have a value, build the `<label>` with the error message. Finally, add the value in the `html` variable to the `innerHTML` property of the `<div>` to display the result.

You may be wondering why you're writing this simple code. Later in this article, you are going to perform this type of logic using templating engine tokens to check for true or false values and display two different values based on the result.

Try It Out

Display the `index.html` file in your browser, wipe out the movie name and click the **Display Data** button to see the error message show up, as shown in **Figure 5**. After doing this, put in a movie name, click the **Display Data** button, and ensure that the movie name is still displayed.

Build Multiple Labels to Add to DOM Using JavaScript

The code you have written so far in JavaScript hasn't been too bad and is easy to read. However, as you add more elements to the dynamic HTML you want to build, this code can become harder to manage. Open the `index.html` file and replace the `<main>` element with the HTML shown in **Listing 6**.

Replace the `displayData()` function with the code shown in **Listing 7**. As you can see, the code that builds all the `<label>` and `
` elements is harder to read compared to the previous code. There's more opportunity to miss a closing bracket, or a forward slash because you're mixing code and HTML. Using template literals helps immensely compared to normal string concatenation, but you still don't have color coding or IntelliSense.

Listing 5: Add some logic to the `displayData()` function to display an error message if no movie name is entered.

```
function displayData() {
  let html = "";
  let name = getValue("name");
  if (name) {
    html = `<label class='movieInfo'>
      ${name}
    </label>`;
  } else {
    html = `<label class='movieInfo'>
      No Movie Input
    </label>`;
  }
  // Add label to div
  getElement("output").innerHTML = html;
}
```

Manipulate the DOM Using Templating

Movie Name
Display Data

No Movie Input

Figure 5: Display an error message to the user when no data is input.

Manipulate the DOM Using Templating

Movie Name
Movie Rating
Movie Genre
Year Released
Rental Price
Display Data

Young Frankenstein
PG
Comedy
1974
\$3.99

Figure 6: You may build as many labels as you wish and add them to the `<div>`.

Try It Out

Display the `index.html` file in your browser, click on the **Display Data** button, and your page should look like **Figure 6**.

Use an Object Literal to Add Elements to DOM Using JavaScript

Before you start to learn about template engines, there's one more JavaScript technique you can employ to help make your code even more readable. Use a literal object, named `movie`, as shown in **Listing 8**, and populate each property of the object from the data entered. Add a `buildHTML()` method into this object to build your string with the `<label>` elements to display the movie information.

Listing 6: Add more input elements to the web page.

```
<main>
  <div>
    <label for="name"
      class="form-label">
      Movie Name
    </label>
    <input id="name" name="name"
      value="Young Frankenstein"
      class="form-control" />
  </div>
  <div>
    <label for="rating"
      class="form-label">
      Movie Rating
    </label>
    <select id="rating"
      class="form-control">
      <option>G</option>
      <option selected>PG</option>
      <option>PG-13</option>
      <option>R</option>
    </select>
  </div>
  <div>
    <label for="genre"
      class="form-label">
      Movie Genre
    </label>
    <input id="genre" name="genre"
      value="Comedy" class="form-control" />
  </div>
  <div>
    <label for="year"
      class="form-label">
      Year Released
    </label>
    <input id="year" name="year"
      value="1974"
      class="form-control" />
  </div>
  <div>
    <label for="price"
      class="form-label">
      Rental Price
    </label>
    <input id="price" name="price"
      value="3.99"
      class="form-control" />
  </div>
  <div>
    <label class="form-label">
      Display Data
    </label>
    <button onclick="displayData();">
      Display Data
    </button>
  </div>
  <div id="output">
  </div>
</main>
```

Listing 7: Add more code to display more labels.

```
function displayData() {
  let name = getValue("name");
  let rating = getValue("rating");
  let genre = getValue("genre");
  let year = getValue("year");
  let price =
    formatCurrency(getValue("price"));

  // Build movie information in labels
  let html =
    <label class='movieInfo'>${name}</label>
    <label>${rating}</label><br />
    <label>${genre}</label><br />
    <label>${year}</label><br />
    <label>${price}</label><br />`;

  // Add movie labels into <div>
  getElement("output").innerHTML = html;
}
```

Listing 8: Use an object literal to wrap up the building of the HTML for the movie data display.

```
function displayData() {
  let movie = {
    "name": getValue("name"),
    "rating": getValue("rating"),
    "genre": getValue("genre"),
    "year": getValue("year"),
    "price": getValue("price"),
    "buildHTML": function () {
      return
        <label class='movieInfo'>
          ${this.name}
        </label>
        <label>${this.rating}</label><br />
        <label>${this.genre}</label><br />
        <label>${this.year}</label><br />
        <label>${formatCurrency(this.price)}</label><br />;
    }
  };

  // Add movie labels into <div>
  getElement("output").innerHTML =
    movie.buildHTML();
}
```

Try It Out

Display the `index.html` file in your browser, click on the **Display Data** button, and your page should still look like [Figure 6](#).

Place HTML to Generate in a Template Element

A literal object makes the JavaScript code easier to read and maintain. However, you're still mixing code and HTML. Since 2015, the HTML language has included a `<template>` element that's meant to hold HTML for use with templating engines. Prior to this, developers used the `<script>` element to hold this HTML, as shown in the following code snippet.

```
<script id="template"
  type="text/html">
<input id="name"
  name="name"
  type="text"
  value="{{name}}">
</script>
```

It's recommended to use the `<template>` element moving forward, as it semantically helps distinguish between what is script and what is a template of HTML, as shown in the following snippet:

```
<template id="template"
  type="text/html">
<input id="name"
  name="name"
  type="text"
  value="{{name}}">
</template>
```

Using the `<template>` element, your editor can now format the HTML and provide color coding ([Figure 7](#)). There's no

```
<template id="template" type="text/html">
  <input id="movieName" name="movieName"
    value="{{name}}>
</template>
```

Figure 7: Placed into a template element, the editor will give you color coding.

color coding in JavaScript strings to distinguish between code and HTML (**Figure 8**). In the template, wrap your replaceable items with a token, in this case, `{{name}}` , so they can be replaced using JavaScript.

Once the HTML is in a `<template>` element, read the `innerHTML` property to retrieve the HTML. After reading the HTML with the token in it, search for the token, and replace it with the value from the `movie.name` property, as shown in the following code snippet:

```
window.onload = function () {
  let movie = {
    "filmId": 1,
    "name": "Young Frankenstein",
    "buildHTML": function () {
      let html = getElement("template")
        .innerHTML;

      return html.replace("{{name}}",
        movie.name);
    }
  };

  getElement("output").innerHTML =
    movie.buildHTML();
}
```

If you run this code, you'd have an `<input>` on the web page with the value of "Young Frankenstein" in the text box. You now have a nice separation of concerns between the HTML and the code with the data. This is much easier to read, understand, maintain, and debug.

Use a Template Element to Add Multiple Elements to DOM Using JavaScript

Go back to your web project and replace the code in the `displayData()` method to use a `<template>` element. Just after the closing `</main>` tag, add the following code. Assign a unique `id` attribute to the `<template>` tag. It isn't necessary to add the attribute `type` to this `<template>` tag with the value set to "text/html". However, I like adding this attribute, as it makes this `<template>` element self-documenting.

```
<!-- Create a template for
  displaying movie information -->
<template id="template" type="text/html">
  <label class="movieInfo">{{name}}</label>
  <label>{{rating}}</label><br />
  <label>{{genre}}</label><br />
  <label>{{year}}</label><br />
  <label>{{price}}</label><br />
</template>
```

```
<script>
  window.onload = function () {
    let movie = {
      "filmId": 1,
      "name": "Young Frankenstein",
      "buildHTML": function () {
        return `<input type='text'
          id='movieName'
          name='movieName'
          value='${this.name}' />`;
      }
    };
  };
}
```

Figure 8: You do not get any color coding when HTML is in a string.

Locate the `displayData()` function in the `<script>` tag and replace the `buildHTML()` method with the code shown below. The first line of code retrieves the HTML from the `<template>` element. The succeeding lines of code perform a search and replace on each of the tokens contained in the HTML template.

```
"buildHTML": function () {
  // Get the template HTML
  let html = getElement("template").innerHTML;

  // Perform search and replace on movie info
  html = html.replace("{{name}}", this.name);
  html = html.replace("{{rating}}",
    this.rating);
  html = html.replace("{{genre}}", this.genre);
  html = html.replace("{{year}}", this.year);
  html = html.replace("{{price}}",
    formatCurrency(this.price));

  return html;
}
```

Try It Out

Display the `index.html` file in your browser, click on the `Display Data` button, and your page should still look like Figure 6.

Use a Template to Add a Row to a Table Using JavaScript

As another example of using a `<template>` element to dynamically add HTML to the DOM, let's build an HTML table from the data entered on the page. Replace the `<div id="output">` element on the index page with the following `<table>` element:

```
<table id="movieTable">
  <thead>
    <tr>
      <th>Movie Name</th>
      <th>Rating</th>
      <th>Genre</th>
      <th>Year Released</th>
      <th class="pull-end">Price</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>{{name}}</td>
      <td>{{rating}}</td>
      <td>{{genre}}</td>
      <td>{{year}}</td>
      <td>{{price}}</td>
    </tr>
  </tbody>
</table>
```

```
</thead>
<tbody>
</tbody>
</table>
```

Replace the `<template>` element with the following HTML. As you enter this HTML, notice how you get IntelliSense to help you enter the elements. You do not get this when you are building a string in JavaScript. Using a `<template>` element once again has great advantages over using a string.

```
<!-- Create a template for
     a row of movie data -->
<template id="template" type="text/html">
  <tr>
    <td>{{name}}</td>
    <td>{{rating}}</td>
    <td>{{genre}}</td>
    <td>{{year}}</td>
    <td class="pull-end">{{price}}</td>
  </tr>
</template>
```

Movie Name	Rating	Genre	Year Released	Price
Young Frankenstein	PG	Comedy	1974	\$3.99

Figure 9: Build an HTML table by replacing the input element values into the tokens in the template element.

Listing 9: Add styles for the HTML table.

```
table {
  border-collapse: collapse;
  margin-top: 2em;
}

table,
th,
td {
  border: 0.02em solid black;
}

table th,
table td {
  padding: 0.5em;
}

th {
  text-align: left;
}

.pull-end {
  text-align: right;
}
```

Because you removed the `<div id="output">` and you have a table, you must change the code that adds the results from the `buildHTML()` method. Locate these last two lines of code in the `displayData()` function.

```
// Add movie labels into <div>
getElement("output").innerHTML =
  movie.buildHTML();
```

Remove those two lines of code and add the following lines to add the HTML as a row of data in the HTML table. This code gets an object reference to the `<table>` element. The `<tbody>` object reference is retrieved and placed into the `body` variable. The `insertRow()` method is invoked on the `body` object passing in the total number of rows in the table. Call the `movie.buildHTML()` method to create the HTML and assign that result into the `innerHTML` property of the row object.

```
// Get reference to the table element
let table = document
  .getElementById('movieTable');

// Get reference to the tbody element
let body = table
  .getElementsByTagName('tbody')[0];

// Create a blank row
let newRow = body.insertRow(body.rows.length);

// Add Generated HTML to the new row
newRow.innerHTML = movie.buildHTML();
```

Add Styles for the Table

To make our table look somewhat decent compared to the default table styles the browser uses, add some styles to your project. Open the `styles\site.css` file and add the CSS rules shown in Listing 9 into this file.

Try It Out

Display the `index.html` file in your browser, click on the **Display Data** button, and your web page should look like Figure 9.

Looping to Build a Table

In many applications, you'll be dealing with an array of data that you must load into a table or similar list element. To load a table of HTML data from an array, use the same `<template>` and `<table>` elements. Remove all the input fields from the `<main>` element, and just leave the HTML table, as shown in the following code snippet:

```
<main>
  <table id="movieTable">
    <thead>
      <tr>
        <th>Movie Name</th>
        <th>Rating</th>
        <th>Genre</th>
        <th>Year Released</th>
        <th class="pull-end">Price</th>
      </tr>
    </thead>
    <tbody></tbody>
```

Manipulate the DOM Using JavaScript

Movie Name	Rating	Genre	Year Released	Price
2012	PG-13	Action	2009	\$3.95
9 to 5	PG	Comedy	1980	\$2.95
African Queen, The	G	Adventure	1952	\$3.95

Figure 10: Loop through an array to build an HTML table.

```
</table>  
</main>
```

Remove the `displayData()` function within the `<script>` element and add an array, as shown in Listing 10. This array, named **movies**, contains a collection of movie object literals. Each object literal contains the five properties as the movie object literal used in the previous samples.

Just below this array, and before the closing `</script>` tag, add a `window.onload()` function, as shown in Listing 11. When this web page is loaded, the `onload()` method is called. In this method, retrieve a reference to the table, from which you can get a reference to the `<tbody>` element. Also, retrieve the HTML contained in the `<template>` element and place that into a variable named **template**. Use a `foreach()` method to iterate over each element in the **movies** array. On each iteration, a movie object is placed into the **row** variable.

For each row in the array, assign to a variable named **html** the HTML from the **template** variable. Take the data from the **row** variable and replace each property into the appropriate token contained in the template. This is similar code to that created when loading a single row into the table. Insert a blank row at the end of the rows in the `<tbody>` and insert the HTML generated from the current movie object into this new row. After all objects have been created, the HTML table is built and should look like Figure 10

Try It Out

Display the `index.html` file in your browser to see the HTML, as shown in Figure 10.

The Mustache.js Templating Engine

To illustrate a templating engine, I'm going to use Mustache.js. Mustache.js is a JavaScript library that became the standard for most other templating engines. Check out Mustache at <https://github.com/janl/mustache.js>. You may wonder why it's called mustache. It's because the tag or token delimiters are curly braces. Rotate the opening curly brace 90 degrees to the right and it looks like a mustache. Mustache is what's known as a logic-less template syntax. Mustache doesn't contain any if, else, or looping statements; there are only tokens. When you read the mustache documentation, they refer to `{{name}}` as a tag. I prefer to use the name token, as in HTML, you already have a tag such as the `<template>` tag.

Build an HTML Project Using Mustache

Create a new folder to place the samples for working with Mustache. Keep these new samples separate from

Listing 10: Add an array of movie data to the `<script>` element.

```
<script>  
  'use strict';  
  
  let movies = [  
    {  
      "name": "2012",  
      "rating": "PG-13",  
      "genre": "Action",  
      "year": "2009",  
      "price": 3.95  
    },  
    {  
      "name": "9 to 5",  
      "rating": "PG",  
      "genre": "Comedy",  
      "year": "1980",  
      "price": 2.95  
    },  
    {  
      "name": "African Queen, The",  
      "rating": "G",  
      "genre": "Adventure",  
      "year": "1952",  
      "price": 3.95  
    }  
  ];  
</script>
```

Listing 11: Write code in the `window.onload()` to create an HTML table.

```
window.onload = function () {  
  // Get reference to the table element  
  let table = getElement('movieTable');  
  // Get reference to the tbody element  
  let body = table  
    .getElementsByTagName('tbody')[0];  
  // Get the template HTML  
  let template = getElement("template")  
    .innerHTML;  
  
  movies.forEach(row => {  
    // Assign template to html variable  
    let html = template;  
  
    // Perform search and replace on movie info  
    html = html.replace("{{name}}", row.name);  
    html = html.replace("{{rating}}",  
      row.rating);  
    html = html.replace("{{genre}}", row.genre);  
    html = html.replace("{{year}}", row.year);  
    html = html.replace("{{price}}",  
      formatCurrency(row.price));  
  
    // Create a blank row  
    let newRow = body  
      .insertRow(body.rows.length);  
    // Add Generated HTML to the new row  
    newRow.innerHTML = html;  
  });  
}
```

the JavaScript samples previously created so you can compare the two side-by-side. Create a new folder named **mustacheSamples**. Copy the folder named **scripts** from the **javascriptSamples** folder into the **mustacheSamples** folder. Copy the **styles** folder from the **javascriptSamples** folder into the **mustacheSamples** folder. You now need to get the `mustache.js` library onto your computer so you can copy it into the **scripts** folder in the **mustacheSamples** folder.

Install mustache.js Using Node Package Manager (npm)

If you have `npm` installed on your computer, create a directory on your computer named **mustache**. Open a com-

mand window in the new **mustache** folder and execute the following command:

```
npm install mustache -save
```

After running this command, you'll find the **mustache.min.js** file located in the `\mustache\node_modules\mustache` folder. Copy that file into the scripts folder in the `mustacheSamples` folder.

Install mustache.js by Downloading from GitHub

If you do not have npm on your computer, you can download the `mustache.min.js` file from my GitHub folder located at <https://github.com/PaulDSheriff/Training-Samples/tree/main/mustacheJS>. Download the `mustache.min.js` file to the scripts folder in the `mustacheSamples` folder.

Create the Index Page

Create an `index.html` file in the `mustacheSamples` folder and add the HTML shown in **Listing 12**. This HTML should look familiar, as it's almost identical to the `index.html` file first created in this article. The major difference is the reference to the `mustache.min.js` file.

Try It Out

Display the `index.html` page in your browser and it should look like **Figure 3**.

Build a Single Label and Add to DOM Using Mustache

In the `index.html` file, just after the closing `</main>` tag, add a `<template>` element. Into this `<template>` element, you add the label into which you will take the value entered and replace that value into the `{{name}}` token.

```
<!-- Create a template for
    displaying movie information -->
<template id="template" type="text/html">
```

```
<label class="movieInfo">
  {{name}}
</label>
</template>
```

Modify the `displayData()` function to read the HTML from this `<template>` element and send it to the Mustache engine for replacing the tokens with data. The global Mustache object is defined in the `mustache.min.js` file and has a `render()` method. The `render()` method accepts two arguments; the template string, and an object literal. The object literal contains properties with the same name as those used in each token. Mustache takes care of performing searches for each token and replacing it with data from the object literal. No need to call the individual `replace()` methods to build the HTML; Mustache takes care of that for you. As you can see, this is greatly simplified code from what you used to write.

Try It Out

Display the `index.html` file in your browser, enter a name of a movie or take the one already defaulted, and click on the **Display Data** button. Clicking the button calls the `displayData()` function that uses Mustache to build the web page that looks like **Figure 4**.

Check forTruthy Expressions Using a Section

In addition to tokens that render a single value, mustache also has tags called sections. A section tag starts with `{{#name}}` and the section ends with `{{/name}}`. Placing a pound-sign (#) before the name informs mustache to check if the value in the property `name` has a value. If the value is null or empty, a false is returned from this section, and everything between the section tokens is not rendered. Think of this as an **if** statement and only if the value is true does everything between these section tokens get rendered. Replace the `<template>` element in the `index.html` file with the following template.

Listing 12: Create an index page that uses the `mustache.js` templating engine.

```
<!DOCTYPE html>
<html>

  <head>
    <meta charset="utf-8" />

    <title>
      Manipulate the DOM Using Templating
    </title>

    <link href="styles/site.css" rel="stylesheet" />
  </head>

  <body>
    <header>
      <h1>
        Manipulate the DOM Using Templating
      </h1>
    </header>

    <main>
      <div>
        <label for="name"
          class="form-label">
          Movie Name
        </label>
        <input id="name" name="name"
          value="Young Frankenstein"
          class="form-control" />
      </div>
      <div>
        <label class="form-label"></label>
        <button onclick="displayData();">
          Display Data
        </button>
      </div>
      <div id="output">
      </div>
    </main>

    <script src="scripts/mustache.min.js">
    </script>
    <script src="scripts/common.js">
    </script>
    <script>
      'use strict';

      function displayData() {
        var output = document.getElementById('output');
        var name = document.getElementById('name').value;
        if (name) {
          output.innerHTML = 'The movie you entered is ' + name;
        } else {
          output.innerHTML = '';
        }
      }
    </script>
  </body>
</html>
```

Listing 13: Add multiple input elements to the HTML page.

```

<main>
  <div>
    <label for="name"
      class="form-label">
      Movie Name
    </label>
    <input id="name" name="name"
      value="Young Frankenstein"
      class="form-control" />
  </div>
  <div>
    <label for="rating"
      class="form-label">
      Movie Rating
    </label>
    <select id="rating"
      class="form-control">
      <option>G</option>
      <option selected>PG</option>
      <option>PG-13</option>
      <option>R</option>
    </select>
  </div>
  <div>
    <label for="genre"
      class="form-label">
      Movie Genre
    </label>
    <input id="genre" name="genre"
      value="Comedy"

```



```

      class="form-control" />
  </div>
  <div>
    <label for="year"
      class="form-label">
      Year Released
    </label>
    <input id="year" name="year"
      value="1974"
      class="form-control" />
  </div>
  <div>
    <label for="price"
      class="form-label">
      Rental Price
    </label>
    <input id="price" name="price"
      value="3.99"
      class="form-control" />
  </div>
  <div>
    <label class="form-label">
    </label>
    <button onclick="displayData();">
      Display Data
    </button>
  </div>
  <div id="output">
  </div>
</main>

```

```

<template id="template" type="text/html">
  {{#name}}
    <label class='movieInfo'>
      {{name}}
    </label>
  {{/name}}
</template>

```

Try It Out

Display the **index.html** file in your browser, delete the movie name in the text box, and click on the **Display Data** button and nothing is displayed in the output area because the **<label>** element is never rendered.

Check for Falsy Expressions Using an Inverted Section Tag

Another kind of section tag is called an **inverted section**. These sections begin with **{{^name}}** and the section ends with **{{/name}}**. The up-caret symbol (^) informs mustache to check the value in the property **name** and invert the value. If the **name** property has a value, return false, and if it doesn't have a value, return a true. Replace the **<template>** element in the **index.html** file with the following template.

```

<template id="template" type="text/html">
  {{^name}}
    <label class='movieInfo'>
      No Movie Input
    </label>
  {{/name}}
</template>

```

Try It Out

Display the **index.html** file in your browser, delete the movie name in the text box and click on the **Display Data** button and it should look like **Figure 5**. Put a movie name into this text box, and nothing is displayed.

Combine Section Tags in a Template

If you combine the use of both these section tags in a template, it effectively becomes an if...else statement. For example, replace the **<template>** element with the following snippet's HTML and tokens. A **<label>** element is to appear regardless of the user input, so both section tags go within the **<label>**. However, only one value is displayed depending on whether some data is entered or not.

```

<template id="template" type="text/html">
  <label class='movieInfo'>
    {{#name}}{{name}}{{/name}}
    {{^name}}No Movie Input{{/name}}
  </label>
</template>

```

Try It Out

Display the **index.html** file in your browser, with a movie name in the text box click on the **Display Data** button. The movie name should appear on your web page. Delete the movie name from the text box and click on the **Display Data** button to see the error message **No Movie Input** appear.

Build Multiple Elements and Add to DOM

Now that you're used to working with Mustache tokens and section tags, let's go back to the code where you're working with multiple elements. This is where you really see how much less code you must write. Replace the **<main>** element with the code shown in **Listing 13**.

Locate the **<template>** element and replace the contents with the following HTML and tokens. If you're working in VS Code or similar editor, notice how nice it is to get IntelliSense and color coding while building this template.

SPONSORED SIDEBAR

Ready to Modernize a Legacy App?

Need advice on migrating yesterday's **legacy applications** to today's modern platforms?

Take advantage of **CODE Consulting's** years of experience and contact us today to schedule a **FREE** consulting call to discuss your options.

**No strings.
No commitment.**

For more information: www.codemag.com/consulting or email us at info@codemag.com.

Listing 14: Read all data from the input elements and use Mustache to generate the HTML.

```

function displayData() {
  let movie = {
    "name": getValue("name"),
    "rating": getValue("rating"),
    "genre": getValue("genre"),
    "year": getValue("year"),
    "price": getValue("price")
  };

  // Read template with Mustache tokens
  let tmpl = getElement("template").innerHTML;
  // Use Mustache to render the HTML
  let html = Mustache.render(tmpl, movie);
  // Add the rendered HTML to the div
  getElement("output").innerHTML = html;
}

```

```

<template id="template" type="text/html">
  <label class='movieInfo'>
    {{#name}}{{{name}}}{{/name}}
    {{^name}}No Movie Input{{/name}}
  </label>
  <label>{{rating}}</label><br />
  <label>{{genre}}</label><br />
  <label>{{year}}</label><br />
  <label>{{price}}</label><br />
</template>

```

Modify the `displayData()` function to build a `movie` object literal and populate each property with the data input by the user (**Listing 14**). Notice that you don't need to write a `buildHTML()` method, you only need to populate the individual properties. Using Mustache re-

duces the code needed and provides the separation of concerns discussed earlier.

Try It Out

Display the `index.html` file in your browser, click on the **Display Data** button and the resulting web page should look almost like **Figure 6**.

Display Price as Currency Value

Notice on your web page that the `price` value is not displayed in a U.S. currency format. This is because you didn't use the `formatCurrency()` function when putting the input value into the `price` property. I'd suggest that you read the data directly from the inputs and place those values directly into the properties. If you send this data to a back-end, you most likely want to store the raw numbers without any currency symbols.

To handle these types of situations, create a method on your literal object that transforms your property into the appropriate string for display. For example, for the `price` property, create a `priceFormatted()` method that returns the `price` property formatted as a U.S. currency, as shown in the following code snippet:

```

let movie = {
  "name": getValue("name"),
  "rating": getValue("rating"),
  "genre": getValue("genre"),
  "year": getValue("year"),
  "price": getValue("price"),
  "priceFormatted": function () {

```

Listing 15: Create a web page to render a select list.

```

<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>Templating a Select List</title>
  <link href="styles/site.css" rel="stylesheet" />
</head>

<body>
  <header>
    <h1>Templating a Select List</h1>
    </header>
    </body>
    </html>
    </main>
    <script src="scripts/mustache.min.js"></script>
    <script src="scripts/common.js"></script>
    <script>
      </script>
    </body>
    </html>

```

Listing 16: Add a movie list and the code to render this list into the select list.

```

<script>
  'use strict';

  window.onload = function () {
    let movies = {
      "movieList": [
        {
          filmId: 1,
          name: "Casablanca"
        },
        {
          filmId: 2,
          name: "Caddy Shack"
        },
        {
          filmId: 3,
          name: "Young Frankenstein"
        }
      ];
    };

    // Read template with Mustache tokens
    let tmpl = getElement("template").innerHTML;
    // Use Mustache to render the HTML
    let html = Mustache.render(tmpl, movie);
    // Add the rendered HTML to the select
    getElement("movies").innerHTML = html;
  }
</script>

```

```

        return formatCurrency(this.price);
    }
};

```

Modify the <template> to display **priceFormatted** instead of **price**, as shown in the following code snippet. The Mustache rendering engine looks at each token name and determines if it's a simple property or a function, and makes the appropriate call for replacement into that token.

```

<template id="template" type="text/html">
    <label class='movieInfo'>
        {{#name}}{{name}}{{/name}}
        {{^name}}No Movie Entered{{/name}}
    </label>
    <label>{{rating}}</label><br />
    <label>{{genre}}</label><br />
    <label>{{year}}</label><br />
    <label>{{priceFormatted}}</label><br />
</template>

```

Try It Out

Display the **index.html** file in your browser, click on the **Display Data** button, and it should now look exactly like **Figure 6**.

Build a Select List Using Templating

Section tags, besides checking for true or false, also check a property to see if it's a list. Each item of the list is iterated over and the tokens specified within the template are replaced. Create a new HTML file named **select-list.html** and add the HTML shown in **Listing 15** into this file. Notice the <select> element with the **id** set to "movies". In this <select> is where <option> elements are added to create a drop-down list of movies.

Add the code shown in **Listing 16** into the empty <script> element. This code creates a simple list of data in an object literal named **movies**. This object literal has a single array property named **movieList**. Each element of this array is another object literal with two properties; **filmId** and **name**. Each of these properties is used to create the <option> elements in the <select> element.

Just after the closing </main> tag, add a <template> element, as shown in the following code snippet. Within the section {{#movieList}} is the HTML needed to build each <option> element from the **filmId** and **name** properties. The section tag needs to be the same name as the array property within the **movies** object literal, in this case **movieList**.

```

<!-- Create a Mustache Template -->
<template id="selectTemplate"
    type="text/html">
    {{#movieList}}
        <option value="{{filmId}}">
            {{name}}
        </option>
    {{/movieList}}
</template>

```

Try It Out

Display the **select-list.html** file in your browser and you should see a drop-down list of movie names, as shown in **Figure 11**.

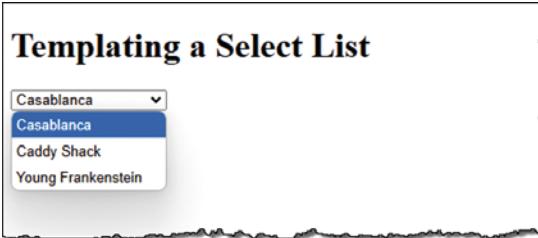


Figure 11: Create a select list using templating.

Build a Table Using Templating

If several properties of an object literal in an array are needed to display on an HTML page, a <table> element is ideally suited for this. Create a new folder named **classes** and add a new file named **Movies.js** into this new folder. Add the code shown in **Listing 17** to the **Movies.js** file.

In the **Movie.js** file, you create a **Movies** class using the **class** keyword. A single private field name **#list** is created to hold the data created in the **getAll()** method. A single public property named **movieList** is created to return the array of movie data contained in the **#list** field. This class is going to be used to populate the <table> in the next web page you are creating.

Create a Page to Display the Table

Create a new HTML file named **table.html** and add the HTML shown in **Listing 18** into this new file. The <table> element has an **id** attribute set to "movieTable". Into the empty <tbody> element is where each row of movie data

ADVERTISERS INDEX

Advertisers Index	
CODE Consulting--AI Services www.codemag.com/ai-services	2
CODE Consulting--Copilot www.codemag.com/copilot	63
CODE Consulting--Executive Briefing www.codemag.com/executivebriefing	7
CODE Consulting www.codemag.com/Code	75
CODE Staffing www.codemag.com/staffing	76
DevIntersection www.devintersection.com	5
dtSearch www.dtSearch.com	15
Power Platform Community Conference www.powerplatformconf.com	35



CODE
Does Angular End RxJS Boilerplate?
Make Movies with AI
Embracing JavaScript Templates
Async Techniques Explored

Advertising Sales:
Tammy Ferguson
832-717-4445 ext 26
tammy@codemag.com

This listing is provided as a courtesy to our readers and advertisers. The publisher assumes no responsibility for errors or omissions.

Listing 17: Create a closure that provides a list of movie data.

```

class Movies {
  // Private fields
  #list = [];

  // Public properties
  get movieList() {
    return this.#list;
  }

  // Public Methods
  getAll() {
    this.#list = [
      {
        "filmId": 1,
        "name": "2012",
        "genres": "Action,Adventure,Sci-Fi",
        "actors": "John Cusack,Amanda Peet",
        "runTimeInMinutes": 158,
        "rating": "PG-13",
        "price": 3.95,
        "releaseDate": "2009-11-13"
      },
      {
        "filmId": 2,
        "name": "9 to 5",
        "genres": "Comedy",
        "actors": "Jane Fonda,Lily Tomlin,
                  Dolly Parton",
        "runTimeInMinutes": 109,
        "rating": "PG",
        "price": 2.95,
        "releaseDate": "1980-12-19"
      },
      {
        "filmId": 5,
        "name": "African Queen, The",
        "genres": "Adventure,Drama,Romance",
        "actors": "Humphrey Bogart,
                  Katharine Hepburn",
        "runTimeInMinutes": 105,
        "rating": "G",
        "price": 3.95,
        "releaseDate": "1952-02-20"
      },
      {
        "filmId": 6,
        "name": "Alien",
        "genres": "Horror,Sci-Fi",
        "actors": "Sigourney Weaver",
        "runTimeInMinutes": 117,
        "rating": "R",
        "price": 3.95,
        "releaseDate": "1979-05-25"
      }
    ];
  }
}

```

Listing 18: Create a web page to display a table of movie data.

```

<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>
    Templating an HTML Table
  </title>
  <link href="styles/site.css" rel="stylesheet">
</head>

<body>
  <header>
    <h1>
      Templating an HTML Table
    </h1>
  </header>
  <main>
    <table id="movieTable">
      <thead>
        <tr>
          <th>Movie Name</th>
          <th>Actors</th>
          <th>Genres</th>
          <th class="pull-end">Rental Price</th>
        </tr>
      </thead>
      <tbody></tbody>
    </table>
  </main>
  <script src="scripts/mustache.min.js"></script>
  <script src="scripts/common.js"></script>
  <script src="classes/Movies.js"></script>
  <script>
    </script>
  </body>
</html>

```

is to be inserted. Be sure to include the **Movies.js** script file in your HTML page.

Add the code shown in **Listing 19** into the empty `<script>` element. This code creates a new instance of the **Movies** class and assigns it to a variable named **entity**. The `getAll()` method is invoked on the **entity** object to load all the movie data into the private `#list` field. A literal object named **vm** is created to wrap up the list of movies and to provide additional methods used to format the movie data when displayed in the table.

Just below the closing `</main>` tag, add a new `<template>` element with one section to build the row of movie data (**Listing 20**). The inverted section is for when there are no movies and you wish to display a message to the user.

Try It Out

Display the **table.html** file in your browser and it should look like **Figure 12**.

Go back to the **vm** object literal and uncomment the line that sets the **movieList** property to **null**. Comment out the other **movieList** property that's assigned data from the **entity.movieList** property. Display the **table.html** file again in your browser and you should now see the error message appear, as shown in **Figure 13**.

Other Templating Engines

There are many different templating engines for use on web pages. Some of the more common ones are **handlebars**, **pug**, **EJS**, **Ember**, and **Vue**. The syntax is similar to

Listing 19: Create the table from the Movies class and the template in this page.

```

<script>
'use strict';

window.onload = function () {
  // Create instance of class
  let entity = new Movies();

  // Load all movies into
  // the movieList property
  entity.getAll();

  // Create a view model
  let vm = {
    // "movieList": null,
    "movieList": entity.movieList,
    "priceFormatted": function () {
      return formatCurrency(this.price);
    },
    "runTime": function () {
      let value = new
        Number(this.runTimeInMinutes);
      let hours = Math.floor(value / 60);
      let minutes = value % 60;
      return `${hours} hours and
${minutes} minutes`;
    }
  };

  // Read template with Mustache tokens
  let tmpl = getElement("template")
    .innerHTML;
  // Use Mustache to render the HTML
  let html = Mustache.render(tmpl, vm);
  // Add the rendered HTML into the table
  getElement("movieTable")
    .getElementsByTagName("tbody")[0]
    .innerHTML = html;
}

</script>

```

Listing 20: Use two section tags to display an error when there is no movie data or to display the list of movies.

```

<!-- Template to Build a Table -->
<template id="template"
  type="text/html">
  {{#movieList}}
    <tr>
      <td>{{name}}</td>
      <td>{{actors}}</td>
      <td>{{genres}}</td>
      <td class="pull-end">
        {{priceFormatted}}
      </td>
      <td>{{runTime}}</td>
    </tr>
  {{/movieList}}
  {{^movieList}}
    <tr>
      <td colspan="6">
        ** No Movies Available **
      </td>
    </tr>
  {{/movieList}}
</template>

```

what you learned in this article, although the delimiters used for their tokens may be a little different. Many of these engines have more features than Mustache; however, check to make sure you need all those features before you add the extra library size to your web project. If you're already using jQuery in your web pages, you might want to explore jQuery Templates.

Are Templating Engines Still Relevant?

With the advent of JavaScript frameworks such as Angular, Vue, and React, are templating engines like Mustache even needed anymore? I think they still serve a purpose if you're focusing mainly on client-side code calling Web APIs. Templating engines are also used to build websites with node.js and Express. In a future article, I'll explore creating a website using node.js, Express, and a templating engine.

Summary

Use templates whenever you find yourself embedding HTML inside of JavaScript strings. Whether you code the search and replace yourself, or use a templating engine like Mustache, take advantage of the `<template>` element to store your HTML. Using templating engines makes it much easier to create dynamic web pages while making your code more maintainable and readable. Other benefits include separation of concerns, the ability to get IntelliSense when typing in your HTML, color coding, and code that's easier to debug.

Templating an HTML Table

Movie Name	Actors	Genres	Rental Price	Run Time	Release Date
2012	John Cusack,Amanda Peet	Action,Adventure,Sci-Fi	\$3.95	2 hours and 38 minutes	2009-11-13
9 to 5	Jane Fonda,Lily Tomlin,Dolly Parton	Comedy	\$2.95	1 hours and 49 minutes	1980-12-19
African Queen, The	Humphrey Bogart,Katharine Hepburn	Adventure,Drama,Romance	\$3.95	1 hours and 45 minutes	1952-02-20
Alien	Sigourney Weaver	Horror,Sci-Fi	\$3.95	1 hours and 57 minutes	1979-05-25

Figure 12: Create a table using Mustache templating.

Templating an HTML Table

Movie Name	Actors	Genres	Rental Price	Run Time	Release Date
** No Movies Available **					

Figure 13: Display an error message when there's no data available for the table.

Wait, What? Async-Await Explained

I've probably written a thousand async-await functions and you probably have too. But I have a confession: It wasn't until quite recently that I understood what async-await actually does and what purpose it serves. In this article, I want to take you through my discovery process and perhaps give you some insight into this very common, but often misunderstood, C# language feature.



Matthew Hess

csharpartisan@gmail.com
YouTube: @csharpartisan

Matthew Hess is a Microsoft MVP and a software engineer from Albuquerque, NM. Matthew got his start programming using Borland's Delphi IDE and has been working with C# and the .NET Framework since version 2. Matthew has developed software for many different industries from legal to financial services, to biotechnology, to non-profit. Matthew currently works for CG Infinity, a Dallas-based software consulting firm. His YouTube Channel, @CSharpAristan, is meant to help programmers of all levels become better at the craft of programming in C#.



What Confused Me about Async-Await

When you think about asynchronous programming, the basic idea is very simple: You want your code to do more than one thing at a time. Rather than block and wait on some function call that might take a while, you call it asynchronously so that the code can go work on something else and then process the results of that function when they arrive. It's a clear and intuitive idea, but to really get it in focus, let's compare an asynchronous call pattern to a synchronous one.

Synchronous Call Patterns

Let's say you have a function that needs to call two other functions in its body. In **Figure 1**, you can see what this looks like if the primary function calls the sub functions using a synchronous call pattern. The calls are made serially, one after the other and the Main Function waits for Sub Function 1 to complete before calling Sub Function 2.

Asynchronous Call Patterns

Now let's say that Sub Function 1 and Sub Function 2 are independent of each other. In other words, Sub Function 2 does not need the result of Sub Function 1 and vice-versa. This means they can be called in any order and, in fact, can safely run in parallel. To achieve this, you decide to call these functions asynchronously. You kick

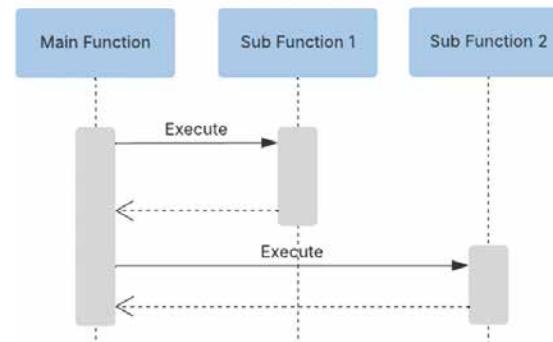


Figure 1: Synchronous calls

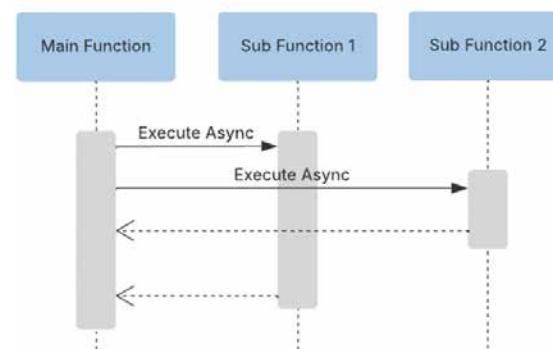


Figure 2: Asynchronous calls

them both off and wait for them both to return. **Figure 2** shows what this call pattern looks like:

In this particular sequence diagram, Sub Function 2 happens to complete first, but it doesn't have to, and it doesn't really matter. The point is that you have parallelized the code. The two functions run at the same time.

This is simple and intuitive. Unfortunately, this is **not** what the C# async-await pattern does, and that's what I found so confusing.

An Await Statement Blocks Local Execution

To understand why I was so confused, let's translate the asynchronous call pattern into C#. Imagine that both Sub Function 1 and Sub Function 2 call some REST API endpoint to retrieve data that the Main function eventually needs. Here's the code:

```

async Task MainFunction()
{
    await SubFunction1();
    await SubFunction2();
}

async Task SubFunction1()
{
    HttpResponseMessage response = await
        _myHttpClient.GetAsync("http://someuri.net");
}

async Task SubFunction2()
{
    HttpResponseMessage response = await
        _myHttpClient.GetAsync("http://another.net");
}
  
```

If you run this code, you'll see that the call to SubFunction1 blocks the local code execution. The call to SubFunction2 doesn't start until the first function finishes. In other words, this looks like the synchronous call pattern in **Figure 1**, not the asynchronous call pattern in **Figure 2**.

When I first saw this, my reaction was, "Wait, what?!?!" I fully expected my code to keep running with the async task going in the background. When I saw that my code blocked, I was honestly kind of shocked and I thought, "What's the point? What is this even doing?" Maybe you've had the same feeling or reaction.

What Async-Await Actually Does

The reality is that **async-await** does do something very important. To understand what is going on here, you need to take some time to understand Tasks and the Thread Pool.



Figure 3: A Thread Pool with three threads

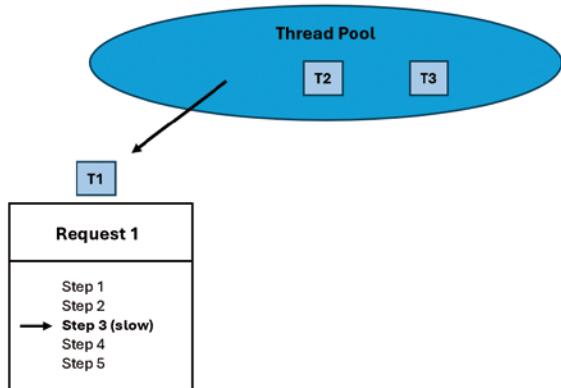


Figure 4: Thread Pool thread assigned to a task

Tasks and the Thread Pool

In C#, an **async-await** call is shorthand for creating, running, and awaiting a **Task**, and as you may know, **Tasks** run on **Thread Pool Threads**. Let's walk this through step-by-step to see how it works.

The Thread Pool that a real .NET application gets has thousands of threads in the pool by default. But let's imagine that you have a sad little Thread Pool with exactly three threads, as shown in **Figure 3**.

Now imagine that you have a server application that's listening for requests. Whenever it gets a request, it calls a request handler. This causes a task to get created and assigned a thread from the Thread Pool, as shown in **Figure 4**.

The request handler contains a few normal steps (Step 1 and Step 2) and then it gets to Step 3, which, for some reason, is slow. Maybe it's calling some REST API that takes a bit to reply. The result is that Request 1, and therefore T1, is waiting on the result of Step 3.

Now imagine that two more requests come in. This causes two more tasks start running, each of which gets a thread pool thread, as shown in **Figure 5**.

What happens if the server receives a fourth request? Because there are no available threads, the fourth request cannot even start. This is called "thread starvation," as shown in **Figure 6**.

Solving Thread Starvation

Thread starvation is the problem that **async-await** solves. Here's how it works.

Let's go back in time to the moment when the very first client request comes in. Request 1 starts running. It does Step 1 and Step 2. Then it gets to Step 3. Rather than calling Step 3 synchronously, call it using an **await**. This causes .NET to create what is called a **Continuation**. You

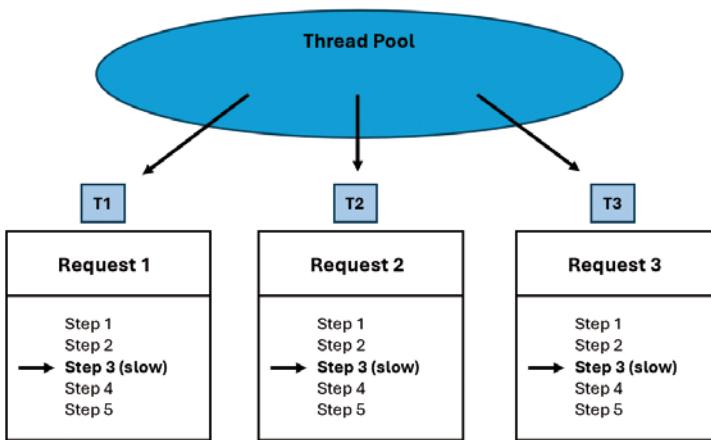


Figure 5: All three threads assigned to tasks

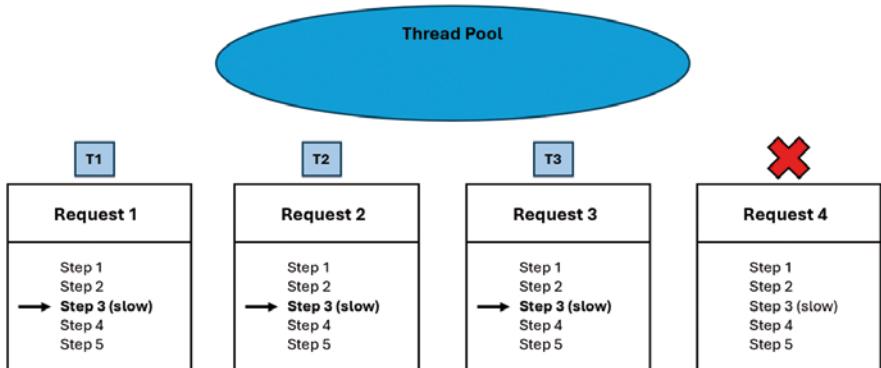


Figure 6: Thread starvation

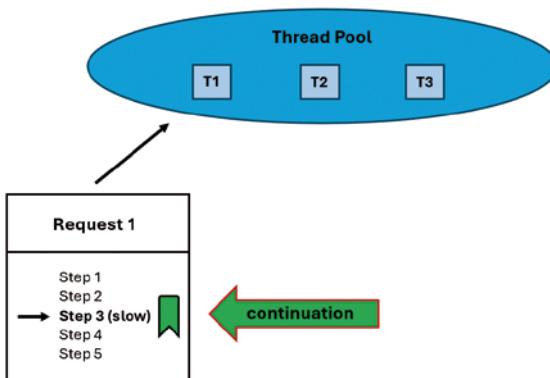


Figure 7: A Continuation frees a Thread Pool thread

can think of a **Continuation** as a sort of bookmark. Essentially, .NET is saying, okay, this function is awaiting the result of an asynchronous operation. Let's suspend that function for the time being and, when asynchronous operation completes, you can restart (continue) that function where it left off. In the meantime, the thread can **return to the pool**, as shown in **Figure 6**.

You can see how this solves the problem of thread starvation. Because the threads return to the pool, when that fourth request comes in, you have no trouble at all servicing it. There are three threads to choose from, as shown in **Figure 8**.

Request 4 happened to be assigned T2 to start with, but it really doesn't matter. With tasks, there's no affinity between a task and the thread that's servicing it at any given point in time. The key is that using an **await** allows a thread to return to the pool during the async operation, and this allows a real .NET application to service many thousands of simultaneous requests using a limited thread pool.

To complete the story, now let's imagine that the **async** call in Request 1 completes. At that point, .NET clears the **Continuation**. It assigns a Thread Pool thread to the task and resumes execution at Step 4, as shown in **Figure 9**.

Did you notice how when **async** Function 1 resumed, it resumed on a different Thread Pool thread from the one it started on? Once again, there's no affinity between a thread and a task. After every **await**, there's a high likelihood that your code will continue on a different Thread Pool thread, which is why using techniques like the **ThreadStatic** attribute can be disastrous. But that's a topic for another day. For now, the important thing is to understand what **async-await** is **really** doing. It's not letting the code **inside** your function do multiple things at once. Rather, it's freeing up Thread Pool threads so your application **overall** can do multiple things at once.

Implications and Analysis

This understanding of **async-await** raises some questions and has some interesting implications. Let's address some of the most common ones.

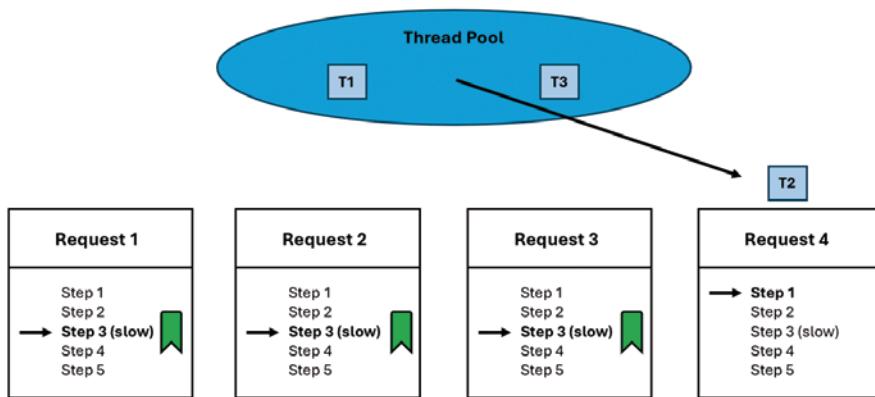


Figure 8: Thread Starvation solved

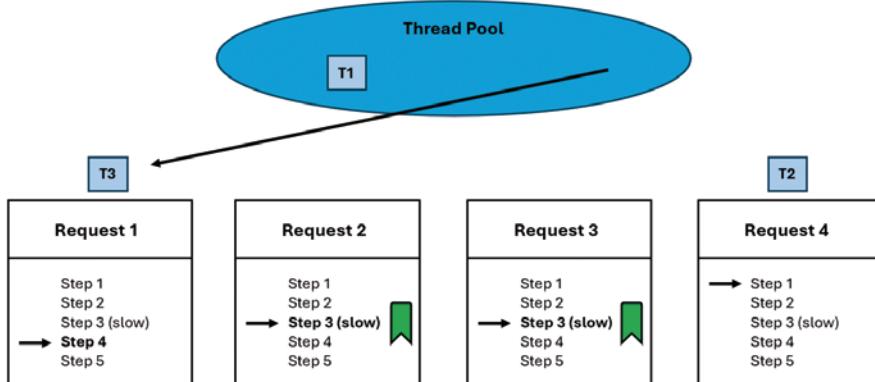


Figure 9: A Continuation cleared and execution resumed

Running Tasks in Parallel

I've established that **async-await** doesn't allow the code inside of a function to do multiple things at once. But what if that's what you want to do? What if you want some true parallel processing where one request can await multiple operations at a time, as you hoped for in **Figure 2**? How do you do that?

The C# tasks model makes this very easy. You can modify the MainFunction from above as follows:

```
async Task MainFunction()
{
    // start both tasks
    Task t1 = SubFunction1();
    Task t2 = SubFunction2();

    // wait for both tasks to finish
    await Task.WhenAll(new Task[] { t1, t2 });
}
```

Now both SubFunction1 and SubFunction2 run at the same time and neither holds onto a Thread Pool thread while awaiting the result of their respective HTTP GetAsync calls. Importantly, the thread associated with MainFunction is also able to return to the Thread Pool and be reused while both SubFunction1 and Subfunction2 complete. This is the best of all worlds in terms of parallel processing.

Turtles All the Way Down

Sometimes when you introduce **async-await** code into your project, it creates a sort of problem of "infinite regress" where it seems everything must become **async**. Imagine that you have a sequence of three functions where Function1 calls Function2 and Function2 calls Function3.

```
void Function1()
{
    Function2();
}

void Function2()
{
    Function3();
}

void Function3()
{
}
```

Now imagine that the body of Function3 contains an **async** call, therefore when you implement Function3 you change it to be marked as **async** like this:

```
async Task Function3()
{
    HttpResponseMessage response = await
        _myHttpClient.GetAsync("http://someuri.net");
}
```

This causes a problem for Function2. Function2 can no longer call Function3 as written because Function3 is now **async**. So, you fix Function2 and make it **async** also. This then causes the same problem for Function1. After you fix all three, your code ends up looking like this:

```

async Task Function1()
{
    await Function2();
}

async Task Function2()
{
    await Function3();
}

async Task Function3()
{
    HttpResponseMessage response = await
        _myHttpClient.GetAsync("http://abc.net");
}

```

What ends up happening is that as soon as you introduce a single **async-await**, it seems to percolate all the way up the entire call chain. I like to think of this as a version of the “turtles all the way down” problem, which is a way of describing “infinite regress.” Perhaps you’ve experienced this phenomenon.

What should you do? One approach is to simply make everything **async** all the way to the external entry point of your code. In other words, stack the turtles all the way! If your application is a console app or a Windows executable that has a Main function at the root, you can declare your Main function as **async** like this:

```

static async Task Main(string[] args)
{
    await Function1();
}

```

If you’re working with an ASP.NET web service where your Controller functions serve as the root of each request, you can declare your Controller functions as **async** like this:

```

[HttpPost]
public async Task Post()
{
    await Function1();
}

```

Making everything **async** works, but it can be quite burdensome in a large legacy code base. The alternative approach is to explicitly navigate the synchronous/asynchronous boundary yourself. There are several ways to do this and some, like using the Task-based Asynchronous Pattern (TAP), are very sophisticated and give you a lot of control over what your calling synchronous function does while your asynchronous task is executing. A simple, safe, and common approach is to do something like this:

```

void Function1()
{
    Function2().GetAwaiter().GetResult();
}

```

In this code, the Function1 context is synchronous, and the thread associated with Function1 does indeed block and does **not** return to the pool. However, any **awaits** called downstream are asynchronous and release their threads.

It should be noted that when you have an **async** Main or an **async** Controller method, C# is navigating the synchronous/asynchronous boundary for you behind the scenes. For example, with a console or Windows app, your **async** Main function is not exposed to the operating system (Windows has no concept of a task). Rather, there is an invisible function that probably looks something like this:

```

// hidden main which the operating system calls
static void HiddenMain(string[] args)
{
    Main(args).GetAwaiter().GetResult();
}

// your "main" function
static async Task Main(string[] args)
{
    await Function1();
}

```

The point is, there is always a synchronous/asynchronous boundary. The question is where and how it’s navigated. If you don’t want to implement the infinite regress of **async** calls, you can, depending on your application, choose the point at which it makes sense to make the transition.

What Makes Something “Awaitable”?

In the discussion of the “turtles all the way down” phenomena, you saw how you sometimes end up with a chain of **async** functions that works its way back to the root, or entry point, of your program. But what happens at the other end? Looking at the other end of the chain, you’ll find that a series of **async** calls almost always bottoms out with a function like **HttpClient.GetAsync** that’s provided by a .NET library.

Table 1 includes some of the async functions found in the .NET libraries. This list is by no means exhaustive. It’s just meant to highlight some common (and perhaps less common) async functions you might run into:

What do all of these async methods have in common? They’re all IO operations. They all connect to, read from, or write to a file, port, or stream of some kind. This is a very important observation. Think back to the earlier discussion of a Continuation. As you saw, a Continuation can be thought of as a bookmark. The .NET runtime places the bookmark at an **await** in a method and resumes the code there once the asynchronous operation completes. But how does the runtime know that the asynchronous operation has completed? Is there some magic signal that tells the runtime that it’s time to continue?

The answer is: Yes! There is, in fact, a kind of magic signal that makes this happen. In Windows, it involves a Windows facility called IO Completion Port (IOCP). This is a feature of the Windows API that allows a caller to initiate an IO operation and then subscribe to a special Port that will signal when the IO operation completes. This is the “magic signal” that makes **async-await** Thread Pool sharing possible. It may be “turtles all the way down” but the last turtle is implemented using IOCP!

What About CPU-Bound Tasks?

Everything you’ve looked at so far involves IO-bound tasks. This is work where the result is dependent on (bounded by or bottlenecked by) an IO operation, and as you’ve seen,

SPONSORED SIDEBAR

Stay Connected with
CODE Magazine on
[LinkedIn](#)

Join our professional
community on LinkedIn
to stay ahead in the tech
world.

Why Connect with Us?

Exclusive Content:
Get the latest tech insights
and articles.

Industry Updates:
Stay informed with cutting-
edge news.

Networking:
Connect with peers and
industry leaders.

Help us grow by sharing
with colleagues and
friends.

Follow us today:
[www.linkedin.com/
company/code-magazine](http://www.linkedin.com/company/code-magazine)

NameSpace	Class	Method
System.Net.Http	HttpClient	GetAsync
	HttpClient	PatchAsync
	HttpClient	PostAsync
	HttpClient	PutAsync
	HttpClient	DeleteAsync
	HttpClient	SendAsync
Microsoft.Data.SqlClient	SqlConnection	OpenAsync
	SqlCommand	ExecuteReaderAsync
	SqlCommand	ExecuteNonQueryAsync
	SqlCommand	ExecuteScalarAsync
	SqlDataReader	ReadAsync
	SqlDataReader	GetFieldValueAsync
System.IO	File	ReadAllLinesAsync
	File	ReadAllBytesAsync
	File	ReadAllTextAsync
	File	WriteAllLinesAsync
	File	WriteAllBytesAsync
	File	WriteAllTextAsync
	FileStream	ReadAsync
	FileStream	WriteAsync
Azure.Messaging.ServiceBus	FileStream	CopyToAsync
	ServiceBusSender	SendMessageAsync
	ServiceBusSender	SendMessagesAsync
	ServiceBusSender	ScheduleMessageAsync
	ServiceBusSender	CancelScheduledMessageAsync
	ServiceBusReceiver	ReceiveMessageAsync
	ServiceBusReceiver	RecieveMessagesAsync
	ServiceBusReceiver	PeekMessageAsync
	ServiceBusReceiver	CompleteMessageAsync
	ServiceBusReceiver	AbandonMessageAsync
	ServiceBusReceiver	RenewMessageLockAsync
	ServiceBusReceiver	DeadLetterMessageAsync
Windows.Storage	FileIO	ReadTextAsync
	FileIO	ReadLinesAsync
	FileIO	WriteBufferAsync
	FileIO	WriteBytesAsync
	FileIO	AppendTextAsync
System.IO.Pipes	PipeStream	ReadAsync
	PipeStream	WriteAsync
	PipeStream	FlushAsync
System.Security.Cryptography	CryptoStream	ReadAsync
	CryptoStream	WriteAsync
	CryptoStream	FlushAsync

Table 1: Some async functions in the .NET libraries

the “awaitability” of these tasks is made possible by a special operating system mechanism that the .NET runtime can take advantage of. But what about CPU-bound tasks? These are tasks that are dependent on (bounded by or bottlenecked by) the number and speed of CPU cores. Are these also “awaitable?” And if so, what does that mean? Do they also support continuations and thread sharing?

To answer this, let’s look at a snippet of code that launches and awaits a set of CPU-Bound tasks.

```

async Task DoABunchOfCPUBoundWork()
{
    List<Task> tasks = new();
    for (int i = 1; i <= 10; i++)
    {
        tasks.Add(Task.Run(() =>
            { SomeCPUBoundJob(); }));
    }
    await Task.WhenAll(tasks);
}

void SomeCPUBoundJob()
{
    // processor intensive work here
}

```

The first thing to note is that the CPU-bound function, **SomeCPUBoundJob**, is not marked as **async** and does not return a task. It’s a regular C# function. So how does it get run as a task? The magic is the call to **Task.Run()** which takes the supplied function and executes it in a task. This is how you run CPU-bound work in a task: by using **Task.Run()**.

As this code executes, it queues ten tasks, each of which eventually gets scheduled on its own Thread Pool thread. Meanwhile, the thread associated with **DoABunchOfCPUBoundWork** is, in fact, freed back to the Thread Pool. This happens when you **await Tasks.WhenAll()**. At this point, a normal Continuation is indeed created and the thread can be assigned other work.

With IO-bound work, the **async-await** pattern allows an application to potentially service many thousands of requests with a small number of threads. With CPU-bound work, this is not the case. For CPU-bound work, the application will be limited by the number and speed of the CPU cores available to the .NET task scheduler that’s queuing and running all those tasks. The code looks similar, but the parallelization behavior turns out to be quite different for IO-bound tasks that are truly awaitable and CPU-bound tasks that need a processor to run on. But at least the call that launches all the CPU-bound tasks doesn’t sit on the controlling thread. You can still use **async-await** to make efficient use of the Thread Pool even in a CPU-bound situation.

Conclusion

At a high level, the idea of asynchronous programming is easy to understand: You simply want your code to be able to do more than one thing at a time. In practice, as you’ve seen, there’s a bit of nuance and complexity to it and simply using **async-await** doesn’t automatically make your code run in parallel. What **async-await** does do, is allow your application overall to make more effective use of the Thread Pool, and by using other techniques, such as **Task.WhenAll()**, you can combine this efficient Thread Pool usage with true parallel processing for both IO-bound and CPU-bound work.

When I first started using **async-await**, I was a bit frustrated and confused, but now I think I have a clearer understanding of how it works and what it’s for. I hope you do, too.

THINK POWER PLATFORM IS JUST POINT-AND-CLICK? *THINK AGAIN.*

Where Professional Developers Build the Future of AI Agents

As a seasoned developer, you've probably dismissed Power Platform as "citizen developer territory." We get it. But here's what you might be missing: **Copilot Studio**—Microsoft's most advanced AI agent development platform that's redefining how intelligent automation gets built.

Build AI Agents That Actually Work in the Enterprise

Forget chatbots that break when users ask anything unexpected. With Copilot Studio in Power Platform, you're building **custom AI agents** with the architectural sophistication you'd expect: autonomous decision-making, complex workflow orchestration, and enterprise-grade integrations that scale.

The AI Development Platform You've Been Waiting For

Copilot Studio isn't just another conversational AI tool; it's a complete agent development environment:

- **Autonomous Agents** - Build agents that act independently, make decisions, and execute complex business logic
- **Conversational Intelligence** - Create agents that users can talk to that understand context, maintain conversation flow, and integrate with your existing systems
- **AI-First Workflows** - Orchestrate intelligent processes that adapt and learn from user interactions
- **Enterprise Integration** - Connect agents to Dataverse,

custom APIs, and third-party systems with the security and compliance you need

- **Custom Skills Development** - Extend agent capabilities with your own code and specialized functions

Connect with Developers Building the Next Generation

Network with Microsoft MVPs and professional developers who are already deploying AI agents in production. Learn how they're solving real business problems with autonomous agents that handle everything from customer service to complex data analysis.

Hands-On Agent Development Workshops That Respect Your Skills

Full-day technical workshops are designed for developers who want to see the architecture, understand the AI orchestration, and build agents that solve real problems. See the website for full-day workshops Oct 26-27 and Oct 31.

Ready to Build the Future of Intelligent Automation?

Give Copilot Studio three days (or more). You'll discover why AI agents are the next evolution of enterprise development, and why Power Platform is such a power place to deploy them.

- Visit powerplatformconf.com
- PPCC 2025 - *Where professional developers discover there's more than meets the eye*

4TH ANNUAL



Power Platform
COMMUNITY CONFERENCE

OCTOBER 28-30, 2025
WORKSHOPS OCTOBER 26, 27 & 31
MGM GRAND LAS VEGAS

REGISTER
TODAY

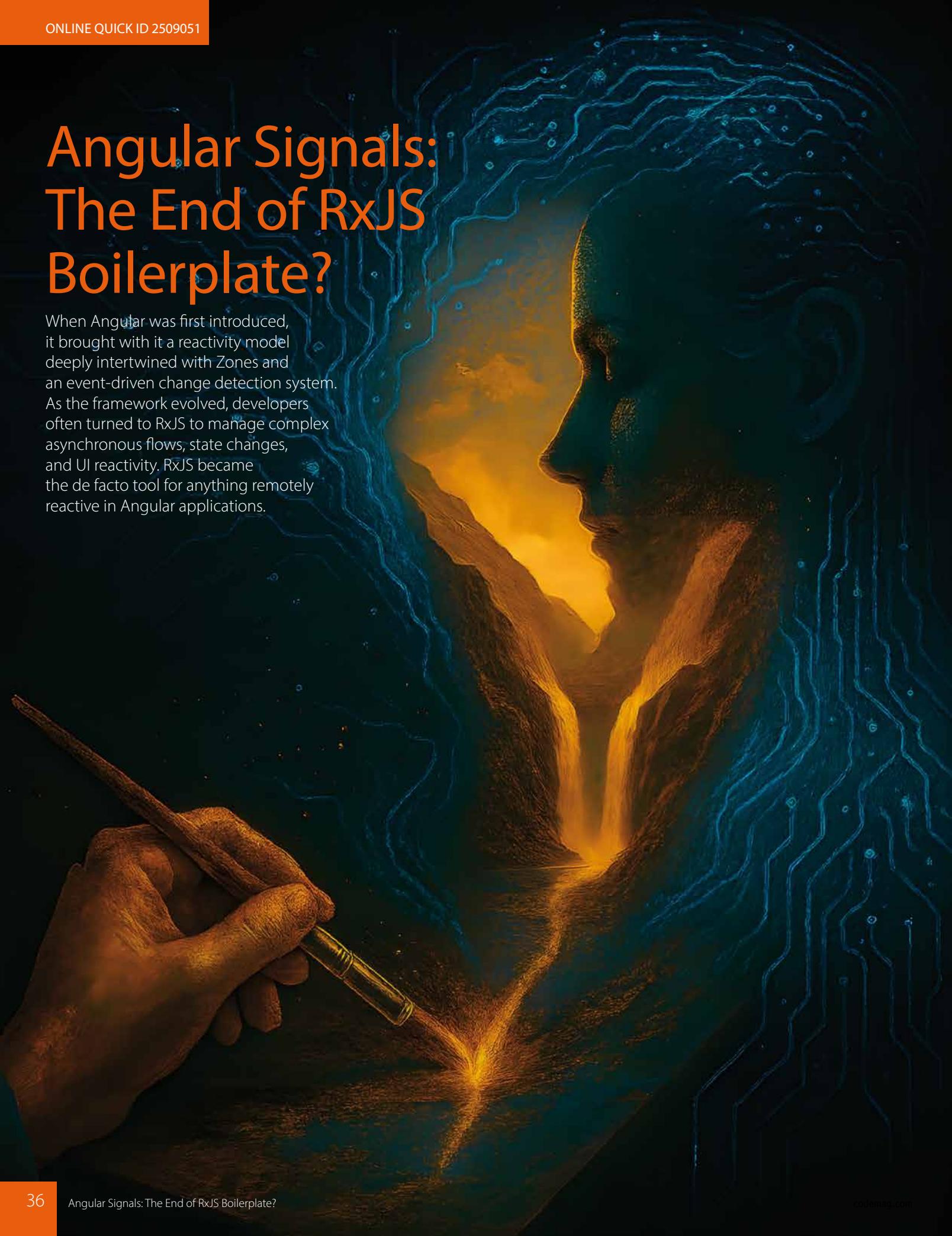


#PPCC25

PowerPlatformConf.com

Angular Signals: The End of RxJS Boilerplate?

When Angular was first introduced, it brought with it a reactivity model deeply intertwined with Zones and an event-driven change detection system. As the framework evolved, developers often turned to RxJS to manage complex asynchronous flows, state changes, and UI reactivity. RxJS became the de facto tool for anything remotely reactive in Angular applications.



However, as applications scaled, many developers encountered the same challenge: the abundance of boilerplate code, the difficulty of tracing data flows, and the steep learning curve associated with mastering streams, subscriptions, and operators.

Today, Angular enters a new era of reactivity with the introduction of **Signals**. Designed to be intuitive, explicit, and fine-grained, Signals offer a new reactive primitive built into the core of Angular itself. Signals aren't just a minor addition; they represent a fundamental shift in how you model data and UI state in Angular applications. For the first time, developers have a tool that is both simple enough for local state management and powerful enough to replace large chunks of RxJS boilerplate in the UI layer.

In this article, I'll take a deep dive into Angular Signals, not merely as a new API, but as a new way of thinking about building applications. I'll explore their core primitives, `signal()`, `computed()`, and `effect()`, and understand how they seamlessly fit into Angular's vision for a **zoneless, standalone component-driven future**. I'll introduce the concept of `LinkedSignal`, a powerful feature that dramatically improves component-to-component reactivity without the tedious ceremony developers are used to.

This journey isn't about pitting Signals against RxJS in a binary competition. RxJS still has its place for modelling complex event streams, real-time data, and sophisticated asynchronous workflows. Rather, this article will help you understand where Signals excel, where RxJS remains essential, and how the two can coexist in modern Angular applications. Through real-world examples, including building typeahead inputs, form handling, route guards, and data fetching with the new `HttpResource`, you'll see how Signals simplify common front-end challenges, eliminate boilerplate, and enhance maintainability.

As I proceed, I'll also dive into the performance optimizations under the hood, the new mental models developers must adopt, and strategies for gradually migrating existing Angular applications to embrace this simpler, more predictable reactivity model. I'll examine not only the technical benefits but also the profound impact on developer experience, debugging, testing, and scalability.

By the end of this exploration, it should become clear: Signals are not just another feature. They are the key to a leaner, cleaner, and more intentional Angular. They mark the end of unnecessary RxJS boilerplate in the UI, and the beginning of a new, more elegant way to build Angular applications.

Core Concepts: `signal()`, `computed()`, `effect()`

Angular's new reactive system is built on three essential primitives: `signal()`, `computed()`, and `effect()`. These primitives create a direct and predictable relationship between application state and the user interface, streamlining how changes flow through an Angular application.

A `signal()` is the starting point. It encapsulates a single piece of state, providing both a getter and setter for its value. When a signal's value is updated, Angular automatically tracks which parts of the application have read from it and triggers updates precisely where needed. This behavior removes the need for explicit subscriptions, manual event listeners, or redundant change detection triggers. A signal can be thought of as a reactive variable, but one that carries with it a deep understanding of its consumers across the application.

Built on top of signals are `computed()` values, which represent derived data. A `computed()` function automatically recalculates itself whenever any of the signals it depends on change. Unlike manual recalculations sprinkled throughout component logic, `computed()` values offer a declarative way to express relationships between different pieces of state. This encourages a design where derived data is defined once and reused consistently, without risk of stale values or unintended inconsistencies. Computed values are lazy by design, updating only when actively used, which leads to highly efficient UI updates.

The final primitive, `effect()`, introduces controlled side effects into the reactive system. An `effect()` function runs whenever one of its dependent signals changes, executing logic that falls outside pure state transformations. This is where Angular developers can interact with services, perform DOM manipulations, or trigger external processes in a safe, signal-driven manner. Effects provide a clean boundary between pure reactivity and the inevitable imperative code that all real-world applications require. Effects must be used inside an injection context, such as the constructor.

What makes this system particularly powerful is Angular's runtime dependency tracking. Developers do not need to declare what to watch or remember to clean up listeners. Angular automatically infers dependencies at runtime, ensuring that only the necessary parts of the application rerun when data changes. This eliminates a common class of bugs where stale subscriptions, forgotten listeners, or over-triggered change detection lead to unpredictable behavior and performance issues.

Rather than introducing an entirely new way of programming, these primitives reinforce principles that developers already know: manage state carefully, derive values logically, and trigger side effects intentionally. What changes is the level of mechanical sympathy Angular now provides natively, resulting in simpler, safer, and more performant applications.

To visualize this flow, consider a simple example. Imagine a count signal that feeds into a doubleCount computed value, which in turn triggers a DOM update through an `effect()`. This automatic dependency wiring ensures that when count changes, everything downstream updates precisely once, without the developer having to orchestrate the process manually.

To visualize how these primitives work together in practice, consider the following simple Angular component example. Here, you create a count signal, derive a doubleCount computed value from it, and set up an `effect()` that logs whenever the derived value changes.



Sonu Kapoor

Sonukapoor@gmail.com
linkedin.com/in/sonu-kapoor
@sonukapoor1978

Sonu Kapoor has been a software developer since 2003. He's currently focused on web development using the Angular Framework and is a Microsoft MVP and a GDE in Angular. Sonu has written numerous articles that are available on the internet.

Sonu is the author of "Beginning JavaScript Syntax: Understanding Syntactical Rules and Structures for Better JavaScript Programming," published by Apress in 2025.



Component:

```
import {
  signal,
  computed,
  effect
} from '@angular/core';

@Component({})
export class CounterComponent {
  // 1. Create a base signal
  const count = signal(0);

  // 2. Create a computed value based on the signal
  const doubleCount = computed(() => count() * 2);

  // 3. Create an effect that reacts to changes
  constructor() {
    effect(() => {
      console.log(`Double count is now: ${doubleCount()}`);
    });
  }

  increment() {
    // Example mutation
    // Console output: Double count is now: 10
    count.set(5);
  }
}
```

Template:

```
<div>
  <h2>Simple Counter Example</h2>
  <p>Count: {{ count() }}</p>
  <p>Double Count: {{ doubleCount() }}</p>
  <button (click)="increment()">Increment</button>
</div>
```

In the example above, I create a simple CounterComponent using Angular Signals. The count signal holds the primary state, and doubleCount is derived from it using computed(). An effect() runs inside the constructor to log whenever doubleCount changes.

In the template, notice how accessing the value of a signal simply involves calling it as a function with parentheses: count() and doubleCount(). No async pipes, no subscriptions, no observable unwrapping. Updating the state is just as straightforward: calling increment() mutates the signal, triggering all dependent computations and effects automatically.

LinkedSignal: Inputs, Reusability, and Component Boundaries

Angular Signals have already shown their power in simplifying local state management with signal(), computed(), and effect(). But what happens when you need to connect components across boundaries, handle dynamic inputs, or create reusable reactive logic? Enter LinkedSignal, a lesser-known but transformative feature that ties component inputs to reactive state, slashing the boilerplate traditionally associated with RxJS or manual

input tracking. In this section, I'll explore how LinkedSignal streamlines component-to-component reactivity, enhances reusability, and respects Angular's component-driven architecture all while keeping things intuitive and predictable.

What Is a LinkedSignal?

A LinkedSignal is a reactive primitive that binds a component's input (defined as a Signal) to a computed value, automatically updating whenever the input changes. Introduced alongside Angular's Signal API, LinkedSignal (created via linkedSignal()) acts like a computed() value but is specifically designed to work with input Signals. It eliminates the need for lifecycle hooks, manual subscriptions, or RxJS Observables to track input changes, providing a declarative, signal-driven approach to transforming incoming data.

Think of LinkedSignal as a bridge: It connects the external world of component inputs to the internal reactive system of a component, ensuring seamless data flow without the ceremony of traditional Angular approaches. This makes it ideal for scenarios where components need to react to parent-supplied data while maintaining encapsulation and reusability.

Let's start with a practical example to see LinkedSignal in action. Imagine a UserDisplay component that receives a userId input and needs to display a derived value, such as a formatted username. Here's how it looks with LinkedSignal:

```
import { Component, input } from '@angular/core';
import { linkedSignal } from '@angular/core';

@Component({
  selector: 'app-user-display',
  template: `
    <h3>User Display</h3>
    <p>User ID: {{ userId() }}</p>
    <p>Formatted Name: {{ formattedName() }}</p>
  `,
  standalone: true,
})
export class UserDisplayComponent {
  userId = input<number>(0); // Input as a Signal
  formattedName = linkedSignal(() => {
    const id = this.userId();
    return id ? `User-${id}` : 'Unknown User';
  });
}
```

In a parent component, you can pass the userId like this:

```
import {
  Component, signal
} from '@angular/core';
import {
  UserDisplayComponent
} from './user-display.component';

@Component({
  selector: 'app-parent',
  template: `
    <app-user-display
      [userId]="userId()"
    >
  
```

```

<button
  (click)="changeUserId()"
  Change User
</button>
,
standalone: true,
imports: [UserDisplayComponent],
})
export class ParentComponent {
  userId = signal(1);

  changeUserId() {
    this.userId.set(this.userId() + 1);
  }
}

```

Here's what's happening:

- The userId input is defined as a Signal using `input<number>(0)`.
- The formattedName is a LinkedSignal that transforms userId into a formatted string.
- When the parent updates userId (e.g., via a button click), the userId Signal in the child updates, and formattedName recomputes automatically.
- The template binds to both `userId()` and `formattedName()`, rendering updates without async pipes or subscriptions.

Compare this to the RxJS equivalent, where you'd need a `BehaviorSubject`, a pipe with `map`, and lifecycle hooks to manage subscriptions and cleanup. LinkedSignal delivers the same reactivity with a fraction of the code, aligning perfectly with Angular's push toward simplicity.

Reusability: Encapsulating Reactive Logic

One of the standout features of LinkedSignal is its ability to encapsulate reusable reactive logic. By defining LinkedSignal computations in utility functions or services, you can share state transformations across components, reducing duplication and improving maintainability. This is especially valuable in large applications or design systems, where consistent reactivity is critical.

Let's refactor the previous example to make the formatting logic reusable:

```

import { Signal } from '@angular/core';
import { linkedSignal } from '@angular/core';

export function formatUserId(
  userId: Signal<number>
): Signal<string> {
  return linkedSignal(() => {
    const id = userId();
    return id ?
      `User-${id}` :
      'Unknown User';
  });
}

```

Now, any component can use this utility:

```

import {
  Component, input
} from '@angular/core';

```

```

import {
  formatUserId
} from './user.utils';

@Component({
  selector: 'app-user-display',
  template: `
    <h3>User Card</h3>
    <p>{{ formattedName() }}</p>
  `,
  standalone: true,
})
export class UserCardComponent {
  userId = input<number>(0);
  formattedName = formatUserId(
    this.userId
  );
}

```

This approach mirrors the modularity Angular developers love in services or pipes but leverages Signals' reactive nature. It's a clean, reusable way to handle input-driven state without tying logic to a specific component, making it ideal for shared libraries or design systems.

Crossing Component Boundaries

LinkedSignal truly shines when managing reactivity across component boundaries. In traditional Angular, passing reactive data between components often meant using RxJS Observables, which required careful subscription management to avoid memory leaks. LinkedSignal simplifies this by leveraging Signal-based inputs, ensuring reactivity flows naturally from parent to child without manual orchestration.

Consider a dashboard where a parent component passes a selected item's ID to a child component for display:

```

import { Component, input } from '@angular/core';
import { linkedSignal } from '@angular/core';

@Component({
  selector: 'app-item-detail',
  template: `
    <h3>Item Detail</h3>
    <p>Item ID: {{ itemId() }}</p>
    <p>Detail: {{ itemDetail() }}</p>
  `,
  standalone: true,
})
export class ItemDetailComponent {
  itemId = input<number>(0);
  itemDetail = linkedSignal(() => {
    return `Detail for Item ${this.itemId()}`;
  });
}

@Component({
  selector: 'app-dashboard',
  template: `
    <app-item-detail
      [itemId]="selectedId">
    </app-item-detail>
    <button (click)="selectNext()">
      Next Item
    </button>
  `
})

```

```

    ,
  standalone: true,
  imports: [ItemDetailComponent],
})
export class DashboardComponent {
  selectedId = signal(1);

  selectNext() {
    this.selectedId.set(this.selectedId() + 1);
  }
}

```

In this setup:

- The parent's selectedId Signal drives the child's itemId input.
- The child's itemDetail LinkedSignal reacts to itemId changes, computing a derived value.
- Updates flow automatically when selectedId changes, itemId and itemDetail update, and the template reflects the new state.

This eliminates the need for RxJS Subjects, subscriptions, or lifecycle hooks, creating a clean, predictable data flow. It also respects component encapsulation, as the child's logic remains self-contained and unaware of the parent's implementation. Signals also respect Angular's change detection boundary optimizations automatically, without additional developer configuration.

LinkedSignal vs. RxJS: A Side-by-Side Comparison

To appreciate how LinkedSignal reduces complexity, let's rewrite the dashboard example using RxJS without Signals. This

approach relies on traditional Angular patterns with Observables, showcasing the boilerplate that LinkedSignal eliminates.

Listing 1 shows the RxJS version of the ItemDetailComponent and DashboardComponent.

Let's break down the differences:

- **Boilerplate:** The RxJS version requires a BehaviorSubject to track itemId changes, a pipe with map to transform the value, and explicit subscription management in ngOnInit and ngOnDestroy. This adds significant overhead compared to the LinkedSignal version, which handles reactivity declaratively.
- **Lifecycle Management:** With RxJS, you must manually unsubscribe to prevent memory leaks, whereas LinkedSignal leverages Angular's automatic dependency tracking, eliminating lifecycle hooks.
- **State Handling:** The RxJS approach uses a plain number (selectedId) in the parent, requiring manual updates to propagate changes. Signals, on the other hand, ensure reactivity flows naturally from parent to child.
- **Template Simplicity:** In the RxJS version, the template binds to plain properties (itemId and itemDetail), but the logic to keep them in sync is hidden in the component. With LinkedSignal, the template binds directly to Signals (itemId() and itemDetail()), making the reactivity explicit and predictable.

LinkedSignal is more than a convenience: It's a step toward Angular's vision of a simpler, more intentional reactivity

Listing 1: The RxJS version of the ItemDetailComponent and DashboardComponent

```

import {
  Component,
  Input,
  OnInit,
  OnDestroy
} from '@angular/core';
import {
  BehaviorSubject,
  Subscription
} from 'rxjs';
import { map } from 'rxjs/operators';

@Component({
  selector: 'app-item-detail',
  template: `
    <h3>Item Detail</h3>
    <p>Item ID: {{ itemId() }}</p>
    <p>Detail: {{ itemDetail() }}</p>
  `,
  standalone: true,
})
export class ItemDetailComponent implements
  OnInit, OnDestroy {
  @Input() set itemId(value: number) {
    this.itemIdSubject.next(value);
  }
  itemId: number;
  itemDetail: string;
  private itemIdSubject = new BehaviorSubject<number>(0);
  private subscription: Subscription;

  ngOnInit() {
    this.subscription = this.itemIdSubject.pipe(
      map(id => `Detail for Item ${id}`)
    ).subscribe(detail => {
      this.itemId = this.itemIdSubject.value;
      this.itemDetail = detail;
    });
  }

  ngOnDestroy() {
    this.subscription.unsubscribe();
  }
}

@Component({
  selector: 'app-dashboard',
  template: `
    <app-item-detail
      [itemId]="selectedId()">
    </app-item-detail>
    <button
      (click)="selectNext()">
      Next Item
    </button>
  `,
  standalone: true,
  imports: [ItemDetailComponent],
})
export class DashboardComponent {
  selectedId = 1;

  selectNext() {
    this.selectedId++;
  }
}

```

model. By replacing RxJS-driven input handling with a Signal-based alternative, it reduces boilerplate, eliminates common pitfalls (like forgotten unsubscriptions), and aligns with the zoneless, fine-grained change detection Angular is moving toward. It also empowers developers to write reusable, modular code that scales gracefully in large applications.

Compared to RxJS, LinkedSignal is less powerful for complex async workflows (e.g., debouncing or combining streams), but it's unmatched for UI-driven reactivity. As I'll explore in the next section, Signals and RxJS can co-exist, with LinkedSignal handling component state and RxJS tackling advanced event streams.

This comparison highlights why Signals, and LinkedSignal in particular, are a game-changer for Angular developers. They eliminate the need for RxJS in many UI-driven scenarios, reducing complexity and making code easier to reason about.

When to use LinkedSignal:

- Dynamic Inputs:** Use LinkedSignal when a component needs to react to input changes in a declarative way.
- Reusable Logic:** Encapsulate state transformations in utilities for shared components or libraries.
- Component Communication:** Simplify parent-child reactivity without RxJS or lifecycle hooks.
- Simplicity:** Opt for LinkedSignal when you want reactive UI updates without the overhead of Observables.

If you need advanced stream manipulation, stick with RxJS or combine it with Signals (e.g., using `toSignal()` to convert Observables). The beauty of Angular's approach is that you're not forced to choose; you can use the right tool for the job.

Table 1 underscores how LinkedSignal simplifies the developer experience, making reactivity intuitive and reducing the potential for errors like memory leaks or stale data.

Signals vs. RxJS: When to Use What

The introduction of Angular Signals raises a natural question for developers who have long relied on RxJS for reac-

tivity: Where do these two powerful systems overlap, and where does each shine? Although Signals and RxJS may appear similar at first glance, especially to those new to reactive programming, they are fundamentally designed for different purposes. Understanding when to reach for Signals versus when to embrace the full power of RxJS is essential for building clean, maintainable, and high-performance Angular applications.

At their core, Signals are about state management. They provide a simple, synchronous, and fine-grained mechanism to model local or UI-driven state within a component or a closely related set of components. Signals excel when you need a value to update predictably in response to user interaction, a parent input, or a computed relationship between other pieces of state. Because Signals are synchronous and immediately reflect their latest value, they feel intuitive for tasks like form control, toggling UI elements, managing user selections, or deriving secondary state based on one or more primary signals.

RxJS, on the other hand, was built to model streams over time. It shines in situations where you are dealing with asynchronous flows, cancellation, complex coordination between multiple sources, debouncing, throttling, retries, backoff strategies, or real-time event streams like WebSocket connections. RxJS offers operators like `switchMap`, `mergeMap`, `combineLatest`, and `retryWhen`, which are indispensable for sophisticated asynchronous workflows that extend beyond simple UI reactivity. Unlike Signals, observables are inherently lazy and asynchronous, meaning they only emit when subscribed to, and they can model both discrete events and continuous processes over time.

In a typical modern Angular application, a pragmatic division of responsibilities emerges. Signals dominate inside components. They drive the UI, handle local state, manage form values, and compute derived properties for rendering. RxJS remains critical outside components, particularly in services and infrastructure layers, where you interact with external systems like HTTP APIs, WebSockets, authentication providers, and large event-based architectures. This separation of concerns aligns beautifully with Angular's architectural vision: Signals simplify the

Aspect	LinkedSignal	RxJS
State Management	<code>selectedId</code> is a Signal, inherently reactive.	<code>selectedId</code> is a plain number, requiring manual updates to propagate changes.
Input Handling	<code>itemId</code> is an input Signal, automatically reactive with <code>linkedSignal()</code> .	<code>itemId</code> uses a <code>BehaviorSubject</code> , requiring a setter to update the Subject manually.
Derived Values	<code>itemDetail</code> is computed declaratively with <code>linkedSignal()</code> .	<code>itemDetail</code> requires a pipe with <code>map</code> and a subscription to compute the derived value.
Lifecycle Hooks	None needed, Angular handles reactivity automatically.	Requires <code>ngOnInit</code> and <code>ngOnDestroy</code> to manage subscriptions and prevent memory leaks.
Template Binding	Binds directly to Signals (<code>itemId()</code> , <code>itemDetail()</code>), explicit and reactive.	Binds to plain properties, with reactivity logic hidden in the component.
Code Complexity	Minimal: ~10 lines of logic, declarative.	Higher: ~20 lines of logic, imperative, with manual subscription management.

Table 1: Comparison: LinkedSignal vs. RxJS

reactive story at the UI layer, while RxJS remains a powerful tool for managing complex, asynchronous business logic and system interactions.

Consider a simple typeahead search feature. With Signals, you can model the current input value and the loading state of the UI. When the user types, a signal() holds the latest query string, and a computed or effect can trigger a data fetch. However, if you want to debounce the input (e.g., wait 300ms after typing stops) before triggering a search, you'd still lean on RxJS's debounceTime() operator, possibly converting the debounced stream back into a signal using toSignal(). This hybrid approach leverages the strengths of both systems: Signals for clean, declarative state updates, and RxJS for precise timing control over asynchronous user input.

It's important to adopt the right mental model when working with these tools. With Signals, you think in terms of values and direct dependencies. A change to one value immediately flows through all dependent computations and effects. With RxJS, you think in terms of events and transformations over time, where data may be buffered, delayed, canceled, or merged with other streams before reaching its destination. Signals offer immediacy and predictability, and RxJS offers flexibility and control over complex asynchronous flows.

Rather than replacing RxJS, Signals complement it by handling the vast majority of simpler, synchronous UI state management scenarios that previously required verbose observable pipelines. The result is cleaner, more declarative components with less boilerplate, fewer subscriptions to manage, and a clearer separation between UI and service layers. Developers no longer need to force-fit RxJS into every part of their applications, freeing RxJS to focus on the scenarios where its strengths truly matter.

As Angular continues to evolve, the trend is clear: Signals will increasingly be the default choice for most UI-driven reactivity, and RxJS will remain a vital tool for dealing with asynchronous complexity. Mastering both and know-

ing when to apply each is the key to building robust, scalable, and maintainable Angular applications in the Signals era.

In large-scale Angular applications, state management libraries like NgRx provide a concrete example of where RxJS continues to play a critical role. NgRx's traditional Store, Effects, and Entity modules are deeply based on RxJS streams, and they remain an excellent choice for complex, event-driven architectures requiring time-based event coordination, advanced side effects handling, and scalability across large teams.

That said, even within the NgRx ecosystem, Signals are beginning to reshape the reactive story. The recently introduced SignalStore package offers a new, Signals-first API for state management, blending the fine-grained reactivity of Signals with the robustness of NgRx tooling. SignalStore internally leverages some RxJS capabilities, but developers interact with state primarily through Signals and computed properties, drastically reducing observable boilerplate for common patterns.

This evolution illustrates Angular's broader reactive future: RxJS remains essential for advanced workflows, and Signals dramatically simplify UI and local state management, even inside libraries traditionally built entirely on observables.

Real-World Patterns with Signals

Theory alone cannot capture the transformative simplicity that Angular Signals bring to modern front-end development. Real power comes from seeing how common patterns, often complex and verbose when built with traditional observables, become drastically simpler, more declarative, and more maintainable when rebuilt using Signals. In this section, I'll explore practical, real-world problems that every Angular developer encounters and show how Signals provide clean solutions that reduce boilerplate while enhancing clarity and performance.

Typeahead

Typeahead is a user interface feature that provides real-time suggestions as the user types into an input field. By predicting and displaying possible matches from a dataset, typeahead helps users complete their queries faster and with fewer errors. This improves efficiency and user experience, especially in search bars, forms, and dropdowns where quick, relevant results are important. Most websites, such as Google and YouTube, use this feature to enhance the user experience.

Aspect	Signals	RxJS	NgRx Store	NgRx SignalStore
Purpose	Fine-grained local UI state, synchronous	Asynchronous event streams, complex flows	Global app-wide state management	Global state with Signals-first API
Best for	UI bindings, forms, derived state, component inputs	HTTP polling, debouncing, cancellations, websockets	Large applications, cross-feature communication	Large applications, but with simpler Signal-driven APIs
Learning Curve	Low	Medium to High	High (needs understanding of actions, reducers, effects)	Medium (simpler API, but still NgRx patterns)
Boilerplate	Minimal	Moderate	High	Low
Zone.js dependency	None	None	None	None
Example Operators	N/A (synchronous)	debounceTime, switchMap, mergeMap	combineReducers, createEffect	createComputed, createSignalSelector
Ideal Usage	Inside components	In services, APIs, complex side effects	Cross-app global state, enterprise apps	Same, but with Signals as the first-class citizen

Table 2: Signals, RxJS, and NgRx: Choosing the Right Tool for Reactive State Management

A Reactive Typeahead Input

The typeahead pattern, where users type into a search box and see suggestions appear, is a classic front-end challenge. Traditionally, building a robust typeahead involved juggling several concerns: capturing user input, debouncing rapid keystrokes, handling loading states, triggering API calls, and rendering the results reactively. In a pure RxJS approach, this often required chaining multiple operators, carefully managing subscriptions, and dealing with race conditions manually. With Signals, much of this complexity falls away.

Local UI state, such as the current search term, the loading indicator, and the list of search results, can be modelled directly as Signals. As previously explained, a signal() can track the user's input in real-time, a computed() can derive debounced input if needed, and an effect() can handle fetching new data when the debounced input changes.

Listing 2 shows a simple TypeaheadComponent.

In this example:

- The user's raw input is captured in the query signal.
- To debounce rapid input changes, toObservable(this.query) converts the signal into an observable, which is then piped through debounceTime(300). The de-

bounced observable is converted back into a signal using toSignal(), resulting in a reactive, debounced query value.

- An effect() monitors the debounced query. When it changes, the component triggers a new HTTP request to fetch search results.
- The loading and results states are managed as Signals, ensuring the template updates automatically without async pipes or manual subscription management.

This design leads to a smooth, declarative flow where Signals manage state, effects handle side effects, and the template reflects the latest state naturally. Compared to traditional RxJS-heavy approaches, the code is cleaner, easier to understand, and easier to maintain.

The Typeahead pattern demonstrates a larger principle: by combining Signals for local UI state and selectively using RxJS for stream transformations, Angular developers can build dynamic, highly responsive applications without the ceremony and cognitive overhead of earlier reactive patterns.

Conditional Rendering Using the New Control Flow Syntax

Angular's new control flow syntax, introduced alongside standalone components and Signals, unlocks a far more declarative and composable way to handle conditional

Listing 2: A simple TypeaheadComponent

```
import { Component, effect, signal } from '@angular/core';
import { toObservable, toSignal } from '@angular/core/rxjs-interop';
import { debounceTime } from 'rxjs/operators';

@Component({
  selector: 'app-typeahead',
  standalone: true,
  template: `
    <input
      #searchInput
      type="text"
      [value]="query()"
      (input)="updateQuery(searchInput.value)"
      placeholder="Search..." />
    <if(loading()) {
      <p>Loading...</p>
    }
    <ul>
      <for(item of results()); track item) {
        <li>{{ item }}</li>
      }
    </ul>
  `,
})
export class TypeaheadComponent {
  query = signal('');
  loading = signal(false);

  debouncedQuery = toSignal(toObservable(this.query))
    .pipe(debounceTime(300)), {
      initialValue: '',
    });

  results = signal<string>([]);

  constructor() {
    effect(() => {
      const search = this.debouncedQuery();
      if (!search) {
        this.results.set([]);
        return;
      }

      this.loading.set(true);

      const mockResults = [
        'Apple',
        'Banana',
        'Cherry',
        'Date',
        'Elderberry',
        'Fig',
        'Grape',
        'Honeydew',
      ];

      // Simulating a delay to mimic an HTTP request
      setTimeout(() => {
        this.results.set(
          mockResults.filter((item) =>
            item.toLowerCase().includes(search.toLowerCase())
          )
        );
        this.loading.set(false);
      }, 3000);
    });

    updateQuery(value: string) {
      this.query.set(value);
    }
  }
}
```

rendering in templates. Previously, developers relied heavily on structural directives like `*ngIf` and `*ngFor`, which, while powerful, could become verbose and hard to nest cleanly inside more reactive patterns.

Signals integrate naturally with the new control flow syntax, allowing for fine-grained template updates based on reactive state changes without the ceremony of traditional observables or manual event handling. By using Signals as the source of truth for visibility conditions, Angular can optimize change detection and DOM updates, resulting in both cleaner code and better runtime performance.

Consider a simple example: a component that toggles between showing a login form and a dashboard view based on a user's authentication status.

```
import { Component, signal } from '@angular/core';

@Component({
  selector: 'app-login-dashboard',
  standalone: true,
  template: `
    @if (isAuthenticated())
      <h2>Welcome Back!</h2>
      <button (click)="logout()">Logout</button>
    } @else {
      <h2>Please Log In</h2>
      <button (click)="login()">Login</button>
    }
  `

})
export class LoginDashboardComponent {
  isAuthenticated = signal(false);

  login() {
    this.isAuthenticated.set(true);
  }

  logout() {
    this.isAuthenticated.set(false);
  }
}
```

In this example:

- The `isAuthenticated` signal drives the template logic reactively.
- Angular's `@if ... @else` control flow blocks switch the rendered content based on the current authentication state.
- Because `isAuthenticated` is a signal, Angular automatically knows when to rerender the correct block, without needing any explicit calls to `ChangeDetectorRef`, manual subscriptions, or async pipes.

The tight integration between control flow syntax and Signals allows developers to think more declaratively about UI states. Instead of wiring conditions manually in the component class, developers express their intent directly in the template, powered by reactive primitives. This approach reduces boilerplate, improves readability, and enables Angular's fine-grained reactivity system to optimize DOM operations at runtime.

By combining Signals with the new control flow syntax, Angular developers can build highly dynamic interfaces that remain clean, predictable, and maintainable even as applications grow in complexity.

Guard Logic with `computed()` for Route Protection

Routing in Angular often involves protecting certain routes based on application state: whether a user is authenticated, has a certain role, or has completed an onboarding flow. Traditionally, route guards were written imperatively, relying on injected services and manually checking conditions in `canActivate()` methods, often returning Booleans or observables.

With Signals and `computed()`, Angular developers can now model guard conditions reactively, creating a clean, declarative way to manage access control based on live application state. Instead of scattering authorization logic across services and guards, Signals allow for centralizing and composing guard conditions in a predictable and testable way.

Here's a practical example: protecting a dashboard route based on whether a user is logged in.

```
import {
  Injectable,
  computed,
  signal
} from '@angular/core';
import {
  CanActivateFn,
  Router
} from '@angular/router';

@Injectable({
  providedIn: 'root'
})
export class AuthService {
  user = signal<{
    id: number,
    name: string
  } | null>(null);

  isLoggedIn = computed(() => !!this.user());
}

export const canActivateDashboard:
  CanActivateFn = (route, state) => {
  const authService = inject(AuthService);
  const router = inject(Router);

  if (authService.isLoggedIn()) {
    return true;
  } else {
    router.navigate(['/login']);
    return false;
  }
};
```

In this example:

- The `AuthService` uses a `signal()` to track the current user state.
- A `computed()` signal `isLoggedIn` derives its truthiness based on whether the user is present.

- The route guard canActivateDashboard reads the computed signal directly at the time of route activation.
- If the user is not logged in, the guard redirects them to the login page.

This approach is both more declarative and more reactive than traditional service-based guards. If additional rules were needed, such as checking user roles or subscription status, new computed signals could be composed cleanly on top of the existing ones, without breaking the reactive graph or introducing side effects.

Signals make guard logic more transparent, more testable, and better aligned with Angular's direction toward fine-grained, zoneless architectures. They reduce the imperative ceremony around route protection and allow guards to respond naturally to the live, reactive state of the application.

HttpResource: Signal-Based Data Fetching

Data fetching has always been a central part of building Angular applications. Managing HTTP requests, tracking loading states, handling errors, caching responses, and updating the UI have historically required developers to combine several concepts, including HttpClient, observables, subscriptions, and manual state management. Although these tools are powerful, they often lead to verbose patterns, especially in components where fine-grained reactivity is critical.

In Angular 19.2, a new experimental feature called HttpResource was introduced to simplify this problem dramatically. HttpResource is a signal-powered abstraction over data fetching, designed to handle the common lifecycle concerns of remote requests, loading, success, error handling, and caching without requiring developers to manually manage subscriptions or async pipes.

HttpResource is created by calling the httpResource() function from @angular/core. This function takes a fetcher function as input, typically one that uses HttpClient to return an observable or promise. In return, it provides several Signals that automatically track the status of the request:

- data()** : A signal holding the successfully fetched data.
- error()** : A signal holding any error encountered during fetching.
- status()** : A signal indicating the current fetch state (initial, pending, success, or error).

- isLoading()** : A Boolean signal that becomes true during a request.
- reload()** : A method to manually re-trigger the fetch.

HttpResource fits perfectly into Angular's broader move toward fine-grained, declarative reactivity. Instead of wiring manual subscriptions, managing multiple loading flags, and manually triggering change detection, HttpResource allows developers to work with Signals directly, bringing a new level of simplicity and predictability to data fetching.

Let's revisit our earlier TypeaheadComponent and enhance it using httpResource(), as shown in Listing 3.

In this version:

- User input is captured via the query signal.
- The input is debounced using toObservable and toSignal.
- The additional check for the debounceQuery Signal ensures that the request is not executed until that Signal has been set.
- httpResource() watches the debounced query and automatically triggers a new HTTP request whenever it changes.
- The template uses Signals exposed by HttpResource (isLoading(), error(), value()) to declaratively render the appropriate UI.

There is no need for subscriptions, manual loading flags, or defensive async pipes. State changes propagate naturally through Signals, making the component smaller, clearer, and easier to maintain.

HttpResource also introduces built-in caching. If the same parameters are used again, Angular avoids re-sending the request unless explicitly instructed to reload(). Error handling is first-class as well: Developers can easily respond to fetch failures without extra boilerplate or duplicated error checks in the template.

Table 3 offers a quick comparison.

In short, HttpClient is still the right choice when:

- You're making API calls inside a service layer.
- You need full control over request/response pipelines.
- You're composing complex RxJS streams (e.g., retries, joins, batching).

Aspect	HttpClient	HttpResource
State Management	Manual (loading flags, error handling)	Automatic via Signals (isLoading(), data(), error(), status())
Subscriptions	Required (.subscribe())	Not needed, data is pushed into Signals
Caching	Manual or via interceptors	Built-in default caching
Template Binding	async pipe often required	Direct binding with signal() calls
Boilerplate	Moderate to high	Very low
Best For	Bulk API calls, service layers, complex flows	UI-driven data fetching, fine-grained component reactivity

Table 3: A comparison between HttpClient and HttpResource

Listing 3: Enhanced TypeaheadComponent

```
import { httpResource } from '@angular/common/http';
import {
  Component, computed, signal
}
from '@angular/core';
import {
  toObservable, toSignal
}
from '@angular/core/rxjs-interop';
import { debounceTime } from 'rxjs';

@Component({
  selector: 'app-typeahead-with-http-resource',
  imports: [],
  template: `
    <h2>Typeahead with HTTP Resource</h2>

    <input
      #searchInput
      type="text"
      [value]="query()"
      (input)="updateQuery(searchInput.value)"
      placeholder="Search...">
  >

  @if (isLoading()) {
    <p>Loading...</p>
  } @else if (error()) {
    <p>Error loading results.</p>
  } @else {
    <ul>
      @for(item of products() || [] ; track item) {
        <li>{{ item }}</li>
      }
    </ul>
  }
  ,
})

```

```
export class TypeaheadWithHttpResourceComponent {
  query = signal('');
  debouncedQuery = toSignal(toObservable(this.query))
    .pipe(debounceTime(300)),
    initialValue: '',
  });

  #searchResource = httpResource<{
    products: { id: number; title: string }[]
  }>()

  () => {
    if (!this.debouncedQuery()) {
      // Skip the request if the query is empty
      return undefined;
    }

    const query = encodeURIComponent(this.debouncedQuery());
    const url = `.../search?q=${query}`;
    return url;
  };

  isLoading = computed(() => this.#searchResource.isLoading());
  error = computed(() => this.#searchResource.error());
  data = this.#searchResource.value;
  products = computed(() => {
    return this.data()?.products.map(
      (product) => product.title) || [];
  });
  updateQuery(value: string) {
    this.query.set(value);
  }
}
```

HttpResource is ideal when:

- You want to fetch data directly inside a component.
- You need automatic loading/error handling.
- You want the data-fetching state to drive the UI declaratively.
- You want minimal boilerplate and maximum clarity.

HttpResource enables a new mental model for front-end developers: Treat remote data as live state, just like local variables, and let the framework handle the complexities of the HTTP lifecycle reactively and predictably.

Gradual Migration Strategies

One of the key strengths of Angular's Signals system is that it was designed with gradual adoption in mind. Unlike some previous architectural shifts that required all-or-nothing rewrites, Angular Signals allow teams to introduce fine-grained reactivity incrementally, alongside existing codebases built with RxJS, `@Input()`/`@Output()`, and traditional change detection mechanisms. This flexibility ensures that projects of any size, complexity, or legacy burden can benefit from Signals without introducing instability or major technical debt.

Signals can coexist peacefully with existing patterns because Angular provides deliberate interop bridges be-

tween the old and the new. Developers can convert an observable into a signal using `toSignal()` or expose a signal as an observable using `toObservable()`. Similarly, traditional `@Input()` bindings can be upgraded to `input()` to create linked signals reactively, without breaking component APIs. Effects and computeds can live side-by-side with older imperative event handlers, making migration paths organic rather than disruptive.

A pragmatic migration strategy often begins at the component level. Instead of trying to refactor entire services or state management layers immediately, developers can first refactor simple, isolated components that primarily deal with local UI state. Components with form controls, visibility toggles, tab switching logic, or basic computed displays are perfect candidates. These migrations tend to be straightforward: replace class properties with `signal()`, replace derived getters with `computed()`, and replace imperative methods like `markForCheck()` with reactive `effect()` blocks.

After developers learn to manage local state using Signals, the next step is to connect service-layer observables to signals. By using the `toSignal()` function, you can convert an RxJS stream from a service into a reactive signal within your component. This keeps your component signal-based and removes the need to handle observables directly. As a result, template bindings become much sim-

pler: Instead of using async pipes and checking for null values, you can bind directly to signal values using the `{} signal()` syntax.

As adoption grows, projects can begin to upgrade critical interaction points like route guards, loading states, and even reactive forms to signal-driven versions. New features and modules can be built natively with Signals, and legacy areas of the application continue using observables and services as before. This incremental opt-in model ensures that teams never need to pause feature development simply to adopt Signals; they can phase in improvements naturally over time.

One of the most effective ways to begin migrating to Signals is by targeting simple, low-risk parts of the application first. Local state variables such as counters, toggles, form field values, or loading flags can be easily rewritten as `signal()` instances with minimal code changes. The accompanied code contains several examples of these using Signals. Derived state, often expressed today as getters or manual recalculations inside methods, can be naturally upgraded to `computed()` signals, offering cleaner declarative data flows. Child components that currently use `@Input()` properties can transition to using `input()`, automatically gaining reactive linked signals without breaking their external API. In templates, structural directives like `*ngIf` and `*ngFor` can be rewritten using the new `@if` and `@for` control flow syntax, unlocking finer-grained rendering optimizations driven by signals rather than coarse view checks. By starting with these accessible, low-complexity migrations, teams can quickly build confidence with Signals while laying the groundwork for broader adoption across larger, more critical parts of the codebase.

It's important, however, to be thoughtful during migration. Signals and RxJS are powerful tools but solve slightly different problems. Replacing observables purely for the sake of using Signals can lead to unnecessary complexity, especially in asynchronous, event-driven workflows where RxJS still excels. A good rule of thumb is to use Signals for local synchronous UI state and RxJS for asynchronous operations, system events, and service communications.

A good rule of thumb is to use Signals for local synchronous UI state and RxJS for asynchronous operations, system events, and service communications.

Tooling and developer ergonomics are improving as well. The Angular Language Service, component-level DevTools inspections, and community libraries are rapidly evolving to offer better support for Signals, making it easier to spot reactive flows, debug dependency graphs, and optimize fine-grained reactivity across the stack.

Ultimately, the gradual migration path for Signals is a significant achievement for Angular's ecosystem. It empow-

ers teams to adopt the future of reactive programming on their own terms, balancing innovation with stability. By starting small, iterating carefully, and applying the right tool for each job, developers can transform Angular applications to be faster, cleaner, more declarative, and more maintainable, without ever facing a "rewrite or die" dilemma.

Signals and Performance: What Changes Under the Hood?

As with any powerful abstraction, Signals unlock new possibilities for building reactive applications, but they also introduce new pitfalls for developers unfamiliar with the model. Being aware of common mistakes early on can save teams from unnecessary complexity, performance regressions, and confusing bugs. Fortunately, most of these mistakes stem from incorrect mental models inherited from working with RxJS or traditional Angular state management.

One of the most frequent mistakes is misusing `effect()` for derived state. New users often reach for `effect()` whenever they want to compute something reactively. However, `effect()` is intended for triggering side effects, such as making service calls, interacting with the DOM, or dispatching actions, not for computing new values. When you need to derive a new value from one or more signals, you should always prefer `computed()`. This distinction ensures that your reactive graph remains pure and declarative, while side effects are isolated and intentional.

Another common trap is forgetting that Signals are synchronous. Unlike observables, Signals update immediately when their dependencies change. There is no concept of asynchronous emission inside a Signal's getter. Developers sometimes mistakenly expect Signals to behave like observables, thinking that downstream updates will be deferred or batched. This misunderstanding can lead to confusing bugs where state changes happen earlier than anticipated. When working with Signals, always assume synchronous propagation unless you intentionally introduce asynchronous behavior via RxJS operators at the service layer.

Over-engineering is another frequent issue. Signals offer a natural, lightweight way to model local and derived state, but in some cases, developers wrap simple values or create unnecessary layers of indirection "just to be reactive." Not every piece of local data needs to be a signal; sometimes a plain field or a straightforward method is sufficient. Remember: Signals shine most when you are modeling dynamic, shared, or computed state that benefits from fine-grained reactivity.

Developers may also recreate observables unnecessarily when working with hybrid codebases. Although `toObservable()` is a powerful bridge, it should be used thoughtfully. Signals are optimized for direct, synchronous dependency tracking inside components. Rewrapping them into observables inside templates or unnecessary service layers often reintroduces the very complexity that Signals were designed to remove.

Finally, a subtle but important consideration is managing resource lifecycles when combining Signals with other

Angular constructs. For example, when creating effects inside services that outlive their components, it's crucial to ensure that side effects and subscriptions are properly disposed of to avoid potential memory leaks. Although Angular automatically handles cleanup for effects created inside component trees, long-lived singleton services require more care. The Angular team has acknowledged the need for formal lifecycle management hooks for Signals and Effects, and this is currently under active discussion for future releases. Until such APIs are available, developers must be cautious when wiring Signals and Effects inside services or globally scoped objects.

Signals are a powerful addition to Angular's toolkit, but they demand a slight shift in mindset toward simpler, more synchronous, more declarative design. By being aware of these common mistakes early on, developers can leverage Signals to their fullest potential, building applications that are cleaner, faster, easier to reason about, and easier to scale.

Conclusion: A Simpler, More Predictable Angular

Angular's evolution has always been about balancing power with pragmatism. The introduction of Signals marks one of the most profound shifts in how developers model state, data flows, and UI reactivity within Angular applications. Rather than layering more complexity onto an already mature framework, Signals simplify. They allow developers to reason about application state synchronously, declaratively, and with an unprecedented level of fine-grained control.

Signals unlock a new reactive mental model, one that aligns with how modern users expect web applications to behave: fast, responsive, and intelligent. They remove the need for heavy-handed change detection strategies, reduce boilerplate around inputs, outputs, and subscriptions, and offer a composable, ergonomic way to build complex interfaces without drowning in streams and lifecycles.

At the same time, Signals respect Angular's commitment to stability. They do not force existing projects into painful rewrites. They provide bridges, gradual migration paths, and tooling to ensure that teams can adopt this next-generation model on their own terms. Whether you are building small UI interactions, enhancing form management, optimizing rendering, or rethinking global state management strategies, Signals open the door to simpler, more predictable, and more maintainable Angular applications.

The future of Angular is reactive, declarative, and powered by Signals. It has already begun.

are not merely another API to learn: They represent a re-thinking of Angular's core reactive story. Teams that embrace Signals today will find themselves best positioned to build the fast, scalable, and resilient applications that tomorrow's web demands.

Although this article focused on introducing Signals and their role in modern Angular development, there are even more powerful patterns emerging. In future installments, I'll explore advanced topics such as integrating Signals with Reactive Forms, building localization workflows with reactive primitives, designing custom Signal utilities, and evolving large-scale state management strategies. Signals aren't just a feature. They're a foundation, and their possibilities are only beginning.

As it matures beyond experimental status, `HttpResource` promises to be a major part of building more responsive, maintainable, and performant Angular applications.

Sonu Kapoor


As Angular moves toward a zone-less future, with faster initial render times, smarter hydration strategies, and enhanced concurrency, Signals will be the foundation. They

The New Features and Enhancements in .NET 10 (Part 2)

Microsoft plans to release .NET 10 later this year. It will be a Long-Term Support (LTS) release, providing all you need to build and deploy high-performance, scalable, reusable, and extensible applications that can meet the demands of today's businesses. The addition of .NET 10 in the .NET ecosystem features a collection of new features and enhancements throughout the .NET stack.

As of this writing, .NET 10 Preview 5 is available. You can download a copy of .NET 10 Preview here: <https://dotnet.microsoft.com/en-us/download/dotnet/10.0>.

This article presents an overview of the new features and enhancements in .NET 10 Preview 6. To work with the code examples discussed in this article, the following software must be available:

- Visual Studio 2022 Preview
- .NET 10.0 Preview
- ASP.NET 10.0 Preview Runtime

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

Migrating from Previous Versions to .NET 10

There are several enhancements in this release. The key enhancements in .NET 10 include:

- Enhanced performance
- Better scalability
- Improved security and compliance
- Support for cloud-native applications
- Enhancements in AI and machine learning
- Enhancements in developer productivity

To migrate your application from an earlier version of .NET to .NET 10, keep specific points in mind.

First, review the system requirements and confirm that they are satisfied. For example, you should have Visual Studio 2022 (or its Preview Version) installed on your computer. Alternatively, if you're using JetBrains Rider as the IDE, you should install the latest version of JetBrains Rider.

The next step is to download the .NET 10 SDK from Microsoft's download site. You can check the version of .NET installed on your computer using the following command in the Command window:

```
dotnet -version
```

You can get a list of all .NET SDKs installed on your computer using the following command:

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

The next step is to change the project files (i.e., .csproj files) in your application to target the .NET 10 framework, as shown in the code snippet below:

```
<TargetFramework>net10.0</TargetFramework>
```

Finally, upgrade your NuGet packages to make them compliant with .NET 10.

To avoid compatibility issues with the version(s) of .NET and the SDKs installed on your computer, upgrade the NuGet packages by executing the following commands:

```
dotnet nuget locals all --clear  
dotnet restore  
dotnet outdated  
dotnet add package Microsoft.AspNetCore.App  
--version 10.0.0
```

Using the .NET 10 Preview Features in the Visual Studio IDE

To use .NET 10 Preview features in your Visual Studio IDE, edit the project files and specify the LangVersion and TargetFramework elements as shown below:

```
<Project Sdk=>Microsoft.NET.Sdk</>  
<PropertyGroup>  
    <OutputType>Exe</OutputType>  
    <TargetFramework>net10.0</TargetFramework>  
    <LangVersion>preview</LangVersion>  
    <ImplicitUsings>enable</ImplicitUsings>  
    <Nullable>enable</Nullable>  
</PropertyGroup>  
</Project>
```



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.



Enhancements to the .NET Libraries

With .NET 10, several new features and enhancements have been introduced in .NET libraries. In this section, I'll examine the enhancements made to .NET libraries in this release.

String-Based Normalization APIs

With .NET 10, Microsoft has included support for string-based normalization APIs that can help you normalize `Span<char>` or `ReadOnlySpan<char>` with the need to create new strings. The following code snippet illustrates the `StringNormalizationExtensions` class that includes the newly added methods:

```
public static class  
StringNormalizationExtensions {  
    public static int GetNormalizedLength(  
        this ReadOnlySpan<char> source,  
        NormalizationForm normalizationForm  
    = NormalizationForm.FormC);
```

```

public static bool IsNormalized(
    this ReadOnlySpan<char> source,
    NormalizationForm normalizationForm
    = NormalizationForm.FormC);
public static bool TryNormalize(
    this ReadOnlySpan<char> source,
    Span<char> destination,
    out int charsWritten,
    NormalizationForm normalizationForm
    = NormalizationForm.FormC);
}

```

The methods of the `StringNormalizationExtensions` class in the preceding code snippet are extension methods. The following code sample shows how to use the extension methods of the `StringNormalizationExtensions` class:

```

char[] buffer = new char[50];
ReadOnlySpan<char> input =
"Hello World!".AsSpan();
Span<char> output = buffer;

if (!input.IsNormalized
(NormalizationForm.FormC)) {
    if (input.TryNormalize(output,
    out int written,
    NormalizationForm.FormC)) {

        // Write your code here to
        // display the serialized data
    }
}

```

Extension methods are a special type of method used to attach additional functionality to an existing type without the need to create a new derived type, recompile the source code, or alter the original type.

Enhancements in ML and AI

As modern businesses lean on predictive analytics and automated decision systems, a reliable framework can help them streamline their business processes and shorten the time-to-market for new insights. With .NET 10, you can now take advantage of ML.NET framework, for a more effortless integration with applications that leverage AI. In addition, you can take advantage of these enhancements in .NET 10 to streamline predictive analytics, automation, and AI-powered decision-making. These enhancements offer streamlined project workflows, and improved performance for your AI workloads within the .NET ecosystem, thereby enabling you to plug machine learning models straight into your existing applications that leverage AI/ML, empowering you to build, test, and deploy smarter applications with confidence.

HybridCache Enhancements in .NET 10

A hybrid cache combines the best of both worlds: It combines both in-memory and distributed caching. In essence,

it's a caching strategy that provides support for two-level caching, that is, L1 (for in-memory cache) and L2 (for distributed cache, such as Redis). With .NET 10, HybridCache has been significantly enhanced. Incidentally, HybridCache is a part of the `Microsoft.Extensions.Caching.Hybrid` namespace.

The key features of HybridCache include:

- **Abstraction:** HybridCache provides a single abstraction for in-memory and distributed caching. When you're using HybridCache, you can just call the `GetOrCreateAsync` method. The HybridCache library takes care of storage, serialization, stampede prevention, etc.
- **Tagging:** A great new feature added to HybridCache in .NET 10 is the support for tagging that enables you to add tags to the cached items and invalidate all of them when required. You can use these tags to categorize your cache data, i.e., create and manage groups of related cache entries.
- **Compatibility:** Because HybridCache is built on top of the `IDistributedCache` interface, you can use it when working with SQL Server, Redis, Azure Cache, or any existing distributed cache provider.

Using HybridCache

To be able to use HybridCache in your application, you must install the HybridCache NuGet package using the following command:

```
Install-Package Microsoft.Extensions.Caching.Hybrid
```

Add HybridCache services to the ASP.NET Core request processing pipeline using the following code snippet in the `Program.cs` file:

```
builder.Services.AddHybridCache();
```

Let's understand how to use tags in HybridCache using a code example. Consider the following piece of code that shows a record type named `Product`:

```

public record Product
{
    public int Product_Id
    { get; set; }

    public string Product_Name
    { get; set; } = default!;

    public string Product_Description
    { get; set; } = default!;

    public int Product_Category_Id
    { get; set; }

    public string Product_Category
    { get; set; } = default!;

    public decimal Product_Price
    { get; set; } = default!;

    public int Product_Quantity
    { get; set; } = default!;
}

```

Listing 1 illustrates how you can retrieve products from the database based on a particular product category. Note how tags have been created with each tag representing a particular product category.

Improved Data Handling

With .NET 10, there have been a few critical updates to how you can work with text and data in your applications. This release of .NET provides support for improved string normalization with span-based APIs, improved JSON serialization, and elegant handling of UTF-8 strings. The new ISOWeek class pertaining to the System.Globalization namespace works nicely with the DateOnly type now. In the earlier versions of .NET, the ISOWeek class supported only DateTime.

With C# 14, you can compare strings numerically instead of lexicographically. To take advantage of the CompareOptions enum, install the System.Globalization NuGet package into your project and include the System.Globalization assembly in your programs.

The following code snippet illustrates how you can work with the ISOWeek class in C#:

```
using System;
using System.Globalization;

DateOnly date = ISOWeek.ToDateTime(
    2025, 30, DayOfWeek.Monday);
int week = ISOWeek.GetWeekOfYear(date);
```

Improvements in JSON Serialization

When it comes to converting objects into JSON, .NET 10 offers a lot of flexibility. JSON serialization receives an upgrade in .NET 10, introducing new options for handling object references. With the new JsonSourceGenerationOptionsAttribute, you can fine-tune reference handling during

source generation, giving fine-grained control over how source-generated serializers operate. That means lighter payloads and faster processing when every millisecond counts. As a result, it reduces serialization overhead, which is especially welcome in time-critical real-life applications

The following code snippet illustrates how you can use JsonSourceGenerationOptionsAttribute to decorate your class:

```
[JsonSourceGenerationOptions(
    WriteIndented = true,
    DefaultIgnoreCondition
    = JsonIgnoreCondition
        .WhenWritingNull,
    PropertyNamingPolicy
    = JsonKnownNamingPolicy.CamelCase)]
[JsonSerializable(typeof(Author))]

public partial class
    AuthorSerializationContext
    : JsonSerializerContext {
    // Implementation code of the
    // AuthorSerializationContext class
    // is not provided here for brevity
}
```

To work with the JsonSourceGenerationOptionsAttribute, include the following namespaces in your code:

In this release, the JSON serialization has been enhanced by adding a few best practices. A new JsonSerializerOptions. Strict preset has been added that can disable the AllowDuplicateProperties property, enable the RespectNullableAnnotations and RespectRequiredConstructorParameters settings, apply the JsonUnmappedMemberHandling. Disallow policy, and preserve property binding in a case-sensitive manner.

Listing 1: Retrieving products based on Product Category using tags

```
public async ValueTask<List<Product>>
GetProductWithTagsAsync (
    int productCategoryId,
    CancellationToken cancellationToken
    = default)
{
    var tags
        = new[] { "Mobile", "Laptop" };

    var cacheKey
        = $"product-{productCategoryId}";

    return await _cache.
GetOrCreateAsync(
    cacheKey,
    async
        cancel => await db.Products
            .Where (
                p => p.Product_Category_Id
                == productCategoryId)
            .Include (
                p => p.Product_Category)
            .ToListAsync (
                cancellationToken),
            new HybridCacheEntryOptions {
                Expiration =
                    TimeSpan.FromMinutes(30),
                LocalCacheExpiration =
                    TimeSpan.FromMinutes(30)
            },
            tags: tags,
            cancellationToken:
            cancellationToken);
}
```

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

Performance Optimization of ZipArchive

There have been significant performance improvements in ZipArchive, reducing memory usage and making file operations faster. Incidentally the ZipArchive class is defined in the System.IO.Compression namespace. Applications that involve automatic backups, cloud storage, and massive datasets will find it particularly helpful. Unlike in the previous versions of .NET, all entries are not loaded into the memory when you're updating the files present in a ZipArchive. Moreover, when you extract the archive, multiple threads run in parallel to obtain the files present in the archive, improving performance.

```
using (FileStream fileStream
= new FileStream
(@"D:\test.zip",
 FileMode.Open))
{
    using (ZipArchive zipArchive =
new ZipArchive(fileStream,
ZipArchiveMode.Update))
    {
        ZipArchiveEntry zipArchiveEntry =
zipArchive.CreateEntry("Sample.txt");
        using (StreamWriter writer = new
StreamWriter(zipArchiveEntry.Open()))
        {
            writer.WriteLine
("This is a sample piece of text.");
        }
    }
}
```

OrderedDictionary Tweaks

With the advent of .NET 10, the OrderedDictionary class includes a few new features that make indexing far less cumbersome. For example, both TryAdd and TryGetValue can return an index, so you no longer need to loop through the entire collection to find where an item is located. Those little shortcuts streamline data handling in business applications, caching systems, and any data-heavy tool you may be developing.

With .NET 10, additional overloaded methods have been introduced for working with OrderedDictionary<TKey,TValue>, as shown in the code snippet below:

```
public class OrderedDictionary
< TKey, TValue >
{
    //Additional overloads
    public bool TryAdd(TKey key,
    TValue value, out int index);
    public bool TryGetValue(TKey key,
    out TValue value, out int index);
}
```

The following code snippet shows how you can use the OrderedDictionary class in C#:

```
OrderedDictionary<string, int>
orderedDictionary
= new OrderedDictionary<string,
int>();
string key = "P001";
```

```
if (!orderedDictionary.TryAdd(
key, 1215, out int index)) {
    int value
    = orderedDictionary.GetAt(index)
    .Value;
    orderedDictionary.SetAt(
index, value + 1);
}
```

Circular References in JSON Source Generators

In the previous versions of .NET, the System.Text.Json namespace didn't provide support for handling references when handling self-referencing or circular relationship objects. Hence, if you serialized a self-referencing object with a source-generated context, it resulted in exceptions being thrown by the runtime.

With .NET 10, you can specify a ReferenceHandler in your [JsonSourceGenerationOptions] attribute, allowing you to preserve or ignore reference loops, as shown in the next code example:

```
public static void CreateSelfReference()
{
    var referenceObj
    = new MyCustomSelfReferenceType();
    referenceObj.Self = referenceObj;

    string data
    = JsonSerializer.Serialize(
referenceObj,
ContextWithPreserveReference
.Default.SelfReference);
    Console.WriteLine(data);
}

[JsonSourceGenerationOptions(
    ReferenceHandler
    = JsonKnownReferenceHandler
    .Preserve)]
[JsonSerializable(
    typeof(SelfReference))]

internal partial class
ContextWithPreserveReference
: JsonSerializerContext { }

internal class
MyCustomSelfReferenceType {
    public MyCustomSelfReferenceType
    Self { get; set; }
    = null!;
}
```

With .NET 10, Microsoft has introduced support for PQC (Post-Quantum Cryptography), to secure applications against attacks in potential future quantum computers.

Enhancements to the .NET Runtime

With .NET 10, several performance enhancements have been introduced to the .NET Runtime. For example, Microsoft has made several improvements to JIT compilation, making it more efficient than earlier versions. For instance, better support for method inlining and loop unrolling optimizations can help frequently used code pathways execute significantly faster, which improves the performance of long-running apps.

In this section, I'll examine the key enhancements made to the .NET runtime in this release.

Optimization of Loops and Inlining

One of the key objectives of .NET 10 is enhancing performance and eliminating performance overhead when working with types such as arrays. Consider the following piece of code that illustrates a foreach loop used to iterate through a collection and cast data to an instance of type `IEnumerable<T>`:

```
int total = 0;

foreach (IEnumerable<int> n in array)
{
    total += n;
}
```

In the previous versions of .NET, the JIT compiler wasn't able to optimize such code by inlining. With .NET 10, the JIT can now remove the virtual method calls and inline your code and apply optimizations for your loop as appropriate. Additionally, .NET 10 is adept at optimizing how enumerators are created. For example, enumerator structs such as `ArrayEnumerator<T>` are now inlined, stack allocated to eliminate any heap allocation and deallocation overheads, and loop cloned for improved performance.

Improved Code Generation When Calling Methods with Instances of Structs

With .NET 10, the JIT compiler provides support for an optimization named physical promotion. This is a strategy that can place the members of a struct in registers, thereby eliminating the need for any memory access. This is a great optimization, especially when you need to pass an instance of a struct to a method and the calling method is required to pass the struct members in registers.

Consider the following struct named `Distance`:

```
[MethodImpl
(MethodImplOptions.NoInlining)]
static void CalculateDistance
(Distance distance)
{
```

Listing 2: New Clipboard APIs added in .NET 10

```
public static void SetDataAsJson<T>
(string format, T data)

public static bool TryGetData<T>
(string format, out T data)

public static bool TryGetData<T>
(string format, Func<TypeName, Type>
resolver, out T data)

public void SetDataAsJson<T>
(T data)

public void SetDataAsJson<T>
(string format, T data)

public void SetDataAsJson<T>
(string format,
bool autoConvert, T data)

public bool TryGetData<T>
(out T data);

public bool TryGetData<T>
(string format, out T data);

public bool TryGetData<T>
(string format,
bool autoConvert, out T data);

public bool TryGetData<T>
(string format,
Func<TypeName, Type> resolver,
bool autoConvert, out T data);

protected virtual
bool TryGetDataCore<T>
(string format,
Func<TypeName, Type> resolver,
bool autoConvert, out T data)

public bool TryGetData<T>
(out T data);
```



```
public bool TryGetData<T>
(string format, out T data);

public bool TryGetData<T>
(string format, out T data);

public bool TryGetData<T>
(string format,
Func<TypeName, Type>
resolver, out T data)
```

```

        Console.WriteLine
        ($"Feet: {distance.Feet.
        ToString()}, Inch:
        {distance.Inch.ToString()}");
    }

    struct Distance
    {
        public long Feet;
        public long Inch;
        public Distance(long feet,
        long inch)
        {
            if (inch > 12)
            {
                feet++;
                inch -= 12;
            }

            Feet = feet;
            Inch = inch;
        }
    }
}

```

The following code snippet shows how you can create an instance of the struct named Distance and invoke the CalculateDistance method:

```

using System.Runtime.CompilerServices;
Distance distance = new Distance(5, 15);
CalculateDistance(distance);

```

Now, assume that you've changed the members of the struct named Distance to integer instead of long. Because an integer occupies four bytes in the memory and a register requires eight bytes on a x64 system, you're required to pass the members of the struct in a single register. In the earlier versions of .NET, the JIT compiler wasn't able to represent values that share a register. Hence, the JIT compiler has to store the values of the instance of the struct in the memory and then load the entire eight-byte chunk into a register. With .NET 10, the JIT compiler is now capable of storing the promoted members of the struct arguments into shared registers.

Enhanced Loop Inversion

With .NET 10, the JIT compiler is now capable of transforming the condition of a while loop in C# into a do-while loop. Consider the following piece of code that shows the structure of a typical while loop:

```

while(condition)
{
    //body of the loop
}

```

With .NET 10, the JIT compiler can now translate this while loop to a do-while loop as shown in this code snippet:

```

if (condition)
{
    do
    {
        //body of the loop
    } while (condition);
}

```

This strategy of moving the condition of a loop to the bottom of the loop in C# is known as loop inversion. By shifting the condition of the loop in the preceding code example to the bottom of the loop in this way, the JIT compiler improves code layout because there's no need of branching to the top of the loop each time the loop is iterated.

Creating Container Images for Console Apps

In the earlier versions of .NET, if you were to publish a .NET console application as a container image using the dotnet publish command, you were constrained to specify the following in your project file, i.e., the .csproj file in your application:

```

<EnableSdkContainerSupport>true
</EnableSdkContainerSupport>

```

With .NET 10, you no longer need to specify this because you can publish a console application directly as a container image using the following command:

```
dotnet publish /t:PublishContainer
```

Controlling the Image Format of Containers

In the previous versions of .NET, when you published container images using the dotnet publish /t:PublishContainer command, the format of the generated image was automatically chosen based on the architecture and the base image. With .NET 10, you can control the image format explicitly using the newly added <ContainerImageFormat> property, as shown in the next code snippet:

```

<PropertyGroup>
    <ContainerImageFormat>OCI
    </ContainerImageFormat>
    <!-- <ContainerImageFormat>OCI
    </ContainerImageFormat> -->
</PropertyGroup>

```

You can force generating the OCI format by specifying the following code in the project file.

```

<Project Sdk="Microsoft.NET.Sdk">
    <PropertyGroup>
        <TargetFramework>net10.0
        </TargetFramework>
        <ContainerImageName>myapp
        </ContainerImageName>
        <ContainerImageFormat>OCI
        </ContainerImageFormat>
    </PropertyGroup>
</Project>

```

Enhanced Code Layout

With .NET 10, JIT compiles source code into basic blocks, which makes it faster at runtime. In earlier versions of .NET, the JIT leveraged a reverse post order (RPO) traversal approach. Although this approach had some degree of performance improvement, it had its downsides as well. With .NET 10, the JIT models the reordering problem as an asymmetric Travelling Salesman Problem to determine an optimal or almost optimal traversal, thereby improving hot path density and facilitating better runtime performance.

It should be noted that block reordering is done using a model based on a reduction to the asymmetric Travel-

ling Salesman Problem. With these techniques, .NET 10 optimizes hot path density while decreasing branch distances, thereby improving overall performance.

Efficient Search for PEM-Encoded Data

The previous version of .NET required the user to transform bytes into a string before being able to use the `PemEncoding.Find()` method. This was an unnecessary performance overhead. With .NET 10, this conversion is no longer necessary, and you can use the `PemEncoding.FindUtf8()` method to read bytes directly without requiring you to convert the byte data to a string anymore.

Let's understand this with a code example. Consider the following piece of code that shows how the old approach in the previous versions of .NET worked:

```
string path = "D:\\mycertificate.pem";
byte[] data = File.ReadAllBytes(path);
string text = Encoding.ASCII.GetString(data);
PemFields pemFields = PemEncoding.Find(text);
```

In the new approach, when working with .NET 10, you no longer need this conversion, as shown in the code snippet given next:

```
string path = "D:\\mycertificate.pem";
byte[] data = File.ReadAllBytes(path);
PemFields pemFields = PemEncoding.FindUtf8(data);
```

Optimized Just-In-Time (JIT) Compilation

The Just-In-Time or JIT compiler in .NET 10 is significantly faster compared to the previous versions of .NET. Several changes have been made in method inlining and loop unrolling, enabling frequently used code paths to execute faster. Additionally, there have been enhancements in tiered compilation, allowing for continuous optimization of frequently used code and improving performance for long-running applications in real-time.

A Just-in-Time (JIT) compiler translates compiled code into machine code at runtime, hence the name. Although the JIT compiler in .NET is a component of the Common Language Runtime (CLR) that converts the IL (earlier known as MSIL) code to machine code, it converts bytecode to machine code if you're working with Java.

Stack Allocation for Small Fixed-Sized Arrays

The CLR splits memory into two types: Stack and Heap. The former refers to memory area where value types are stored, and the latter contains objects of reference types. When you create objects of reference types, garbage collection (GC) needs to keep track of their lifetime

including allocation and deallocation of the memory in the managed heap. On the other hand, when objects are stored on the stack, the GC needs to track fewer objects, thereby minimizing the overhead.

In .NET 9, you were able to create objects on the stack, reducing GC overhead. This feature has been further enhanced in .NET 10 by facilitating the allocation of small, fixed-size arrays of value types that do not have any GC references in the stack. With .NET 10, small and fixed-sized arrays are allocated in stack memory, thereby eliminating garbage collection overhead, which is particularly helpful in applications that perform AI or ML calculations, as well as those that process data in real time.

Garbage Collection Enhancements

Garbage collection is a strategy used by the CLR to clean up managed objects, i.e., objects that are created in the managed heap. Support for GC is available as an in-built feature in .NET. With .NET 10, significant enhancements have been made to garbage collection and memory management.

Such improvements in .NET 10 have made GC more efficient, thereby reducing latency in real-time applications. Enhancements have also been made to background GC to prevent memory fragmentation, thereby improving application performance. Additionally, enhancements have been made to memory compaction to ensure that long-running applications are able to maintain stable memory footprints.

Security Enhancements and Updates in .NET 10

With cyber-attacks on the rise, .NET 10 prioritizes security by helping businesses secure their applications using zero trust architecture, introducing enhanced authentication methods, and implementing stricter encryption standards. This release adds several security changes, improvements, and updates that include resolving the security gaps, improving the coverage of the existing security features, and providing new capabilities especially relevant to developers.

Prior to .NET 10, there were certain limitations related to encryption and hashing algorithms for exported certificates that used 3DES as the legacy default algorithm. With .NET 10, you have the flexibility to define how certificates are encrypted during export. Essentially, it's now possible to dictate the encryption and hashing algorithms that are to be applied in exporting certificates. With this release, Microsoft has introduced several new APIs that allow specifying encryption and digest algorithms for PKCS#12/PFX export.

Better Certificate Handling

.NET 10 makes working with security certificates more seamless than ever before. The renewed `FindByThumbprint` method now works with several hashes, not just SHA-1, so you can build login flows in a secure, reliable way. This change boosts your overall security posture and aligns with the latest guidance on how certificates should really be verified.

The following code snippet illustrates how you can search using the SHA-256 instead of the default SHA-1 algorithm.

```
X509Certificate2Collection  
certificateCollection =  
store.Certificates.FindByThumbprint  
(HashAlgorithmName.SHA256, thumbprint);
```

```
builder.AddProject  
<MyDemoAspireProject>("frontend")  
.WithReference(redisCache)  
.WaitFor(redisCache);
```

Microsoft provides the Microsoft Windows HTTP Services (WinHTTP) Certificate Configuration Tool to set up client certificates in any certificate store. This utility, available as part of the file named WinHttpCertCfg.exe, makes it easier to ensure that certificates are available wherever the Internet Server Web Application Manager needs them.

New Features and Enhancements in .NET Aspire

Microsoft built .NET Aspire, a cloud-native toolkit for .NET developers primarily meant for enabling the building of high-performance and scalable applications with less hassle. The .NET Aspire development stack constitutes an assortment of templates, libraries, and tools that enable developers to build and deploy microservices and distributed systems seamlessly. One of the great new features in .NET 10 MAUI is the support for .NET Aspire integration. I'll discuss MAUI later in this article.

The key features of .NET Aspire include the following:

- Support for orchestration of applications at development-time
- Seamless support for integration
- Support for project templates and tooling for Visual Studio Code and Visual Studio

Typical use cases of .NET Aspire include the following:

- Building Microservices-based enterprise applications
- Building cloud-native back-ends for web or mobile applications
- Applications that leverage caching, messaging, service discovery, and telemetry services
- Applications that leverage Kubernetes, Azure Container Apps, or Docker-based environments

The following piece of code demonstrates how you can create a Redis container, wait for it to be available, and then configure it appropriately:

```
var builder =  
DistributedApplication.CreateBuilder(args);  
var redisCache =  
builder.AddRedis("redis_cache");
```

There are several enhancements to .NET Aspire such as user experience, optimization of local development, and customization of the dashboard.

Improved Cloud-Native Features

Cloud native refers to an approach to building and deploying software where software is developed and maintained within a cloud computing environment. This enables your application to leverage the benefits of cloud computing technology in terms of scale, flexibility, and resiliency.

With .NET 10 around and with the availability of .NET Aspire 9.1, you can now integrate your applications with the cloud in a much simpler way than before. If you're using containerized applications using containers and Kubernetes, the new features in .NET Aspire will help you build and deploy modern cloud applications seamlessly.

Enhanced Support for Logging and Telemetry

As the name implies, distributed tracing is a method used in distributed applications for troubleshooting and analysis. In a distributed system, you can leverage distributed tracing to capture application events and diagnose errors effectively. Remember, decreased performance issues can not only hinder a business's operation but also, over time, cripple any applications that are in use.

In .NET 10, structured logging enables developers to understand how the application's performance is functioning. Moreover, it will also help in identifying the actual problem behind these bottlenecks easily, including, but not limited to, security, performance, and scalability issues. The enhanced support for OpenTelemetry and Application Insights will enable you to troubleshoot problems much easier than ever before.

In a distributed system, capturing data proactively helps determine the current state of the system, enhances its operational visibility, and facilitates better issue resolution by monitoring the health of the system over time. Observability thrives on three key elements: metrics, logs, and traces. By observing applications in real-world production environments over time, you can identify bottlenecks (in performance, scalability, and security).

New Features and Enhancements in Windows Forms for .NET 10

In .NET 10, there have been several changes in Windows Forms such as clipboard serialization and deserialization, and introduction of new APIs to help you in handling JSON data. At a glance, the key features and enhancements in Windows Forms for .NET 10 include:

- Clipboard serialization and deserialization changes
- Obsolete clipboard methods
- New JSON-based clipboard APIs

Windows Forms (also known as WinForms) in .NET 10 comes up with several enhancements aimed at improving developer productivity, modernizing the design experience, and enhancing interoperability. WinForms and Windows Presentation Foundation (WPF) now share a common clipboard implementation, streamlining data exchange between the two frameworks. There have been several enhancements to improve compatibility with screen readers like NVDA.

Several new methods have been added to boost JSON-based serialization and phase out the obsolete BinaryFormatter. These advancements enhance security and modernize the Windows Forms applications and make it aligned with Microsoft's objectives of enhancing security and maintainability.

Create a New .NET 10 Windows Forms Project in Visual Studio 2022 Preview

Let's create a Windows application project now to work with the C# 14 code examples given in the sections that follow in this article. You can create a WPF 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 Windows Application Project in Visual Studio 2022:

1. Start the Visual Studio 2022 Preview IDE.
2. In the Create a new project window, select **Windows Forms Application**, and click Next to move on.
3. Specify the project name and the path where it should be created in the **Configure your new project** window.
4. If you want the solution file and project to be created in the same directory, optionally check the **Place solution and project in the same directory** checkbox. Click Next to move on.
5. In the Additional information screen, specify the target framework as .NET 10.0 (Preview), as shown in **Figure 1**.
6. Click Create to complete the process.

Clipboard Changes

With .NET 10, the clipboard has been redesigned so that you can share the code of the clipboard API with Windows Presentation Foundation (WPF) applications. As a result, both Windows Forms and WPF can now share the same code base, which will help unify how these two technologies interact with the clipboard. Another point to note here is that the BinaryFormatter class was made obsolete in .NET 9. Because this class was used in some clipboard operations, Microsoft has introduced additional methods to serialize clipboard data in a JSON format. These additional methods have been incorporated to help enable the .NET 10 transition away from BinaryFormatter seamlessly.

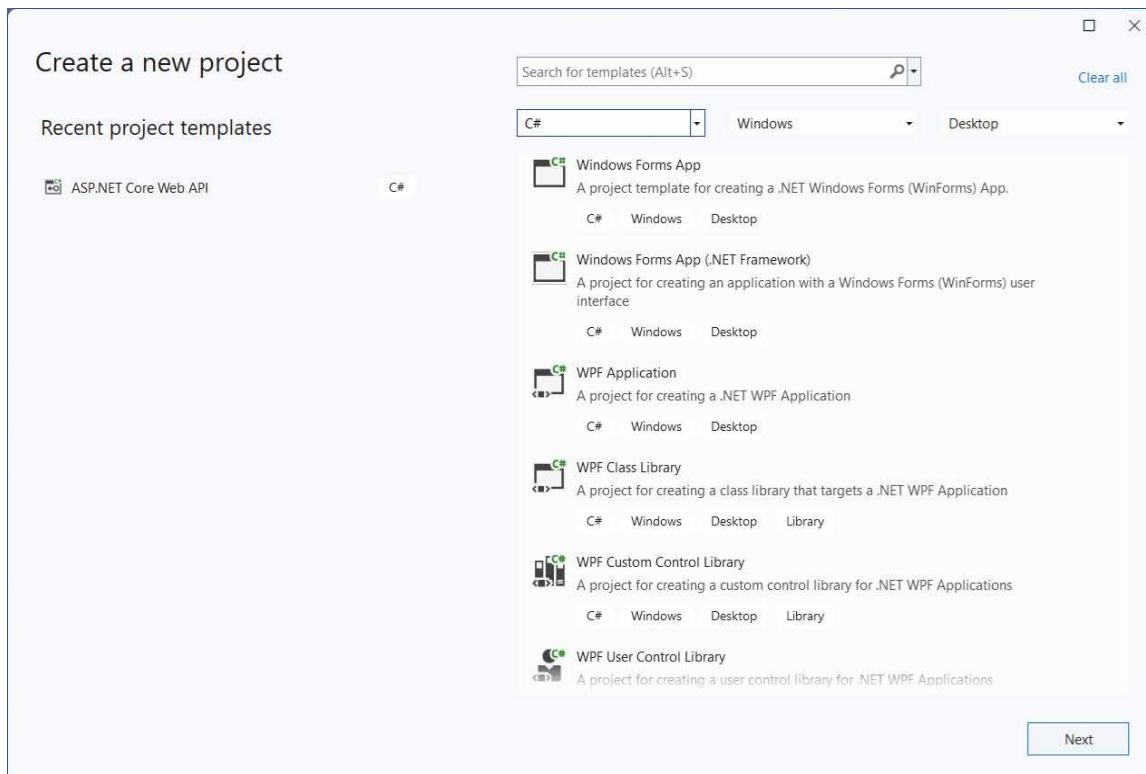


Figure 1: Create a new Windows Forms Application in Visual Studio.

With .NET 10, support for serialization has been enhanced considerably. For example, the `JsonSourceGenerationOptionsAttribute` has been introduced to customize how your application handles serialization of data in JSON format.

Clipboard APIs Marked as Obsolete

With .NET 10, several APIs have been marked as obsolete in Windows Forms such as the following:

```
OnClosing(CancelEventArgs)
System.Windows.Forms.Form.OnClosed(EventArgs)
System.Windows.Clipboard.GetData(String)
System.Windows.Forms.ContextMenu
System.Windows.Forms.DataGrid
System.Windows.Forms.MainMenu
System.Windows.Forms.Menu
System.Windows.Forms.StatusBar
System.Windows.Forms.ToolBar
```

Say goodbye to clipboard:

```
public static object? GetData(string format)
```

DataObject is also gone:

```
public virtual object? GetData
(string format)

public virtual object? GetData
(string format, bool autoConvert)

public virtual object? GetData
(Type format)
```

The Clipboard class given below is available as a part of both the `System.Windows.Forms` and `System.Windows` namespaces, thereby enabling you to use it in both Windows Forms as well as WPF applications.

```
public static partial class Clipboard
{
    public static void SetDataAsJson<T>
(string format, T data) { }

    public static object? GetData
(string format) { }

    public static bool TryGetData<T>
(string format, out T data) { }

    public static bool TryGetData<T>
(string format,
Func<Reflection.Metadata.TypeName, Type>
resolver, out T data) { }

    //Other members of the Clipboard class
}
```

Listing 3 shows the complete source code of the Clipboard class and its dependent types. Microsoft has introduced several new Clipboard related APIs in Windows Forms for .NET 10, as shown in **Listing 4**.

Custom Designer Improvements

Classic `UITypeEditors`, such as the `ToolStripCollectionEditor` and several `DataGridView` editors, have been migrated from legacy .NET Framework. You can now see them in both the `PropertyGrid` and the actions panel of the Windows Forms Designer.

Performing Clipboard Operations Prior to the New Clipboard API

In this section, I'll examine how you could work with the Clipboard before the advent of the new Clipboard API.

Consider the following class named `Product`.

```
[Serializable]
public class Product
{
    public Guid Product_Id { get; set; }
    public string Product_Name { get; set; }
    public string Description { get; set; }
    public decimal Price { get; set; }
    public int Quantity { get; set; }
}
```

The following code snippet shows how you can call the `SetData` static method of the `Clipboard` class to store `Product` data in the specified format:

```
public void SetClipboardData()
{
    Product productModel = new()
    {
        Product_Id = Guid.NewGuid(),
        Product_Name = "HP ZBOOK",
        Description = "i9 Laptop",
        Price = 2500.00M,
        Quantity = 100
    };

    Clipboard.SetData("customFormat",
productModel);
}
```

The following code snippet shows how you can retrieve data in the specified format from the Clipboard by invoking the `GetData` static method of the `Clipboard` class:

```
public static Product? GetClipboardData()
{
    if (Clipboard.GetData("customFormat") is
Product productDTO)
    {
        //Write your code here.
        return productDTO;
    }

    return null;
}
```

Listing 5 contains the complete source code for this example.

Performing Clipboard Operations Using the New Clipboard API

In .NET 10, you can use the SetDataAsJson static method of the Clipboard class, as shown in the next code snippet:

```
Product productModel = new()
{
    Product_Id = Guid.NewGuid(),
```

```
    Product_Name = "HP ZBOOK",
    Description = "i9 Laptop",
    Price = 2500.00M,
    Quantity = 100
};

Clipboard.SetDataAsJson
("customFormat", productModel);
```

Listing 3: The Clipboard class and its dependent types

```
public static partial class Clipboard {
    public static void SetDataAsJson<T>(
        string format, T data)
    {
    }
    public static object? GetData(string format) { }
    public static bool TryGetData<T>(
        string format, out T data)
    {
    }
    public static bool TryGetData<T>(string format,
        Func<Reflection.Metadata.TypeName, Type> resolver,
        out T data)
    {
    }
}

public partial class DataObject
: IDataObject,
  Runtime.InteropServices.ComTypes
  .IDataObject ITypedDataObject {
    public void SetDataAsJson<T>(T data) { }
    public void SetDataAsJson<T>(string format, T data) { }
    public void SetDataAsJson<T>(
        string format, bool autoConvert, T data)
    {
    }
    public virtual object? GetData(
        string format, bool autoConvert)
    {
    }
    public virtual object? GetData(string format) { }
    public virtual object? GetData(Type format) { }
    public bool TryGetData<T>(out T data) { }
    public bool TryGetData<T>(string format, out T data) { }
    public bool TryGetData<T>(string format,
        bool autoConvert, out T data)
    {
    }
}

public interface ITypedDataObject {
    bool TryGetData<T>(out T data);
    bool TryGetData<T>(string format, out T data);
    bool TryGetData<T>(
        string format, bool autoConvert, out T data);
    bool TryGetData<T>(string format,
        Func<Reflection.Metadata.TypeName, Type> resolver,
        bool autoConvert, out T data);
}

public sealed class DataObjectExtensions {
    public static bool TryGetData<T>(
        this IDataObject dataObject, out T data);

    public static bool TryGetData<T>(
        this IDataObject dataObject, string format,
        out T data);
    public static bool TryGetData<T>(
        this IDataObject dataObject, string format,
        bool autoConvert, out T data);
    public static bool TryGetData<T>(
        this IDataObject dataObject, string format,
        Func<Reflection.Metadata.TypeName, Type> resolver,
        bool autoConvert, out T data);
}
```

Listing 4: The ClipboardManager and the Product Model Classes

```
[Serializable]
public class Product
{
    public Guid Product_Id { get; set; }

    public string Product_Name { get; set; }

    public string Description { get; set; }

    public decimal Price { get; set; }

    public int Quantity { get; set; }
}

public class ClipboardManager
{
    public void SetClipboardData()
    {
        Product productModel = new()
        {
            Product_Id = Guid.NewGuid(),
            Product_Name = "HP ZBOOK",
            Description = "i9 Laptop",
            Price = 2500.00M,
            Quantity = 100
        };

        Clipboard.SetData("customFormat", productModel);
    }

    public static Product? GetClipboardData()
    {
        if (Clipboard.GetData
            ("customFormat") is Product productDTO)
        {
            //Write your code here.
            return productDTO;
        }

        return null;
    }
}
```

Listing 5: Performing Clipboard operations prior to the new Clipboard API

```
[Serializable]
public class Product
{
    public Guid Product_Id { get; set; }
    public string Product_Name { get; set; }
    public string Description { get; set; }
    public decimal Price { get; set; }
    public int Quantity { get; set; }
}

public class ClipboardManager
{
    public void SetClipboardData()
    {
        Product productModel = new()
        {
            Product_Id = Guid.NewGuid(),
            Product_Name = "HP ZBOOK",
            Description = "i9 Laptop",
            Price = 2500.00M,
            Quantity = 100
        };

        Clipboard.SetDataAsJson("customFormat", productModel);
    }

    public static Product? GetClipboardData()
    {
        if (Clipboard.TryGetData("customFormat",
            out Product? productDTO))
        {
            //Write your code here.
            return productDTO;
        }

        return null;
    }
}
```

To retrieve the Product instance from the JSON serialized data, you can use the following code:

```
if (Clipboard.TryGetData
("customFormat",
out Product? productDTO))
{
    //Write your code here.
    return productDTO;
}
```

You don't need to use a BinaryFormatter even when implementing drag/drop functionality in the Clipboard, as illustrated in **Listing 6**.

Listing 6: Performing Clipboard operations using the new Clipboard API

```
private Form formObj = new();
private Control dragControlObj = new();

dragControlObj.MouseDown += beingDrag_MouseDown;
formObj.DragDrop += Form1_DragDrop;

void beingDrag_MouseDown
(object sender, MouseEventArgs e)
{
    Supplier supplier = new Supplier
    {
        Supplier_Id = Guid.NewGuid()
    };

    dragControlObj.DoDragDropAsJson
    (weatherForecast, DragDropEffects.Copy);
}

void DragDropExampleForm_DragDrop
(object sender, DragEventArgs e)
{
    DataObject dataObject = e.Data;
    if (dataObject.TryGetData
        (typeof(Supplier), out Supplier? supplier))
    {
        //Write your code here to work
        //with the deserialized data.
    }
}
```

New Features and Enhancements in Windows Presentation Foundation (WPF) for .NET 10

Windows Presentation Foundation, (WPF) is a flexible user interface framework developed by Microsoft and included as part of .NET Framework 3.0 initially. In WPF 10, several enhancements and bug fixes are primarily focused on improving performance and fluent style support. In this release, fluent styles have been added for controls such as Label, Hyperlink, GroupBox, GridSplitter, and NavigationWindow.

Performance Enhancements

In .NET 10, performance was improved considerably by substituting base.NET types like `ReadOnlyCollection<T>` for internal data structures. Application performance was also enhanced by input composition, regex use, XAML parsing, optimizing array handling, font rendering, dynamic resources, trace logging, and migration of the font collection loader to managed code.

Performance enhancements have been achieved by optimizing cache operations, moving the font collection loader to managed code, and improving array handling. Besides this, several bug fixes and changes have been made that can enhance the framework's functionality and developer experience.

There are several enhancements to code quality and performance in this WPF release. These include the following:

- Memory allocations have now been optimized to use more efficient methods, such as generic collections and arrays, replacing boxed collections like `ArrayList`, `Hashtable`, and `IList`.
- This release discards the unused fields in the `DpiHelper` and `XamlSchema` utilities to optimize internal logic.
- There's an appreciable runtime performance enhancement because the static constructor from `KnownColors` is removed.
- `ReflectionHelper` for WinRT and other unused or legacy code has been discarded.

Clipboard Changes

With .NET 10, WPF and Windows Forms share the same clipboard API. The two desktop technologies have converged in how they communicate with the clipboard. Moreover, since .NET 9, Microsoft has obsoleted the BinaryFormatter class, which supports certain clipboard operations. These operations with a clipboard meant that you needed to subscribe to a compatibility package or a workaround for the operation.

To ensure that the phasing out of the BinaryFormatter class is seamless, Microsoft has marked several clipboard methods to indicate that you should no longer use those methods. Instead, new methods have been introduced to assist in JSON data serialization using the clipboard, moving away from using the BinaryFormatter class.

The Need for a Unified Clipboard API

In the earlier versions of .NET, both WPF and Windows Forms used different code bases for working with the clipboard. The System.Windows.Clipboard contained the source code for working with clipboard in WPF applications, and the Windows Forms applications used the used System.Windows.Clipboard namespace for the same.

There are several benefits of the unified Clipboard API used by both WPF and Windows Forms in .NET 10:

- Consistency:** One of the key benefits of unifying the Clipboard API is that both the WPF and Windows Forms leverage the same source code for performing clipboard operations such as Copy, Paste, Cut, etc., thereby ensuring consistency.
- Enhanced Interoperability:** The unification of the Clipboard API enables you to build hybrid applications that can take advantage of both these technologies or share components between them.
- Code simplification:** Because the API is more consistent, it lessens the need for creating separate code paths based on whether you're using WPF or Windows Forms.
- Easier updates and enhancements:** Because the API is unified, introducing updates or enhancements that can be leveraged by both WPF and Windows Forms will now be much easier.

New Clipboard APIs Introduced

In .NET 10, the MessageBoxButton and MessageBoxResult enumerations have been enhanced, as shown in the next code snippet:

```
namespace System.Windows
{
    public enum MessageBoxButton
    {
        OK = 0,
        OKCancel = 1,
        AbortRetryIgnore = 2,
        YesNoCancel = 3,
        YesNo = 4,
        RetryCancel = 5,
        CancelTryContinue = 6,
    }

    public enum MessageBoxResult
    {
        None = 0,
        OK = 1,
        Cancel = 2,
        Abort = 3,
        Retry = 4,
        Ignore = 5,
        Yes = 6,
        No = 7,
        TryAgain = 10,
        Continue = 11,
    }
}
```

```
None = 0,
OK = 1,
Cancel = 2,
Abort = 3,
Retry = 4,
Ignore = 5,
Yes = 6,
No = 7,
TryAgain = 10,
Continue = 11,
}
```

Fluent UI Style Changes

Microsoft has resolved several bugs in WPF for .NET 10, thereby enhancing the Fluent UI style support in WPF. In this release of WPF, Fluent styles have introduced additional controls, such as the following:

- DatePicker
- GroupBox
- GridSplitter
- Hyperlink
- Label
- NavigationWindow
- RichTextBox
- TextBox

Create a New .NET 10 WPF Project in Visual Studio 2022 Preview

Let's create a WPF application project now to work with the C# 14 code examples in the sections that follow. You can create a WPF 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 WPF Application Project in Visual Studio 2022:

1. Start the Visual Studio 2022 Preview IDE.
2. In the Create a new project window, select **WPF Application**, and click Next to move on.
3. Specify the project name and the path where it should be created in the **Configure your new project** window.
4. If you want the solution file and project to be created in the same directory, optionally check the **Place solution and project in the same directory** checkbox. Click Next to move on.
5. In the Additional information screen, specify the target framework as .NET 10.0 (Preview), as shown in **Figure 2**.
6. Click Create to complete the process.

In the sections that follow, you'll use this project to write and run the WPF code listings and snippets.

New Features and Enhancements in MAUI in .NET 10

Microsoft's .NET Multi-platform App UI is a cross-platform framework that lets developers write native apps that can run on Android, iOS, Windows, and macOS using C# and XAML code. With .NET 10, the MAUI team has focused on optimizing the framework and improving the developer

Memory Fragmentation and Compaction

Memory fragmentation and compaction are key concepts in OS memory management. In the former case, available memory segments are scattered into non-contiguous blocks of memory due to loading and unloading processes over time, making it challenging to allocate large contiguous blocks even though enough total free memory exists. In the latter case, the non-contiguous blocks of memory are combined to form a large contiguous memory block.

experience and performance of applications built using .NET MAUI across Android, iOS, and macOS.

Install .NET MAUI

You can install .NET MAUI using the Visual Studio Installer by selecting the **.NET MAUI** workload under the **Mobile development with .NET** workload in the Visual Studio Installer. Alternatively, you can install .NET MAUI using the following command at the Windows Shell:

```
dotnet workload install maui --version  
<<Specify the desired version here.>>
```

Replace the text inside the << brackets with the desired version of .NET MAUI. As of this writing, .NET 10 Preview 5 is available. You can install .NET MAUI by specifying the desired version using the following command:

```
dotnet workload install maui  
--version 10.0.100-preview.4.25263.1
```

Installing MAUI Templates

If you're unable to see the MAUI templates in your Visual Studio IDE, you may have to install them manually. To install these templates, follow the steps outlined below.

In the first step, press the Windows and the R keys together to invoke the Run command window.

To install the MAUI workload along with its essential components and dependencies, type the following command into the Run command window and then hit the Enter key on your keyboard:

```
dotnet workload  
install maui
```

If the MAUI workload is installed successfully on your computer, the next and final step is to execute the following command in the Run command window to install MAUI templates:

```
dotnet new install  
Microsoft.Maui.Templates
```

That's all you need to do. If the installation is complete and successful, you should see the MAUI templates in the Visual Studio IDE when creating a new MAUI project.

Create a New .NET 10 MAUI Project in Visual Studio 2022 Preview

Let's create a WPF application project now to work with the C# 14 code examples in the sections that follow. You can create a WPF 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 WPF Application Project in Visual Studio 2022:

1. Start the Visual Studio 2022 Preview IDE.
2. In the Create a new project window, select **.NET MAUI App**, and click Next to move on.
3. Specify the project name and the path where it should be created in the **Configure your new project** window.
4. If you want the solution file and project to be created in the same directory, optionally check the **Place solution and project in the same directory** checkbox. Click Next to move on.
5. In the Additional information screen, specify the target framework as .NET 10.0 (Preview), as shown in **Figure 3**.
6. Click Create to complete the process.

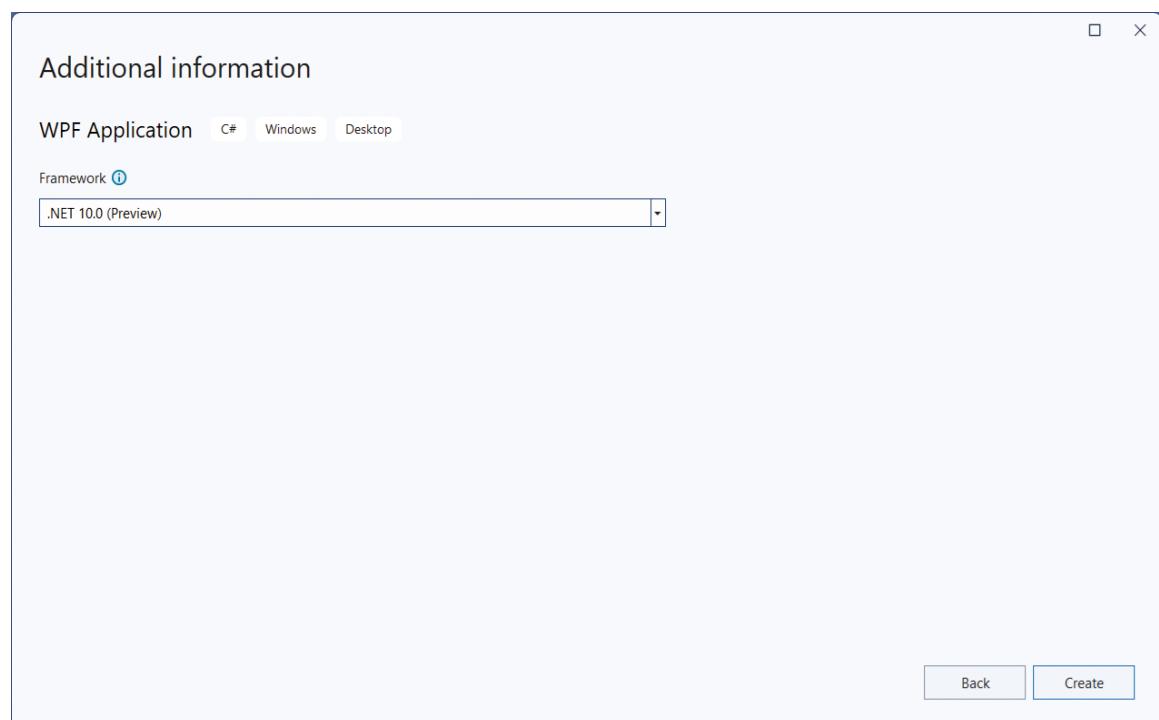


Figure 2: Specify the .NET Framework version you'll use in the Additional Information screen.

APPS WITHOUT COPILOTS ARE NOW LEGACY!



Microsoft has introduced Copilot-driven applications and announced that all Microsoft applications will follow this approach going forward. This aims to capitalize on the extreme productivity gains this new paradigm promises. The same applies to your own applications and systems.

We can help you build Copilots into your own applications. Tailored to what you do.

**VISIT OUR COPILOT PAGE TO FIND OUT MORE ABOUT OUR SERVICES
AND TO SEE DEMONSTRATIONS AND EXAMPLES!**

codemag.com/copilot

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

Stability Improvements

.NET 10 brings a friendly boost to your MAUI applications by reducing the memory footprint, enabling the user interface screens to load much faster, and enabling animations to glide along without the occasional stutter that used to appear in previous versions. The background tasks run much quicker now, allowing long-running processes to run smoothly and keeping the UI responsive.

Enhanced Device APIs

With .NET 10 around, you can explore native features like fingerprint log in, stronger NFC, and smoother and more polished background services. Thanks to enhanced sensor compatibility, cross-platform apps are much more responsive across all platforms, such as Android, iOS, Windows, and macOS.

Android Enhancements in .NET 10

With .NET 10, Android now provides support for Android API 36 and JDK 21. If you want to target the .NET 10 framework, you need to set your project's target to net10.0-android36. Additionally, the default supported Android API is now Android 24 (Nougat), updated from Android 21 (Lollipop).

In the previous versions, you weren't able to use the dotnet run command in your Android for .NET projects. With .NET 10, you can execute your Android projects using the dotnet run command. Moreover, you can now specify which device you'd like your Android application to use, as shown in the code snippet below.

```
dotnet run -p:AbdTarget="-s emulator-5554"
```

In .NET 9, a new marshal method mechanism was introduced. Although it was turned off by default in .NET 9,

in .NET 10, this feature is enabled by default. You can enable/disable this as shown in the code snippets given below.

To enable:

```
<PropertyGroup>
<AndroidEnableMarshalMethods>true
</AndroidEnableMarshalMethods>
</PropertyGroup>
```

To disable:

```
<PropertyGroup>
<AndroidEnableMarshalMethods>false
</AndroidEnableMarshalMethods>
</PropertyGroup>
```

CollectionView Enhancements

Although the two new handlers named CollectionView and CarouselView on iOS and Mac Catalyst for enhancing stability and performance were optional in .NET 9, they are now available by default in .NET 10. However, you can also choose to opt out of these handlers if you want to, using the following piece of code in your MauiProgram class:

```
#if IOS || MACCATALYST
builder.ConfigureMauiHandlers(
    handlers =>
{
    handlers.AddHandler
        <Microsoft.Maui.Controls.CollectionView,
        Microsoft.Maui.Controls.Handlers.
        Items.CollectionViewHandler>();
```

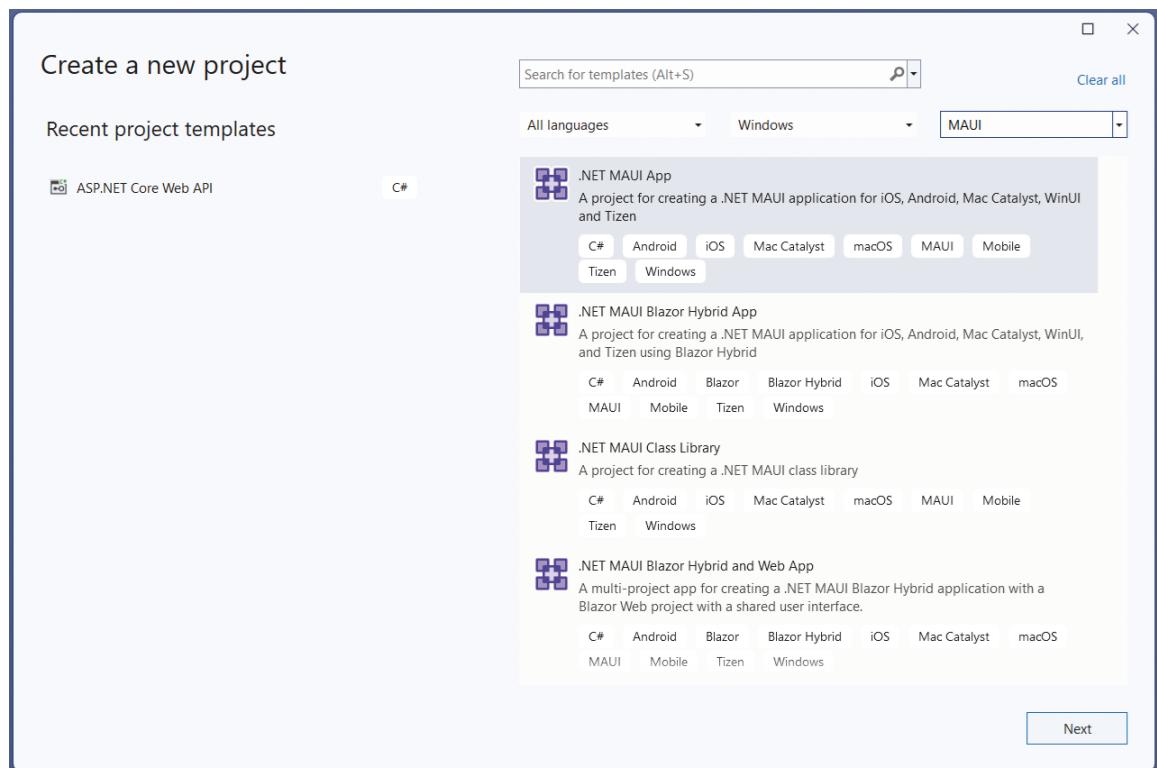


Figure 3: Create a new .NET MAUI Project in Visual Studio

```

handlers.AddHandler
<Microsoft.Maui.Controls.CarouselView,
Microsoft.Maui.Controls.Handlers.
Items.CarouselViewHandler>();
};

#endif

```

Cleaner XML-namespace Experience

With .NET 10, you can now have a much cleaner XML-namespace experience for your .NET MAUI applications by removing most of the boiler-plate xmlns: statements you'd otherwise have to specify at the top of each XAML file.

Here's how a typical XAML looked in the previous versions of .NET:

```

<!-- .NET 8 style -->
<ContentPage
    xmlns="http://schemas.microsoft.com/
dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/
winfx/2009/xaml"
    xmlns:models="clr-namespace:
    MyApp.Models"
    xmlns:controls="clr-namespace:
    MyApp.Controls"
    x:Class="MyApp.MainPage">
    <controls:TagView x:DataType=
    "models:Tag" />
</ContentPage>

```

In .NET 10, you can write the same in a much cleaner way, like this:

```

<!-- .NET 10 style -->
<ContentPage
    xmlns="http://schemas.microsoft.com/
dotnet/maui/global"
    xmlns:x="http://schemas.microsoft.com/
winfx/2009/xaml"
    x:Class="MyApp.MainPage">
    <TagView x:DataType="Tag" />
</ContentPage>

```

In .NET 10, you won't have to declare the xmlns:models or xmlns:controls statements because they're declared globally in a file named GlobalXmns.cs.

To adapt to this change, you should follow these steps:

1. Upgrade your existing project to .NET 10 by setting the target framework as .NET 10.0.
2. Add the assembly-level file named GlobalXmns.cs that map. to your CLR namespaces.
3. Remove the redundant xmlns: lines and prefixes from your XAML code to make your application's code much cleaner.

After this change, a typical GlobalXmns.cs file contains the following lines of code:

```

[assembly: XmlnsDefinition(
    "http://schemas.microsoft.com/
dotnet/maui/global",

```

```

    "MyApp.Views")]
[assembly: XmlnsDefinition(
    "http://schemas.microsoft.com/
dotnet/maui/global",
    "MyApp.Controls")]
[assembly: XmlnsDefinition(
    "http://schemas.microsoft.com/
dotnet/maui/global",
    "MyApp.Converters")]

```

.NET Aspire Integration

.NET MAUI for .NET 10 ships with a fresh project template that spins up a .NET Aspire service defaults project for .NET 10 MAUI. Inside this starter project, you'll find a tidy set of extension methods that can wire telemetry and service discovery into your application. To leverage those capabilities, just tweak the CreateMauiApp method in MauiProgram class to invoke the AddServiceDefaults method from the .NET Aspire service defaults project, on the MauiAppBuilder object, as shown here:

```

builder.AddServiceDefaults();

```

The AddServiceDefaults method performs the following tasks:

- Sets up OpenTelemetry metrics and tracing
- Adds service discovery functionality to your application
- Tweaks HttpClient so that it can work with service discovery

Wrapping Up

The most recent release version of .NET (.NET 10) provides a significant boost in developer productivity over its predecessors because it provides a variety of new capabilities, substantial performance, scalability, and extensibility improvements, and enhanced security. There are several tools, features and improvements that Microsoft has included in this release, making it more polished and stable as ever, and again strengthening its position as a stable cross-platform framework in .NET ecosystem.

Regardless of whether you're attempting to build enterprise, mobile, or modern web applications, this release has all the ingredients you require when building world-class modern-day applications. Because .NET 10 is a Long-Term Support (LTS) release, you can use it in production because it will receive steady support while still offering a stack that keeps moving forward. So, take a spin with this release to not only accelerate your development processes and software releases but also ensure that your application's source code is lean, clean, and maintainable.

Joydip Kanjilal


SPONSORED SIDEBAR

CODE Is Hiring!

CODE Staffing is accepting resumes for various open positions ranging from junior to senior roles.

We have **multiple openings** and will consider candidates who seek full-time employment or contracting opportunities.

For more information: www.codestaffing.com.

Veo 3: The Clockwork Hand

In the summer of 2022, I used Midjourney to generate my first artificial intelligence image. It was a clockwork hand. I did, without irony, choose the very thing that Generative AI was terrible at. It returned some images that vaguely resembled hands. These weren't just any hands, of course. They were abominations in the eyes of God. It was accidental Salvatore Dali,



Jason Murphy

[@jason-murphy.bsky.social](https://thestrangerous.substack.com)

Jason is a writer, AI enthusiast, and media creator. As the producer and co-host of **Hacking the System** on the National Geographic Channel, he stole cars in Hollywood, made improvised smoke bombs, and prepared for the apocalypse. On YouTube, he co-created and hosted **the Modern Rogue**, where he explored hacking, lock-picking, and trade-craft. After publishing multiple speculative fiction novels and writing a produced screenplay, Jason is currently exiled in the desert, where he stares into the future with awe and terror. And he still wants to make a video game.



a warped thing with wobbly appendages. Vague suggestions of numberless clock faces emerged from this generation. Despite the crude response, I was fascinated and immediately set to spending way too much money on hundreds of pictures that were completely unusable for any purpose other than to say, "Hey, isn't this weird and cool?" Take a look at the monstrosity in **Figure 1**.

This was just with Midjourney 2.0. Other iterations followed, and they came quickly. Through the updates, the hand started to take shape. It took a few years to get the appropriate number of fingers worked out, but all along, I watched the technology change in real time. With every upgrade, my clockwork hand looked more and more like I imagined. I'm a little slow on the uptake sometimes, but around Midjourney 3.0, the thing that had been nagging at me since the beginning finally revealed itself. If you could make an image of something, you could get Generative AI to make lots of those images in a row, and if you wanted to make lots of images in a row, you could pack them in tightly—24 images every second, maybe.

The Myth of Filmmaking

Growing up, filmmaking was some mystical art. It was an occulted thing, a special club for people with exciting lives, like astronauts and karate champions. Only rich people and wizards got to make movies. Then film gave way to digital. We started to have affordable, non-linear editing systems in our homes and the "prosumer" camera market exploded. In 2001, Danny Boyle directed the zombie flick **28 Days Later** on one such camera, the Canon XL1. I bought my own right after that for around \$1500. As you can see in **Figure 2**, it was portable enough to carry in one hand and it used tape storage instead of film.



Figure 1: My first attempt is freakish.

Before long, tapes were swapped out for digital storage. The plunging cost per compute put complex FX systems into the hands of hobbyists. Blender, an open-source 3D creation suite you can see in **Figure 3**, supports everything from modeling and animation to compositing and video editing. And it's free. The idea of filmmaking as an exclusive club started to erode. In 2015, filmmaker Sean Baker shot the award-winning **Tangerine** entirely on an iPhone 5S. Just a few years later, Steven Soderbergh used an iPhone 7 Plus to make the thriller **Unsane**. Movies are shot on phones all the time now, with built-in cameras that outperform the XL1. Filmmaking has been democratized, in many ways. Aspiring filmmakers have near limitless tools available.

A New Toy in the Toybox

2023 saw the mainstream adoption of primitive video generation. Diffusion models were improving rapidly, and open-source frameworks proliferated. Earlier that year, Runway ML introduced video-to-video transformations with Gen-1. Gen-1 was one of the first tools that let neophytes with no expertise in coding or filmmaking generate video content, all packaged in a user-friendly tool. Then the dam broke. Developers like Kaiber and Pika Labs introduced advanced workflows to the ecosystem. Just a year after Gen-1, Runway introduced Gen-2, providing actual text-to-video capabilities. After much anticipation, Sora (by OpenAI) hit, only to be quickly eclipsed by Google's Veo 3 model.

Veo 3 is Google DeepMind's latest and most sophisticated video model. Although it's capable of impressive text-to-video, Google also implemented a suite of other perks, giving users more control. With these new features, users can generate native audio, achieve greater consistency



Figure 2: The Canon XL1 was groundbreaking.

from shot-to-shot, and use prompting for shot composition. Since launch, Google has integrated Veo 3 into their own AI filmmaking suite Flow, as well as Gemini, their primary LLM.

How Does It Work?

To generate small clips of video, Veo 3 accepts multi-modal inputs from text prompts, images, or frames of videos. It's built using a diffusion model, which functions by generating data such as images, audio, or text through a two-step process. In the **forward process**, the model progressively adds noise to the image. It does this over and over until the data becomes pure noise. This intentional muddying of the information instructs the model on how a clean piece of data can transition into noisy data. At this point, you enter the **reverse process**, where the model learns, step by step, to strip the noise from the noisy data. It does this in iterations until it rebuilds the original data or, in this case, generates something entirely new from the noise. Through ex-

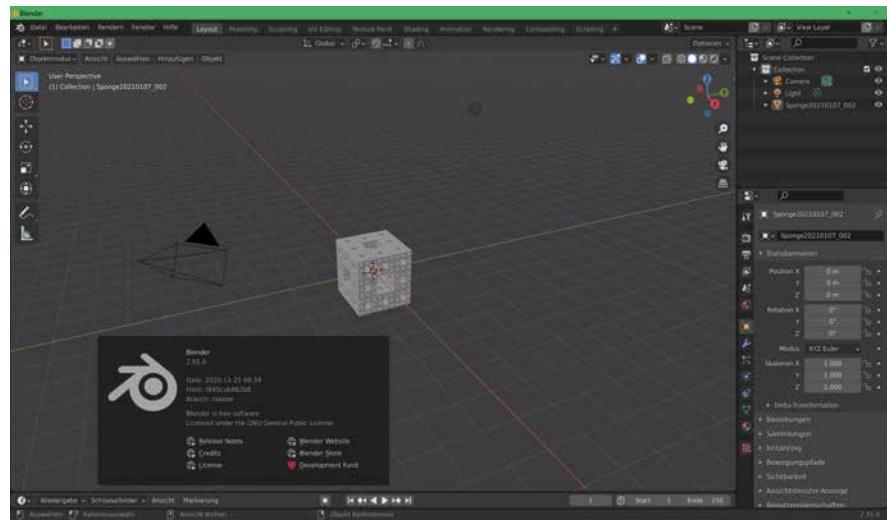


Figure 3: Blender 2.91

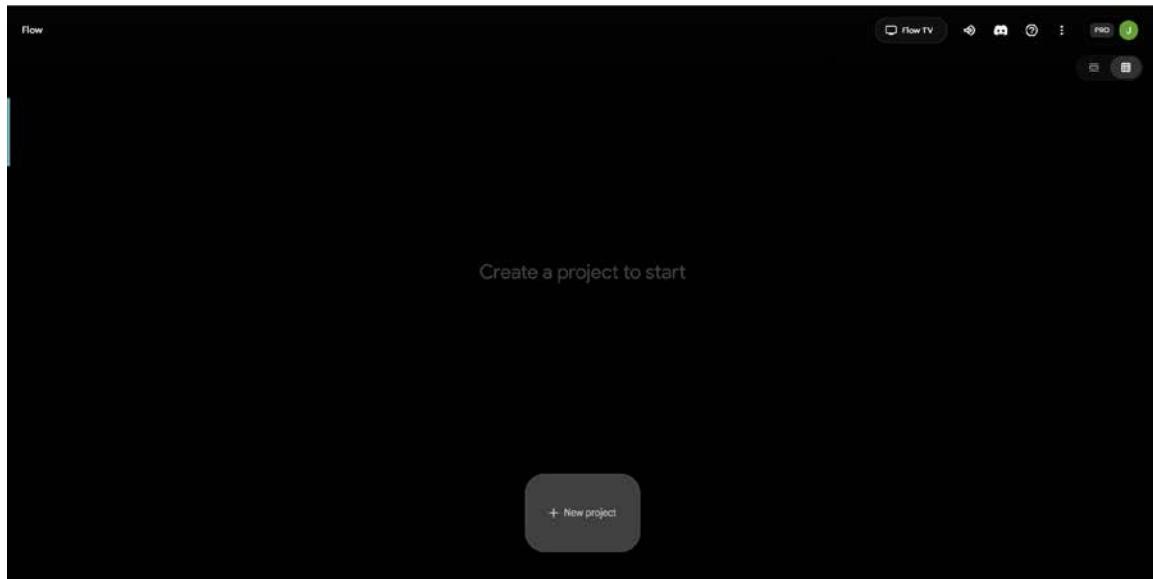


Figure 4: At first glance, Veo 3 is bare bones.

tensive and repetitive training, the model learns about the underlying distribution of the data. This enables it to generate new images **tabula rasa** by reversing the noise process.

Hands On

Access to the tool is available with a subscription to Google AI Pro. If you're hoping to just get the tool a la carte, you may be disappointed. The bundle includes an entire host of other options you can take or leave, like Gemini 2.5 Pro, supplemental storage, Notebook LM, and more. The basic tier is \$19.99 per month, which is market competitive. There's a pricier tier with higher monthly generation limits, if you're ready to drop \$250 per month. With the standard package, you get 1000 AI Credits. The per-generation cost can be anywhere from 10 credits to 100, depending on speed, quality, and model (Veo 2 vs. Veo 3). For more control, I access Veo 3 through their simplified editing system, Flow.

As you see in **Figure 4**, the interface is stripped down and simple.

Once a project is created, the rest of the interface is spare and clean. This isn't a full non-linear editing system like Final Cut or Adobe Premiere. You'll find more complex options on TikTok or Instagram. The primary interface at the bottom of the screen puts all the focus on the prompt, shown in **Figure 5**. It's your standard text prompt, little

Prompt Engineering

Prompt engineering is about shaping a very specific request in natural language to steer an AI model to the required output. A prompt engineer attempts to communicate the tone, context, and specific details to get as close to the intended result as possible. A well-engineered prompt ensures that the user's intent is clearly conveyed to the Large Language Model in a structured manner.

Prompt engineering is already being employed in literally every industry that uses generative AI. As these systems are integrated into various fields, prompt engineers will be responsible for customizing and tweaking inputs ensuring that these systems offer relevant responses.

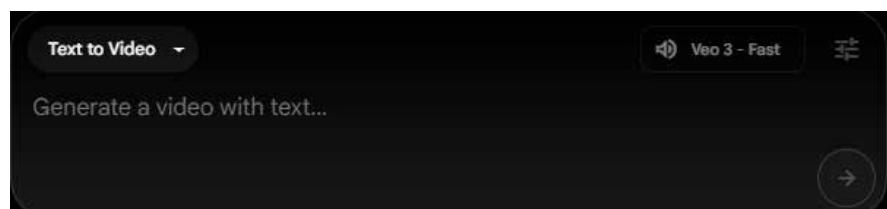


Figure 5: The Veo 3 prompt window is focused on simplicity.

different from the design used by ChatGPT for Sora or Runway ML's interface.

The prompt window provides a few simple options. The Settings section in the upper right allows the user to choose the number outputs per prompt, from one to four instances, as well as the model choice. Veo 2 costs fewer tokens but is less advanced and offers no native sound. Veo 3 has a Fast and a Quality setting. To give you an idea of how you're spending your credits, the Fast setting costs 20 AI credits while the Quality setting costs a hefty 100 credits. If you're not ready to drop a considerable amount of cash, the 1000-credit limit to this plan is quite a barrier.

Users can dive right in with a simple Text to Video prompt, but for more granular control, the Frames to Video option accepts two keyframes. A starting frame and an ending frame can be uploaded (.png, .jpg, etc.) or generated via AI using the tool. Additionally, this function comes with a buffet of camera directions made available as simple icons, seen in **Figure 6**.

Prompting in Veo 3 is where you get to try to flex your directing muscles. For the best results, prompt it like you're

sitting in the director's chair. Specificity is key. Name your subject, what they're doing, and where it's happening, along with how it feels, the style of the piece. The model responds well to the use of cinematic language. Craft your prompt with everything like camera movement, lighting, and style in mind, providing references when possible (Stanley Kubrick, 70s telefilm, found footage). Intentional and detailed writing skills will go much farther than a generic suggestion. Stay away from vague prompts, don't abuse adjectives, and make sure every noun has a verb. Treat each prompt like the beats for a scene.

After much consideration, I craft the following:

Dolly in on an old wizard in his shadowy lair as he examines his own steampunk clockwork hand. The mechanical hand opens and closes.

The First Attempt

Within seconds, Veo 3's Fast setting gives me two options, as seen in **Figure 7**. And I'm kind of stunned. I've performed this exercise with other tools like Sora, but it

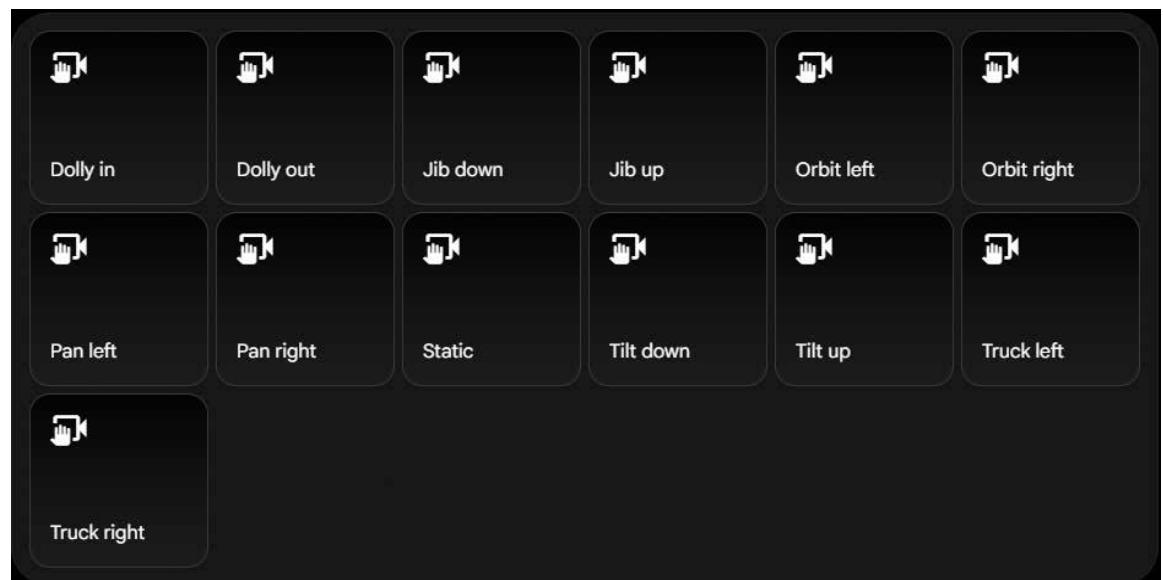


Figure 6: Frames to Video offers all the standard camera movements.

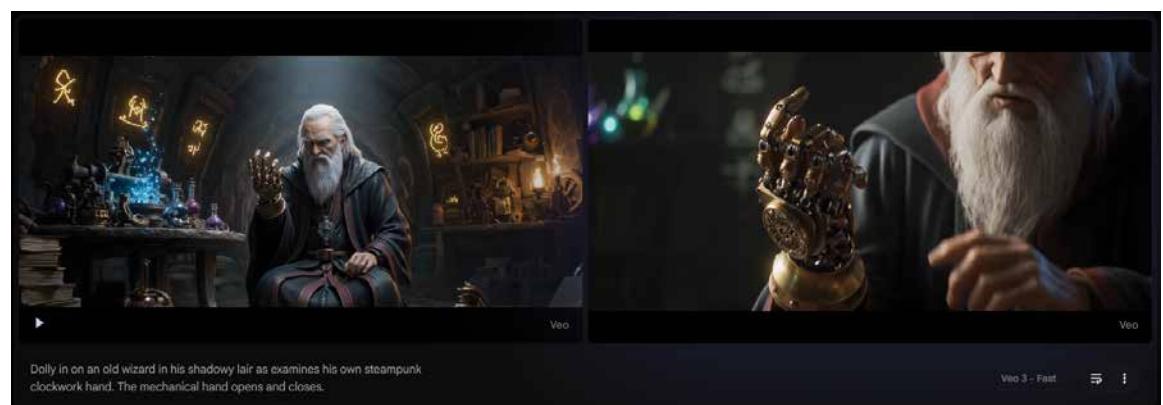


Figure 7: The first attempt is surprisingly close to what I imagined.

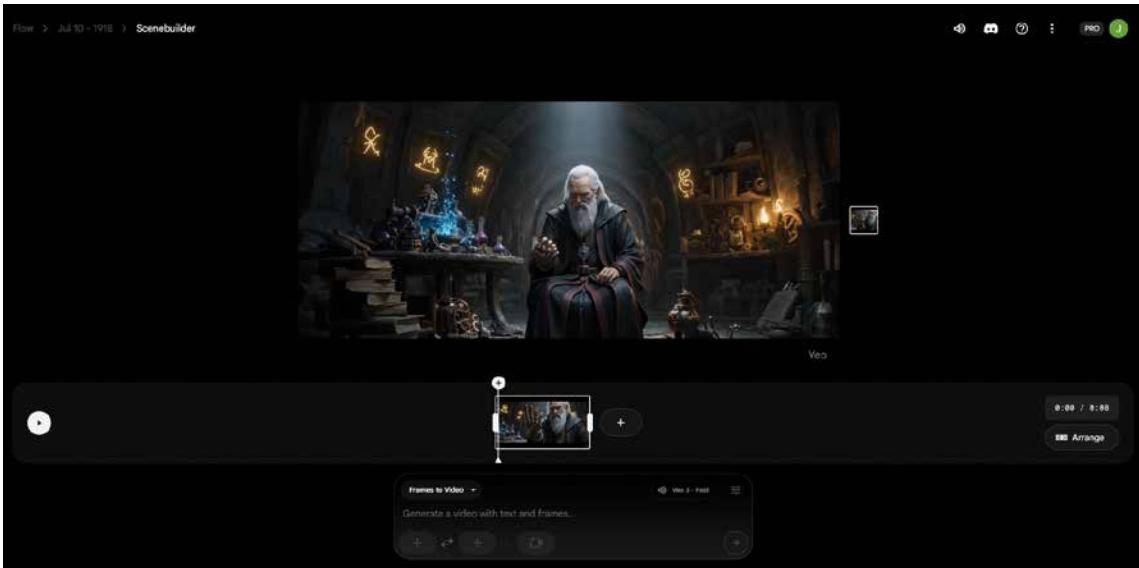


Figure 8: Flow's Scenebuilder editing interface leaves lots of room for growth.

never resulted in anything this complex and accurate. The model followed my instructions perfectly. From the jump, the results blew my expectations out of the water. It understood the assignment in every way, adding mysterious sigils and a sparkling cauldron to the image.

With image prompting, the first pass is usually a question of horseshoes and hand grenades. With every iteration, I'm often asking myself, "Is this close enough?" It's rare that it spits out exactly what I want, requiring prompting revisions and uploaded images to influence the result. Can I craft the perfect query, coaxing the model to do exactly what I'm envisioning? Yes, but it will take some effort, time, and money. That's the nature of **prompt engineering**, refining to find the just the right words that allow you to communicate your message perfectly to the Large Language Model. If you're trying to craft something with very specific details, that amount of trial and error can get expensive.

Basic Editing Tools

Once I've chosen my preferred output, I can add this beat to a greater scene with the click of a button in the Scenebuilder tool. For clarity, Flow is Google's platform through which you access models like Veo 3. Scenebuilder is the editing tool inside Flow. This ports the eight-second clip over into a rudimentary editing system, as seen in **Figure 8**. The options here are incredibly limited. The sliding bar over the video allows me to shorten it as much as I like but not make any specific cuts. If your project requires more flexibility—and it likely does—stick with your favorite non-linear editing system or grab something free, like CapCut or DaVinci Resolve. With Veo 3, as with Open AI's Sora, Google isn't trying to compete in the professional editing space. These options are functional, but the main focus is to offer an uncluttered look at what the model is capable of.

Adding another beat to the sequence is as easy as generating the first, but for this entry, I'll try the Frames to Video feature. This will require two keyframes: one for the beginning of the clip and one for where it will end

up. Rather than providing an uploaded example from the internet, I go ahead and use the tool to generate them directly. Keep in mind that the description you provide for this part isn't for video. This is just the starting frame, so prompt accordingly.

≡ An old door on the cliffside of a snowy mountain. It is carved with magical sigils, the entrance to a wizard's lair.

Veo 3 comes back with two fine options, as seen in **Figure 9** and **Figure 10**. The results for these still images are perfect, aligning with the arcane touches of the first video. The generic quality of the results is something to consider. Part of the reason the output here is so good is that we're dealing with the median. AI excels at presenting archetypes. When I ask for an old wizard, it comes directly from the mold of **GENERIC OLD WIZARD**. Good art



Figure 9: Here's a door, a mountain, and the arcane magics.



Figure 10: Is this the back door to Ironforge? Or a secret path through Caradhras?



Figure 11: This feels like it's back in Midjourney territory.



Figure 12: I don't think that's a 5E spell.

direction with specific, focused human effort is still required to make your project stand out from the infinitely expanding pool of slop. That takes more work, time, and effort. It needs your touch—your special sauce—to make it something new and different.

Let's go with the first one from **Figure 9**. It's a tiny bit spookier. For the ending keyframe, I want the wizard standing on the cliffside, casting a mighty spell. Sounds simple enough, right? But here's where I'm afraid it may get tricky. With this new generation, will the wizard look the same? I need consistency, which has been an ongoing issue with earlier video models. Faces will shift. Limbs will twitch. Maybe an entirely different wizard person will emerge. I hope it's not that Potter nerd.

An old wizard with a steampunk clockwork hand stands on a snowy cliffside, holding his hand out to cast a powerful spell into the sky.

I believe this is what we call "the other shoe dropping." Veo 3 was on such a roll. The wizard looks the same-ish. He's still an old guy with a white beard. In **Figure 11** and **Figure 12**, you can see where things start to go awry. Here begins the game of horseshoes. Is this close enough? No, I don't think so. I'd like to spend a few more credits to see if I can coax something better out of it.

An old wizard with a steampunk clockwork hand stands on a snowy cliffside, holding his hand out to cast a powerful spell at the camera.

The change in the prompt was subtle, nudging the model to closer to what I want. One of them still has that video game sheen to it, while the other looks decidedly more photo-real, as you can see in **Figure 13** and **Figure 14**. Neither of them is exactly what I want. Without the ability to upload a reference image, the model starts from scratch with the old wizard. The wizard from the first clip is nowhere to be found. If I were so inclined, I could port them into a more powerful AI editing tool and make tweaks within the image. Many Gen AI image tools offer in-painting, where specific areas of an image can be altered via targeted prompting. For the purposes of this exercise, however, I'll just go with **Figure 13** as the final frame of this beat.

Now that I have my keyframes, I can tell Veo 3 what I want to happen and instruct the camera in which direction to move, if I want a dynamic shot. The transition will move the video from **Figure 9** to **Figure 13**. That's usually where you see a lot of artifacts and shenanigans, things shapeshifting on the fly, and the world becoming rubbery and amorphous.

The door opens. An old wizard with a clockwork steampunk hand steps out onto the snowy cliffside. He raises his clockwork hand to the camera, casting a powerful spell into the air.

I select Dolly Out for the camera movement and cross my fingers, but I'm immediately met with an error. As of this experiment, the camera movement buttons only work with Veo 2. Once I go ahead with the lesser model, Google serves up its first disappointment, as you can see in **Figure 15**.

Veo 2 drops the ball. The door doesn't open. The photo-real Not-Gandalf slides unnaturally across the image, holding his glowing hand out like he's begging for wizard drugs. It looks ridiculous. Everything people hate about AI video manifests within this eight-second clip. It may just be the limitations of Veo 2, which I'm having trouble reconciling. Why switch to the lesser model for a major feature of your product?

Rather than wipe the entire effort, I go back and start with the door. I like my magic door. It can stay. The prompt remains the same, but I'll only provide the starting keyframe. Trying to force it to go from one keyframe and end up in the next is causing it to contort in ways that make it useless. **Figure 16** shows you the folly of my ways.

In Veo 2's defense, the walking animation on the Warlock Rick Rubin is pretty good. Everything else, like the way he just kind of squeezes through a crack in the door, is terrible. As bad as these images look, the videos are worse, but that doesn't mean the keyframe method is worthless. Far from it. If you use a frame from the original video as the starting point, you can extend the original by another eight seconds and maintain consistency of the images.

The end of the initial clip can be seen in **Figure 17**. That's the starting point for the new clip.

 The old wizard stands, turns, and strides through a door out onto a snowy cliffside.

The results are much better! It looks like the same wizard this time, exiting the same lab I had before. Now when I say "the same" here, I mean "pretty similar." The details shift as I move from one clip to the next and the ambient noise is inconsistent throughout the sequence. With the second clip, I can hear the room tone drop out, immediately replaced with alpine winds. There's a very clear warning in the interface that apologizes for errors with the native sound generation. At this stage in the product's life cycle, that's not a dealbreaker.

Now that my spellcaster has stepped out into the snow, I have to see if I can kick the action up a notch. The end of the second clip gives the beginning of the third clip in the sequence, which you can see in **Figure 18**.

 The old wizard steps out into the snow, raises his clockwork hand to the sky, and casts a powerful spell.

It works brilliantly. The third clip is generated as instructed. The clockwork hand isn't the same one it designed before, but as the gears and cogs spin, so too does a swirl of sparkling magic, as seen in **Figure 19**. The spell crackles in the audio. Now I have a 24-second sequence. If you're



Figure 13: He looks familiar.



Figure 14: Veo 3 took the "clockwork hand" a little too literally.



Figure 15: Inconsistent styles within the same image are a big fat fail.



Figure 16: Now it looks like something from a bad Full Motion Video game from the 90s.



Figure 17: This will serve as the new starting keyframe.



Figure 18: It's not quite John Ford, but I like it.

anything like me, you know what the wizard needs to do next. He needs to blow something up with that magic he's working.

The wizard needs a formidable foe, so let's go big. After all, the small \$20/month movie budget is going a long way already. Because this will be a different shot that won't continue from the wizard's path, I can't use any-

thing from the sequence as a starting reference frame. From here, I switch back to Text to Video.

- Wide shot of a clockwork titan that looms over the snowy mountains. Something on the ground far below gets its attention, so the 200-foot-tall behemoth turns to look.

As seen in **Figure 20**, this new character suits my needs pretty well. It's a colossus of cogs and gears, billowing smoke high above the frosty landscape. It moves as I asked. More or less. Within tolerance, anyway.

Once I get a grasp of Scenebuilder and its limitations, I go all in, burning through credits in a frenzy. Every clip adds eight seconds to the sequence. I trim them and move them around on the timeline, grabbing stills from each one to use as my next starting keyframe.

- The magic from the wizard's spell distorts the sky with arcane energies. It arcs and blazes with power.
- A storm of purple and blue lightning billows through the sky, attacking the titan. The magic storm tears it apart.
- The titan falls to pieces. It collapses violently into the snow.
- Wide shot of this titan rampaging through medieval alpine villages, stomping on homes. Villagers flee the destruction.
- Dolly out to show cheering villagers celebrating among the rubble of their village.
- The old wizard lowers his clockwork hand, straightens his robes, and turns to head back inside.
- The titan thrashes and flails as the powerful magics tear it apart.

My wizard doesn't return to his lair, for some reason. He decides to mosey off into the sunset. I lean into the mistakes, letting them steer the story. "Vibe coding" is a thing. Is this "vibe directing?" The two terms seem antithetical, but sometimes fighting the "happy accidents" makes things so much worse.

- Dolly up and out as the old wizard walks off into the distance.

I suppose it's a more fitting end for a legendary hero after a feat of wizardry. With each generation, the output surprises me in good, bad, and very strange ways. As you see in **Figure 21**, the villagers are indeed celebrating. What you can't hear is the song they're singing. In unison, they belt out, "The darkness is goooooone!" It's not the tone I was going for. Not at all.

I'm ready to compile the final cut, happy with an amusing bit of generic high fantasy when Scenebuilder collapses under the weight. Once I get more than eight or nine clips, it becomes impossible to rearrange them on the timeline as the interface promises. What should be a simple click-and-drag operation is an exercise in frustration. I can't see most of the clips and scrolling through them just doesn't work. It results in me putting clips

in the wrong spots, deleting the wrong segments, and ultimately losing the sequence altogether. I made the mistake of refreshing the page. There was no method to save my work other than downloading the entire thing. Luckily, each clip I generated remains in the library, but the overall edit is gone. My 90-second epic disassembled itself.

Assessment

Instead of using settings, dials, and lenses, you use language. That's the L of NLP. In natural language processing, clarity of message is key. Dress it up with a fancy name like "prompt engineering" all you want, there's still an element of sorcery. Craft your wish as best you can. Be precise, but don't overload it with details, and hope that the genie isn't feeling fickle.

We Are So Close

The raw video clips produced by Veo 3 are remarkable but still have one foot planted in the uncanny valley. If you know what to look for, you can't help but spot the telltale signs of AI. Most of what you can produce with it will still be derided as AI slop, because it is. Visual storytelling is about making choices. Veo 3 doesn't let you make many choices outside of what you can prompt. It's difficult to imagine someone using Veo 3 to craft an entire film. When hundreds or thousands of shots come into play, that requires iterative generation, review, and refinement. That becomes a massive logistical challenge. The consistency and specificity required to truly shape your creative vision just isn't there. That's a lot of time and money to just get close enough.

The toolset provided for editing the timeline is slipshod and ineffective. The Scenebuilder fails on almost every level. It borders on useless, but that's very telling. Right now, Google isn't in the business of making non-linear editing suites. They don't care about that. That's not the point. They care about providing this stunning video generation model. I wouldn't expect many more bells and whistles to be added. That's better left in the hands of full editing environments like Adobe Premiere, and Google knows it.

It's that raw video material that's ripe with promise. The real benefit is found by adding it to existing processes. True utility can be had here by generating drone shots, temp scenes for workprints, or quickly converting storyboards to animatics. It should be one of many tools in a creator's arsenal, where the video can be ported into a proper post-processing system for color correction, audio mastering, polishing the motion graphics, etc. Human intervention is still a must for a fully realized product.

That said, even now we're starting to see the first batch of Gen AI commercials on the market. The canned water company, Liquid Death, just introduced a commercial full of cultists, explosions, and a whale on the highway. It was made by one person. Although the uncanny valley is unfit for what most people consider cinema, expect to see a surge of commercials and other shortform content made on the cheap. If a company can hire one person to produce a video instead of budgeting for an entire cast and crew, what do you think they'll choose?



Figure 19: My wizard means business.



Figure 20: It's the kind of enemy that will make D&D players hate you.



Figure 21: I don't know why they're singing. No one gave them permission to sing.

SPONSORED SIDEBAR

AI Executive Briefing

Experience the game-changing impact of **AI** through **CODE Consulting's** Executive Briefing service. Uncover the immense potential and wide-ranging benefits of **AI** in every industry. Our briefing provides **strategic guidance** for seamless implementation, covering crucial aspects such as infrastructure, talent acquisition, and leadership.

Discover how to effectively **integrate AI** and propel your organization into future success. **Contact us today** to schedule your executive briefing and embark on a journey of AI-powered growth. www.codemag.com/ai

Accelerating Returns

Veo 3 is a groundbreaking tool in a time when groundbreaking tools are being released every day. In January of 2025, this article was originally about Sora. It was ready for print, but print is a slow medium and artificial intelligence is advancing at a speed that's difficult to comprehend. The trajectory of video models over the last year shows exponential growth. Futurist Ray Kurzweil describes these leaps as part of the "law of accelerating returns." Kurzweil's law holds that each new jump in capabilities builds on the last, compressing development timelines into a blur of progress. Less than a year ago, the best video models in the hands of consumers could only generate incoherent horror shows. Hands warped. Faces twisted into surreal nightmares. Native audio was a fantasy. Veo 3 illuminates the road ahead, showing us that as the technology iterates and stacks, full cinematic consistency isn't far off.

This is as bad as it will ever be. Eight-second clips will become five-minute short films. Those short films will grow into prompt-driven movies. Not long after that, we'll be able to tell our favorite movie generating service that we're looking for a murder mystery set on Venus, starring Jean Claude Van Damme and Betty White, directed by David Fincher, and it's an unofficial sequel to "Madam Web." It will be packaged for streamers like Netflix, who will be able to look at your preferences and manifest a surgically targeted film. Ready or not, we'll all have movies made by clockwork hands.

Jason Murphy
CODE



Sep/Oct 2025
Volume 26 Issue 5

Group Publisher
Markus Egger

Editor-in-Chief
Rod Paddock

Managing Editor
Ellen Whitney

Content Editor
Melanie Spiller

Writers in This Issue
Matthew Hess
Joydip Kanjilal
Sonu Kapoor
Sahil Malik
Jason Murphy
Paul Sheriff

Technical Reviewers
Markus Egger
Rod Paddock

Production
Friedl Raffeiner Grafik Studio
www.frigraf.it

Graphic Layout
Friedl Raffeiner Grafik Studio in collaboration with on sight (www.on sightdesign.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: Accelerate360
EBSCO
Magzter
Zinio

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

YOUR PARTNER FOR CUSTOM SOFTWARE SOLUTIONS



REAL BUSINESS VALUE FOR AI
TRAINING / MENTORING

CUSTOM APPLICATION DEVELOPMENT
CONTINGENT IT STAFFING

Is your development team struggling to complete business-critical projects on time?
Are you looking to harness cutting-edge technologies, including AI, for maximum impact?

CODE EXCELS IN:

- **AI integration for enhanced functionality**
- **.NET web and desktop development**
- **Azure cloud migration and transformation**
- **Blazor development**
- **Mobile app creation**
- **Staffing, training and mentoring**

Let CODE transform your software challenges into competitive advantages.

CONTACT US TODAY. NO STRINGS. NO COMMITMENT.

codemag.com/code

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



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