C#, Laravel, Note.js, Chrome Debugging, DuckDB

CODE

MAY
JUN
2023

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

Writing Programs with Programs

A Simple Guide to Moq

Understanding .NET Code Generators

Using DuckDB

0 5>
7 25274 56130 8

CODE SECURITY

shutterstock_rangizzz/Africa Studio/onsight

INCREASE YOUR CYBER SECURITY DEFENSES

IDENTIFY POTENTIAL SECURITY PROBLEMS

CODE Security offers application security reviews to help organizations uncover security vulnerabilities. Let us help you identify critical vulnerabilities in complex applications and application architectures and mitigate problems before cybercrime impacts your business.

- **CODE Security** offers three types of security testing for many applications and services, including simulating a real-world attack scenario. We can test web applications, internal applications, mobile applications and do code audits and test hardware. We can even reverse engineer software if you no longer have the source code.
- After our security audit, our **CODE Security** experts provide a technical report including all identified vulnerabilities along with a severity rating according to industry standards.

ADDRESS POTENTIAL SECURITY PROBLEMS

With a list of identified vulnerabilities, what do you do next? We have two suggestions:

- Give your internal developer team the knowledge to address security problems in your applications. **CODE Security** offers two flavors of Secure Coding for Developers courses. Each course is three days but follow the same basic outline. One course is primarily for C# developers and the other is for Java developers. See our training page for a list of upcoming classes.
- If you cannot spare developers to overhaul your software, hire **CODE Consulting** to work with your teams and we'll address the security vulnerabilities together.

Let 2023 be the year you mitigate cyber security risk in your applications.

CODE Security experts can help you identify and correct security vulnerabilities in your products, services and your application's architecture. Additionally, CODE Training's hands-on **Secure Coding for Developers** training courses educates developers on how to write secure and robust applications.

Contact us today for a free consultation and details about our services.

codemag.com/security

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

Features

Columns

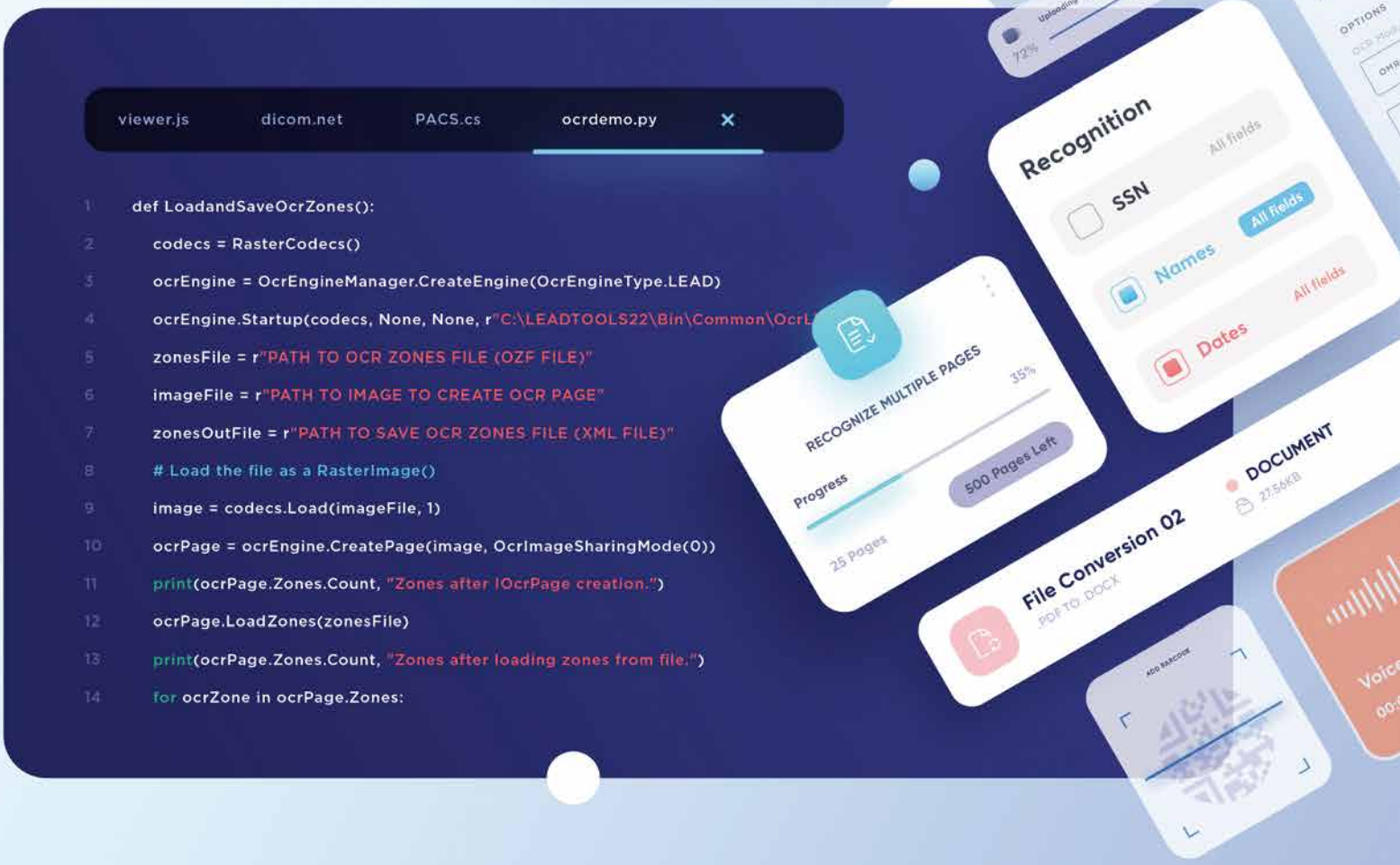
Departments

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

Subscribe online at www.codemag.com

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

BUILD BETTER APPS WITH LEADTOOLS®



Powered by patented artifical intelligence and machine-learning algorithms, LEADTOOLS SDKs help you create desktop, web, server, and mobile applications with the most advanced document, recognition, medical, and multimedia technologies, including:

OCR | DOCUMENT VIEWING & EDITING | BARCODE | PDF | IMAGE PROCESSING | + MANY MORE

DICOM | OFFICE FORMATS | FORMS RECOGNITION & PROCESSING | VIEWERS

Get Started

DOWNLOAD OUR FREE EVALUATION

LEADTOOLS.COM

The Power of the Pivot

The queue to the 2023 SXSW keynote seemed manageable. Our intrepid editor was ready to squat at the opening session in order to guarantee a spot in a later keynote that was sure to be popular. But on this day, the conference gods didn't smile on our venturesome attendee, as the line was cut off just as entrance was imminent. What could be done? One possibility would be to play the "press card," which had only a limited chance of success, or he could simply find another session to fill time until a seat in the main hall could be acquired. "Let's pivot," he said to himself. A new session was quickly targeted.

With a few minutes to spare, our valiant editor found a seat where he would spend the next few hours catching multiple sessions The first session was chosen for the same reason as the first keynote: to fill a timeslot in order to guarantee a spot in the following session that seemed likely to be popular.

Once again, things didn't turn out as expected as the "throw away" session was anything but. This talk was titled "Beyond the Podium: How Athleta and Allyson Fenix Reinvented the Brand/Athlete Partnership." It was an interview with Allyson Ferix, an Olympic gold medalist, mother, and entrepreneur. Allyson helped change the discriminatory nature of endorsement deals that's applied to female athletes. You see, female athletes are penalized when they decide to have a family—in particular, when they or their partners become pregnant. This speaker changed that. When her sponsorship was negotiated, she made Nike remove the penalties from her contract when the time came for her to have a child. I was floored that this was still a thing.

This is where our astonished editor learned the power of the pivot. The missed session led to an opportunity to catch another session that he wouldn't have chosen otherwise, and he learned not only a powerful lesson about discrimination (and fairness), but that staying open to new ideas was the best way to deal with nearly any situation.

Cut away to QCON London 2023. Another keynote. Jetlag provided a good excuse to sleep a bit longer and not worry about the opening keynote, but duty to CODE Magazine and the QCON hosts pestered. A few short rides on the London Underground later and our groggy editor finds himself seated in the Churchill Room of the Queen Elizabeth conference center. The keynote was delivered by Leslie Miley, advisor to the CTO of Microsoft. The keynote was entitled "AI Bias and Sustainability," and, to be honest, the title really didn't do justice to the content that was about to be delivered. '

This keynote focused on the negative impact that the current push into generative AI will have on our world resources, with negative

Figure 1: A close look (copyright 2023, Rod Paddock)

consequences being felt for generations to come. Focus was spent on the amount of energy consumed by data centers, the amount of water consumed to cool those data centers, the land consumed by data centers, and, finally, the physical affects these all have on the people who live and work near these data centers. The picture painted isn't rosy.

A parallel was drawn to the construction of the federal highway system that began in the mid-50s and continued for a decade and a half. This project came with a large number of benefits: Thousands of jobs would be created, interstate travel would be greatly simplified, and unseen opportunities would be reaped, including the population of the western states. The bad news—a harsh reality—is that these benefits came at a cost primarily to minority communities. To better understand these costs, you need to understand the discriminatory culture of the United States during that era. The concept of "separate but equal" was anything but, and the highway project reflected this attitude. Neighborhoods were divided based on race. Here's a truth that I'd never heard: Certain bridges in New York were deliberately created where buses could not cross them in order to keep "those people" out of certain boroughs. This is an astonishing fact, much like the contractual issues facing female athletes.

Figure 2: The whole picture (copyright 2023,Jessica Cargill)

It's become important to ask ourselves whether the creation and usage of AI is going to continue the divide of the haves and the have nots. To better understand, here are a few facts that were mentioned.

- The amount of water generative AI takes to cool its hardware is prohibitive. Scarcity of water causes problems on the lower rungs of the socioeconomic cohort.
- Building data centers is costly, especially in terms of land use. Where is that data center going to be located? Is the creation of a data center going to displace people from their homes, and if so, who will be displaced? If history is any indicator, it will likely be the same cohort that is already affected negatively by previous generations of "improvements."
- Now is the time to consider the negative side-effects of this technology and do something about it rather than when it's already released, too late, and irreparable harm has been done.

It's great that you're thinking about these concepts now. So is our earnest editor. He's also recognizing that the power of the pivot is in getting unexpected insights that are important and might change the way you develop, do business, vote, or live your life. Our erstwhile editor went to sessions he'd no desire to see and came away enriched with new insights that will change the way he goes forward in the world.

Consider the figures. They were captured by two photographers. The two photos, taken minutes apart, are of the same wall, yet provide entirely different perspectives. **Figure 1** is zoomed in on the little purple guy and **Figure 2** is of the same location, only zoomed way out. Notice anything different? You know, like the BIRD? The person who shot the zoomed in image COMPLETELY missed the bird, as his focus was on the purple dude.

Such a simple change in perspective clearly illuminates the power of the pivot. So take a chance and pivot. A perspective change is a powerful thing. Don't be afraid to learn something new.

Rod Paddock
CODE

More Chrome Debugging Tips

In my previous article in CODE Magazine (https://codemag.com/Article/2303021/Chrome-Debugging-Tips), I shared some tips and tricks I use with Chrome DevTools. To refresh your memory, Chrome DevTools is that window that launches when you hit F12 on Windows or CMD_OPT_I on a Mac. If you've developed for the web, be it HTML, CSS, or JS, I'm sure you've used them at some point.

Sahil Malik
www.winsmarts.com
@sahilmalik

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

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

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

My focus wasn't to talk about the basics of Chrome DevTools, but more about the hidden, and very interesting, tips and tricks I use to super charge my dev lifecycle. As I started writing the article, it became clear that there's so much more! So here I am with even more Chrome debugging tips.

Find Unused JavaScript

As is common when you write code for any project, the codebase gets larger and larger. Over time, the individuals who checked in certain code, with inexplicable comments, leave the company, retire, get hit by a bus, or win a lottery. And now you're paralyzed. You don't know if removing certain code will cause a missile to launch elsewhere in the world. JavaScript is no different. You will frequently have gobs and gobs of code, and no idea how useful it is. This can become a real problem in sites where performance matters.

Chrome DevTools have a convenient mechanism to allow you to find what percentage of any JavaScript file has been used, as per the actions performed on the page so far. Additionally, it can show you block by block or function by function what portions of code have been executed, and what portions are just sitting there taking up space.

This is invaluable in finding JavaScript on a page that's never been called. Here's how you can find unused JavaScript on any page. Open any popular site; in my case, I went to Google's home page. Open DevTools and go to the sources tab.

Now launch the coverage tab using CTRL_SHIFT_P (Windows) or CMD_SHIFT_P (Mac), and choose to reload the page. As you can see in **Figure 1**, the red portions clearly show you which portions of your JavaScript have so far not been executed. Additionally, it shows you a bar chart saying which files are the least used, along with total bytes and unused bytes. This is a great way to focus on the top optimizations you can perform on the page. You can continue to interact with the page and see how this data changes, and use it to make informed decisions around optimizing your page.

Finding and Working with Elements

One of the things Chrome DevTools is really good at is working with elements. There are so many tips and tricks hidden here! At the very simplest, if you glance at the top left of **Figure 1**, you can see an arrow icon on a rectangle. Go ahead and click on it and point to any UX element on the page. You'll see that chrome immediately selects, in code, the element you have visually pointed to.

But it doesn't end there. Hit ESC to reveal the console drawer, or just tab to the console tab, and type $0. This gives you a shorthand variable for debugging purposes that represents the current element. Now point to another element and type $0—it shows you the newly selected element. You'll find that $1 shows you the previous element you pointed to.

Now with an element selected, do a right click, and go to copy\copy selector, as shown in **Figure 2**.

This copies the jQuery selector as text into your clipboard. Chrome DevTools support the jQuery selector syntax for debugging purposes. You can try this by typing $("") in the console window and typing in the selector you just copied. It should immediately select the element you'd pointed to. I find this tip very helpful in figuring out complex jQuery selector syntaxes.

In the menu shown in **Figure 2**, there are a bunch of other useful tricks. My favorite one is "Break on," which allows you to force a breakpoint in JavaScript if the DOM of a particular element changes. This is great when you know some piece of code is changing some UX, but you just can't figure out what. You can hit the breakpoint and easily glance at the stack trace to see what portion of your code caused the UX change to occur.

XHR Breakpoints

JavaScript code can get complex. When you write AJAX code, one of the challenges is dealing with multi-threaded code. As you know, JavaScript doesn't really support multi-threading like C++ or C# do. To be precise, you can have threads, but you can't have strict control over how threads step over each other. This means that it can be difficult to pinpoint exactly what sequence of events caused an XHR request to fire.

To solve exactly this problem, you have XHR breakpoints. In the sources tab on the right-hand side, there's a section for XHR breakpoints. As you can see from **Figure 3**, I've set a breakpoint such that anytime an XHR request is sent to *.google.com, a breakpoint is automatically hit.

Now, I just have to sit back and wait. To my surprise, without my intervention in downtimes, www.google.com on a Mac was making XHR requests to **play.google.com**. Puzzling! But now I have the ability to debug why.

Conditional Breakpoints

When debugging JavaScript, you use breakpoints. But there's a good chance that you have a loop that gets called over and over again. Wouldn't it be nice if you could hit the breakpoint only if a certain code condition matched? This is the job of a conditional breakpoint. Simply set a breakpoint and right-click on it and choose to **Edit breakpoint**. You'll be presented with a user interface, as shown in **Figure 4**, where you can now input an

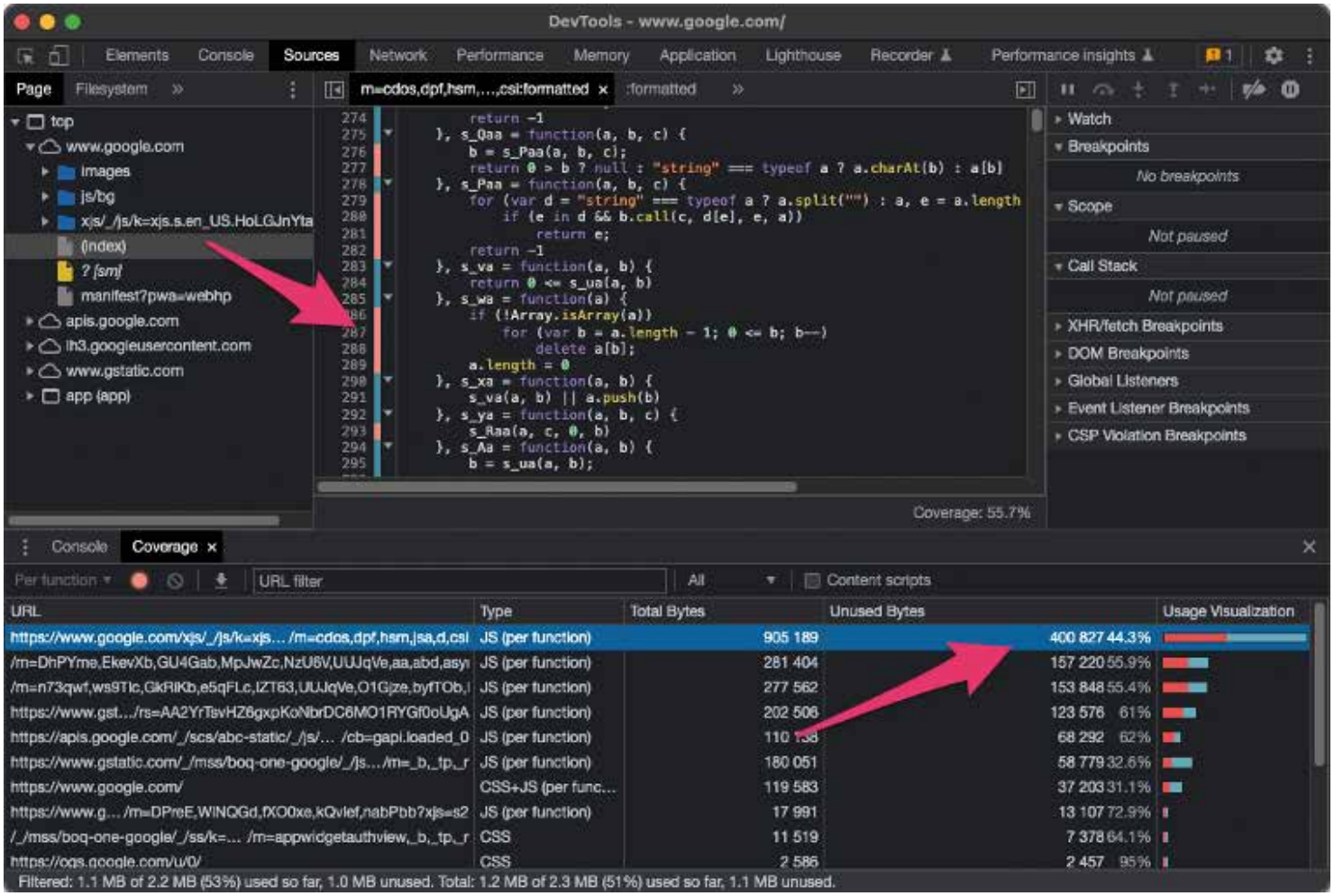


Figure 1: Code coverage in DevTools

expression. Whenever that expression evaluates to true, that's when the breakpoint is hit. Otherwise, the breakpoint is simply ignored.

Easily Convert Images to Data URIs

Data URIs are a recent enhancement to web standards. They're a great way to include data inline on web pages. A good example is an image. You can reference an image using the **img** tag. But this causes a separate HTTP request. Each request has its own lifecycle and generally slows things down. When working in development mode, you have numerous files with various extensions. Some are for fonts, some are JS, some are CSS, and some are images. These are usually kept separate. What helps you during development, works against you in production.

Luckily Chrome DevTools let you easily convert any image into a data URI format. This means that for commonly used images on a page, you can avoid an extra HTTP call. Not only does this help your page load faster, it also means items are available as soon as the page is available because they're all downloaded together.

Converting an image to data URI is quite simple. Go to the network tab and look for the image you're interested in. Click on the image, and go to the Preview tab. On the

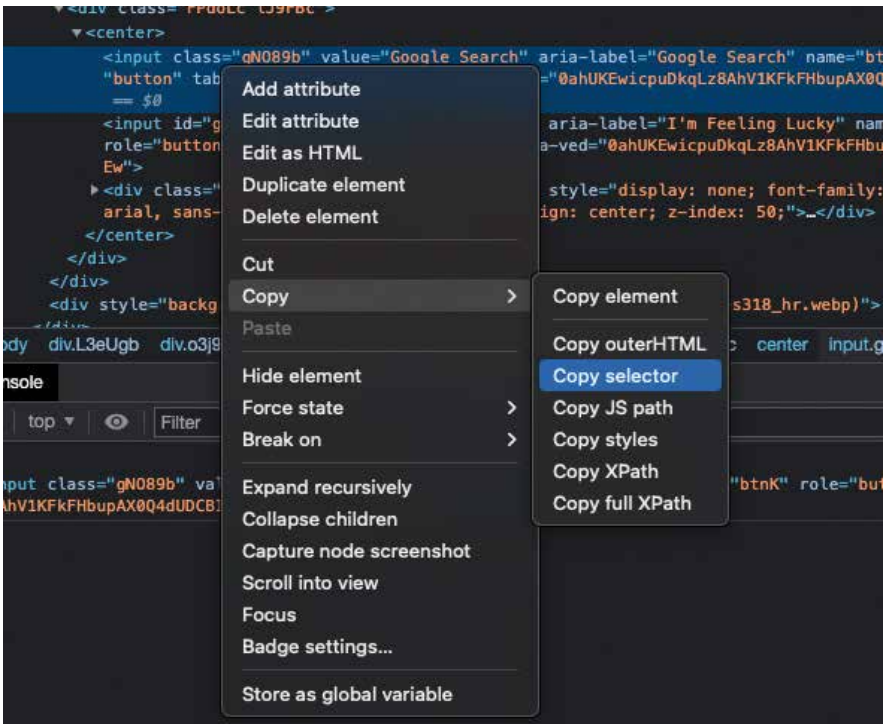


Figure 2: Copy selector

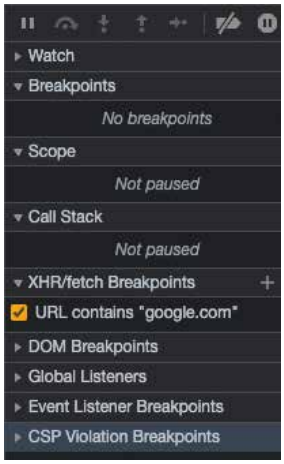


Figure 3: XHR breakpoints

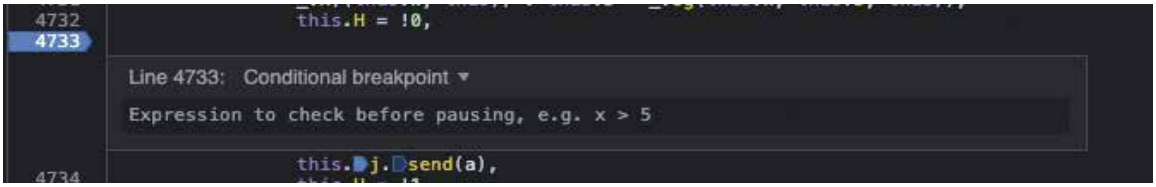


Figure 4: Conditional breakpoints

Figure 5: Image URIs from DevTools

image itself, do a right click and choose "Copy image as data URI," as shown in **Figure 5**. That's it, now replace your **img src** with this data URI, and the image is served inline with your HTML text.

Simulate Network Conditions

Developing for the internet is hard. Developers usually have high-end computers, but their customers may not. You could easily write a website that someone on a low bandwidth connection on an old device may have a hard time using. How do you simulate such situations? Chrome DevTools to the rescue.

In the network tab, hit ESC to open the console window, and on the left-hand side, look for the triple ellipsis button. Click on it to open **Network conditions**. Here you can do some interesting things, such as disabling caching, throttling network speeds, or changing user agents. This can be seen in **Figure 6**.

Figure 6: Simulating low bandwidth networks

Additionally, you can selectively block certain requests by adding patterns under the **Network request blocking** tab. This is useful to simulate overzealous network admins that you're trying to work around, just in case they block access to CDNs etc.

In-Browser Code Editing

How do you develop? I typically have my IDE such as VSCode open, and maybe I use live-reload if the project supports it. But mostly, it's a matter of editing code, pressing save, compiling if necessary, and hitting F5 to see if my code worked. This is terribly inefficient, especially when I'm trying to debug a hard to corner issue.

Chrome lets you do inline code editing right in the browser. For instance, I visited www.codemag.com and in the Sources tab, I noticed there was a file at /Content/2015 called Reset.css. I could simply open it and start editing it, and see the results apply live in the browser. This works with any kind of file. But it gets better.

The challenge is that once I've fixed a problem, I still need to collect the changes from the browser session and make them permanent by copying them to my source code in VSCode.

Go to the Sources tab in Chrome DevTools and hit ESC to open the console drawer. Here, click on the ellipsis button and look for the Changes menu. You can visually examine all the changes you had to make to get something working. Not only do Chrome DevTools show you which files are changed, it also shows you exactly what changes were done.

And when you are done making changes, simply hit the Copy icon at the bottom in the status bar to copy the changes to the clipboard. Now you can paste them into your original source code and make the code changes permanent. This can be seen in **Figure 7.**

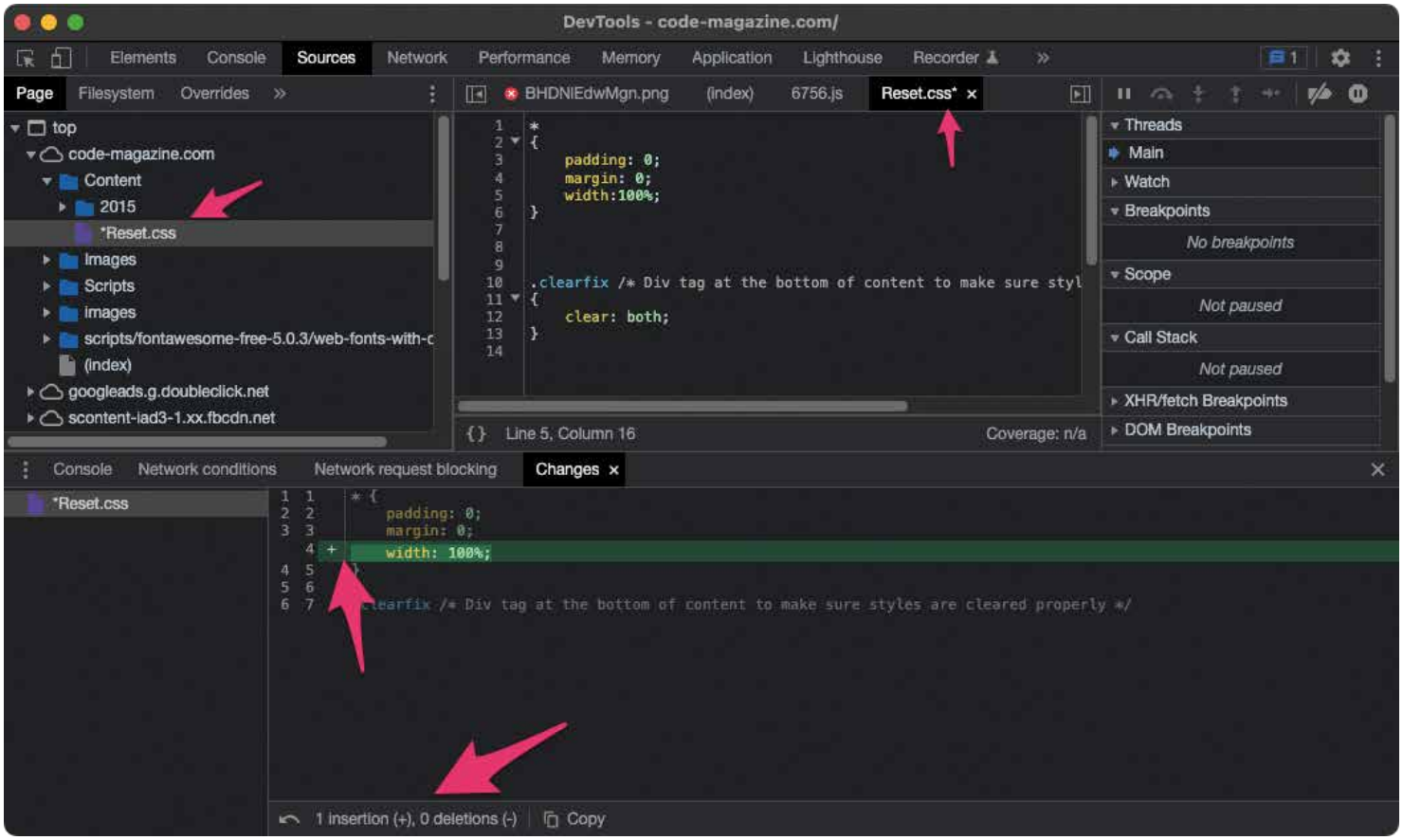


Figure 7: In-browser code editing

The CSS Overview Panel

The CSS overview panel is a great way for you to analyze the CSS on any given page, and even identify potential improvements. Open the Chrome browser and visit www.codemag.com. Open DevTools and press CMD_SHIFT_P (Mac) or CTRL_SHIFT_P (Windows) and launch the CSS overview panel.

You'll be greeted with a new panel that gives you a button to capture an overview. Go ahead and click on it. Clicking that button analyzes the CSS in use on the page. The created report has five sections.

The overview section gives you a high-level summary of your page, the elements, the external stylesheets, etc. Here, you can easily capture whether you have any inline styles in use for instance. It also shows you possible contrast issues that may make it harder to read text. This can be seen in **Figure 8**. Clicking on any specific contrast issue also shows you exactly where that contrast issue is present and double-clicking on that element takes you to the element in the source code of your HTML so you can examine it and experiment with fixing it right within DevTools.

The colors section, as the name indicates, gives you an overview of the colors in use on your page. Here also, each color is clickable so you can easily examine where a color is in use.

The Font Info section, as the name indicates, tells you which fonts are in use and where they're coming from. Additionally, it'll show you the number of occurrences of each font.

The unused declarations section shows you redundant elements that are present but may not have any effect on your page. For instance, on CODE Magazine's home page, it tells me that the vertical alignment style is applied in four places where the element is neither inline nor a table-cell. Removing that has no effect on the page, so you might as well clean up your code and remove it. Again, you can click on the section and navigate in code to where those elements are, and then easily locate the problems and clean up your code.

Finally, the media queries section shows you all of the media queries defined on your page sorted by the number of occurrences. As you may be aware, media queries allow you to specify different behaviors for different resolutions. If you've used media queries before, you know how easily they can get messy and it's a whole lot easier to keep them at a top level and clean to keep them manageable. This section will help you achieve that.

Easily Identify Memory Leaks

You'll find this tip extremely useful when writing complex pages. To try this out, visit a heavy website: I visited maps.google.com. Open DevTools and open the Performance tab. Here, choose to profile for memory in the checkboxes at the top and start recording. Now interact with the page as you normally would. And stop recording.

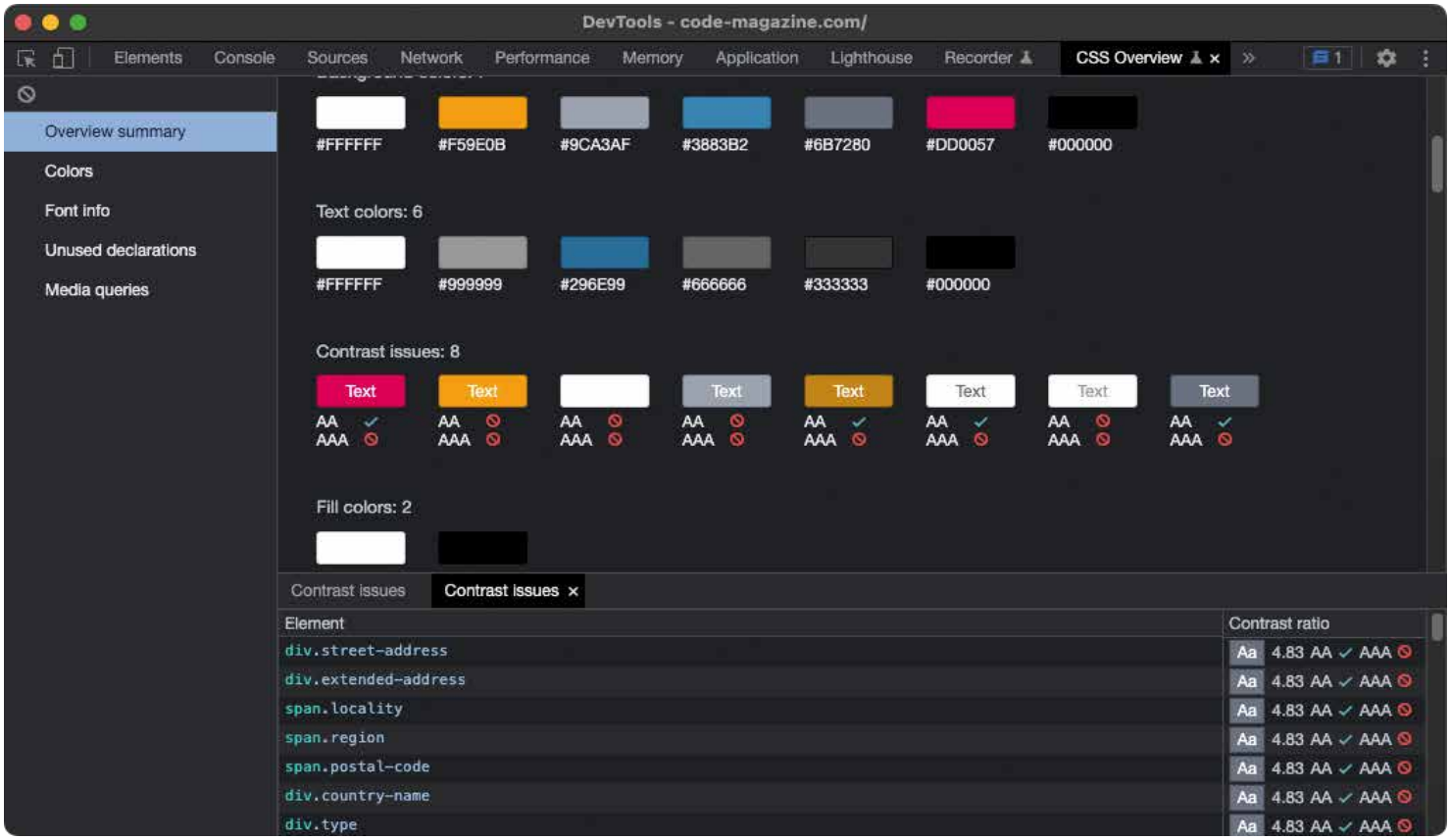


Figure 8: Contrast issues in DevTools CSS overview

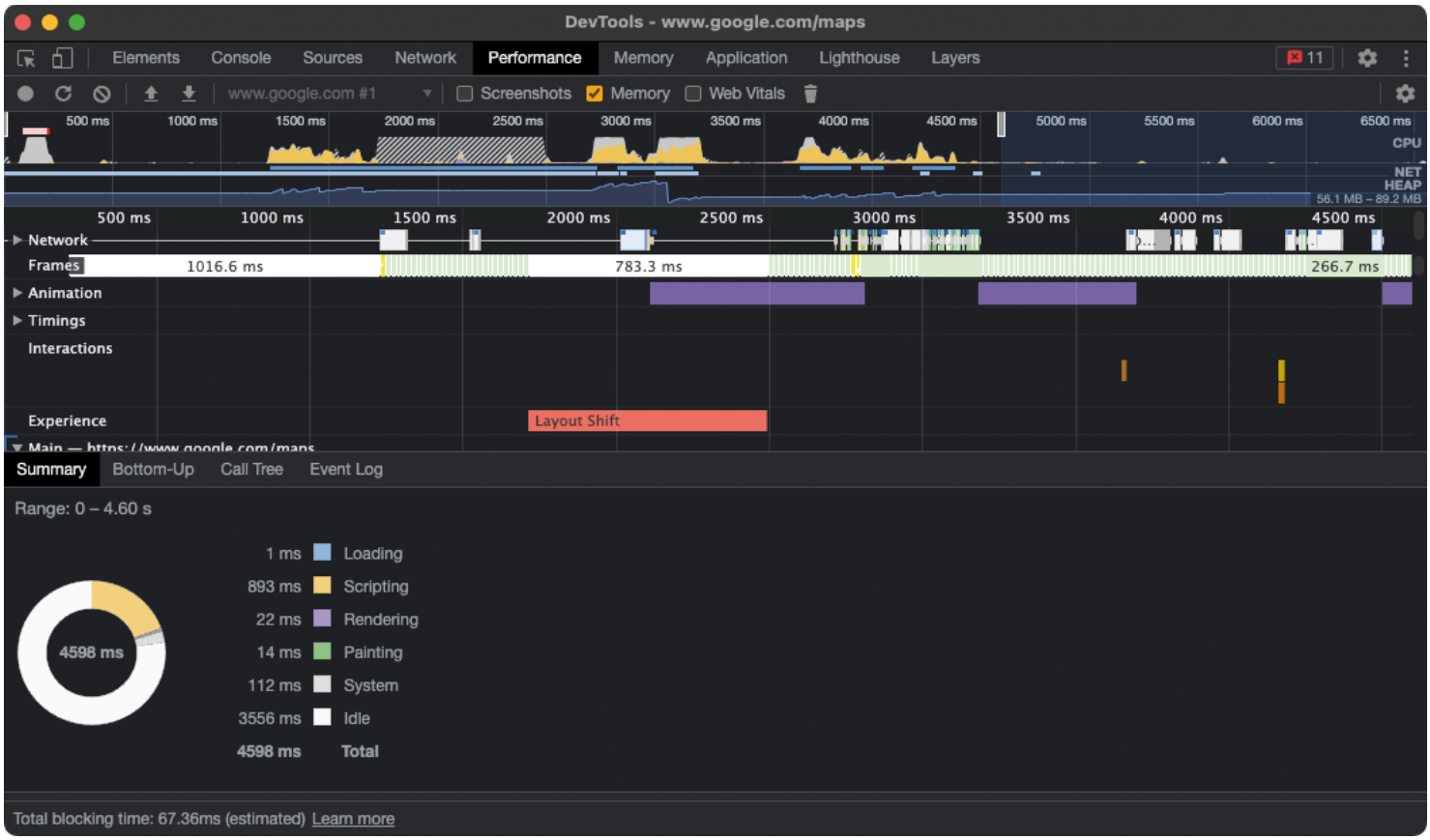


Figure 9: Performance capture for memory

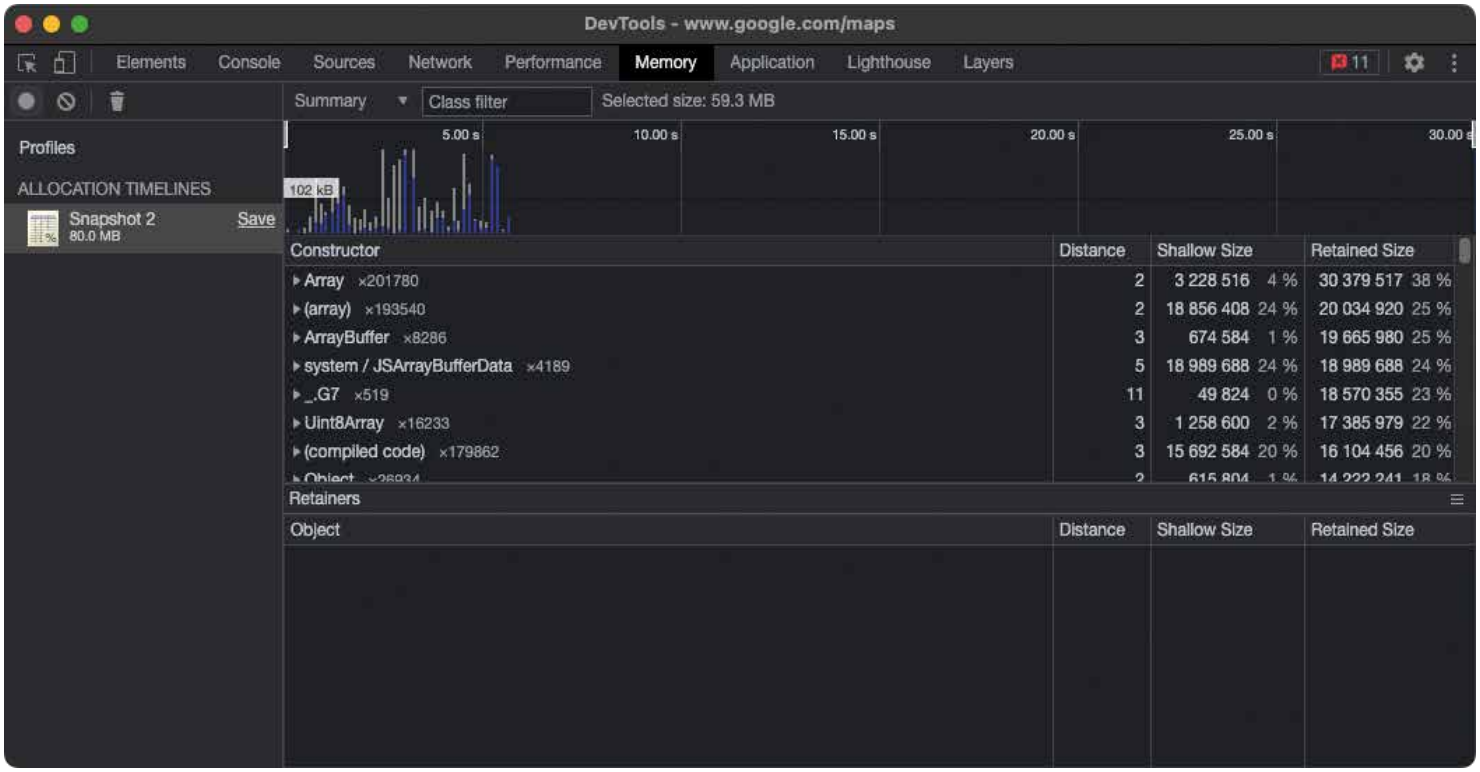


Figure 10: Memory snapshot capture

You should see a capture, as shown in **Figure 9.**

There's a wealth of information captured here. You can easily identify where the bottlenecks are, whether they're in network, or whether they're animations. The blue graph you see is memory usage; if you see that rising constantly, you know that you have a problem. Underneath, you see exactly where time was spent. You can see an overall summary, but by navigating through various tabs, you can also see exactly which function call took how much time, and at what point it occurred.

To do an even deeper analysis, open the Memory tab of DevTools, and choose the **Allocation instrumentation on timeline** option and start recording. As usual, interact with the page normally, and stop recording after a while. This produces an output similar to that shown in **Figure 10**.

Here, you can see that memory that's allocated and not deallocated is shown in blue. Memory that was freed up is shown in gray. You can look at exactly where and how the memory is being used, sorted by usage. You can select a small portion in the timeline of the memory footprint and select a specific element. This shows you the full object tree and show you exactly what object had consumed how much memory at what time. This can help you identify the largest memory piggies on your page.

Simulate Mobile Devices

Let's be honest, these days, if you're targeting only desktops, you're missing the party. Chrome DevTools includes capabilities to help you develop for mobile while being on a desktop. These tools can be seen in **Figure 11.**

At the most basic, you can click the button indicated by the red arrow in **Figure 11**. Clicking that button lets you simulate almost any mobile device as far as dimensions and DPI go. There are many presets to choose from or you can define custom presets. Additionally, you can choose to pick throttling behavior to simulate loading on different wireless networks.

However, developing for mobile isn't just resolution and DPI. Mobile devices frequently make use of capabilities such as location, orientation, or touch. All of these are grouped under Sensors, which can be opened next to the console tab, as can be seen in **Figure 11**.

Mobile devices also frequently make use of local storage, Web SQL, or IndexedDB. All of these can be explored under the application tab.

Finally, if you can't simulate something on a laptop and it must be diagnosed on an actual mobile device, you can physically connect an Android phone to your Mac or PC, and use Chrome DevTools to debug a webview running on the physical mobile phone. This, naturally, only works for Android. However, if you must diagnose iOS, there are similar capabilities in Safari.

Master the Console

Okay some days, all these fancy debugging tools take a back seat to the old fashioned console.log. You might already know that the console.log lets you log out some text in the console window. For instance, this is a valid console.log entry:

```
console.log("debug message");
```

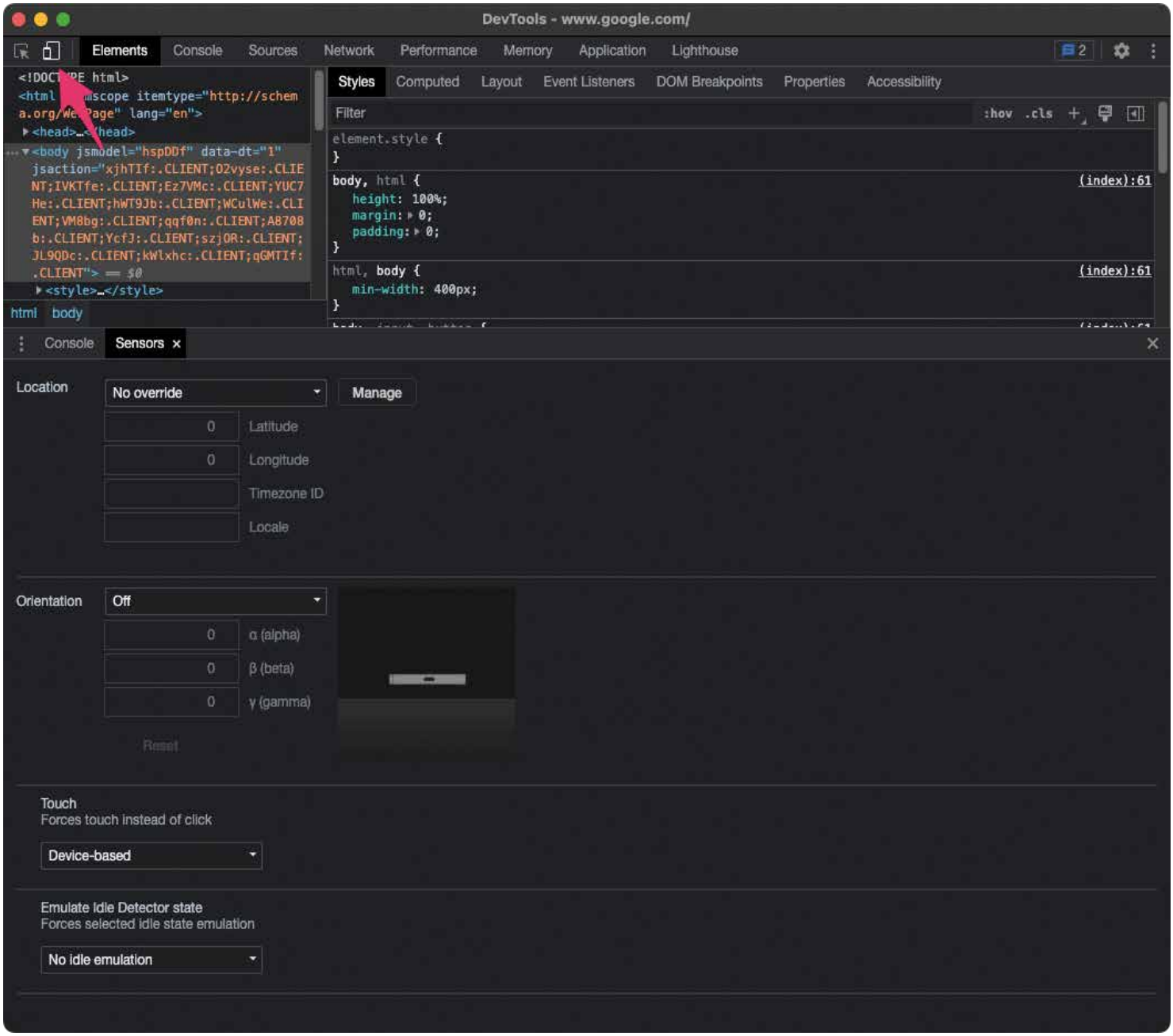


Figure 11: Helping develop for mobile

But console.log is quite powerful. You can use expressions, such as %s, like this:

```
console.log("data %s", "something")
```

This prints out:

```
data something
```

This is great for writing out expressions built using variables. You can even style the output. This can be seen in **Figure 12.**

This is a great mechanism to make certain that messages stick out prominently. Of course, you can also use console.log, console.info, console.error, console.warn, or console.assert to treat messages differently.

Figure 12: Styling console.log entries

Another corollary of console.log is console.table. Sometimes you wish to write out an array, but you wish to view the output as a sortable table. For instance, you want to easily find all the external scripts referenced on a page or all the hrefs on a page. To do so, you can simply issue a console command as follows:

```
console.table(document.scripts, "src")
```

This command writes out all the scripts on a page as a table output and shows the index and src columns.

```
> console.group("mygroup");
  console.log("something");
  console.log("something else");
  console.groupEnd("mygroup");
▼ mygroup
    something
    something else
<- undefined
>
```

Figure 13: Grouping console messages

Yet another trick you can use is console.count. Sometimes you wish to count the number of times a certain function or line was executed. You can simply issue the following statement:

```
console.count("label")
```

As many times as this line is executed, you'll see an incremented count written out. You can change the label text to count more than one occurrence on a page.

Finally, there's console.group. Sometimes you get so many console.log messages that it becomes very difficult to parse through them. You can render console messages in collapse/expand groups using console.group, console.groupCollapsed, and console.groupEnd. This is best seen in **Figure 13.**

The idea here is that you can initiate a group of messages by passing in a label. The label is what allows you to eventually close that group by passing the same label name to console.groupEnd. Now any messages between these two group calls get grouped in a collapse expand section, as can be seen in **Figure 13**. Of course, if you wish to initiate a group in a collapsed state, you can start the group by calling console.groupCollapsed.

This is great when you wish to log a certain method's output, but you want to collapse to out of your view so you can focus on the larger picture.

$

Okay, we all like $, but I'm talking about a better version. There's the $ symbol in the JavaScript world that we're all familiar with. This is what jQuery chooses to use as its shortcut. In Chrome, the $ symbol with its jQuery like effects is built right into Chrome DevTools. Well, it's not all of jQuery, but the general selector syntax works. To be specific, it's a shortcut for document.querySelector, and $$ is a shortcut for document.querySelectorAll.

Now to be clear, this is just a debugging tool and you shouldn't bake dependencies into your production code based on this $ symbol that may not be available in other browsers. But it really helps working in the debug console window. Let's try it out.

dtSearch®

Instantly Search Terabytes

dtSearch's **document filters** support:
- popular file types
- emails with multilevel attachments
- a wide variety of databases
- web data

Over 25 search options including:
- efficient multithreaded search
- easy multicolor hit-highlighting
- forensics options like credit card search

Developers:
- SDKs for Windows, Linux, macOS
- Cross-platform APIs cover C++, Java and current .NET
- FAQs on faceted search, granular data classification, Azure, AWS and more

Visit dtSearch.com for
- hundreds of reviews and case studies
- fully-functional enterprise and developer evaluations

The Smart Choice for Text Retrieval® since 1991

dtSearch.com 1-800-IT-FINDS

```
> var days = new Date("03/01/2023") - new Date("02/13/2023")
< undefined
> days
< 1382400000
> days = days / (1000 * 60 * 60 * 24);
< 16
> $_
< 16
> 848+481
< 1329
> $_/days
< 83.0625
>
```

Figure 14: Using JavaScript to do quick math

```
> var person = new Object()
< undefined
> person.firstName = "Sahil"
< 'Sahil'
> person.lastName = "Malik"
< 'Malik'
> JSON.stringify(person)
< '{"firstName":"Sahil","lastName":"Malik"}'
>
```

Figure 15: Serializing an object into JSON

ADVERTISERS INDEX

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

Advertisers Index

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

Go to the CODE Magazine home page (www.codemag.com) and open DevTools. Now issue a command as follows:

```
$("div")
```

This shows you all the divs on the page as an array. You can expand this array and click on any element to quickly navigate to a particular element. Or you can use console.table to work with this array. If you're a seasoned JavaScript developer, you can use more complex syntax. For instance, I know that there's a div on my page that has an ID of mobileMenu, and inside there, there is a ul, and I want to find whichever element has the "magazineMenu" class. No problem. Here's how you zero in on that.

```
$("#mobileMenu > ul > li.magazineMenu")
```

There are some additional interesting shortcuts built in here as well. For instance, typing $_ or $0 gives you a reference to the current element. $1 gives you the last, $2 gives you the one previous to that, and $3 and $4 goes to the ones before. You can imagine how useful this can be as you're going through elements in debugging and need to step back to the last element you were looking at.

You'll find that $_ is slightly different from $0. That's because $_ is technically the last evaluated expression. So $_ is a bit broader in its applicability than just user interface elements.

For instance, frequently I find myself doing math. And more and more, I'm getting dumber at doing math in my head. I know, not a great habit. I use a Mac, so I find myself doing quick calculations using Spotlight. Like 848+481=1329. Or maybe I'll ask Siri or OK Google to do some quick math for me.

Frequently, I need to something just slightly more complex. I dive into Google Sheets or Excel. But there's a middle ground, where I need to do some quick math, be able to use variables, and do it visually.

Let's say that I have $848 in one bank account and $481 in another bank account, today is February 14, and I wish to know how much can I spend per day till the next paycheck on March 1. You can see me doing this quick math in **Figure 14**.

It's worth pointing out that JavaScript's math is buggy, so I use it for informal calculations. There are npm packages available to do complex and more accurate math if need be, but for my quick and dirty calculations, this works great!

You can extrapolate other uses of $_, such as you can evaluate a code snippet that you drag and drop from your disk, save it in a temp variable as a function, and reuse it in any way you like.

JSON

JSON stands for JavaScript object notation. It's a data transfer format with some basic data type semantics. It lets you store unstructured/semi-structured data. What's great about it is that it doesn't consume a lot of space,

```
> console.log(JSON.stringify(person, null, 2))
  {
    "firstName": "Sahil",
    "lastName": "Malik"
  }
```

Figure 16: Prettyfing JSON

like XML, and all modern browsers have a native JSON parser built into them. That object is called JSON.

A good example of serializing an object into JSON can be seen in **Figure 15.**

Just like you can JSON.stringify an object into a string JSON representation, you can do the reverse using JSON.parse.

There's one issue here. My JSON object here is very simple. It's not uncommon to see JSON objects that are huge, a few megabytes even, and that span multiple pages and have deep nested structures. Reading them can be quite onerous if they aren't formatted well. There are tools online that let you parse them nicely, and JSON objects fetched over XHR can be viewed nicely formatted inside Chrome DevTools' network tab. However, if you wish to arbitrarily format a JSON object in a more readable format in console, you can use the expression seen in **Figure 16.**

A Quick Scratchpad

Chrome is incredibly flexible. More and more, all my tools are being coalesced into a browser. I can almost guarantee that I always have the browser running on my computer. When I need to take a quick note, this tip comes in super handy. Almost like an ephemeral sticky note. I create a bookmark in the bookmark bar and use a convenient emoji for the title, so it doesn't take up too much space, and the following as the URL. Note that line breaks are for clarity.

```
data:text/html,
<html contenteditable spellcheck
style="font-family:helvetica;
line-height:1.6;
background-color:2c292d;color:fdf9f3;
padding:32"/>
```

Now, whenever I wish to take a quick note, I just open a new tab and hit the bookmark button in the bookmark bar. Or you can even make this your start page. Once this tab is open, just start typing in this new browser tab. This can be seen in **Figure 17**. Whenever you're done, just hit **CMD_S/CTRL_S** to save this text, or just close the tab to get rid of it.

Summary

Web pages seem to be stuck together with sticks and spit. It's amazing how we've managed to build some incredibly complex applications using fiddly web technologies that have grown organically with time, such as HTML, CSS, and JavaScript. Frankly, it's amazing that, given all the diverse operating systems, browsers, versions, these

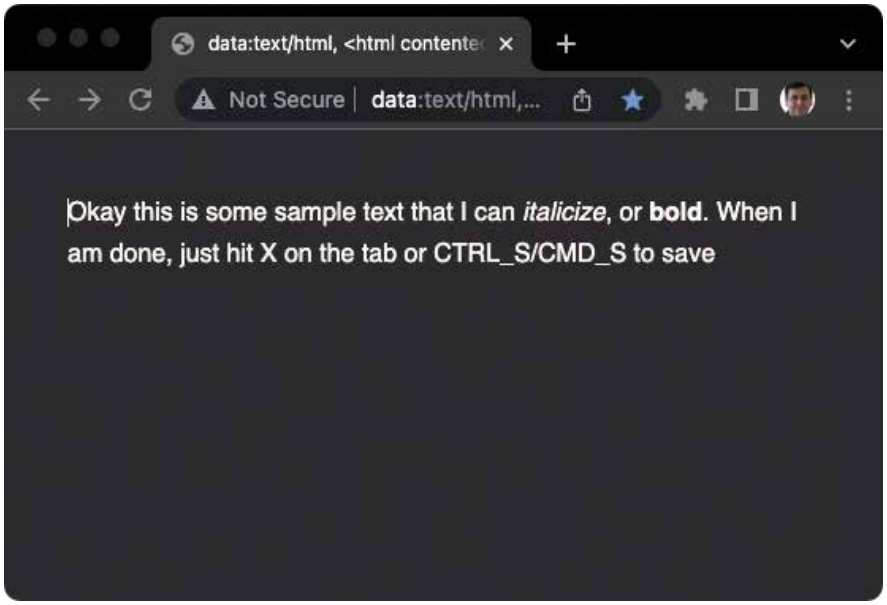


Figure 17: A quick scratchpad

fiddly technologies power the compelling applications that are the bread and butter of a major portion of the world's GDP. Think about it, Facebook, Twitter, Amazon.com, Google, Office 365, Google docs—so much else—are complex web applications. I'm not diminishing the role of back-ends here, but hats off to the engineers that've built the front-ends for all these technological revolutions.

At the heart of all this are good tools, such as Chrome DevTools, that help developers identify some of the hardest to nail down issues in some of the hardest to debug environments.

I hope you found a tip or two useful in this article, something you didn't know before.

Until next time, happy debugging.

Sahil Malik

Building Web APIs Using Node.js and Express: Part 1

Many developers use JavaScript to add client-side functionality to their web pages. JavaScript can be used to create server-side web pages and web APIs too. In this three-part article series, you're introduced to Node.js and Express and you'll learn their roles in the creation of a web server. You're going to build a new web server from scratch, create some routes to return data, and learn how to modularize your web application. You'll be guided step-by-step to creating a set of APIs to return an array of objects or a single object, search for a set of objects, and add, edit, and delete objects. If you've never used Node.js or Express to build a server, don't worry: this article has everything you need to learn the basics of working with these two powerful tools.

Over the next three articles, you'll learn to handle errors in Express using middleware and log errors to a file. You'll learn to use some additional **npm** packages to retrieve settings from a configuration file, and to read and write data to and from SQL Server. You're going to learn to build a website using Node.js and Express and make calls to your Web APIs from the pages on this website. You'll learn to configure CORS to allow cross-domain resource sharing. Using Express and Mustache, you're going to see how to build dynamic web pages with the data retrieved from your Web API calls.

Paul D. Sheriff
http://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/3gvXgvj) and on Udemy.com (https://bit.ly/3WOK8kX) on topics ranging from C#, LINQ, JavaScript, Angular, MVC, WPF, XML, jQuery, and Bootstrap. Contact Paul at psheriff@pdsa.com.

Node.js Explained

Node.js (https://nodejs.org) is a web server like IIS or Apache, but is cross-platform and open-source. Node.js can run on Windows, Linux, Unix, macOS, and many other operating systems. Node.js is a JavaScript runtime environment that runs on Google's V8 JavaScript engine. This means that you get all the good (and bad) of the JavaScript language for programming back-end tasks.

Node.js lets developers use JavaScript to write any type of web server environment they want. Using Node.js, you can serve up static HTML files, dynamic web pages, and Web API calls. One of the best things about using Node.js is that you use JavaScript for all your back-end programming, and on your web pages, you use JavaScript to manipulate your web pages and/or make Web API calls back to your JavaScript web server. Instead of using C#, Visual Basic, or Java for your back-end coding and JavaScript for your front-end coding, you use a single language for both.

Node.js is an event-driven architecture that makes it great for throughput and scalability. Node.js runs as a single-threaded, non-blocking, asynchronous programming paradigm, which is very memory efficient. These features make it a fast and scalable platform for web development.

The Express Framework

Although you can build web applications using Node.js and JavaScript, sometimes the tasks to build a simple web server can take many lines of code. Many frameworks to make this task simpler have been developed around the Node.js infrastructure. One such framework is called Express and is probably the most used. For complete documentation on Express, check out https://expressjs.com. Express is a framework to help you create websites and Web APIs with less code and easier-to-use features than Node.js by itself. You're free to use all the features of Node.js as well.

Why Use Express?

Express is a thin wrapper around many Node.js features and adds new features as well. Some things that Express adds that make creating web applications so easy are middleware, routing, and templating. If you've used Microsoft ASP.NET, you're probably already familiar with these concepts. Middleware is a small piece of code you write that allows you to have access to incoming requests and outgoing responses. This access allows you to look at each request and modify the response that's sent back out. Middleware is often used for exception handling as well as other generic tasks.

Routing is how a web application responds to client requests and calls the appropriate code to perform a task. For example, if you allow a user to call your Web API using http://www.example.com/api/product, this call is routed by Express to a specific JavaScript function that might return a product web page, a simple string, or it might send an array of JSON product objects back to the caller.

Express also supports the ability to build dynamic web pages using templating engines. Templating helps you build dynamic web pages by marrying data and HTML to create a final HTML page to send to the user. In ASP.NET, you might use the Razor syntax combined with C# objects to create a list of products that the user views as an HTML page on their browser. In Express, you can use many different templating engines to generate this same list of products on an HTML page. Express allows you to use the templating engine of your choice such as EJS, Pug, and Mustache.

Set Up Your Development Environment

If you've never used Node.js or Express before, I highly recommend you follow along with the steps in this article to gain proficiency with these tools. You're going to need a few tools (**Table 1**) on your computer to create the Web application in this article.

Of the tools listed in **Table 1**, Postman and Visual Studio Code are optional. Instead of Postman, you can test retrieving data from your Web API calls using a browser. There are other tools, such as SoapUI and Fiddler, that have similar functionality to Postman. Instead of Visual

Tool	Use
Node.js	The web server environment upon which you build your web applications
npm	The Node Package Manager used to publish, discover, and install Node frameworks such as Express
nodemon	A small utility, installed via npm, to monitor your project directory and automatically restart your node application when changes occur in your files
Express	A framework, installed via npm, to make it easy to build your Web API calls or a website
Postman	A tool to help you test your Web API calls
Visual Studio Code	An editor for typing in your JavaScript code and running your Web server

Table 1: A list of tools commonly used to build Web APIs using Node.js

Studio Code, any editor can be used for developing your JavaScript files. You're free to use Visual Studio 2022 if you're familiar with that. I'm using the tools listed in **Table 1** for this article because I'm used to them, and because they are probably the most used tools for developing Web APIs with Node.js.

Install Node and Node Package Manager (npm)

The first step is to see if you have Node.js and the Node Package Manager (npm) on your computer. Open a Command Prompt or open the Windows PowerShell app on your Windows computer and type in the following to see if you have node installed.

```
node -v
```

You'll either see an error, or you'll see a version number appear for node. If you have Node.js installed, you most likely have npm installed, as these two are generally downloaded and installed as one package from the https://nodejs.org website. If Node.js isn't installed on your computer, go to https://nodejs.org and download the **LTS** version (not the **Current** version). If you have a Node.js version older than version 12.x, please download the latest version to upgrade your computer.

Install Visual Studio Code

If you already use Visual Studio Code, or you already have your editor of choice, you can skip to the next section. If you wish to install Visual Studio Code, navigate to http://code.visualstudio.com in your browser and download and install Visual Studio Code. If you're using a different editor, you might have a different way to start the Web API project. Wherever in this article I say to use the terminal window to submit a command, you should use whatever is appropriate for your editor.

Install Postman

If you already have a preferred method of testing your Web APIs, skip to the next section of this article. If you wish to give Postman a try for testing your Web APIs, navigate to https://www.postman.com/pricing in your browser and click the Sign Up button under the Free version of Postman. Download and install Postman on your computer.

Create a Node.js Project Using VS Code

Open a Command Prompt, the Windows PowerShell app, or some other terminal as appropriate for your computer, and navigate to where you normally place your development projects. For this article, I'm using the folder D:\Samples to create my new Node.js project. After opening a command prompt within your development folder, create a new folder under that folder and navigate to that folder using the following two commands.

```
mkdir AdvWorksAPI
cd AdvWorksAPI
```

Open the Visual Studio Code editor in this new folder using the following command. Note, that this is the word **code**, followed by a space, followed by a period (.).

```
code .
```

From the menu system in VS Code, open a new terminal by selecting **Terminal > New Terminal**. Type in the following command in the terminal window to start building a new JavaScript project.

```
npm init
```

The terminal window asks you for some information to describe this project. If you wish to accept the default answers, simply press the Enter key after each prompt, otherwise enter the appropriate information for your project, as shown in **Figure 1**. At the end, answer **yes** to save

```
Press ^C at any time to quit.
package name: (advworksapi)
version: (1.0.0)
description: Adventure Works API
entry point: (index.js)
author: Paul D. Sheriff
license: (ISC)
About to write to D:\Samples\AdvWorksAPI\package.json:

{
  "name": "advworksapi",
  "version": "1.0.0",
  "description": "Adventure Works API",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Paul D. Sheriff",
  "license": "ISC"
}
```

Figure 1: Answer a series of questions to create a package.json file for your project.

a file called **package.json** in your new project folder. The package.json file contains meta-data about your project to help **npm** run the scripts in the project, install dependencies, and identify the initial JavaScript file used to start the application.

Install Express and Nodemon

You're now ready to install any additional utilities you want to use in your application. You're going to use the Express framework for creating the web server, so let's install that now. From within the terminal window, install the Express framework using the following command within the terminal window.

```
npm install express
```

You're also going to use the nodemon utility to automatically detect changes to any files in your project and restart your web server when appropriate. Install nodemon by using the following command within the terminal window.

```
npm install nodemon
```

Modify the package.json File

Open the **package.json** file and you should now see Express and nodemon listed under the Dependencies property. Also notice that there's a Main property that lists **index.js** as its value. This is the starting JavaScript file for the application. Because you want to use nodemon to watch for any changes in your js files, add a Start property under the Scripts property, as shown in the code snippet below.

```
"scripts": {
  "start": "nodemon index.js",
  "test": "echo \"Error: no test
    specified\" && exit 1"
},
```

Be sure to **save** the changes to the **package.json** file at this point.

Create Express Server and Get Data

Now that the configuration and main files of the web application have been created, add a new file named **index.js** file to the root folder of your project. The index.js file is the starting point for web applications. This file is analogous to the Program.cs file in .NET applications. Into the **index.js** file, add the code shown in **Listing 1**.

A Word About Modules

The code shown in **Listing 1** uses the built-in Node.js function called require() to load the express module. A module is a JavaScript file that defines some functionality to be called by your application. Where are these modules located? When you performed the **npm init** command, a folder named **node_modules** was created with the Node.js server JavaScript files and all related dependencies. When you performed the **npm install express** command, a folder named **express** was created under the **node_modules** folder. All the JavaScript files needed for the express framework were downloaded into this folder. The require() function looks into the node_modules folder and locates the folder with the name you specify in the require() function. From there, it figures out where the module named Express is located and instantiates that object.

You can think of modules (very loosely) as a class in C#. A module splits the functionality of an application into separate files. A great thing about Node.js modules is that the scope of the module is just within the one file; they're not in a global memory space, as is the case with JavaScript running in the browser.

Getting Data with Express

Immediately after loading the Express module, call the express() function to create an application object. The application object is used to route incoming HTTP requests, configure middleware, and set up the listening for requests on a specific port. Next, you see the app.get(...) that's used to map a request coming in to the JavaScript code within the app.get() function. For example, if a GET request comes in on http://localhost:3000, the code within the app.Get("/" ...) function is run. The code **res.send("10 Speed Bicycle")** returns the string **10 Speed Bicycle** to the caller. The last lines of this index.js file start the server running on port 3000 on your local computer.

Try It Out

Save the changes you made to all the files in your project. In your terminal window, type in the following command to start the Node.js server and use nodemon to run the JavaScript code in your index.js file.

```
npm start
```

After typing in this command, you should see the message **AdvWorksAPI server is running on http://local-**

Listing 1: Create an index.js file as the starting point for your application.

```
// Load the express module
const express = require('express');
// Create an instance of express
const app = express();
// Specify the port to use for this server
const port = 3000;

// GET Route
app.get('/', (req, res, next) => {
  res.send("10 Speed Bicycle");
});

// Create web server to listen
// on the specified port
let server = app.listen(port, function () {
  console.log(`AdvWorksAPI server is running
    on http://localhost:${port}.`);
});
```

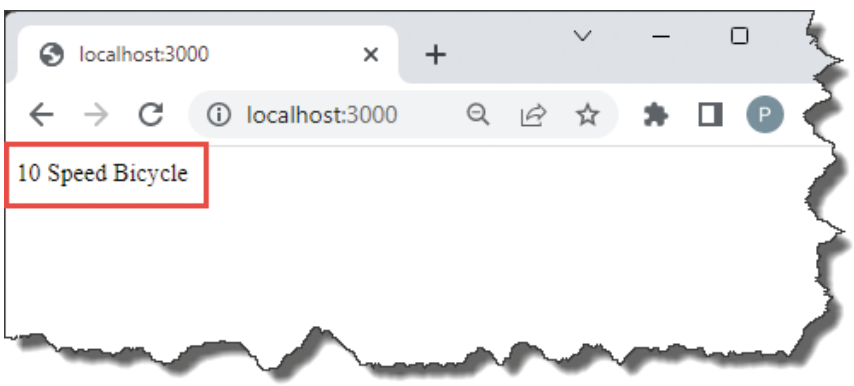


Figure 2: Call the Web API from your browser to have the data returned.

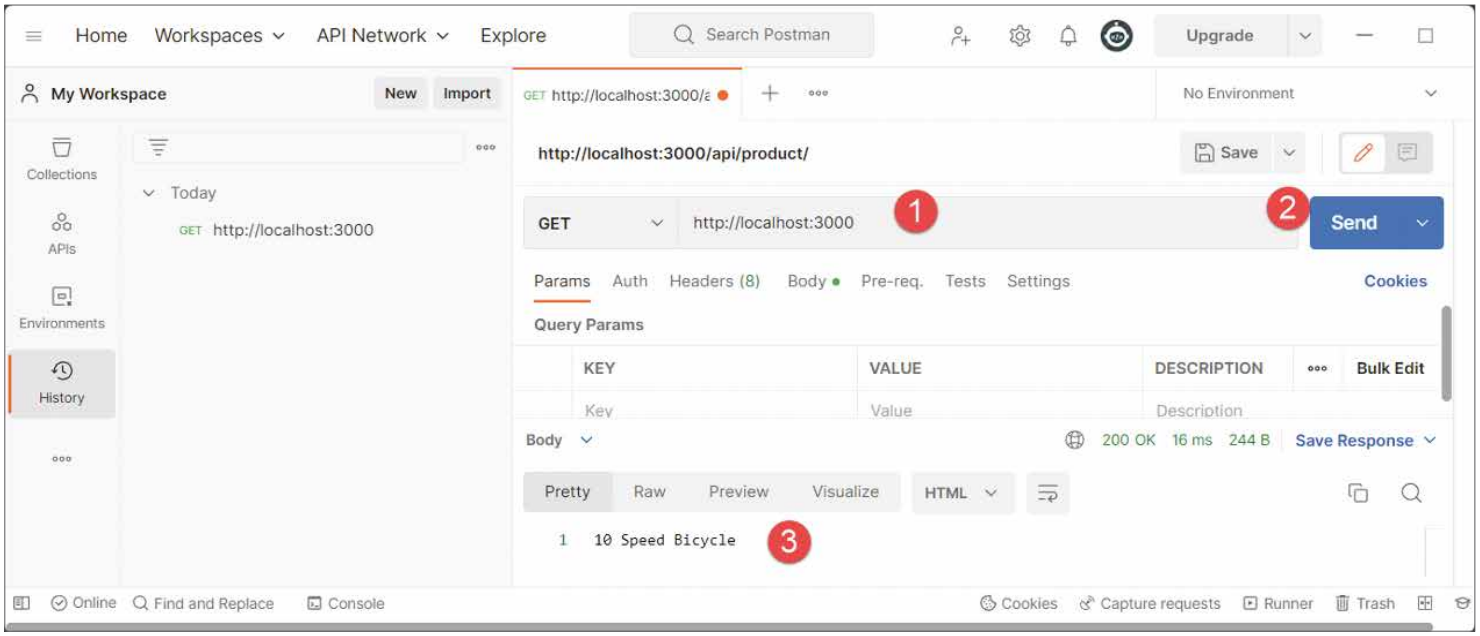


Figure 3: Use Postman to send Web API requests and to view the data returned.

host:3000. This is the message you specified within the app.listen() call. Open your browser and submit the request http://localhost:3000 via the URL line to view the string returned from this Web API call. You should see the data appear in your browser, as shown in **Figure 2**.

Using Postman

If you have Postman installed, open Postman (**Figure 3**) and in the URL line (#1), type in **http://localhost:3000**. Click the Send button (#2) to submit the call to your running Web API application and you should see the string **10 Speed Bicycle** appear in the body window (#3). Congratulations! You have just created your first Web API call using Node.js and Express.

Return an Array of Product Objects

Instead of just returning a single string from a Web API call, let's add an array of product objects to return. Open the **index.js** file and modify the app.get() function you created as your route to look like **Listing 2**.

Try It Out

Save the changes you made the index.js file. From within your browser, or from within Postman, submit the same request, http://localhost:3000. You should see the array of product objects appear. The one nice thing about using Postman is you also see the status codes appear after each call, as shown in **Figure 4**, right with the call. When using your browser, you need to go into the F12 developer tools to see the status code.

Return a Status Code and JSON

By default, all successful API calls return a status code of 200 along with the data. You may change the status code that's returned if you wish. Open the **index.js** file and modify the app.get() function to return a status code of 206 using the code shown in the following snippet:

Listing 2: Return an array of product objects from your Web API.

```
app.get('/', (req, res) => {
  // Create array of product objects
  let products = [{
      "productID": 879,
      "name": "All-Purpose Bike Stand"
    },{
      "productID": 712,
      "name": "AWC Logo Cap"
    },{
      "productID": 877,
      "name": "Bike Wash - Dissolver"
    },{
      "productID": 843,
      "name": "Cable Lock"
    },{
      "productID": 952,
      "name": "Chain"
    }
  ];
  res.send(products);
});
```

```
app.get('/', (req, res, next) => {
  // Create array of product objects
  let products = [
    // PRODUCT ARRAY HERE
  ];

  res.status(206);
  res.send(products);
});
```

Instead of using send() to return the data, you can also use the json() function using the code **res.json(products)**. When passing JSON objects or arrays, both send() and json() are equivalent. When a simple string value is passed to send(), it sets the content-type HTTP header to **text/html** whereas the json() function sets the content-type HTTP header to **application/json**. The json() function takes the input and converts it to a JSON object, then performs a JSON.stringify() on that

object before calling send(). Feel free to use whichever function you want, as your needs dictate. You're allowed to chain the status() and json(), or status() and send() functions together instead of having two separate lines of code, as shown in the following code snippet.

```
res.status(206).json(products);
```

If you want, add the **status(206)** call to your previous code and resubmit your query to see the new status code returned along with the array of product objects.

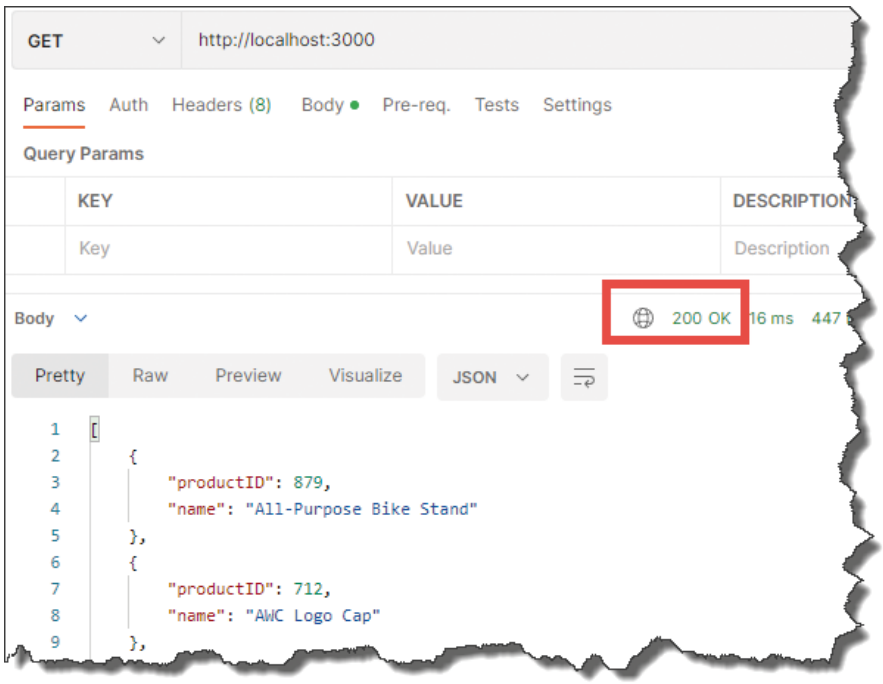


Figure 4: Postman shows the status code along with the data returned from the API call.

Listing 3: Create a module just to hold product data.

```
// Product repository object
let repo = {};

// Retrieve an array of product objects
repo.get = function () {
  return [
    {
      "productID": 879,
      "name": "All-Purpose Bike Stand"
    },
    {
      "productID": 712,
      "name": "AWC Logo Cap"
    },
    {
      "productID": 877,
      "name": "Bike Wash - Dissolver"
    },
    {
      "productID": 843,
      "name": "Cable Lock"
    },
    {
      "productID": 952,
      "name": "Chain"
    }
  ];
}

module.exports = repo;
```

Send JSON Object Instead of Just the Data

If you've ever consumed an API call from JavaScript in a browser, you know that you generally don't just get the data back, you get a JSON object containing the status code, the status text, maybe a message, and then the data. You should follow the same conventions when developing your Web API return values. Open the **index.js** file and modify the app.get() function to look like the following code snippet.

```
app.get('/', (req, res, next) => {
  // Create array of product objects
  let products = [
    // PRODUCT ARRAY HERE
  ];

  res.json({
    "status": 200,
    "statusText": "OK",
    "message": "All products retrieved.",
    "data": products
  });
});
```

Try It Out

Save the changes you made to the index.js file. Resubmit the request, http://localhost:3000, to your web server. You should see the JSON object with the **status**, **statusText**, **message**, and **data** properties appear. If you're using Postman, notice that it formats the data returned very nicely, as shown in **Figure 5**. Unfortunately, when you submit the Web API call through a browser, the JSON object returned is simply dumped unceremoniously onto the page with no formatting. This is another reason I prefer to use Postman instead of a browser for testing API calls.

Create a Module for Your Product Data

Earlier in this article, you were introduced to the concept of modules. As you can see, your index.js file is getting a little large with the hard-coded product data. Imagine if you had customer, employee, and other data that you wished to return from different API calls. You can imagine that your index.js file would get unmanageable. Let's add a module to hold the product data and functionality. Add a new folder to your project and set the name to **repositories**. In the new **repositories** folder, add a new file named **product-file.js** and put the code shown in **Listing 3** into this new file.

Now that you have the product module created, open the **index.js** file and just before the app.get() call, load the repository, and make a call to the get() function created on that repository object, as shown in **Listing 4.**

Try It Out

Save the changes to all the files in your project. Resubmit the same request using your browser or Postman to ensure that you're still getting the same array of product objects.

Read Product Data from a File

Instead of hard-coding the product array in JavaScript, create an array of product objects within a JSON file. You

can then read that file and return that JSON data to the consumer of your API call. Add a new folder to your project named **db**. Add a file in this new **db** folder named **product.json**. Place the code shown in **Listing 5** into this new file.

Open the **respositories\product-file.js** file and replace the entire contents of the file with the code shown in **Listing 6**. In this code, you first load the Node.js file system module. Create a constant named DATA_FILE to hold the name of the file where the array of product objects is located. Within the get() function, call the readFile() function on the file system object. The readFile() function is an asynchronous call to read a file. Once the file reading is done, the resolve() callback is invoked and the data read from the file is passed to resolve(). If an error occurs while reading the file, the reject() callback is invoked and an error object is passed to the reject() function.

You need to change the call to the get() function from within the app.get() function to ensure that it runs asynchronously. When calling the get() function, you need to supply two callback functions that correspond to the **reject** and **resolve** parameters accepted by the get() function. The basic syntax looks like the following code snippet:

```
repo.get(
  function(data) {
    // Code to execute when call is successful
  },
  function (err) {
    // Code to execute when an error occurs
  }
);
```

Open the **index.js** file and replace the app.get() call with the code shown in **Listing 7**. In this code, you still make a call to the repo.get() function, but you're passing two arguments as functions to it. The first argument passed to the get() function is the callback function that receives the product data if the call is successful. A JSON object is created with the normal properties and the data and then is sent to the caller of the API. The second argument passed to the get() function is the callback function that receives the error object. This error object is passed along to the next middleware code within the Express middleware chain using the next() function call. You'll learn more about Express exception handling in Part 2 of this article.

Try It Out

Save the changes to all the files in your project. Resubmit the request to the API to ensure that you're now receiving the data from the product.json file.

Get a Single Product of Data

Instead of always returning the complete list of product data, you may just want to allow a consumer of your API to pass in a product ID to return a single product object. You're dealing with a small amount of JSON objects in the product.json file, so let's just read the complete file each time and then locate the specific product object requested by the user. If you were using a MongoDB, MySQL, or a SQL Server database, you'd submit a query to retrieve a single piece of data instead of reading in the complete file.

Listing 4: Get the data from your new product repository object.

```
// Load product repository module
const repo = require(
  './repositories/product-file');

// GET Route
app.get('/', (req, res, next) => {
  // Get products from repository object
  let products = repo.get();

  res.send({
    "status": 200,
    "statusText": "OK",
    "message": "All products retrieved.",
    "data": products
  });
});
```

Figure 5: A nice feature of Postman is that it formats the JSON object that's returned.

Open the **repositories\product-file.js** file and add a new function to the repo object named getById(). Place this code just before the **module.exports = repo** line at the end of the file. To this function, pass the product ID along with your two callback functions for success or failure, as shown in **Listing 8**. After the file is read successfully, parse the JSON read into an array of product objects. Ap-

Listing 5: Create a file full of product objects.

```
[
{
  "productID": 706,
  "name": "HL Road Frame - Red, 58",
  "productNumber": "FR-R92R-58",
  "color": "Red",
  "standardCost": 1059.3100,
  "listPrice": 1500.0000
},
{
  "productID": 707,
  "name": "Sport-100 Helmet, Red",
  "productNumber": "HL-U509-R",
  "color": "Red",
  "standardCost": 13.0800,
  "listPrice": 34.9900
},
{
  "productID": 708,
  "name": "Sport-100 Helmet (Wireless), Black",
  "productNumber": "HL-U509",
  "color": "Black",
  "standardCost": 13.0863,
  "listPrice": 79.9900
},
{
  "productID": 709,
  "name": "Mountain Bike Socks, M",
  "productNumber": "SO-B909-M",
  "color": "White",
  "standardCost": 3.3963,
  "listPrice": 9.5000
},
{
  "productID": 710,
  "name": "Mountain Bike Socks, L",
  "productNumber": "SO-B909-L",
  "color": "White",
  "standardCost": 3.3963,
  "listPrice": 9.5000
},
{
  "productID": 711,
  "name": "Sport-100 Helmet, Blue",
  "productNumber": "HL-U509-B",
  "color": "Blue",
  "standardCost": 13.0863,
  "listPrice": 34.9900
},
{
  "productID": 712,
  "name": "AWC Logo Cap",
  "productNumber": "CA-1098",
  "color": "Multi",
  "standardCost": 6.9200,
  "listPrice": 8.9900
},
{
  "productID": 713,
  "name": "Long-Sleeve Logo Jersey, S",
  "productNumber": "LJ-0192-S",
  "color": "Multi",
  "standardCost": 38.4923,
  "listPrice": 49.9900
},
{
  "productID": 714,
  "name": "Long-Sleeve Logo Jersey, M",
  "productNumber": "LJ-0192-M",
  "color": "Multi",
  "standardCost": 38.4923,
  "listPrice": 49.9900
},
{
  "productID": 715,
  "name": "Long-Sleeve Logo Jersey, L",
  "productNumber": "LJ-0192-L",
  "color": "Multi",
  "standardCost": 38.4923,
  "listPrice": 49.9900
}
]
```

ply the find() method to the array to locate the product where the **productID** property matches the **id** parameter passed in. If the product is found, the product object is passed to the resolve callback function; otherwise the value **undefined** is passed.

Listing 6: Modify your product repository to read product data from a file.

```
// Load the node file system module
const fs = require('fs');

// Path/file name to the product data
const DATA_FILE = './db/product.json';

// Product repository object
let repo = exports = module.exports = {};

// Retrieve an array of product objects
repo.get = function (resolve, reject) {
  // Read from the file
  fs.readFile(DATA_FILE, function (err, data) {
    if (err) {
      // ERROR: invoke reject() callback
      reject(err);
    }
    else {
      // SUCCESS: Convert data to JSON
      let products = JSON.parse(data);
      // Invoke resolve() callback
      resolve(products);
    }
  });
}
```

Open the **index.js** file and below the call to the app.get("/", ...) route you already have there, add the new route shown in **Listing 9**. To call this new route, make a request to the web server followed by the product ID, such as http://localhost:3000/**711**. The 711 at the end of the URL is the product ID you wish to locate within the product array read from the product.json file. You tell the route you want a parameter passed in by adding the colon (:) and the variable name, **id**, in the first parameter to the app.get() call.

Retrieve the product ID using the **req.params.id** property. The **id** at the end of the **req.params** property is the same name as the parameter you created in the app.get("/:**id**"). That product ID is passed to the repo.getById() function. If no errors occur in the repo.getById() call, the first callback function is invoked and a check is made to see if the **data** parameter contains a JSON object. If there is data, the standard JSON object is built with this product object and sent to the caller. If the **data** parameter is **undefined,** the product ID wasn't found in the array and a 404 response object is sent to the caller.

Try It Out

Save the changes made to all the files in your project and submit a request via your browser or Postman to the route http://localhost:3000/711. If you did everything correctly, you should get a valid product object returned that looks like the following:

```
{
  "status": 200,
  "statusText": "OK",
  "message": "Single product retrieved.",
  "data": {
    "productID": 711,
    "name": "Sport-100 Helmet, Blue",
    "productNumber": "HL-U509-B",
    "color": "Blue",
    "standardCost": 13.0863,
    "listPrice": 34.99
  }
}
```

Now try entering an invalid product ID by submitting the request http://localhost:3000/1. The number one is an invalid product ID, so you should get a 404 response object, as shown in the following code snippet.

```
{
  "status": 404
  "statusText": "Not Found",
  "message": "The product '1' could not be found.",
  "error": {
    "code": "NOT_FOUND"
    "message": "The product '1' could not be found."
  }
}
```

Searching for Product Data

Instead of searching for a product only on the primary key (product ID) field, you might want to search on one or more of the other properties on the product object. For example, you might want to see if the product name contains a certain letter or letters. Or you might want to check whether the list price is greater than a specific value passed in. To accomplish this, submit a request to a search route passing in some URL arguments like any of the following requests shown below.

```
/search?name=Sport

/search?listPrice=100
/search?name=Sport&listPrice=100
```

You can then grab these URL parameters from the **request** object and create your own JSON object to submit to a search() function you're going to create in your product repository object.

```
let search = {
  "name": req.query.name,
  "listPrice": req.query.listPrice
};
```

Create a Search Function

Open the **repositories\product-file.js** file and create a search() function (**Listing 10**) below the getById() function you added earlier. The search() function accepts the search object shown above and uses the properties to filter the records from the array of products and only return those products that match the criteria specified in the search object.

For the purposes of this article, I'm just using the JavaScript filter() function to perform the searching through the

Listing 7: Pass in two callback functions to receive good data or an error object.

```
// GET Route
app.get('/', (req, res, next) => {
  repo.get(function (data) {
    // SUCCESS: Data received
    res.send({
      "status": 200,
      "statusText": "OK",
      "message": "All products retrieved.",
      "data": data
    });
  }, function (err) {
    // ERROR: pass error along to
    // the 'next' middleware
    next(err);
  });
});
```

Listing 8: Add a function to retrieve a single product object.

```
// Retrieve a single product object
repo.getById = function (id, resolve, reject) {
  fs.readFile(DATA_FILE, function (err, data) {
    if (err) {
      // ERROR: invoke reject() callback
      reject(err);
    }
    else {
      // SUCCESS: Convert data to JSON
      let products = JSON.parse(data);
      // Find the row by productID
      let product = products.find(
        row => row.productID == id);
      // Invoke resolve() callback
      // product is 'undefined' if not found
      resolve(product);
    }
  });
}
```

Listing 9: Add a route to retrieve a single product object.

```
// GET /id Route
app.get('/:id', (req, res, next) => {
  repo.getById(req.params.id, function (data) {
    // SUCCESS: Data received
    if (data) {
      // Send product to caller
      res.send({
        "status": 200,
        "statusText": "OK",
        "message": "Single product retrieved.",
        "data": data
      });
    }
    else {
      // Product not found
      let msg = `The product '${req.params.id}'
                 could not be found.`;
      res.status(404).send({
        "status": 404,
        "statusText": "Not Found",
        "message": msg,
        "error": {
          "code": "NOT_FOUND",
          "message": msg
        }
      });
    }
  }, function(err) {
    // ERROR: pass error along to
    // the 'next' middleware
    next(err);
  });
});
```

Listing 10: The search function can return one or more rows.

```
// Search for one or many products
repo.search = function (search,
  resolve, reject) {
  if (search) {
    fs.readFile(DATA_FILE,function (err, data) {
      if (err) {
        // ERROR: invoke reject() callback
        reject(err);
      }
      else {
        // SUCCESS: Convert data to JSON
        let products = JSON.parse(data);
        // Perform the search
        products = products.filter(
          row => (search.name ?
            row.name.toLowerCase()
              .indexOf(search.name.toLowerCase())
                >= 0 : true) &&
            (search.listPrice ?
             parseFloat(row.listPrice) >
             parseFloat(search.listPrice)
               : true));
        // Invoke resolve() callback
        // Empty array if no records match
        resolve(products);
      }
    });
  }
}
```

product array. If you were using a SQL or a NoSQL database, you'd use their query functionality. The filter() method is applied to the array and passed a lambda expression to filter the rows. In the expression shown above, the **name** property in the product object and the **name** object in the search object are converted to lower case letters before seeing if one is contained in the other. Convert the **listPrice** properties in both the search object and product object to floating point types before performing a **greater than** comparison. If both these conditions evaluate to true, that row is returned into the final resulting products array.

Now that you have the search() function created in the product repository, it's time to add a route on your web server to call that function. Open the **index.js** file and add the code shown in **Listing 11** before the call to the app.get('/:id',...). You need to put this search route prior to the route that retrieves a single product, or it may try to map either the name or list price to the **id** parameter in that route.

Try It Out

Save all the changes made to the files in your project and submit the request http://localhost:3000/**search?name=Sport** via your browser or Postman. This request should return a few records where the product name contains the word "Sport". Submit the request http://localhost:3000/**search?name=Sport&listPrice=50** to look for both a **name** that contains the value "Sport" and the **listPrice** is greater than or equal to 50. From this request, you should have a single product object returned. Finally, submit the **request http://localhost:3000/search?name=aaa** to check your error handling. From this request, you should get no records and thus a 404 status code and object is returned from the query.

Use Router Object and Add API Prefix

If you've created Web API calls before, you know that most developers tend to like to put the prefix **api** in front of every route. This helps you distinguish between a route that returns some HTML versus a route that just returns data. To accomplish this with Express, you need to use the **router** object instead of the **app** object. Open the **index.js** file and add the code shown in the snippet below just before the declaration of the **const post = 3000**.

```
// Create an instance of a Router
const router = express.Router();
```

Listing 11: The search route builds a search object to locate rows of product data.

```
// GET /search Route
app.get('/search', (req, res, next) => {
  // Create search object with
  // parameters from query line
  let search = {
    "name": req.query.name,
    "listPrice": req.query.listPrice
  };
  if (search.name || search.listPrice) {
    repo.search(search, function (data) {
      // SUCCESS: Data received
      if (data && data.length > 0) {
        // Send array of products to caller
        res.send({
          "status": 200,
          "statusText": "OK",
          "message": "Search was successful.",
          "data": data
        });
      }
      else {
        // No products matched search
        let msg = `The search for
          '${JSON.stringify(search)}' was
           not successful.`;
        res.status(404).send({
          "status": 404,
          "statusText": "Not Found",
          "message": msg,
          "error": {
            "code": "NOT_FOUND",
            "message": msg
          }
        });
      }
    }, function (err) {
      // ERROR: pass error along to
      // the 'next' middleware
      next(err);
    });
  }
  else {
    // No search parameters passed
    let msg = `No search parameters passed in.`;
    res.status(400).send({
      "status": 400,
      "statusText": "Bad Request",
      "message": msg,
      "error": {
        "code": "BAD_REQUEST",
        "message": msg
      }
    });
  }
});
```

Next, modify all of the route calls you created that start with **app.**get() to **router.**get() as shown in the snippet below:

```
router.get('/', (req, res, next) => {
  // REST OF THE CODE HERE
}
router.get('/search', (req, res, next) => {
  // REST OF THE CODE HERE
}
router.get('/:id', (req, res, next) => {
  // REST OF THE CODE HERE
}
```

Scroll down toward the end of the index.js file and immediately before the call to the app.listen() function, add the following code to setup the prefix for all routes included within the **router** object.

```
// Configure router so all routes
// are prefixed with /api
app.use('/api', router);
```

Try It Out

Save all the changes made to the files in your project and submit the request http://localhost:3000 via your browser or Postman. After submitting this request, you should get a 404 status code and some HTML that says "Cannot GET /". This means that there's no route defined on the root of your web application. Change the request to include the **/api** suffix to the web application root: http://localhost:3000/api. From this request, you should once again retrieve the array of product objects.

Modularize Your API Calls

As you can see, the index.js file is getting quite large. You can imagine that if you add more routes to handle inserting, update, and deleting products, this file is going to get huge. Think about adding on more routes for customers, employees, and other data in your web application. The result would be a maintenance nightmare. Let's move all product routes into a module that can then be loaded into the index.js file using the require() function. Add a new folder to your project named **routes**. Within the **routes** folder, add a new file named **product.js**. Add the code shown in the code snippet below into this new file.

```
// Create an instance of a Router
const router = require('express').Router();

// Load product repository module
const repo =
  require('../repositories/product-file');

// MOVE YOUR ROUTES HERE FROM INDEX.JS

module.exports = router;
```

Open the **index.js** file and cut out all the router.get() routes you created and paste them into the **routes\product.js** file at the location where the comment **// MOVE YOUR ROUTES HERE FROM INDEX.JS** is located. You can remove this comment after you've pasted in all the routes. You can also remove the following code from the index.js file:

```
// Load product repository module
const repo = require(
  './repositories/product-file');
```

Open your **index.js** file and, from where you cut out all the routes, add the following code to register the routes that are now created within the **routes\product.js** file.

```
// Mount routes from modules
router.use('/product', require('./routes/product'));
```

This line of code adds another prefix, **/product**, to all the routes contained within the product module. You want to add this additional prefix because if you add customer routes, you'll want to use **/api/customer** for all those routes. And, if you add employee routes, you'll want to use **/api/employee**, and so on. The **index.js** file should now look exactly like what you see in **Listing 12**. As you can see from the code in **Listing 12**, the index.js file is now significantly smaller. If you add more routes, say for customers or employees, you only add one additional line in the index.js file for each route you add. All of the route definitions are placed into the corresponding files within the **routes** folder.

Try It Out

Save all the changes made to the files in your project and submit the request http://localhost:3000/api/product via your browser or Postman. Check out a couple of the other routes by submitting the request http://localhost:3000/api/product/711 to retrieve a single product object. Now try the search functionality by submitting the request http://localhost:3000/api/product/search?name=Sport. All three of the above requests should return the same data as they did when they were defined within the index.js. The only difference is the prefix **/api/product** and the fact that all the routes are declared within a separate file in your web application.

Inserting Data (POST)

Chances are that you're not going to write an API that simply returns data. You most likely will need to modify data as well. To insert data, you use the POST verb and

Getting the Sample Code

You can download the sample code for this article by visiting www.CODEMag.com under the issue and article, or by visiting www.pdsa.com/downloads. Select "Articles" from the Category drop-down. Then select "Building Web APIs Using Node.js and Express: Part 1" from the Item drop-down.

Listing 12: Your index.js file is now significantly smaller and easier to maintain.

```
// Load the express module
const express = require('express');
// Create an instance of express
const app = express();
// Create an instance of a Router
const router = express.Router();
// Specify the port to use for this server
const port = 3000;

// Mount routes from modules
router.use('/product',
  require('./routes/product'));

// Configure router so all routes
// are prefixed with /api
app.use('/api', router);

// Create web server to listen
// on the specified port
let server = app.listen(port, function () {
  console.log(`AdvWorksAPI server is running
    on http://localhost:${port}.`);
});
```

create a route that maps to that request. Open the **repositories\product-file.js** file and add a new function named insert(), as shown in **Listing 13**.

Listing 13: Create an insert method to map to the POST route.

```
// Insert a new product object
repo.insert = function (newData,
  resolve, reject) {
  fs.readFile(DATA_FILE, function (err, data) {
    if (err) {
      // ERROR: Invoke reject() callback
      reject(err);
    }
    else {
      // SUCCESS: convert data to JSON
      let products = JSON.parse(data);
      // Add new product to array
      products.push(newData);
      // Stringify the product array
      // Save array to the file
      fs.writeFile(DATA_FILE,
        JSON.stringify(products),
        function (err) {
        if (err) {
          // ERROR: Invoke reject() callback
          reject(err);
        }
        else {
          // SUCCESS: Invoke resolve() callback
          resolve(newData);
        }
      });
    }
  });
}
```

Listing 14: Add a router.post to map the insert() function to this route.

```
// POST Route
router.post('/', function (req, res, next) {
  // Pass in the Body from request
  repo.insert(req.body, function(data) {
    // SUCCESS: Return status of 201 Created
    res.status(201).send({
      "status": 201,
      "statusText": "Created",
      "message": "New Product Added.",
      "data": data
    });
  }, function(err) {
    // ERROR: pass error along to
    // the 'next' middleware
    next(err);
  });
});
```

In this function, you pass in a new product object to the parameter **newData**. The complete product.json file is read in and converted to a JSON array. Use the push() method to add the new product object to the end of the array. Stringify the JSON array and write the new array to the product.json file. Please remember that this code is for learning purposes only. You wouldn't want to use this rudimentary file I/O in a production application. Instead, you'd use a real database that can handle multiple requests at the same time.

Add JSON Middleware

When posting data to a route hosted by Express, some middleware needs to be added to allow Express to take the string version of the JSON data and convert it to a real JSON object. Open the **index.js** file and add the following code after the **const router = express.Router()** call to add this middleware:

```
// Configure JSON parsing
// in body of request object
app.use(express.json());
```

Open the **routes\product.js** file and scroll down toward the bottom of the file. Just before the call to **module.exports = router**, add a new router.post() method, as shown in **Listing 14**. This code extracts the JSON object from the **req.body** property and passes it to the insert() function you just created.

Try It Out

Save the changes made to all the files in your project. Open Postman or another API test tool, and submit a POST with the JSON object contained in the body of the post back. Referring to **Figure 6,** follow the steps below to submit a new product object to be inserted into the product.json file.

- Select POST from the drop-down and fill in the URL to call your Web API.
- Click on the Body tab.
- Change the first drop-down to Raw.
- Change the second drop-down to JSON.
- In the Body text box add the following JSON object.

```
{
  "productID": 986,
  "name": "Mountain-500 Silver, 44",
  "productNumber": "BK-M18S-44",
```

Listing 15: Create an update method to map to the PUT route.

```
// Update an existing product object
repo.update = function (changedData, id,
  resolve, reject) {
  fs.readFile(DATA_FILE, function (err, data) {
    if (err) {
      // ERROR: Invoke reject() callback
      reject(err);
    }
    else {
      // SUCCESS: Convert to JSON
      let products = JSON.parse(data);
      // Find the product to update
      let product = products.find(
        row => row.productID == id);
      if (product) {
        // Move changed data into corresponding
        // properties of the existing object
        Object.assign(product, changedData);
        // Stringify the product array
        // Save array to the file
        fs.writeFile(DATA_FILE,
          JSON.stringify(products),
          function (err) {
          if (err) {
            // ERROR: Invoke reject() callback
            reject(err);
          }
          else {
            // SUCCESS: Invoke resolve() callback
            resolve(product);
          }
        });
      }
    }
  });
}
```

Listing 16: Add a router.put() route to map to the update() function.

```
// PUT Route
router.put('/:id', function (req, res, next) {
  // Does product to update exist?
  repo.getById(req.params.id, function (data) {
    // SUCCESS: Product is found
    if (data) {
      // Pass in Body from request
      repo.update(req.body, req.params.id,
        function (data) {
        // SUCCESS: Return 200 OK
        res.send({
          "status": 200,
          "statusText": "OK",
          "message": `Product '${req.params.id}'
                      updated.`,
          "data": data
        });
      });
    }
    else {
      // Product not found
      let msg = `The product '${req.params.id}'
                 could not be found.`;
      res.status(404).send({
        "status": 404,
        "statusText": "Not Found",
        "message": msg,
        "error": {
          "code": "NOT_FOUND",
          "message": msg
        }
      });
    }
  }, function(err) {
    // ERROR: pass error along to
    // the 'next' middleware
    next(err);
  });
});
```

```
  "color": "Silver",
  "standardCost": 308.2179,
  "listPrice": 564.9900
}
```

Click the **Send** button to call the POST route you created. When the response returns, you should see the data in the lower window of Postman (#6). Go back to VS Code and open the **db\product.json** file. At the end of the file, you should see the product you submitted from Postman.

Updating Data (PUT method)

Let's now add the functionality to update a product in your product.json file. Open the **repositories\product-file.js** file and add a new function named update(), as shown in **Listing 15**. This function reads all the text from the product.json file and converts the data to a JSON array. Next, locate the row in the array to update using the find() method and search for the product ID passed into this function. If the product is found, use the Object.assign() method to merge the new data with the data in the existing product object. Stringify the product array and write all the data back to the product.json file.

Open the **routes\product.js** file and scroll down toward the bottom of the file. Just before the call to **module.exports = router**, add a new router.put() function, as shown in **Listing 16**.

Try It Out

Save the changes made to all the files in your project. Open Postman or another API test tool, and submit a PUT with the JSON object contained in the body of the post back. Just like you did when inserting, select **PUT** from the drop-down. Within the Body tab, add the following JSON object. Because you're only making changes to a couple of the properties of the product object, Object.assign() only updates those properties it finds in this JSON object and applies them to the product object read from the file.

```
{
  "productID": 986,
  "name": "Mountain-500 Gray, 44",
  "listPrice": 699.00
}
```

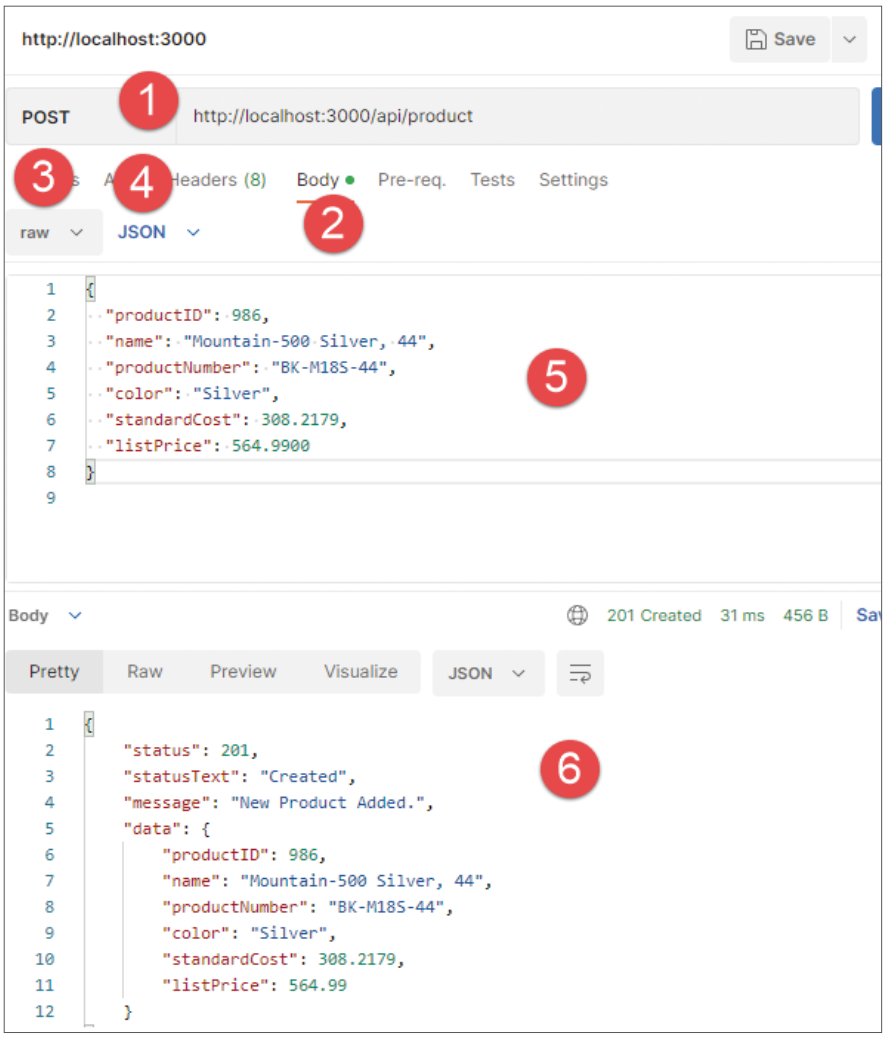


Figure 6: Use Postman to submit a POST to the Web API.

Modify the URL in Postman to add the product ID you wish to update at the end of the request http://localhost:3000/api/product/986. Click the **Send** button to submit the PUT request. Go back to VS Code, open the **db\product.json** file, and format the file. Scroll down to the bottom and view the changes made to product 986.

Listing 17: Create a delete function to map to the DELETE route.

```
// Delete an existing product object
repo.delete = function (id, resolve, reject) {
  fs.readFile(DATA_FILE, function (err, data) {
    if (err) {
      // ERROR: Invoke reject() callback
      reject(err);
    }
    else {
      // SUCCESS: Convert data to JSON
      let products = JSON.parse(data);
      // Find product to delete
      let index = products.findIndex(
        row => row.productID == id);
      if (index != -1) {
        // Remove row from array
        products.splice(index, 1);
        fs.writeFile(DATA_FILE,
          JSON.stringify(products),
          function (err) {
          if (err) {
            // ERROR: Invoke reject() callback
            reject(err);
          }
          else {
            // SUCCESS: Invoke resolve() callback
            resolve(index);
          }
        });
      }
    }
  });
}
```

Listing 18: Add a router.delete() route to map to the delete() function.

```
// DELETE Route
router.delete('/:id',
  function (req, res, next) {
  // Does product to delete exist?
  repo.getById(req.params.id, function (data) {
    // SUCCESS: Product is found
    if (data) {
      // Pass in 'id' from request
      repo.delete(req.params.id,
       function (data) {
        // SUCCESS: Return 204 No Content
        res.status(204).send();
      });
    }
    else {
      // Product not found
      let msg = `The product '${req.params.id}'
                 could not be found.`
      res.status(404).send({
        "status": 404,
        "statusText": "Not Found",
        "message": msg,
        "error": {
          "code": "NOT_FOUND",
          "message": msg
        }
      });
    }
  }, function(err) {
    // ERROR: pass error along to
    // the 'next' middleware
    next(err);
  });
});
```

Return a 404

If you want, try out the 404 Not Found error by appending an invalid product ID to the URL submit the request http://localhost:3000/api/1.

Delete Data

Besides **POST** and **PUT** routes, add the ability to delete a product by adding a **DELETE** route as well. Open the **repositories\product-file.js** file and add a new function named delete(), as shown in **Listing 17**. The delete() function reads in the complete product file and converts it to a JSON object array. Use the findIndex() method to locate the product ID that matches the ID passed into the delete() function. If found, a number greater than minus one (-1) is returned. This is the index number in the array where the product is located. Use the splice() method on the array to remove that row from the array. Stringify the array and write the new array of products back to the product.json file.

Open the **routes\product.js** file and scroll down towards the bottom of the file. Just before the call to **module.exports = router**, add a new router.delete() function, as shown in **Listing 18**.

Try It Out

Save the changes made to all the files in your project. Open Postman or another API test tool, and submit a DELETE with the product ID on the URL line http://localhost:3000/api/product/986. Click the **Send** button and you should receive a 204 No Content status code back from the request. There will be no content in the response body area; you'll simply see the 204 status code.

Summary

In this article, you learned to build a Node.js and Express project for hosting Web API calls. You created a set of routes to read, search, add, edit, and delete product data in a JSON file. A couple of reusable modules were built to help you compartmentalize your application and keep any one file from becoming too large. In the next article, you'll learn to read configuration settings from a JSON file, handle exceptions, and add the ability to interact with a SQL Server database.

Paul D. Sheriff

Using Moq: A Simple Guide to Mocking for .NET

How isolated are your tests and are they truly unit tests? Perhaps you've fallen into the pitfall of having several God Objects within your code and now you just have integration tests. It might be time to start using Moq, the easy-to-implement mocking library, ubiquitously used within Microsoft Docs for .NET. Moq can help ensure that your units under test have the same state

as your application, and even make it easier to enforce shared test structure, such as through the Arrange, Act, Assert (AAA) model. It can help you write tests that are in complete control of the functionality and expectations, where the test adheres to the design pattern of dependency injection and the principle of inversion of control (IOC). Stop focusing your tests on the set up of your context, and instead come with me and learn about how to make your testing code even better.

What are Mocking and Moq?

The process of mocking, in software testing terms, involves creating dummy objects based upon your application's real objects and then using those dummy objects to simulate the behavior of your real objects in various conditions. It allows you to mimic dependencies that your code may have, allowing you to control their behavior in order to better test the functionality of the object you are testing. Mocking also allows you to ensure that a specific method was run with the expected parameters, or enforce that certain methods are only invoked the number of times you are expecting. These mocks can be as simple as faking a virtual method or as complex as ensuring that a call to your database was executed as expected. For this article, I'll be keeping things simple and the examples to the point.

I'm sure you're already thinking of specific situations in your tests where this could be incredibly helpful. Allow me to introduce you to the .NET-flavored mocking library Moq (sometimes also referred to as MOQ, but given that the acronym for MOQ has nothing to do with code mocking, it's generally written to use the initial capped version Moq). This library is freely available on GitHub (https://github.com/moq/moq4) and NuGet (https://www.nuget.org/packages/Moq), fully featured so there's no light version, and makes full use of .NET LINQ expression trees and lambda expressions. Unlike some other mocking library alternatives, Moq is type-safe and enforces that it's setting up expectations on your exact methods, and, where possible, not only relying on strings. When using Moq, you initialize the Mock object with your given interface (interfaces being the primary object you should be mocking) or class (which can be used, too, but results are not as good as for an interface), and then are ready to start mocking the behavior of your code.

Why Should I Use Moq?

There is no doubt that using Moq will help you create better software, particularly at the Enterprise level. Moq's fluent interface allows you to specify the expected behavior of your mock objects in a simple and readable way, which sets it apart from other alternatives, all while ensuring that you're developing loosely coupled software. By loose coupling, I am, of course, referring to using the dependency injection design pattern, so that you don't end up with classes that are dependent on each other. Moq works at its best when used in conjunction with dependency injection, and through its use, it can help to ensure that you don't end up with units of functionality that are each entirely responsible for creation and management of any number of instances. Instead, dependencies are injected into the code, so that a given class or method can focus on doing the functionality for which it was created. Arranging your code using dependency injection results in more modular code that is easier to test, easier to read, and a breeze to extend upon.

To tackle this from the direction of **why** you should use Moq specifically, when you could just create your own dummy objects, I provide this response (which is true for any library you might make use of for any coding language): to make your life easier. Why reinvent the wheel, as it were, and add code to your solution that won't be published as part of your application? Take full advantage of Moq, this free-to-use tool, and elevate your unit tests.

Key Areas of Unit Testing that Moq Can Help With

Like many other developers, I like to structure my unit tests using the AAA model. I feel it keeps things simple by dividing each test into functionality segments, so that it's known what to expect from each, which is useful for when new developers join the code. Fortunately, Moq can help you with each of these sections during the set up of your objects themselves, through to setting expectations on the methods within your mocked object. At the point that you need to access your mocked object, or the mock itself from a mock created instance, Moq can help there, too. This is true, right through to verifying that your object worked as expected and leads to expected assertions..

How to Use Moq

Using the AAA unit testing structure, along with some simple examples, I'll now demonstrate how Moq can be used to create better tests. I've provided a simple example that will be consistently used all the way through so you can notice functionality all the way through development. Although not all of Moq's features will be demonstrated, I cover the most important parts, and cover anything else of note further down in the **Other Moq Functionality** section.

Elliot Moule
emoule@eps-software.com

Formerly a primary school teacher, Elliot has been working in IT since 2013. He taught himself programming using Visual Basic (VB6) Forms and a textbook, and created tools to help with his job. Soon after, he enrolled in a Games Programming degree to tackle mathematically interesting problems. Halfway through this degree, Elliot was awarded an internship opportunity with the private company nsquared, which became permanent position. Elliot now works as part of the Australian division of CODE, trailblazing opportunities there and contributing heavily to various projects worldwide.

Elliot enjoys travelling, writing, reading, video games, tasting wine and coffee, and a good dose of John Denver.

Installing and Using Moq

To start you off with the most basic of parts, you'll first need to add Moq to your project. As in **Figure 1**, with your project open, navigate to the **Tools** menu of Visual Studio, hover over **NuGet Package Manager**, and then select **Package Manager Console**.

With Package Manager Console open, click to ensure that your **Package source** is **nuget.org**, and that your **Default project** is the project where you wish to add Moq. In **Figure 2,** this is the **ExampleProject.Library** project. You're now free to install Moq. Simply click the command line interface so that the caret appears alongside **PM>,** type **Install-Package Moq -Version 4.18.4**, and press **Enter**. The package will be retrieved and installed, and you will likely see a bunch of text within the Package Manager Console. When the **PM>** shows once more, the installation has completed.

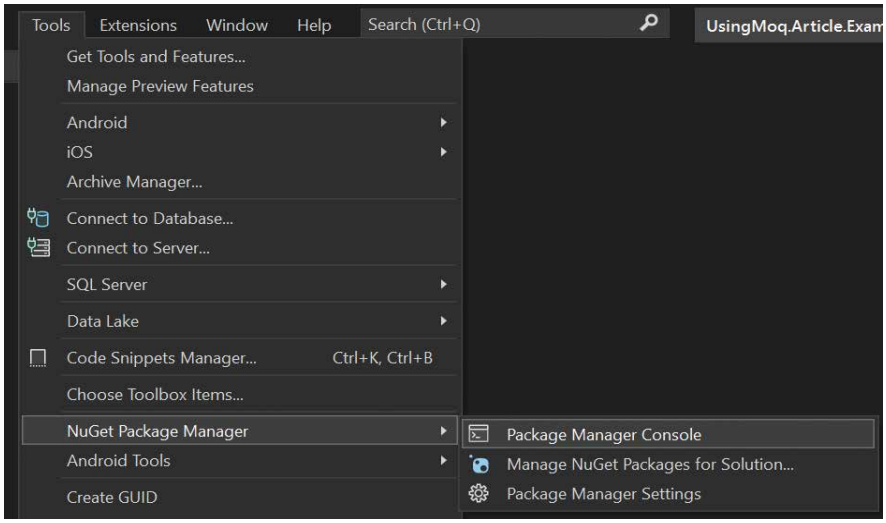


Figure 1: Navigate to NuGet Package Manager, open the Tools menu, and then open the NuGet Package Manager option followed by Package Manager Console.

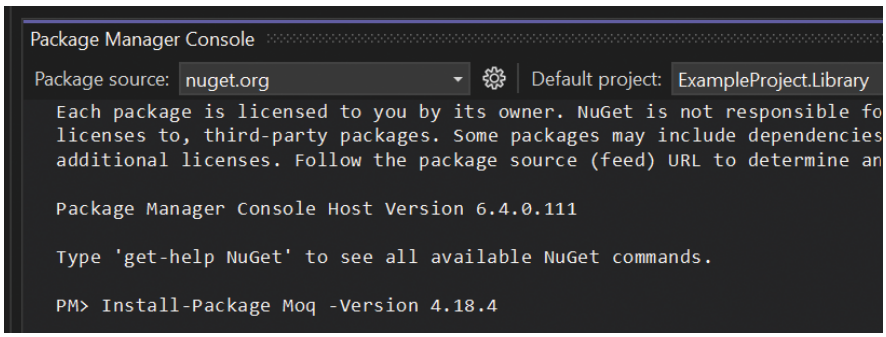


Figure 2: The Package Manager Console, where you can install NuGet packages

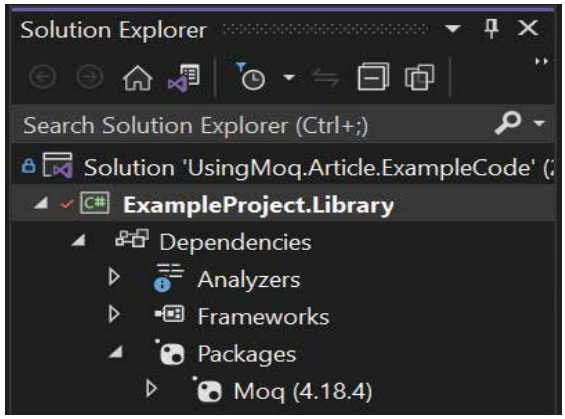


Figure 3: Moq can now be seen installed within your selected project.

You can verify that Moq installed correctly by checking your **Solution Explorer**, opening your **Dependencies** list, and then opening **Packages**, as demonstrated in **Figure 3**.

You've successfully installed Moq and you're now ready to graduate to making use of it. The first step here is implementing the Moq Using statement. When you're mocking a class with a protected method or property—which is discussed at further length later in this article—you'll find that you also need to make use of the **Protected** Using statement. You can see both implemented in the code snippet below.

```
using Moq;
using Moq.Protected;
```

You can add these statements to either the global Using file or at the top of your testing file, whatever is most convenient for you. Note that the latter Protected Using statement must be provided as a declarative, and you can't provide it explicitly in-line because the library is inclusive of extension methods. It's time to dive into making use of the Moq library itself.

Arrange

Consider the next code snippet, which depicts a simple method for determining an employee's pay rate depending on the day of the week.

```
public decimal GetPayRate(decimal baseRate)
{
   return DateTime.Now.DayOfWeek ==
            DayOfWeek.Sunday ?
            baseRate * 1.25m :
            baseRate;
}
```

Usually, methods like this have to go untested or need to be changed so that a test can be created. The issue can be seen when looking at the test below, wherein the test is likely to fail because a different result will be produced depending on which day of the week it is. Creating tests with dependencies on static references, such as the **DateTime.Now** property, results in a test that doesn't have complete control of its context.

```
public void
GetPayRate_IsSunday_ReturnsHigherRate_Snippet2()
{
    // Arrange
    var rateCalculator = new RateCalculator();

    // Act
    var actual = rateCalculator.GetPayRate(10.00m);

    // Assert
    Assert.That(actual, Is.EqualTo(12.5m));
}
```

Updating this to use abstraction, you can separate the fetching of the day of the week out to an interface and take greater control of the employee pay rate method, like the below code snippet. Through doing this, the interface can then be injected into the method, removing a dependency. This is an excellent modification that puts inversion of control on full display, as the method is now agnostic to its environment and context and will now singularly rely on what is passed into it.

```
public interface IDateTimeProvider
{
    DayOfWeek DayOfWeek();
}

public decimal GetPayRate(decimal baseRate,
    IDateTimeProvider dateTimeProvider)
{
    return dateTimeProvider.DayOfWeek() ==
           DayOfWeek.Sunday ?
           baseRate * 1.25m :
           baseRate;
}
```

At this point, you'd expect to create a dummy class that makes use of the interface, so to override it and put back your result. Instead of that though, cut out that arbitrary code, and make use of Moq instead.

```
public void
GetPayRate_IsSunday_ReturnsHigherRate_Snippet4()
{
    // Arrange
    var rateCalculator = new RateCalculator();
    var dateTimeProviderMock =
        new Mock<IDateTimeProvider>();
    dateTimeProviderMock.Setup(m => m.DayOfWeek())
                        .Returns(DayOfWeek.Sunday);

    // Act
    var actual = rateCalculator.GetPayRate(10.00m,
        dateTimeProviderMock.Object);

    // Assert
    Assert.That(actual, Is.EqualTo(12.5m));
}
```

As in the code snippet above, you initialize the mocked object by passing in the interface to be mocked as part of the constructor. Moq dynamically creates a class of its own based upon the **IDateTimeProvider** interface, and fills in any implementations the interface may have.

You can then specify the **Setup** extension, which allows you to provide an expectation for when the **DayOfWeek** method is called, effectively overriding the method. As part of the setup, Moq uses the LINQ-like expression tree and lambda expression format to explicitly grab the method. Giving you access to the method means that, as a developer, you can see what parameters the method requires. You're also assured that the method you're attempting to set expectations on is truly mocked, right down to the specific method overloads.

Directly following the **Setup** extension, as seen in the above snippet, the **Returns** extension is then used. Moq doesn't require the use of a Return for a given setup, but for this example, let's make use of it as we're expecting a result from the method. The **Returns** extension is powerful, and it's possible to provide either an exact result (such as is the case for the above example), or a full expression, wherein you can make use of passed through parameters, and trigger any other code you may wish to run.

The result is not only a better test mechanism, but the code itself, while largely unaltered, can be manipulated and extended to a greater degree. When the test in the above code snippet is run, you can now be assured that the result will be the expected result, regardless of which day of the week it is.

Act

You have now seen how Moq can simplify your testing when arranging your objects, but as demonstrated within the previous code snippet, when a **Mock** instance is initialized, a mocked object is created that's simply **flavored** like your provided interface or class. To demonstrate how to access methods or properties directly from that mocked object, I've updated the previous interface, as seen in the following code snippet, so that it now contains a **Name** property.

```
public interface IDateTimeProvider
{
    DayOfWeek DayOfWeek();
    string Name { get; set; }
}
```

With this new interface in place, at the time when you need to access the object itself, Moq exposes an Object property as part of the dynamically created mocked class. This is demonstrated within the code snippet below, wherein the Name property can be accessed, and even set, directly through the mock.

```
// Arrange
var rateCalculator = new RateCalculator();
var dateTimeProviderMock =
      new Mock<IDateTimeProvider>();

dateTimeProviderMock.Setup(m => m.DayOfWeek()
                    .Returns(DayOfWeek.Sunday);
dateTimeProviderMock.Object.Name =
"DateTimeProvide#1";
```

This is largely all you'll need for the Act sections of your tests when using Moq. Another option is the static Get method, as seen in the next code snippet, which allows you to access the mock wrapper on a given mocked object.

```
IExampleInterface foo =
    ExampleClass.ReturnedFromSomewhereElse();

var fooMock = Mock.Get(foo);
fooMock.Setup(m => m.DoNewAction()).Returns(false);

Assert.That(fooMock.Object.DoNewAction(),
                    Is.EqualTo(false));
```

I don't tend to use this a lot, as altering the arrangement of a test at the point of acting upon it breaks the AAA structure, but it's reassuring to know that, in instances

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. Nothing to buy. For more information, visit www.codemag.com/consulting or email us at info@codemag.com.

where you might be passed a mocked object from somewhere you can't control, you can still append to it.

Assert

The assertion section of AAA is where you ensure that a test ran as expected and you have the expected results. All good unit tests have an assertion, and I try to make it a rule that I only assert one thing in each of my unit tests. Having a single assertion leads to greater accuracy: Should something ever break when new features are added in the future, that developer will have an easy time of identifying the exact location of the break and what needs to be done to create a fix.

When unit testing, you generally have your tests checking the results of a called method, or code interaction, but what if you want to know what happened while your code was under test? Perhaps you want to know that a certain property was fetched a specific number of times, or to ensure that a certain method was not ran. For these cases, Moq gifts us with the verification functionality.

```
// Arrange
var rateCalculator = new RateCalculator();
var dateTimeProviderMock
          = new Mock<IDateTimeProvider>();
dateTimeProviderMock.Setup(m => m.DayOfWeek())
                    .Returns(DayOfWeek.Sunday);

// Act
var actual = rateCalculator.GetPayRate(10.00m,
                 dateTimeProviderMock.Object);

// Assert
Assert.That(actual, Is.EqualTo(12.5m));
dateTimeProviderMock.Verify(m => m.DayOfWeek(),
                                 Times.Once());
```

Building upon the previous code examples, you can see in the most recent snippet that there's a regular test set up for the **IDateTimeProvider** interface. Within it, you wish to know how many times the **DayOfWeek** method was called. You implement the **Verify** extension on the mock within the **Assert** section of the test. You also make use of Moq's static **Times** class, which provides various options for setting expectations for the number of times a method or property was called.

On display once more within the previous code snippet, is the readability benefit of Moq. Usage of the **Times** utility class results in tests that can be easily interpreted for their function: You have the mocked object and you wish to verify that it worked as expected, which you do by checking that the Setup method was called the number of times you expected it to run. Note, too, that there are some other possible usages of Times within the below code snippet, including **Exactly**, which specifies an expectation that several calls are made but no more or less than the given integer. Following the usage of Exactly is the **Never** method, which enforces that the given method or property was never called. There are several other options, but these are some of the more common you may run across while using Moq.

```
// This is expected to run exactly three (3) times
dateTimeProviderMock.Verify(m => m.DayOfWeek(),
                                 Times.Exactly(3));

// And this property should never run.
dateTimeProviderMock.Verify(m => m.Name,
                          Times.Never());
```

Although you don't need to place a setup on a method to make use of the verify functionality, if you do add a setup, it's possible to make a general verification that your setup was run. By making use of Moq's **VerifyAll** method, as seen within the next code snippet, you can ensure that all setups that were placed on your mock were invoked.

```
// Arrange
var rateCalculator = new RateCalculator();
var dateTimeProviderMock = new
    Mock<IDateTimeProvider>();
dateTimeProviderMock.Setup(m => m.DayOfWeek())
                    .Returns(DayOfWeek.Sunday);

// Act
var actual = rateCalculator.GetPayRate(10.00m,
                dateTimeProviderMock.Object);

// Assert
Assert.That(actual, Is.EqualTo(12.5m));
dateTimeProviderMock.VerifyAll();
```

Upon triggering the **VerifyAll** method when running your test, if any of your setups have not been invoked, an exception will be produced. This can be helpful if your called code is multifaceted and has several setups, each of which you require to be called.

The final verification functionality to showcase is the **VerifyNoOtherCalls** method, which does what it says it will do. As seen in the next code snippet, this method is placed after all other verifies, so that you can then enforce that there are no invocations on the mocked object from that point onward to any of its methods or properties.

```
// Arrange
var rateCalculator = new RateCalculator();
var dateTimeProviderMock = new
    Mock<IDateTimeProvider>();
dateTimeProviderMock.Setup(m => m.DayOfWeek())
                    .Returns(DayOfWeek.Sunday);

// Act
var actual = rateCalculator.GetPayRate(10.00m,
                dateTimeProviderMock.Object);

// Assert
Assert.That(actual, Is.EqualTo(12.5m));
dateTimeProviderMock.Verify(m => m.DayOfWeek(),
                            Times.Exactly(1));
dateTimeProviderMock.VerifyNoOtherCalls();
```

If there are further invocations, an exception is produced, allowing you to hunt down the rogue calls and ensure that your code runs specifically as designed.

Other Moq Functionality

The functionality options for Moq are vast, so to keep things brief, only the most common functionality has been demonstrated so far. This section will give you a

taste of some of the other possibilities, including parameter matching, throwing exceptions, accessing protected methods or properties, callbacks, and sequences.

Parameter Matching

When putting together a Setup or Verify for a method, Moq requires the provision of all parameters that method may take, including those that are optional. This ensures that you set up the correct expectation, as, due to the overloading functionality within C#, it's possible to select the wrong method. Within the below code snippet, you'll see that the method **MyCalculatorMethod** requires a string and a decimal.

```
mock.Setup(m =>
          m.MyCalculatorMethod("HelloWorld!",
                               10.00m))
    .Returns(true);

mock.Setup(m =>
          m.MyCalculatorMethod(It.IsAny<string>(),
                               It.IsAny<decimal>()))
    .Returns(true);
```

Providing specific parameters, such as "Provided Name" and "10.00m", means that if a **Verify** is triggered after the **Act** segment, it expects that **MyCalculatorMethod** ran specifically with the provided parameters. This is helpful in ensuring that what you were expecting was indeed provided.

For the second usage within the code snippet, you'll see that it's also possible to provide **stand-in** values, for the times where you don't care what's passed in, just that those values were provided. To use this functionality, Moq provides the static library of **It**, where you can find the static method of **IsAny<T>** that allows you to simulate any given type. This functionality is incredibly helpful, as it works with custom types, too. For the times where you need to pass a reference-type through your method, Moq also provides the **Ref<T>** method as part of the **It** library.

Throwing Exceptions

Not all methods are meant to simply be expectations with returns, as at times you need to ensure that an exception is thrown when certain conditions are met. For these situations, as demonstrated in the following code snippet, Moq provides the **Throws** extension, which can be used in-place of the **Returns** method, following a **Setup**.

```
// Arrange
var mock = new Mock<IDateTimeProvider>();
mock.Setup(m => m.DayOfWeek())
    .Throws(new Exception());

// Act
var result =
ExampleClass.MethodWhichMakesUseOfMock
    (mockedObject: mock.Object);

// Assert
Assert.That(result, Is.EqualTo(true));
mock.VerifyAll();
```

As you know, the benefit of having access to functionality like this means that you don't need to manually arrange some specific circumstance for the exception to be thrown, it just happens when the expected method is invoked. The structure used for the test in the code snippet involves the creation of an expectation, followed up with a **Verify**, to ensure that the exception was thrown. If the verify runs and the throw wasn't triggered, the test produces an exception.

Accessing Protected Setups

For the times when you're mocking a class and some of its functionality is protected, Moq provides the **Moq.Protected** library. The use of this means that the setup of your mocks is slightly different, as seen in the next code snippet, wherein the extension **Protected** is appended to the mock prior to the rest of the setup.

```
// Arrange
var mock = new Mock<InheritingExample>();
mock.Protected()
    .Setup<bool>("MyProtectedMethod",
                 "AStringToPassToThisMethod",
                 ItExpr.IsAny<string>())
    .Returns(true);

// Act
var result =
     mock.Object.AMethodWhichRunsTheProtectedMethod();

// Assert
Assert.That(result, Is.EqualTo(true));
mock.VerifyAll();
```

Structurally, the test is largely the same, meaning that there's still no doubt what this mock will be doing, and it still reads clearly. Now it also informs the developer that it will be working using a protected mocked object.

Protected functionality for Moq is implemented by providing the protected method's name as a string. Usage of this functionality comes at the cost of less explicit code and thus it's ideal that your code is structured around interfaces that mitigate this issue. Moq does provide the ability to mock classes and so the situation wherein you might need to access a protected method is possible. In these cases, albeit not being ideal, it's good to know that there are options for mocking.

Be aware that when using the protected version of a Setup, the **It** library works slightly differently. Instead of **It**, you'll need to use **ItExpr**. This provides the same functionality but it's labelled differently so that they don't conflict.

Callbacks

A powerhouse of a method, **Callback** allows the provision of a provided action that should be run when an expectation is invoked. Callbacks, as demonstrated in the next code snippet, can be helpful for many reasons, such as setting shared objects or running the save operation after a mocked method has been triggered.

```
// Arrange
var values = new bool[] { false, false, false };
var mock = new Mock<IExampleInterface>();
mock.Setup(m => m.DoNewAction())
    .Callback(() => values[0] = true)
    .Returns(true)
```

Resource	Link
Example Code	https://github.com/elliotmoule/CODE.UsingMoq.Article.ExampleCode
Moq (GitHub)	https://github.com/moq/moq4
Moq (NuGet)	https://www.nuget.org/packages/Moq
Quickstart Guide	https://github.com/Moq/moq4/wiki/Quickstart
Events (further reading)	http://www.blackwasp.co.uk/MoqEvents.aspx

Table 1: Resources and Further Reading

```
    .Callback(() => values[1] = true);

// Act
values[2] = mock.Object.DoNewAction();

// Assert
Assert.That(values.All(x => x == true),
    Is.EqualTo(true));
mock.VerifyAll();
```

As demonstrated, the structure for using Callback is similar to Returns and Throws, keeping usage of the extensions consistent. Additionally, Callback can be used alongside these other extensions, such as by stacking them so that code is run before and after an invocation.

Sequences

What about the times you expect a method to be called three times, each with different results? You're looking for Moq's **SetupSequence** method, which can be used in-place of the Setup method. As seen in the following code snippet, it's possible to stack your mock with several expectations that you're expecting to happen in a set sequence order.

```
// Arrange
var mock = new Mock<IDateTimeProvider>();
mock.SetupSequence(m => m.DayOfWeek())
    .Returns(DayOfWeek.Sunday)
    .Returns(DayOfWeek.Tuesday)
    .Returns(DayOfWeek.Saturday);

// Act
var result =
    ExampleClass.ChecksSequenceIsCorrect(
        mockedObject: mock.Object);

// Assert
Assert.That(result, Is.EqualTo(true));
mock.VerifyAll();
```

Extensions can be stacked, so include Returns, Callback, or Throws as necessary, extending further the possibilities for testing various scenarios. For the above code example, the first return will be **Sunday**, the second **Tuesday**, and the last **Saturday**.

The Advantages of Using Moq Over Other Libraries?

If a project lends itself to mocking, such as by having complex objects or tests that could be improved by using mocking, any mocking library is a positive. For me, mocking helps enforce better structure and thoughtful design for my interfaces and classes. I more frequently use dependency injection when I also use Moq, and now, looking back at older projects where I've used Moq, I find that I can quickly and easily get back into development, as the structure of my tests guides my understanding of the functionality I created previously.

There are several alternatives to Moq, including NSubstitute, FakeItEasy, Rhino Mocks, and EasyMock, each with strengths and weaknesses. One of the biggest advantages of Moq by comparison, which you will have seen on full display, is its simple and intuitive syntax. Creating and setting up mocked objects is not only easy, but incredibly quick, which is made possible by Moq providing the expression tree structure and the ability to use it alongside lambda expressions. For developers who may not have used Moq, or mocking more generally, Moq provides a friendly library of functionality that will have most developers on their way quickly. As demonstrated, Moq's rich set of features and capabilities allow developers to create complex mocked objects and set up behavior in a wide range of testing scenarios. This makes it a versatile and powerful tool for C# .NET unit testing. Although Moq may not be as flexible or powerful as some of the other mocking libraries mentioned, its simplicity and ease of use make it a great choice for many developers.

Wrapping Up

Moq is a useful tool for improving the isolation and effectiveness of your unit tests. Transform your code by making use of mocking and enforcing patterns such as dependency injection and the principle of inversion of control. Through using Moq's easy-to-implement library, you can ensure that your units under test have the same state as your application, so that there are no more headaches from indecipherable and confusing dummy classes. I've provided several resources in **Table 1** to continue your learning, including the Moq quick start guide and using Moq for events, should you wish to do so. Thank you for joining me on this adventure, and I hope that you're soon well on your way to implementing mocking within your code and making use of all that Moq can provide. And I'm certain that, through its use, the quality of your testing code will just get better and better.

Elliot Moule

Adding Scripting to Existing Code Using Reflection

In this article, we're going to talk about using C# Reflection functionality from a scripting language. The main idea is that very well-known and debugged parts of C# code can be reused in scripting. A use case for using Reflection and a scripting language might be a scenario where there's already a client class library that does something useful (in C#) and you want to additionally control it from the scripting code to get more flexibility. As an example, we're going to use the PCmover service developed by Laplink Software. PCmover helps you to upgrade an old PC to a new one by transferring all of your files and installed programs to your new computer so that you're up and running on your new computer very quickly and easily.

Why would you want to control these operations from the scripting code? With scripting, you can use the technology in new ways that go beyond what the basic user interface allows. This can include special customization or selection of the computers that are involved, among other things.

One customer may want to transfer from one computer to several computers. The service supports that but the GUI doesn't. That customer could write scripts to do that. Another customer may want to transfer only a certain number of files from one part of a file system. Another customer may want to add extra security checks on certain types of transfers. There are many things that could be done but that the GUI isn't well suited to do. Scripting opens up all the capabilities to the customers without having to create custom versions of the GUI for them.

The goal of this article is to show how you can easily take C# code from a complex service and create CSCS code that looks very similar to the C# code, so that the scripting language now has access to a very complicated existing DLL.

As a scripting language, we're going to use CSCS (Customized Scripting in C#). This is an easy and lightweight open-source language that has been described in previous CODE Magazine articles: https://www.codemag.com/article/1607081 introduced it, https://www.codemag.com/article/1711081 showed how you can use it on top of Xamarin to create cross-platform native mobile apps, and https://www.codemag.com/article/1903081 showed how you can use it for Unity programming. CSCS resembles JavaScript with a few differences, e.g., the variable and function names are case insensitive.

Using reflection, very little code is needed to access existing .NET code via scripting. In this article, we'll show you how to do it and how it's implemented.

Let's start by looking at how reflection can be incorporated into scripting.

A Brief Introduction to the PCmover Application

The easiest way to include the CSCS scripting into your application is to download the CSCS project from the GitHub and include it in your work space. You should also include a reference to the CSCS project in your project. See **Figure 1** for details.

Here's the description of the various modules that are part of the workspace in **Figure 1**:

- **PcmoverComponent**: This is presumed to be an existing DLL that implements a PCmover Service. It's very complicated (not in this example, but it is in real life), doing a large amount of work to transfer all of your files and programs from your old computer to the new computer. By using scripting with reflection, you can easily take C# code that uses this service and create CSCS code that looks very similar to the C# code, so that the scripting language now has access to this very complicated existing DLL.
- **ArticleConsole**: This is the top-level console app that you build and run. Before launching the script, it runs a function called TestPcmoverComponent, which calls a series of functions in PcmoverComponent, demonstrating how it would be used with pure C# code.
- **PcmoverCscsModule**: This is the PcmoverModule component that links the PcmoverComponent code into CSCS. It implements only one function: GetPcmover, which returns a PcmoverService object. Everything else in CSCS is done via reflection through that object.
- **CSCS**: Core CSCS code, referenced in the solution file. Implements core CSCS parsing.
- **CSCS.ConsoleApp**: CSCS.ConsoleApp code, referenced in the solution file. Implements the console functionality around CSCS scripting.
- Modules used by CSCS.
  - **CSCS.InterpreterManager**: Implements multiple interpreters in CSCS
  - **CSCS.Math**: Implements CSCS math functions
  - **CSCS.SqlClient**: Implements CSCS client connectivity to a SQL database.

You can add an arbitrary number of custom modules. You do it in C# in the initialization stage. Note that each module is compiled into a DLL and it doesn't have to be added in C# code or even be a part of your workspace. In this case, you can add an arbitrary module at runtime as follows:

```
Import("CSCS.Math");
```

When doing such an import, the CSCS interpreter is looking for files with the name CSCS.Math.dll.

Dan Spear
Dan.Spear@laplink.com
https://go.laplink.com/
https://en.wikipedia.org/wiki/Laplink_PCmover

Dan is the Chief Architect for Laplink Software, and creator and principal developer for their flagship product, PCmover. Previously, in several roles at Quarterdeck Corporation.

Vassili Kaplan
VassiliK@gmail.com

Vassili was a Microsoft Lync developer between 2008 and 2011. He also worked on the Microsoft Maquette Mixed Reality Prototyping tool and CSCS scripting language to Microsoft Maquette.

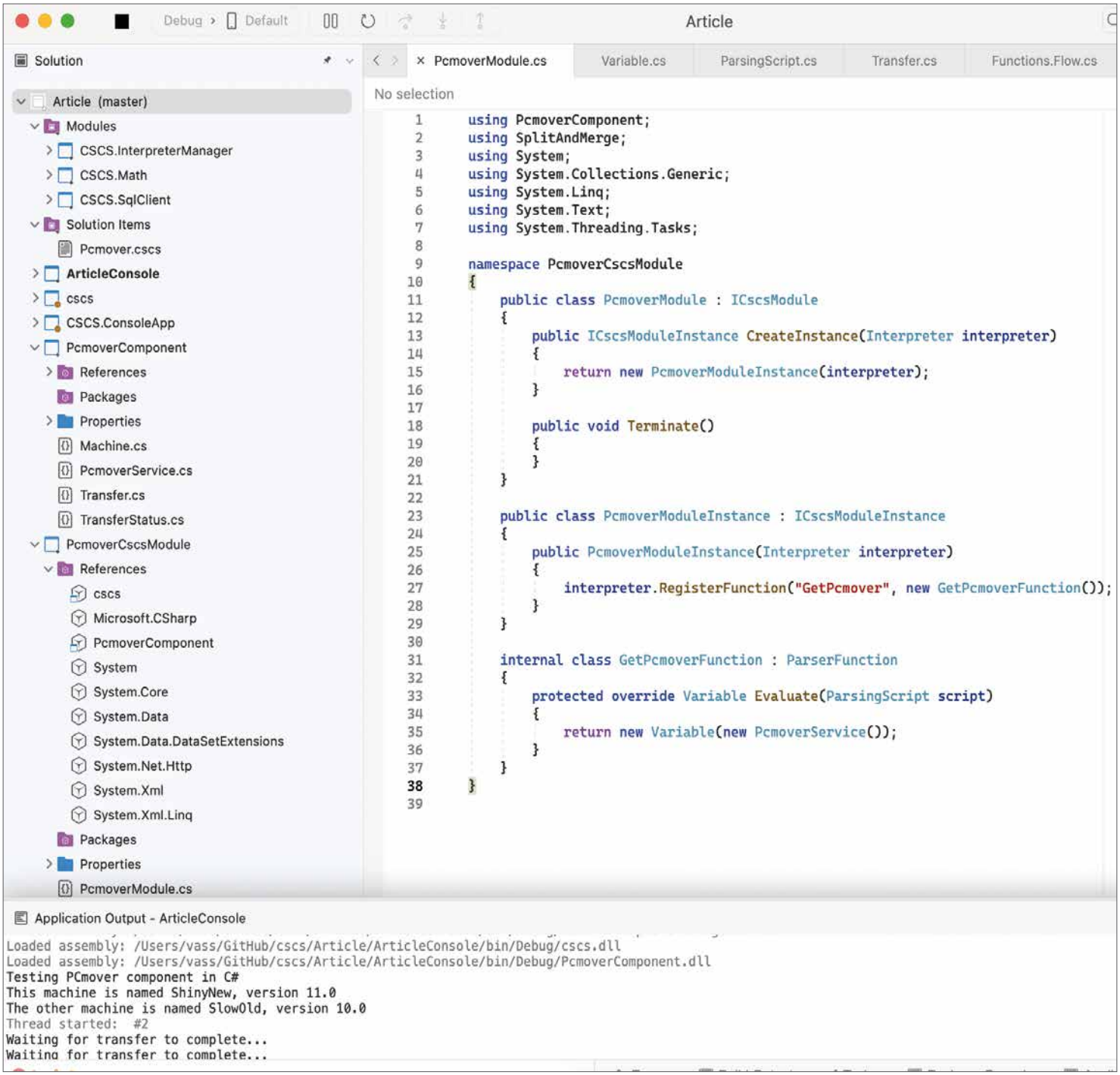


Figure 1: A sample PCmover application Visual Studio project

"Hello, World!" in Reflection with Scripting

Let's start with a very simple class like this one:

```
namespace HelloCscsModule
{
  public class HelloWorldService
  {
    public string Hello(string name) {
      return "Hello, " + name + "!";
    }
  }
}
```

The idea is to use methods declared in this class without also declaring them in a scripting module. In this case, there's just one method, Hello().

To be able to use this HelloWorldService class, you need to define a class deriving from ParserFunction and override its Evaluate() method. This functionality is the same for any other CSCS scripting function. See https://www.codemag.com/Article/2209051 for some examples.

Here's an implementation of such a class, returning a Variable that is a wrapper over the HelloWorldService object:

```
internal class GetHelloWorldFunction:
              ParserFunction
{
  protected override Variable Evaluate(
         ParsingScript script)
  {
    return new Variable(
           new HelloWorldService());
  }
}
```

The last step is to register the GetHelloWorldFunction from the previous step with the interpreter. This is done by executing the following statement in the initialization phase:

```
interpreter.RegisterFunction("GetHello",
 new HelloCscsModule.GetHelloWorldFunction());
```

That's it! As soon as the interpreter extracts the "GetHello" token, it returns a handle to the underlying HelloWorldService object. Now you can execute any method defined for the HelloWorldService class in the C# code. In this case, it's just the Hello() method. It's triggered by interpreter using reflection.

Here's an example in CSCS:

```
hi = GetHello();
hi.Hello("world");
```

Here's the output after executing the two statements above:

```
HelloCscsModule.HelloWorldService
Hello, world!
```

In the same manner, you can invoke any method or a property defined in C#.

In the next section, you're going to see a more real-life example with the PCmover service from Laplink.

Example of PCmover Scripting

Let's look at real C# code that interacts with the PCmover service and see how easy it is to convert that to script code. This is the TestPcmoverComponent function in C# that's used to run a transfer.

First, initialize required components:

```
void TestPcmoverComponent() {
  var pcmoverService = new PcmoverService();
  Print("Testing PCmover component in C#");

  var thisMachine = pcmoverService.ThisMachine;
  Print("This machine is named " +
        thisMachine.Name + ", version " +
        thisMachine.WindowsVersion);

  var otherMachine =
        pcmoverService.GetMachine("SlowOld");
  if (otherMachine == null) {
    Print("Unable to get the other machine");
    return;
  }
  Print("The other machine is named " +
        otherMachine.Name + ", version " +
        otherMachine.WindowsVersion);
```

Then create a Transfer object and mimic a transfer:

```
  Transfer transfer = pcmoverService.
    CreateTransfer(otherMachine, thisMachine);
  if (transfer.Status != TransferStatus.Ready) {
    Print("Transfer is not Ready");
    return;
  }
  if (!transfer.Start()) {
    Print("Error starting transfer");
    return;
  }

  while (transfer.Status ==
         TransferStatus.Active) {
    Print("Waiting for transfer to complete...");
    Thread.Sleep(1000);
  }
  Print("Transfer complete!");
}
```

This code sets up a transfer from the old computer, named SlowOld, to the new computer. It displays some details along the way and some progress while it waits for the transfer to complete. The output of this C# code looks like this:

```
Testing PCmover component in C#
This machine is named ShinyNew, version 11.0
The other machine is named SlowOld, version 10.0
Waiting for transfer to complete...
Waiting for transfer to complete...
Waiting for transfer to complete...
Waiting for transfer to complete...
Waiting for transfer to complete...
Transfer complete!
```

Now you want to write the same functionality in script. Just like with the simple HelloWorld example, all you need to implement in the scripting environment is a function to get the initial PcmoverService object. Everything else works automatically, with a few small tweaks, via reflection. Here's the CSCS code that does the same thing:

```
function TestPcmoverComponent() {
  pcmoverService = GetPcmover();
  Print("Testing PCmover component in CSCS");

  thisMachine = pcmoverService.ThisMachine;
  Print("This machine is named " +
        thisMachine.Name + ", version " +
        thisMachine.WindowsVersion);
  otherMachine =
      pcmoverService.GetMachine("SlowOld");
  if (otherMachine == null) {
    Print("Unable to get the other machine");
    return;
  }
  Print("The other machine is named " +
        otherMachine.Name + ", version " +
        otherMachine.WindowsVersion);
```

Just like its C# counterpart, you create a Transfer object and do the actual transfer after the initialization stage:

```
  transfer = pcmoverService.CreateTransfer(
             otherMachine, thisMachine);
  if (transfer.Status != "Ready") {
    Print("Transfer is not Ready");
    return;
  }
  if (!transfer.Start()) {
    Print("Error stating transfer");
    return;
  }

  while (transfer.Status == "Active") {
    Print("Waiting for transfer to complete...");
    sleep(1000);
  }
  Print("Transfer complete!");
}
```

This code is almost line-for-line identical to the C# code, with just a few differences. Those differences are:

- Syntactically, CSCS does not define functions the same way as C#.
- Syntactically, CSCS does not declare variables with their types. You just use them.
- The CSCS code uses the special GetPcmover function to get the pcmoverService object instead of the "new" syntax of the C# code.
- When checking the value of transfer.Status, C# uses the enum value, but CSCS uses a string matching the enum name.
- CSCS uses the built-in "sleep" function rather than Thread.Sleep().

This is the output of the CSCS version:

```
Testing PCmover component in CSCS
This machine is named ShinyNew, version 11.0
The other machine is named SlowOld, version 10.0
Waiting for transfer to complete...
Waiting for transfer to complete...
Waiting for transfer to complete...
Waiting for transfer to complete...
Waiting for transfer to complete...
Transfer complete!
```

> Good programmers write good code. Great programmers write no code. Zen programmers delete code.
>
> *John Byrd*

That's it! With just a handful of CSCS syntactic changes and a simple function to expose an existing .NET object, you can implement full access to an existing library. Reflection handles access to all of the functionality of that object, and any other objects you retrieve through it. This example uses a PcmoverService object, but it could be a game service, or a geographical service, or any other existing library that you have that you'd like to control with scripting.

Using Reflection in Scripting

In this section, you'll see code snippets of the implementation of reflection in CSCS. The full version is available on GitHub (https://github.com/vassilych/cscs).

As soon as the scripting parsing interpreter encounters a name after a dot, the following method checks whether you can use reflection to either extract the value of the property or to execute a method with that name:

```
Variable GetReflectedProperty(string propName,
         ParsingScript script) {
  BindingFlags bf = BindingFlags.Instance |
                    BindingFlags.Public;

  var property = FindNestedMatchingProperty(
      ObjectType, propName,
      bf | BindingFlags.GetProperty);
  if (property != null) {
    object val = property.GetValue(Object);
    return ConvertToVariable(val,
      property.PropertyType);
  }
```

First, try to find the property match. If not successful, look for methods:

```
  var pConv = new ParameterConverter();
  var bestMethod = pConv.FindBestMethod(
    ObjectType, propName, GetArgs(script), bf);
  if (bestMethod != null} {
    var res = bestMethod.Invoke(Object,
      pConv.BestTypedArgs);
    return ConvertToVariable(res,
      bestMethod.ReturnType);
  }
  return null;
}
```

FindBestMethod() scans the methods to find one that best matches the parameters that are passed from the script.

```
MethodInfo FindBestMethod(Type t, string
 propName, List<Variable> args, BindingFlags bf)
{
  MethodInfo bestMethod = null;
  var methods = t.GetMethods(bf);
  if (methods != null) {
    foreach (var method in methods) {
      if (String.Compare(method.Name, propName,
          true) == 0) {
        var parameters = method.GetParameters();
        if (ConvertVariablesToTypedArgs(args,
            parameters)) {
          bestMethod = method;
          if(BestConversion == Conversion.Exact)
            break;
        }
      }
    }
  }
```

Listing 1: Reflection implementation for scripting in C# (TO BEAUTIFY)

```
public enum Conversion
{
    Exact,
    Assignable,
    Convertible,
    Mismatch
}

public static Conversion ChangeTypes(List<Variable> args, ParameterInfo[]
parameters, object[] typedArgs)
{
    Conversion worstConversion = Conversion.Exact;
    if (args.Count > 0)
    {
        for (int arg = 0; arg < args.Count; ++arg)
        {
            typedArgs[arg] = ChangeType(args[arg].AsObject(), parameters[arg].
ParameterType, out Conversion conversion);
            if (conversion > worstConversion)
                worstConversion = conversion;
        }
    }
    return worstConversion;
}

public static object ChangeType(object value, Type conversionType)
{
    return ChangeType(value, conversionType, out Conversion conversion);
}

public static object ChangeType(object value, Type conversionType, out
Conversion conversion)
{
    try
    {
        Type underlyingType = Nullable.GetUnderlyingType(conversionType);

        if (value == null)
        {
            if (underlyingType == null && conversionType.IsValueType)
            {
                conversion = Conversion.Mismatch;
            }
            else
            {
                    conversion = Conversion.Exact;
            }

            return value;
        }

        Type t = value.GetType();
        if (t == conversionType)
        {
            conversion = Conversion.Exact;
            return value;
        }

        if (t == underlyingType)
        {
            conversion = Conversion.Convertible;
            return value;
        }

        if (conversionType.IsAssignableFrom(t))
        {
            conversion = Conversion.Assignable;
            return value;
        }

        if (underlyingType != null && underlyingType.IsAssignableFrom(t))
        {
            conversion = Conversion.Convertible;
            return value;
        }

        conversion = Conversion.Convertible;
        if (conversionType.IsEnum)
        {
            if (value is string svalue)
            {
                    return Enum.Parse(conversionType, svalue, true);
            }

            if (value is double dvalue)
            {
                return Enum.ToObject(conversionType, (long)dvalue);
            }

            // Let's see if it's some other type that just happens to work
            conversion = Conversion.Mismatch;
            return Enum.ToObject(conversionType, value);
        }

        IList genericList = ConvertToGenericList(value, conversionType);
        if (genericList != null)
        {
            AddToGenericList(genericList, value);
            return genericList;
        }
        try
        {
            return Convert.ChangeType(value, conversionType);
        }
        catch (Exception)
        {
            if (underlyingType != null)
            {
                return Convert.ChangeType(value, underlyingType);
            }
        }
    }
    catch { }
    conversion = Conversion.Mismatch;
    return value;
}
```

```
  return bestMethod;
}
```

The C# implementation details of using reflection from scripting are shown in **Listing 1**.

> Experience is the name everyone gives to their mistakes.
>
> *Oscar Wilde*

Using Events

The code that we've worked on so far had one undesirable aspect. While the transfer was taking place, the code was sitting in a loop polling for whether the transfer was complete, displaying a generic message each time it checked. It would be nice to get asynchronous notification of the progress of the transfer. Fortunately, this PCmover component supports a Progress event. Let's see how to incorporate that into the script client.

The PCmover component defines a ProgressInfo object, which the Transfer object passes to a Progress event, like this:

```
public class ProgressInfo {
  public Transfer Transfer { get; set; }
  public string Action { get; set; }
  public int Percent { get; set; }
}
```

Listing 2: CSCS code to test PCmover implementation in scripting

```
function TestPcmoverComponent()
{
    pcmoverService = GetPcmover();

    Print("Testing PCmover component in CSCS");

    thisMachine = pcmoverService.ThisMachine;
    Print("This machine is named " + thisMachine.Name +
          ", version " + thisMachine.WindowsVersion);

    otherMachine = pcmoverService.GetMachine("SlowOld");
    if (otherMachine == null)
    {
        Print("Unable to get the other machine");
        return;
    }
    Print("The other machine is named " +
     otherMachine.Name + ", version " +
     otherMachine.WindowsVersion);

    transfer = pcmoverService.CreateTransfer(
               otherMachine, thisMachine);
    if (transfer.Status != "Ready")
    {
        Print("Transfer is not Ready");
        return;
    }

    if (!transfer.Start())
    {
        Print("Error starting transfer");
        return;
    }

    while (transfer.Status == "Active")
    {
        Print("Waiting for transfer to complete...");
        sleep(1000);
    }

    Print("Transfer complete!");
}
public class NewObjectFunction : ParsingFunction
{
  protected override Variable Evaluate(ParsingScript  script)
  {
    string className = Utils.GetToken(script);
    className = Constants.ConvertName(className);
    List<Variable> args = script.GetFunctionArgs();

    var csClass = CSCSClass.GetClass(className) as
                  CompiledClass;
    if (csClass != null) {
      ScriptObject obj =
         csClass.GetImplementation(args);
      return new Variable(obj);
    }
    var instance = new CSCSClass.ClassInstance(
      script.CurrentAssign, className, args, script);

    var newObject = new Variable(instance);
    newObject.ParamName = instance.InstanceName;
    return newObject;
  }
}
```

```
public class Transfer {
  public delegate void ProgressHandler(
      ProgressInfo info);
  public event ProgressHandler Progress;

  void FireProgress(string action, int percent){
    Progress?.Invoke(new ProgressInfo(this,
      action, percent);
  }
    …
}
```

To use it, modify the TestPcmoverComponent code to subscribe to the event and have the event handler display the progress message and set an event when the transfer completes. The main code just waits for that event.

```
AutoResetEvent _transferComplete;

void OnProgress(ProgressInfo info) {
  Print(info.Action + " " + info.Percent + "%");
  if (info.Transfer.Status ==
      TransferStatus.Complete)
    _transferComplete.Set();
}

void TestPcmoverComponent() {
// Setup is the same, until the Transfer:
  _transferComplete = new AutoResetEvent(false);
  Transfer.Progress += OnProgress;
  if (!transfer.Start()) {
    Print("Error starting transfer");
    return;
  }
  _transferComplete.WaitOne();
  Print("Transfer complete!");
}
```

Now, when you run TestPcmoverComponent, you get progress messages:

```
Testing PCmover component in C#
This machine is named ShinyNew, version 11.0
The other machine is named SlowOld, version 10.0
Transferring Files 0%
Transferring Files 12%
Transferring Files 23%
Transferring Files 35%
Transferring Files 46%
Transferring Files 58%
Transferring Registry 70%
Transferring Registry 81%
Transferring Registry 93%
Complete 100%
Transfer complete!
```

See **Listing 2** to test PCmover implementation code in scripting.

Getting Events in Script

So, how do you do the equivalent in the script code? Although script code could directly access the Transfer event object, it has no way to directly add a delegate to it. To do that, in the script support module for PCmover,

Laplink PCmover	https://en.wikipedia.org/wiki/Laplink_PCmover
Laplink PCmover	https://go.laplink.com/
CSCS Repository	https://github.com/vassilych/cscs
CSCS Visual Studio Code Debugger Extension	https://marketplace.visualstudio.com/items?itemName=vassilik.cscs-debugger

Table 1: Resources

you need the GetPcmover function to return a wrapper object around the PcmoverService object. This lets you extend the functionality. The extra functionality you want to add is to create a transfer object that fires events.

```
internal class PcmoverServiceWrapper :
              PcmoverService {
  Interpreter InterpreterInstance { get; }

  public PcmoverServiceWrapper(
    Interpreter interpreter){
    InterpreterInstance = interpreter;
  }

  public Transfer CreateProgressTransfer(
    Machine oldMachine,
    Machine newMachine,
    string onProgress) {
     var transfer = new Transfer(oldMachine,
       newMachine);
     transfer.Progress += (info) =>>
     {
        InterpreterInstance.Run(onProgress,
          new Variable(info);
     }
     return transfer;
  }
}
```

The script can now call CreateProgressTransfer, instead of CreateTransfer, passing the name of a script function that should execute when the Progress event fires. All that's left is to modify the script to use it:

```
function OnProgress(info) {
  Print(info.Action + " " + info.Percent + "%");
  if (info.Transfer.Status == "Complete")   {
    Signal();
  }
}

void TestPcmoverComponent() {
// Setup is the same, until the Transfer:
  transfer = pcmoverService.
       CreateProgessTransfer(
       otherMachine, thisMachine, "OnProgress");
…
// Then, instead of polling:
  Wait();
  Print("Transfer complete!");
}
```

You have to call the wrapper's CreateProgressTransfer function, which did the plumbing related to setting up the event handler. The event handler used the interpreter's Run function to execute the requested script code, passing it the same ProgressInfo object. Script used the CSCS Wait and Signal mechanism for the event handling. Otherwise, the code is practically the same as the C# code. And the output is the same as well:

```
Testing PCmover component in CSCS
This machine is named ShinyNew, version 11.0
The other machine is named SlowOld, version 10.0
Transferring Files 0%
Transferring Files 12%
Transferring Files 23%
Transferring Files 35%
Transferring Files 46%
Transferring Files 58%
Transferring Registry 70%
Transferring Registry 81%
Transferring Registry 93%
Complete 100%
Transfer complete!
```

Wrapping Up

In this article, you saw how you can use reflection in a scripting language to access an existing and good working .NET library.

The code shown in this article is just an invitation to explore. After reading it, we hope you can implement scripting via reflection in .NET libraries that you use.

We're looking forward to your feedback, specifically, your experience in applications you're accessing with reflection from scripting and also what other features in CSCS you would like to see.

Dan Spear
CODE

Vassili Kaplan
CODE

SPONSORED SIDEBAR:

Are You Writing Secure C# Code?

Hopefully you are, but hopefully isn't good enough is it? Learn to develop robust and secure C# applications in our online, two-day, hands-on, Secure Coding for C# Developers course. Register today! https://www.codemag.com/training

Writing Code to Generate Code in C#

I've been writing software for nearly 30 years. Over that time, I've learned a lot about what it takes to write code that works well, along with understanding that this educational journey is never over. One rule of thumb I follow is "automate successes." Whenever I encounter a pattern or approach in the programming language I'm using that leads to an improvement, such as better performance or code that's easier to read, I reuse it in the future. What's even better is if I can find a way to enforce these good tactics without having to remember to do them manually.

Jason Bock
jason.r.bock@outlook.com
http://www.jasonbock.net

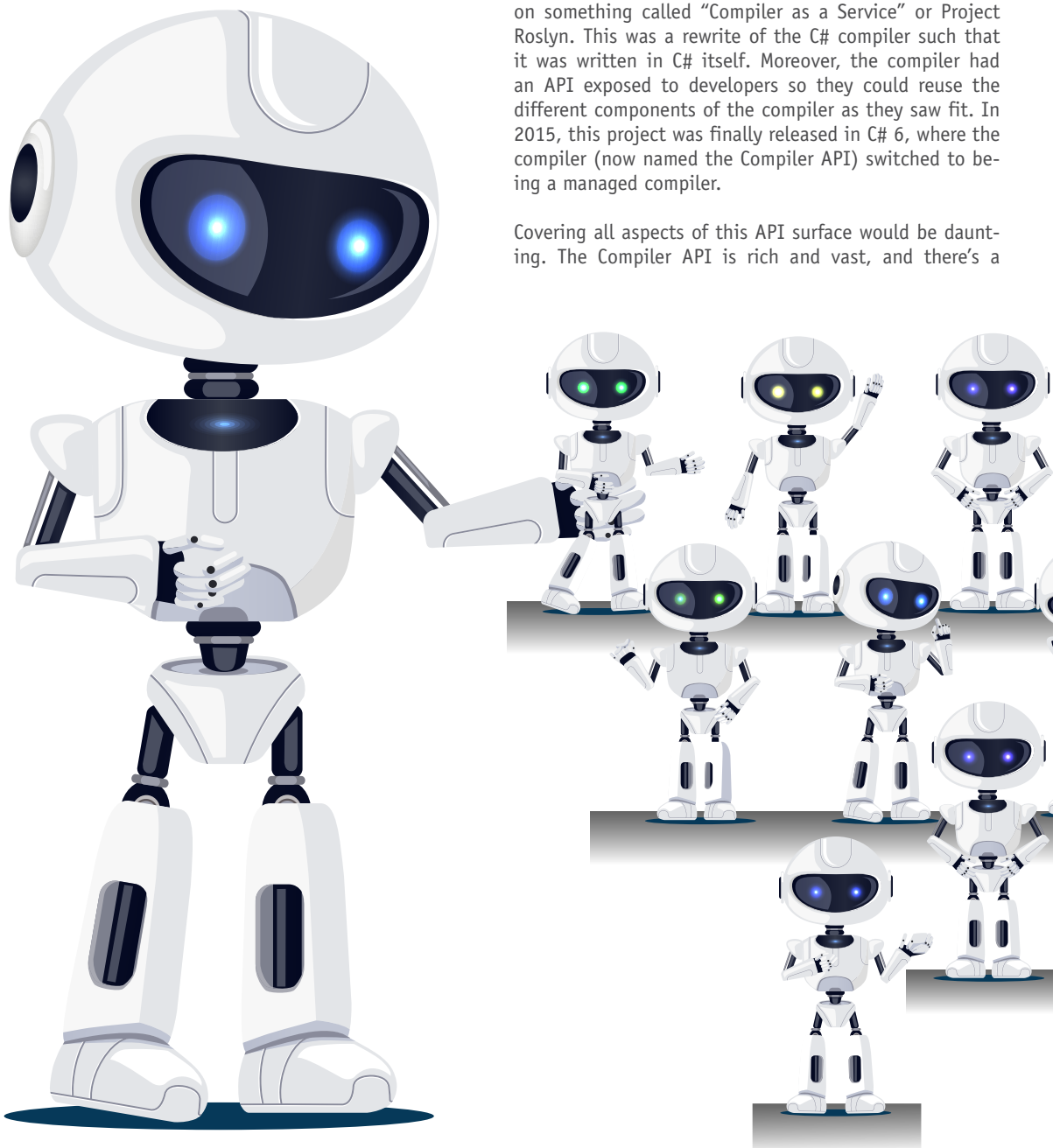
Jason Bock is a Developer Advocate at Rocket Mortgage and a Microsoft MVP (C#). He has nearly 30 years of experience working on several business applications using a diverse set of frameworks and languages. He's the author of ".NET Development Using the Compiler API," "Metaprogramming in .NET," and "Applied .NET Attributes." He's written numerous articles on software development issues and has presented at a number of conferences and user groups. He's a leader of the Twin Cities Code Camp (https://twincitiescodecamp.com/) and holds a master's degree in Electrical Engineering from Marquette University.

In C# 9, a new feature called "source generators" was added. This allows developers to create code that generates new code during the compilation process. This can eliminate repetitive developer tasks and provide a way to improve performance in applications. In this article, I'll describe what source generators are, why they're useful, and how you build one that you can use in your own projects.

A Quick Overview of the Compiler API

Before I dive into source generators, I'll cover what the Compiler API, or Project Roslyn, is. Since the first version of C# was released in 2002, the underlying compiler was written in C++ and C. Over time, Microsoft started working on something called "Compiler as a Service" or Project Roslyn. This was a rewrite of the C# compiler such that it was written in C# itself. Moreover, the compiler had an API exposed to developers so they could reuse the different components of the compiler as they saw fit. In 2015, this project was finally released in C# 6, where the compiler (now named the Compiler API) switched to being a managed compiler.

Covering all aspects of this API surface would be daunting. The Compiler API is rich and vast, and there's a



significant amount of functionality within. Essentially, the compiler translates your source code into a syntax tree. A syntax tree represents the contents of code—this includes all the "trivia" characters, like tabs. **Figure 1** shows a truncated view of a C# syntax tree.

Type definitions, attributes, even whitespace—it's all captured in a tree structure. That's why **Figure 1** is "truncated." Showing the entire contents of a syntax tree for an appreciable amount of C# code would make the tree unreadable. **Figure 1** demonstrates the format of a tree, like a class contains methods and properties and a method has a block body.

The typical result of this tree generation is to transform the tree into an assembly. However, there are APIs that allow you to extend and augment compilation:

- **Analyzers:** You can examine a syntax tree and find issues within that go beyond the issues that the C# compiler looks for. If diagnostics are created in your analyzer, they are manifested as warnings or errors, forcing you to address the problem in a timely manner.
- **Code fixes:** You can provide an automated correction when an analyzer creates a diagnostic. These code fixes can be applied across an entire solution, making it efficient to address analyzer issues.
- **Refactorings:** You can define a transformation to code that helps with readability or assists you move to new coding patterns. Well-known refactorings, such as the Extract method, allow a coder to highlight a section of code, change a method, and add a call to this method where the code block was extracted from. Other refactorings, like changing a verbatim string to a raw string literal, simplify the process of modernizing C# code to new features.

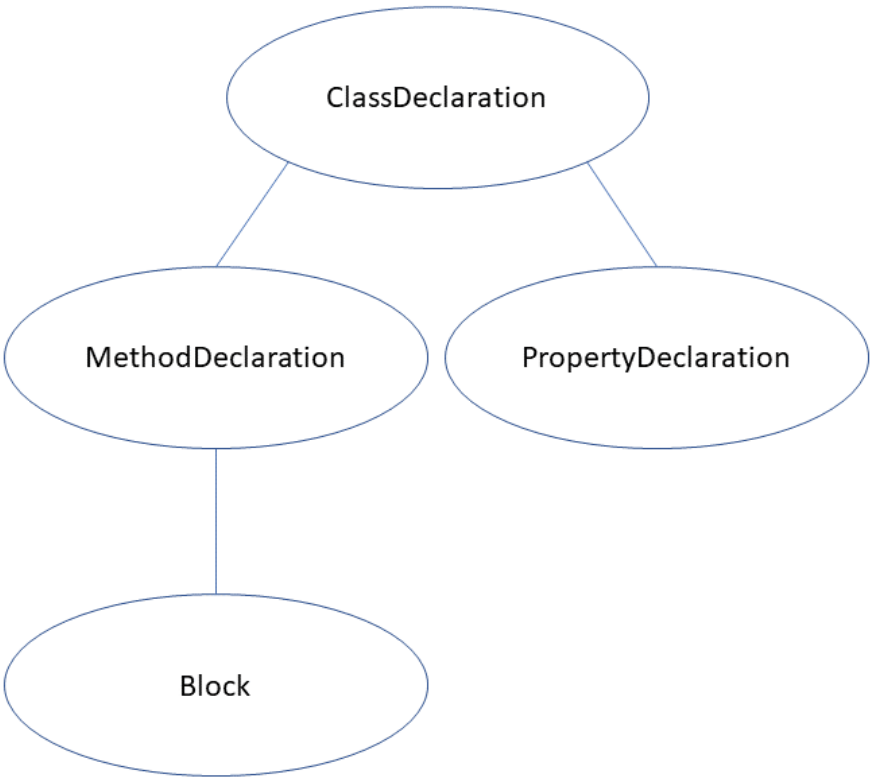
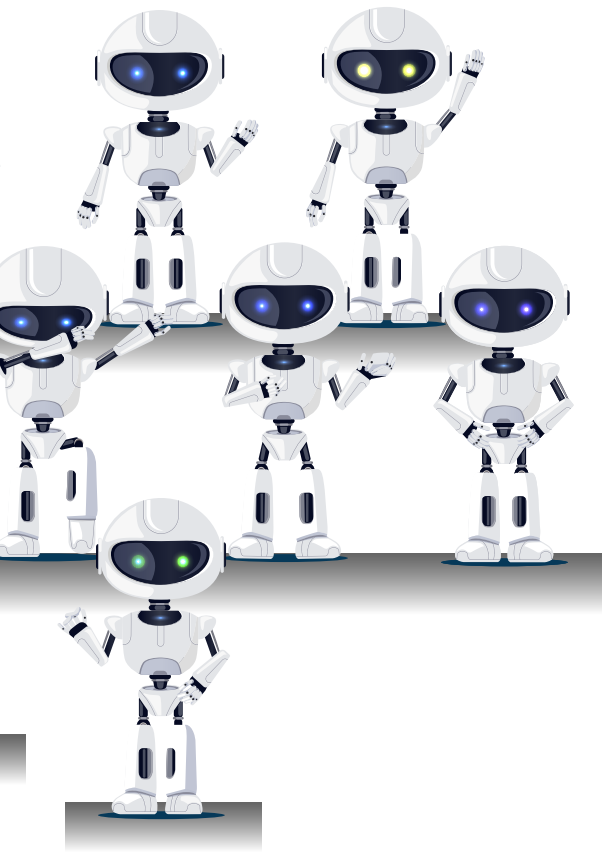


Figure 1: Truncated syntax tree



Source generators were added in 2020 when C# 9 was released. Generators provide a mechanism where you can build code based on the current state of a compilation. I'll explain what generators are in the next section.

The What and Why of Source Generators

Let's start the source generator journey by reviewing what source generators are at a high level. Source generators allow you to plug into the compilation pipeline to inspect the nodes of a syntax tree. These nodes are passed to your source generator, where you decide if a particular node is needed for what your generator needs to do. If that tree information is what you're looking for, you can create a whole new code file. **Figure 2** provides a simple view of this process.

I'll cover the details of implementing a generator in the "Building a Source Generator" section. As you'll see in that section, writing a generator isn't trivial. However, keep in mind that the job of a generator is to get you from point A (a syntax tree) to point B (a new piece of code). Your generator is a factory that takes code as input and produces new code at the end of an assembly line.

At this point, it's worth spending the time answering the question: Why would I want to write a generator in the

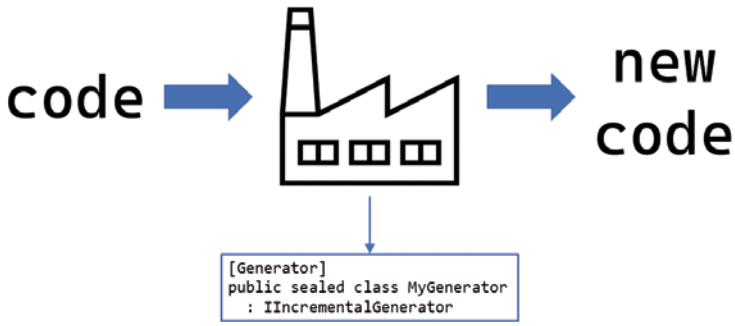


Figure 2: Generator overview

Listing 1: Implementing INotifyPropertyChanged

```
using System.ComponentModel;
using System.Runtime.CompilerServices;

public sealed class Customer
  : INotifyPropertyChanged
{
  private string? name;

  private void NotifyPropertyChanged(
    [CallerMemberName] string name = "") =>
      this.PropertyChanged?.Invoke(
        this, new(name));

  public string? Name
  {
    get => this.name;
    set
    {
      if (value != this.name)
      {
        this.name = value;
        this.NotifyPropertyChanged();
      }
    }
  }

  public event PropertyChangedEventHandler
    PropertyChanged;
}
```

first place? There are two scenarios to consider: eliminating repetition and improving performance.

There are numerous times in code where I've encountered scenarios that required me to duplicate a design. For example, if you ever wrote applications in WPF, you may remember the INotifyPropertyChanged interface. It's a simple interface that has only one member: an event called PropertyChanged. You implement this interface so users of a particular object can receive notifications when a property value changes via the PropertyChanged event. However, the amount of boilerplate code you must write to implement this interface correctly is repetitive and tedious. You need to implement a property setter to determine if the incoming value is different from the current one, and, if so, raise the PropertyChanged event with the right property name. **Listing 1** demonstrates an example of implementing INotifyPropertyChanged.

This is where a source generator makes your life simpler. You can create a generator that creates all the supporting code for property change notification with just one line of code. There's an example of a source generator from Microsoft that does just that—please refer to the "Source Generator Cookbook" article in the Resources table at the end of this article. In that example, you add one attribute to a field called AutoNotifyAttribute and the source generator takes care of everything. The following code snippet shows how simple it is to support INotifyPropertyChanged with this attribute:

```
public partial sealed class Customer
{
  [AutoNotify]
  private string? name;
}
```

Another situation where generators can help is with performance. There are numerous areas in software development where code examines assemblies and, based on the members within that assembly, it reacts accordingly. For example, a unit testing framework like NUnit typically looks for methods marked with a specific attribute—TestAttribute—and it invokes those methods at runtime. Mocking libraries, object mappers—all of these use runtime inspection and code generation techniques like reflection and dynamic expression compilation. Although these approaches are novel and provide useful features, there can be a performance penalty doing these computations while code is examined, and the dynamic code executes. If this dynamic code was generated at compile-time, it's possible that the result will be faster. This generated code can also be examined with other tools, like ahead-of-time compilation or AOT, for further optimizations. A recent example of this in .NET itself is regular expressions. You use the GeneratedRegexAttribute to specify a regular expression in .NET 7, and a source generator looks for the existence of this attribute, creating the necessary code to implement the regular expression. The "Regular Expression Improvements in .NET 7" article in the Resources table covers this new RegEx approach in detail.

Now that I've covered the essentials of source generators, let's see what it takes to write one. In the next section, I'll write a source generator that automatically adds Deconstruct() methods for the types defined in a project.

Building a Source Generator: AutoDeconstruct

In this section, I'll cover the implementation of a source generator I've created called AutoDeconstruct. You can find a link to the GitHub repository in the Resources table at the end of this article. This generator adds object deconstruction for your type definitions for free. By adding this source generator to your projects, you can deconstruct any type in that project.

Let's start by defining what object deconstruction is and how that relates to writing a source generator. Note that although I have code snippets in this article, I highly recommend you visit my GitHub repository to view the code in its entirety (https://github.com/jasonbock/AutoDeconstruct). Generators can be quite complicated to implement, and for brevity's sake, I've only shown key aspects of the generator in the code listings. Visit the repository link to see the full implementation.

Generator Goals

It's important to spend time to determine what the generator is going to do before you write any code to implement

the generator. In this case, this means you need to understand the concept of object deconstruction and how it works in C#, including cases that you may not know of, or use, in your code. Object deconstruction is a way to split the state of an object into distinct parts. These deconstructed parts do not need to match the number of or explicitly map to the fields and/or properties on an object. It's up to you to determine what state is meaningful to provide in deconstruction. With deconstruction, you'll add a Deconstruct() method to the type, as seen in the following code snippet:

```
public class Person
{
  public void Deconstruct(
    out int age, out string name) =>
      (age, name) =
        (this.Age, this.Name);

  public required int Age { get; init; }
  public required string Name { get; init; }
}
```

A Deconstruct() method must be named using the word "Deconstruct," it must return void, and all of its parameters must be Out parameters. Furthermore, method overloading with deconstruct methods doesn't follow the typical rules of methods in C#. Namely, Deconstruct() methods are overloaded purely by the number of parameters. It doesn't matter what the types of these parameters are or the order in which the parameters are listed.

With an object deconstructor in place, you can retrieve the state of the object using a tuple. This is shown in the next code snippet:

```
var person = new Person
{
  Age = 25,
  Name = "Joe Smith"
};

var (age, name) = person;
Console.WriteLine($"{age}, {name}");
```

Although you may have heard of object deconstruction and Deconstruct() methods, you may not have been aware that Deconstuct() methods can be extension methods. This can be useful if you want to provide a custom deconstruction for a type where you don't have access to the source code. The following code snippet is an alternative approach to declaring a deconstructor, where object deconstruction is done with an extension method:

```
public static class PersonExtensions
{
  public static void Deconstruct(
    this Person self,
    out int age, out string name) =>
      (age, name) =
        (self.Age, self.Name);
}
```

This is important, because the source generator should only generate a Deconstruct() if one doesn't currently exist for a given type. If you don't consider extension methods, your generator may do more work than necessary.

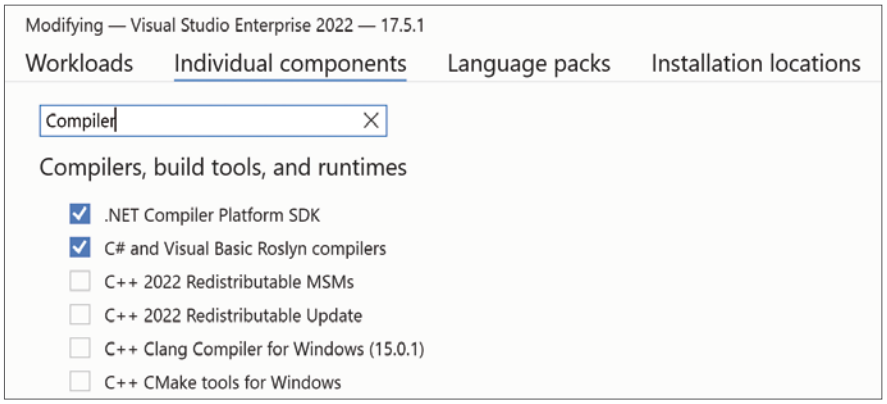


Figure 3: Installing the Compiler SDK in Visual Studio

In general, with source generators, I strongly recommend spending the time thinking about what the generator needs to find in code and all the possible ways this can be done in C#. Some language features are done in multiple ways, and if you don't consider all of them, your generator may not act as expected by the user. For example, although you can add a Deconstruct() extension method for a type, it's arguably unusual to create a Deconstruct() extension method for a type in your current project where you have access to the code. However, this is allowed in C# and you must consider it as a valid possibility. If you don't, you may end up creating a Deconstruct() method that already exists somewhere in the project as an extension method.

For this generator, look for types that have accessible properties and generate a Deconstruct() method if one doesn't exist with the same number of parameters, either on that type definition, or as an extension method. In the next section, I'll cover tools that you can use to understand the syntax tree in the Compiler API.

Determining Targets in Source

Although every developer uses a compiler or an interpreter every day when they're writing code, very few have written such a tool. They are complex beasts that require taking individual characters in a text file to executable code. Every language has different features and nuances, but virtually all of them will, at some point, create a syntax tree to represent that text. Looking at the syntax tree that's provided by the Compiler API can be daunting at first glance. It's frustrating when you're trying to find that node that maps to something you've written as code.

There are a couple of tools you can use to assist you in finding your starting points. I'm a Visual Studio user at heart, and in Visual Studio, you can install the .NET Compiler Platform SDK from the Visual Studio Installer. The easiest way to find this SDK is to modify the current Visual Studio installation, select **Individual components**, and search for Compiler. **Figure 3** is a screenshot of this installer screen:

Once you've installed this SDK, you'll get a number of available analyzer and refactoring project templates. This component also adds a window called the Syntax Visualizer. You can launch this window by traversing the View > Other Windows > Syntax Visualizer menu hierarchy. **Figure 4** shows what this looks like:

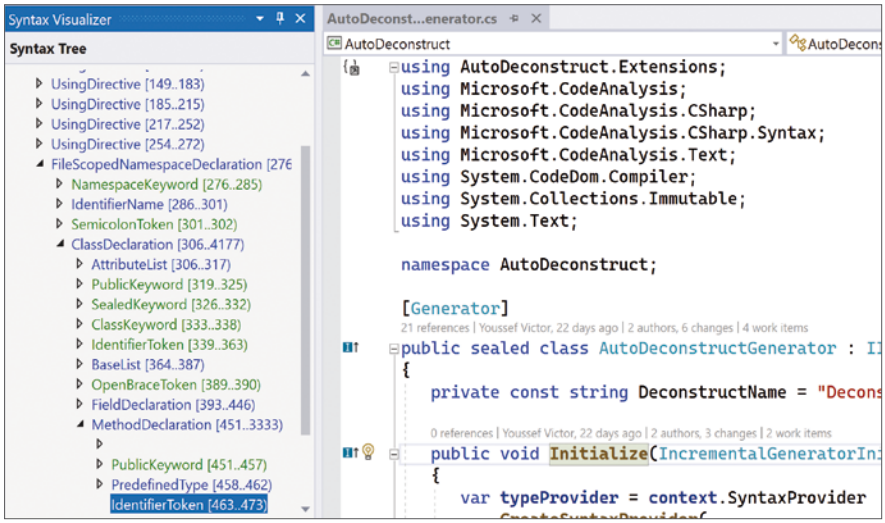


Figure 4: Syntax Visualizer in Visual Studio

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, visit www.codestaffing.com.

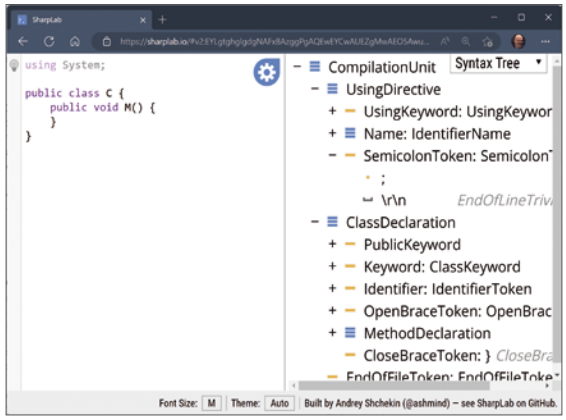


Figure 5: The syntax tree in sharplab.io

I can't demonstrate this tool dynamically in a magazine article—hopefully **Figure 4** gives you a feel for what it does. If you click on a piece of code in Visual Studio, the Syntax Visualizer moves to the corresponding node or token in the tree. Conversely, you can click anywhere in the tree and the related code is highlighted. Furthermore, information about the node object is displayed. For example, when you click on a method, you'll discover that it's a MethodDeclaractionSyntax node. This is valuable knowledge, as you don't have to know every type that exists in a syntax tree. Just write some code, open the visualizer, and see how your code is represented in a syntax tree. The more you use this visualizer, the more you'll discover what can exist in a tree and where it lives.

There's also a wonderful website called sharplab.io, which lets you explore C# code using experimental Compiler API branches and intermediate language (or IL) views. It also has a syntax tree visualizer, as shown in **Figure 5**.

I've found that I use both—it depends on the scenario. Sometimes, using sharplab.io is convenient when I'm thinking about writing a generator, but I don't have Visual Studio open. If I'm already in Visual Studio, I can write a piece of code in a file and quickly see how that's represented in the Syntax Visualizer.

For AutoDeconstruct, I've found that I need to look for TypeDefinitionSyntax nodes as my starting point. Then, I'll look for MethodDefinitionSyntax nodes within that type to see if it has Deconstruct() methods with a set of parameters equal to the number of accessible PropertyDefinitionSyntax nodes. I also need to look at individual MethodDefinitionSyntax nodes to find Deconstruct() extension methods. In the next section, you'll see how the AutoDeconstruct generator is defined.

Filtering Syntax Nodes

Now that you have a conceptual model in place for AutoDeconstruct, let's explore how the generator works. Generators are housed in class library projects that are typically deployed as NuGet packages for other projects to reference. A source generator implements the IIncrementalGenerator interface, which can be found by referencing the Microsoft.CodeAnalysis.CSharp NuGet package. This interface has one member: the Initialize() method. The following code snippet shows the definition of IIncrementalGenerator:

```
namespace Microsoft.CodeAnalysis
{
  public interface IIncrementalGenerator
  {
    void Initialize(
      IncrementalGeneratorInitializationContext
        context);
  }
}
```

Note that the first release of source generators defined an interface called ISourceGenerator. Although it's still available, you should ignore this interface and always use IIncrementalGenerator.

The Initialize() method lets you plug into an incremental compilation pipeline, where you can make step-wise decisions to determine if you need to create code based on the contents of a syntax tree. The incremental approach lets the pipeline optimize the operations if the results from the pipeline haven't changed from a previous invocation. Keep in mind that your generator will be invoked numerous times as developers change code in a project. However, these changes may not require your generator to regenerate code. If you can show the compiler that the content you're looking for is the same as before, the pipeline can catch this and determine that no further steps need to be taken. This means you should create your own object model that supports equality. This model should contain just enough information for your generator.

With AutoDeconstruct, you need to interrogate type definitions. This means that you need to filter for TypeDeclarationSyntax nodes. You could try to look at all the parts of a TypeDeclarationSyntax node, but syntax nodes can be a bit difficult to work with. Using compilation symbols provides more semantics around a given node. The next snippet shows how you can translate the current node in the pipeline to a symbol:

```
var typeProvider = context.SyntaxProvider
  .CreateSyntaxProvider(
    static (node, _) =>
      node is TypeDeclarationSyntax,
    static (context, token) =>
```

```
  {
    var symbol = context.SemanticModel
      .GetDeclaredSymbol(
        (TypeDeclarationSyntax)context.Node,
          token);
```

In the Compiler API, you can use the semantic model, which provides rich information about a syntax node. In this case, if the node is a TypeDeclarationSyntax, and you can load an INamedTypeSymbol instance for that node through the semantic model, you can investigate the definition of that type to see if you need to generate an object deconstructor. This is shown in **Listing 2**.

The GetAccessibleProperties() method is an extension method I wrote to find all of the public, readable properties on an INamedTypeSymbol instance. Note that the type of the return value is EquatableArray<PropertySymbolModel>. You can't use a collection type like an array because you need to be able to determine if the contents of that collection are "equal." This custom EquatableArray<> type handles that scenario for you. This type comes from the PolySharp project, which is listed in the Resources table at the end of the article.

Our PropertySymbolModel definition contains just two values: the name of the property and the fully-qualified name of the property's type. The latter definition is shown in the next code snippet:

```
internal record PropertySymbolModel(
  string Name, string TypeFullyQualifiedName);
```

Because records automatically define equality for you, this is all that's needed to create the custom data model. Also, if you need to reference the name of a type in your generated code, you should always use fully-qualified names, like "global:: System.String". This eliminates the possibility of name collisions, although it can make the generated code a bit hard to read. That said, developers typically won't read your generated code and will just use it for what it's designed to do.

Once you get all of the accessible properties, do a search on the type for IMethodSymbol instances; specifically, you're looking for methods that aren't object deconstructors with the same number of Out parameters. If you don't have that, you return a TypeSymbolModel instance—another custom record defined to contain just enough information to build a new object deconstructor. Let's see how that code is generated in the next section.

Generating Code

The code generation phrase of a source generator is arguably simpler, in that all you need to do is create C# code in a string. How you want to do that is completely up to you, so long as the product is a string that passed into a SourceText object. My preference is to use the IndentedTextWriter class, a custom TextWriter type from the System.CodeDom namespace. The IndentedTextWriter makes it easy to define indents and outdents in the lines of text. You may choose to use a templating engine like Scriban. As long as the result is a string, use whatever production strategy you're comfortable with.

To participate in the output portion of the pipeline, you define a callback method, which is passed to RegisterSourceOutput() on the IncrementalGeneratorInitializationContext object. **Listing 3** shows what happens in the CreateOutput() callback method.

The central point of CreateOutput() is the call to Build() on AutoDeconstructBuilder. This is where all of the generated code is built. For example, the following code snippet illustrates how all of the Out parameters for the object deconstructor are created:

```
var outParameters = string.Join(", ",
  properties.Select(p =>
    $"out {p.TypeFullyQualifiedName} " +
    $"@{p.Name.ToCamelCase()}"
  ));
```

As the Deconstruct() method you're generating is an extension method, the type that you're extending may be a reference type, which means that you might get a null value. With a generator, you can create code tailored for specific scenarios. This is demonstrated in the following code snippet:

```
if (!type.IsValueType)
{
  writer.WriteLine($"global::" +
    "System.ArgumentNullException" +
    $".ThrowIfNull(@{namingContext["self"]});");
}
```

In this case, if the target type isn't a value type, you can generate a null check such that you'll throw an ArgumentNullException if the Self parameter is null.

Listing 2: Using Symbols for Interrogation

```
var attributeName =
  "AutoDeconstruct.NoAutoDeconstructAttribute";

if (symbol is null)
{
  return null;
}
else if (symbol is INamedTypeSymbol typeSymbol &&
  typeSymbol.GetAttributes().Any(a =>
    a.AttributeClass?.ToDisplayString() ==
      attributeName))
{
  return null;
}

var accessibleProperties =
  symbol.GetAccessibleProperties();

if (accessibleProperties.IsEmpty ||
  symbol.GetMembers().OfType<IMethodSymbol>()
    .Any(m => m.Name ==
      AutoDeconstructGenerator.DeconstructName &&
      !m.IsStatic && m.Parameters.Length ==
        accessibleProperties.Length &&
      m.Parameters.All(
        p => p.RefKind == RefKind.Out)))
{
  // There is an existing instance deconstruct.
  return null;
}

return new TypeSymbolModel(
  symbol.ContainingNamespace.ToString(),
  symbol.Name, symbol.GetGenericParameters(),
  symbol.GetFullyQualifiedName(),
  symbol.GetConstraints(), symbol.IsValueType,
  accessibleProperties);
```

Listing 3: CreateOutput() Callback

```
private static void CreateOutput(
  ImmutableArray<TypeSymbolModel> types,
  ImmutableArray<TypeSymbolModel> excludedTypes,
  SourceProductionContext context)
{
  if (types.Length > 0)
  {
    using var writer = new StringWriter();
    using var indentWriter =
      new IndentedTextWriter(writer, "\t");
    indentWriter.WriteLines(
      """
      #nullable enable

      """);

    var wasBuildInvoked = false;

    foreach (var type in types.Distinct())
    {
      var accessibleProperties =
        type.AccessibleProperties;

      if (!excludedTypes.Contains(type))
      {
        AutoDeconstructBuilder.Build(
          indentWriter, type,
          accessibleProperties);
        wasBuildInvoked = true;
      }
    }

    if (wasBuildInvoked)
    {
      context.AddSource("AutoDeconstruct.g.cs",
        SourceText.From(writer.ToString(),
        Encoding.UTF8));
    }
  }
}
```

Listing 4: Writing a Generator Test

```
var code =
  """
  using System;

  namespace TestSpace
  {
    public struct Test
    {
      public string? Namespace { get; set; }
    }
  }
  """;

var generatedCode =
  """
  #nullable enable

  namespace TestSpace
  {
    public static partial class TestExtensions
    {
      public static void Deconstruct(
        this global::TestSpace.Test @self,
        out string? @namespace)
      {
        @namespace = @self.Namespace;
      }
    }
  }

  """;

await TestAssistants.RunAsync(code,
  new[] { (typeof(AutoDeconstructGenerator),
  "AutoDeconstruct.g.cs", generatedCode) },
  Enumerable.Empty<DiagnosticResult>());
```

Also, notice that I have a VariableNamingContext object in my generator code. This is a custom type I've used in a number of source generators I've created to guarantee unique variable names. When you're generating code, you may need to create names for parameters or variables. You can try to come up with something unique, like using the string value of a new GUID, but most of the time, your choice of variable names won't collide. There's always a chance that a duplication will occur. This type provides either the desired name or a unique name for the desired name. Keep in mind that this only works within the scope that the VariableNamingContext is used. If you want to use this approach, you need to ensure that it's being used within the right scope, like a method body.

Once you have all of the code generated, call AddSource() on the SourceProductionContext object. Your generated code is now available to be used by the referencing project. Before you publish the NuGet package so others can use it, you should test the generator. I'll show you how to do this testing in the next section.

Testing Generators

Testing code is a standard practice I follow on any project I work on. With source generators, there are libraries produced by Microsoft that make generator testing easier. The AutoDeconstruct.Tests project uses the Microsoft.CodeAnalysis.CSharp.SourceGenerators.Testing.NUnit library to test generators for NUnit tests (there are libraries for xUnit and MSTest as well). **Listing 4** shows a test where you can specify the generated code you expect to see when your generator runs.

In this test, you expect that a code file should be generated with the content specified in generatedCode. You can also write tests where you can state that a set of diagnostics should be produced.

You can also write tests that let you write code that uses the generated code directly. These tests are defined in the AutoDeconstruct.IntegrationTests project. You need to reference your class library project with the OutputItemType attribute set to Analyzer, as shown in the following code snippet:

```
<ProjectReference
  Include="AutoDeconstruct.csproj"
  OutputItemType="Analyzer" />
```

When you do this, your generator runs directly against the code within that test project. **Listing 5** shows how the generated object deconstructor is used to extract object state.

I find it very satisfying when I get to the point where I can use the outputs of my source generator. It illustrates the power and flexibility of source generators. In this case, it's so nice to see where a developer doesn't have to write an object deconstructor if they don't want to. Furthermore, as an implementor of a generator, it's really nice to be able to view the generated code. If you want to see the results of your source generator, you can go to the Analyzers node in the Dependencies section of a project in the Solution Explorer window of Visual Studio. **Figure 6** shows this list.

Listing 5: Writing Integration Tests

```
public static class NoDeconstructExistsTests
{
  [Test]
  public static void RunDeconstruct()
  {
    var id = Guid.NewGuid();
    var value = 3;

    var target = new NoDeconstructExists
    {
      Id = id,
      Value = value
    };
    var (newId, newValue) = target;

    Assert.Multiple(() =>
    {
      Assert.That(newId, Is.EqualTo(id));
      Assert.That(newValue, Is.EqualTo(value));
    });
  }
}

public sealed class NoDeconstructExists
{
  public Guid Id { get; set; }
  public int Value { get; set; }
}
```

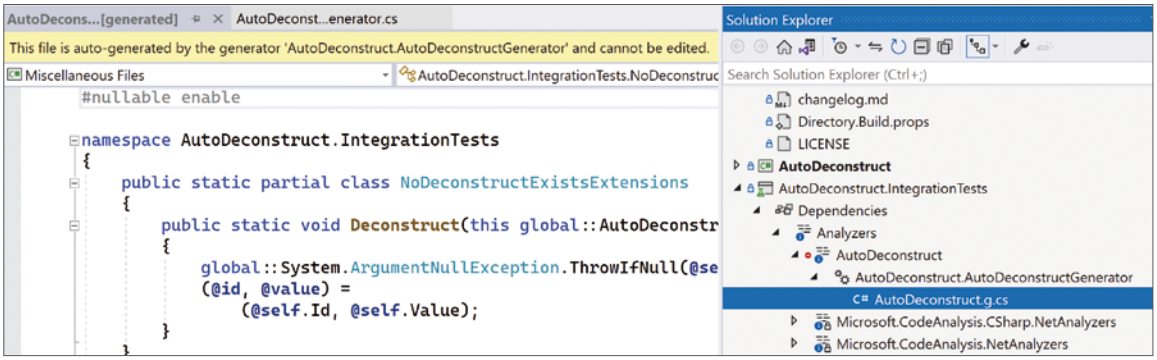


Figure 6: Generated code list in Solution Explorer

By default, these files are not saved to storage. If you want to persist the generated code files to disk, you set the EmitCompilerGeneratedFiles project setting to true. You can also specify a folder to save the files via the CompilerGeneratedFilesOutputPath property.

Not only can you view the generated code, but you can also step into this code in a debugging session. This is a tremendous advantage for a generator author compared to the other techniques I mentioned in the "The What and Why of Source Generators" section. Trying to figure out what isn't working correctly with Reflection or IL generation at runtime can be very tricky and difficult. With a source generator, it's the same as any other code in that project. If the generator emits incorrect code, the compiler finds that issue and I won't be able to build an assembly. This makes finding and fixing bugs arguably quicker.

One technicality with this second test approach is that Visual Studio tends to hold onto the assembly that contains your source generator, even after you've made changes to your generator. You may need to restart Visual Studio if you want to test changes in your generator. I've found that I'll write a number of unit tests first and only write integration tests when I feel like my changes are good. This minimizes the number of times I'd need to restart Visual Studio. It's an unfortunate aspect of testing generators that's due to how Visual Studio loads analyzer-based projects in other projects, but I've found that this is a minor inconvenience.

Start the Journey

In this article, I've covered source generators in C#. Generators can be used to reduce the amount of code a developer has to write or to define performant code at compile-time. Writing generators may seem daunting at first, and I'll freely admit that it can be quite frustrating to implement one. The amount of documentation on source generators is anemic at times and it can be confusing to figure out how to get to the point where you generate the code you want to. That said, source generators can be a positive addition to your coding experience. I encourage you to dive into source generators. Who knows: You may end up writing one that simplifies the code you write, or, more exactly, the code you no longer must write.

"Refactoring," by Martin Fowler	https://martinfowler.com/books/refactoring.html
The .NET Compiler Platform SDK	https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/
Source Generators Cookbook	https://github.com/dotnet/roslyn/blob/main/docs/features/source-generators.cookbook.md
Incremental Generators	https://github.com/dotnet/roslyn/blob/main/docs/features/incremental-generators.md
Regular Expression Improvements in .NET 7	https://devblogs.microsoft.com/dotnet/regular-expression-improvements-in-dotnet-7/
AutoDeconstruct Code Repository	https://github.com/JasonBock/AutoDeconstruct
PolySharp	https://github.com/Sergio0694/PolySharp
Scriban	https://github.com/scriban/scriban

Table 1: Resources

Table 1 provides a list of helpful resources on this subject.

Jason Bock

Using DuckDB for Data Analytics

Very often, when people talk about data analytics, Pandas is the first library that comes to mind. And, of course, in more recent times, Polars is fast gaining traction as a much faster and more efficient DataFrame library. Despite the popularity of these libraries, SQL (Structured Query Language) remains the language that most developers are familiar with. If your data is stored in SQL-supported databases, SQL is one of the easiest and most natural ways for you to retrieve your data.

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

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

Recently, with Python becoming the lingua franca of data science, most attention has shifted to techniques on how to manipulate data in tabular format (most notably stored as a DataFrame object). However, the real lingua franca of data is actually SQL. And because most developers are familiar with SQL, isn't it more convenient to manipulate data using SQL? This is where DuckDB comes in.

In this article, I'll explain what DuckDB is, why it's useful, and, more importantly, I want to walk you through examples to demonstrate how you can use DuckDB for your data analytics tasks.

What Is DuckDB?

DuckDB is a Relational Database Management System (RDBMS) that supports the Structured Query Language (SQL). It's designed to support Online Analytical Processing (OLAP), and is well suited for performing data analytics. DuckDB was created by Hannes Mühleisen and Mark Raasveldt, and the first version released in 2019.

Unlike traditional database systems where you need to install them, DuckDB requires no installation and works in-process. Because of this, DuckDB can run queries directly on Pandas data without needing to import or copy any data. Moreover, DuckDB uses vectorized data processing, which makes it very efficient—internally, the data is stored in columnar format rather than row-format (which is commonly used by databases systems such as MySQL and SQLite).

Think of DuckDB as the analytical execution engine that allows you to run SQL queries directly on existing datasets such as Pandas DataFrames, CSV files, and traditional databases such as MySQL and Postgres. You can focus on using SQL queries to extract the data you want.

Why DuckDB?

Today, your dataset probably comes from one or more of the following sources:

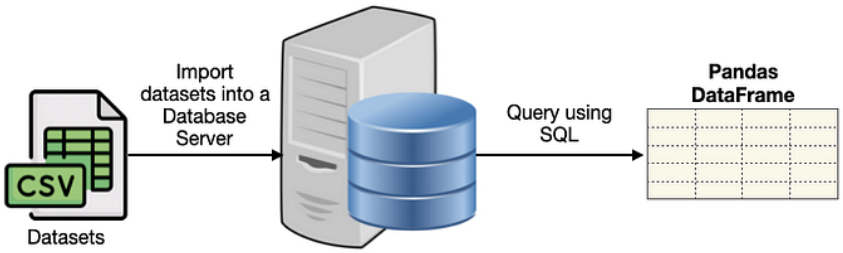


Figure 1: Import your data into a database server before you can use SQL to query your data.

- CSV files
- Excel spreadsheets
- XML files
- JSON files
- Databases

If you want to use SQL to manipulate your data, the typical scenario is to first load the dataset (such as a CSV file) into a database server. You then load the data into a Pandas DataFrame through an application (such as Python) using SQL (see **Figure 1**).

DuckDB eliminates the need to load the dataset into a database server and allows you to directly load the dataset using SQL (see **Figure 2**).

Once the DataFrame is loaded, you can use DuckDB and SQL to further slice and dice the DataFrame (see **Figure 3**).

Data Analytics Using the Insurance Dataset

The best way to understand DuckDB is through examples. For this, I'll use the insurance dataset located at https://www.kaggle.com/datasets/teertha/ushealthinsurancedataset?resource=download. The insurance dataset contains 1338 rows of insured data, where the insurance charges are given against the following attributes of the insured: age, sex, BMI, number of children, smoker, and region. The attributes are a mix of numeric and categorical variables.

Loading the CSV File into Pandas DataFrames

Let's examine the insurance dataset by loading the insurance.csv file into a Pandas DataFrame:

```
import pandas as pd

df_insurance = pd.read_csv("insurance.csv")
display(df_insurance)
```

Figure 4 shows how the DataFrame looks.

The various columns in the DataFrame contain the various attributes of the insurance customer. In particular, the **charges** column indicates the individual medical costs billed by health insurance (payable by the insured).

Creating a DuckDB Database

Before you can create a DuckDB database, you need to install the **duckdb** package using the following command:

```
!pip install duckdb
```

To create a DuckDB database, use the **connect()** function from the **duckdb** package to create a connection (a **duckdb.DuckDBPyConnection** object) to a DuckDB database:

```
import duckdb
conn = duckdb.connect()
```

You can then register the DataFrame that you loaded earlier with the DuckDB database:

```
conn.register("insurance", df_insurance)
```

The **register()** function registers the specified DataFrame object (**df_insurance**) as a virtual table (**insurance**) within the DuckDB database.

Directly Loading the CSV into a Pandas DataFrame Using DuckDB

Instead of loading a DataFrame manually and then registering with the DuckDB database, you can also use the connection object to read a CSV file directly, like this:

```
conn = duckdb.connect()

df = conn.execute('''
    SELECT
        *
    FROM read_csv_auto('insurance.csv')
''').df()

conn.register("insurance", df)
```

In the above code snippet:

- I used a SQL statement with the **read_csv_auto()** function to read a CSV file. The **execute()** function takes in this SQL statement and executes it.
- The **df()** function converts the result of the **execute()** function into a Pandas DataFrame object.
- Once the DataFrame is obtained, use the **register()** function to register it with the DuckDB database.

You can confirm the number of tables in the DuckDB by using the **SHOW TABLES** SQL statement:

```
display(conn.execute('SHOW TABLES').df())
```

The result is shown in **Figure 5.**

To fetch rows from the insurance table, you can directly use a SQL statement using the **execute()** function:

```
df = conn.execute(
        "SELECT * FROM insurance").df()
df
```

The output will be the same as shown earlier in **Figure 4**.

Performing Analytics using DuckDB

Let's now perform some useful data analytics using the **insurance** table in the DuckDB. First, I want to visualize the distribution of charges based on sex:

```
import seaborn as sns
import matplotlib.pyplot as plt
f, ax = plt.subplots(1, 1, figsize=(5, 3))
df = conn.execute('''
    SELECT
        *
```

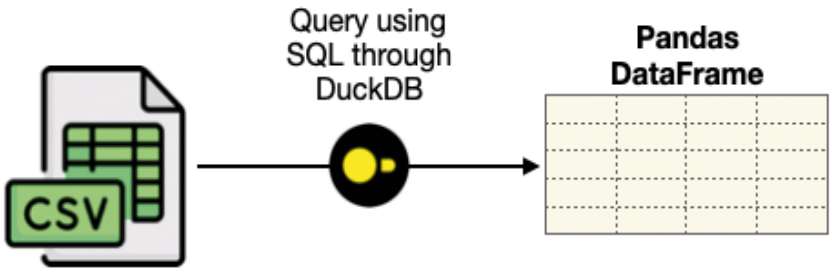


Figure 2: You can use DuckDB to directly query your dataset using SQL.

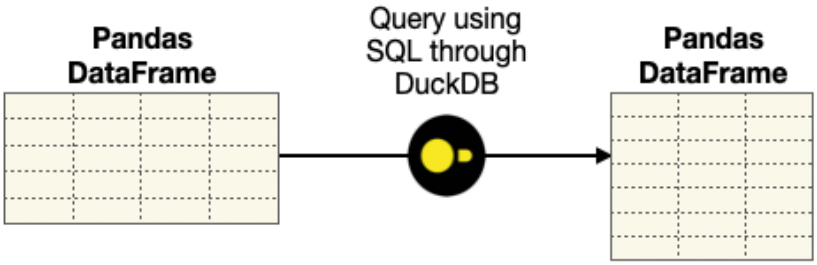


Figure 3: You can also use DuckDB to query Pandas' DataFrames using SQL.

	age	sex	bmi	children	smoker	region	charges
0	19	female	27.900	0	yes	southwest	16884.92400
1	18	male	33.770	1	no	southeast	1725.55230
2	28	male	33.000	3	no	southeast	4449.46200
3	33	male	22.705	0	no	northwest	21984.47061
4	32	male	28.880	0	no	northwest	3866.85520
...	...	...	...	...	...	...	...
1333	50	male	30.970	3	no	northwest	10600.54830
1334	18	female	31.920	0	no	northeast	2205.98080
1335	18	female	36.850	0	no	southeast	1629.83350
1336	21	female	25.800	0	no	southwest	2007.94500
1337	61	female	29.070	0	yes	northwest	29141.36030

1338 rows × 7 columns

Figure 4: The insurance dataset loaded as a Pandas DataFrame

	name
0	insurance

Figure 5: The DuckDB has one associated table named insurance.

```
    FROM insurance
''').df()
ax = sns.barplot(x = 'region',
                 y = 'charges',
                 hue = 'sex',
                 data = df,
                 palette = 'cool',
                 errorbar = None)
```

The above code snippet uses the Seaborn package to plot a bar plot that shows the various insurance charges for each gender in each of the four regions (see **Figure 6** for the output).

Overall, men tend to have higher medical insurance cost for all regions, except the northwest region.

I'm also interested in visualizing the distribution of insurance charges for people based in the southwest region,

based on the number of children a person has. I can do the following:

```
import seaborn as sns
import matplotlib.pyplot as plt
f, ax = plt.subplots(1, 1, figsize=(7, 5))
df = conn.execute('''
    SELECT
        *
    FROM insurance
    WHERE region = 'southwest'
''').df()

ax = sns.barplot(x = 'region',
                 y = 'charges',
                 hue = 'children',
                 data = df,
                 palette = 'Set1',
                 errorbar = None)
```

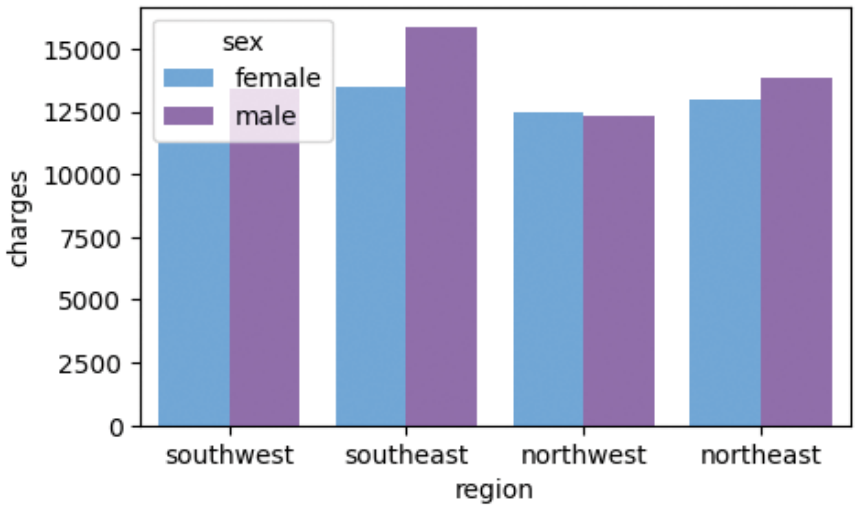


Figure 6: The distribution of charges for customers in each region based on sex

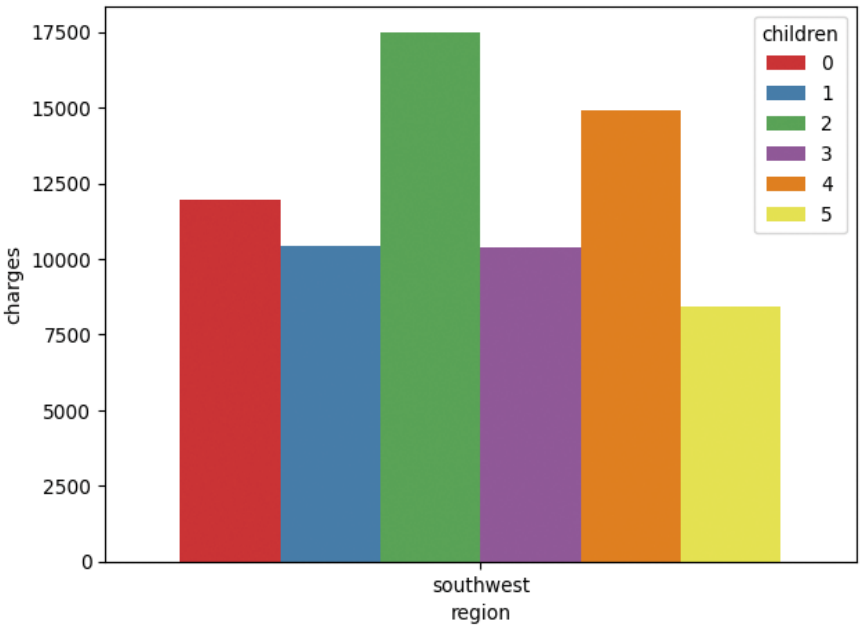


Figure 7: Charges for customers in the southwest region based on number of children

As you can see from **Figure 7**, for the southwest region, the mean insurance charges for a person with two children is close to $17,500, which is the highest. The lowest mean insurance charges are for people with five children.

Based on the value in the DataFrame **df** (which contains all the people in the southwest), you can plot linear models to examine the relationships between the various attributes (such as age, bmi, smoker, children) against the insurance charges:

```
ax = sns.lmplot(x = 'age',
                y = 'charges',
                data = df,
                hue = 'smoker',
                palette = 'Set1')
ax = sns.lmplot(x = 'bmi',
                y = 'charges',
                data = df,
                hue = 'smoker',
                palette = 'Set2')
ax = sns.lmplot(x = 'children',
                y = 'charges',
                data = df,
                hue = 'smoker',
                palette = 'Set3')
```

Figure 8 shows how the insurance charges relates to age, bmi, and number of children, and whether they are smokers or not.

Generally, smokers have to pay much higher insurance charges compared to non-smokers. In the case of smokers, as the age or BMI increases, the amount of insurance charges increases proportionally. The number of children a customer has does not really affect the insurance charges.

Next, I want to visualize the mean insurance charges for all the people in the southeast and southwest regions, so I modify my SQL statement and plot a bar plot:

```
df = conn.execute('''
    SELECT
        region,
        mean(charges) as charges
    FROM insurance
    WHERE region = 'southwest' or
          region = 'southeast'
    GROUP BY region
''').df()

f, ax = plt.subplots(1, 1, figsize=(5, 3))
ax = sns.barplot(x = 'region',
                 y = 'charges',
                 data = df,
                 palette = 'Reds')
```

Figure 9 shows the plot.

Overall, the insurance charges are higher for people in the southeast region than those in the southwest region.

Next, I want to see the proportion of smokers for the entire dataset:

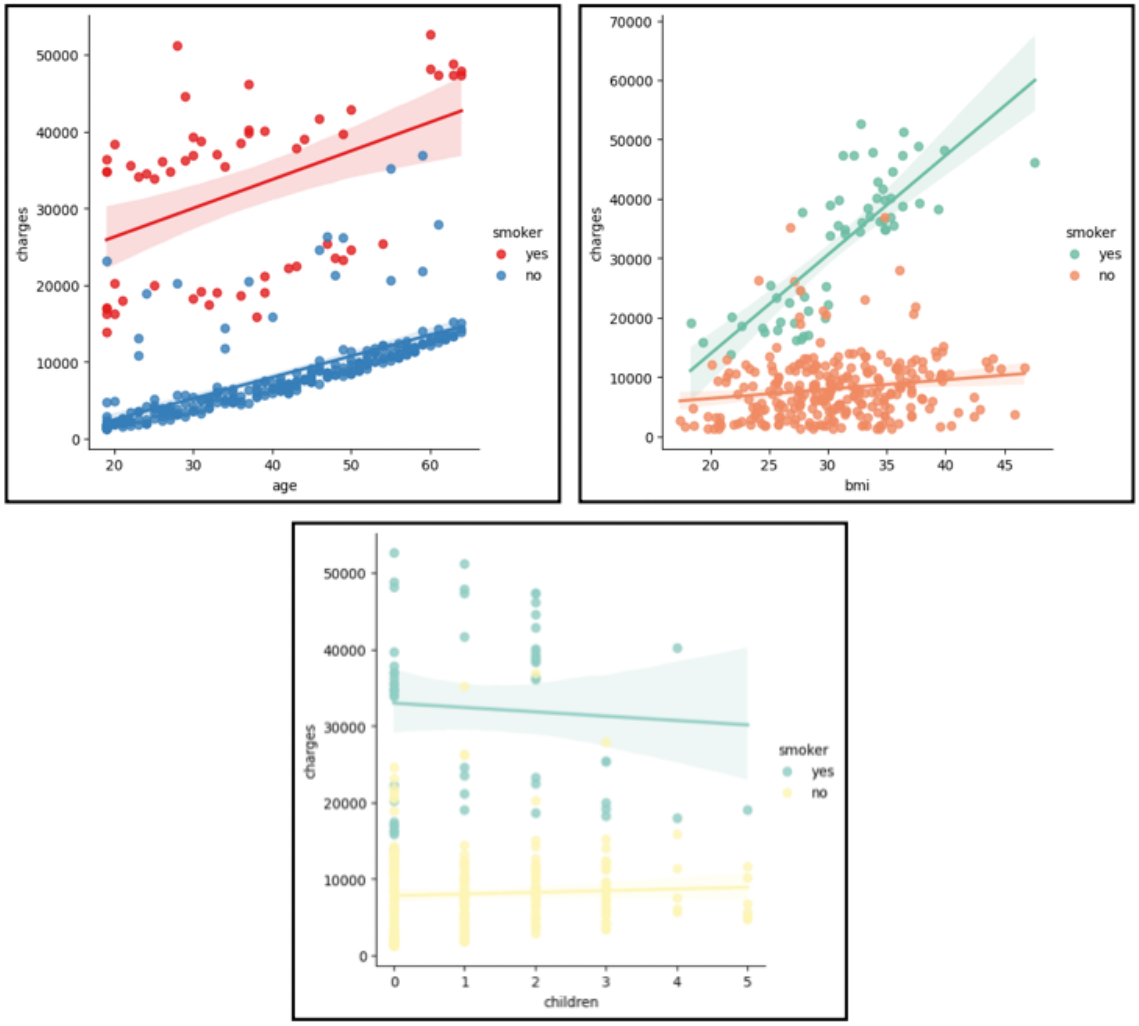


Figure 8: The relationships between the various attributes with respect to charges

```
palette_color = \
    seaborn.color_palette('pastel')
plt.figure(figsize = (5, 5))

df = conn.execute('''
    SELECT
        count(*) as Count,
        smoker
    FROM insurance
    GROUP BY smoker
    ORDER BY Count DESC
''').df()

plt.pie('Count',
        labels = 'smoker',
        colors = palette_color,
        data = df,
        autopct='%.0f%%',)

plt.legend(df['smoker'], loc="best")
```

The pie chart in **Figure 10** shows the proportion of smokers (20%) vs. non-smokers (80%).

It would be more useful to be able to display the numbers of smokers and non-smokers alongside their percentages.

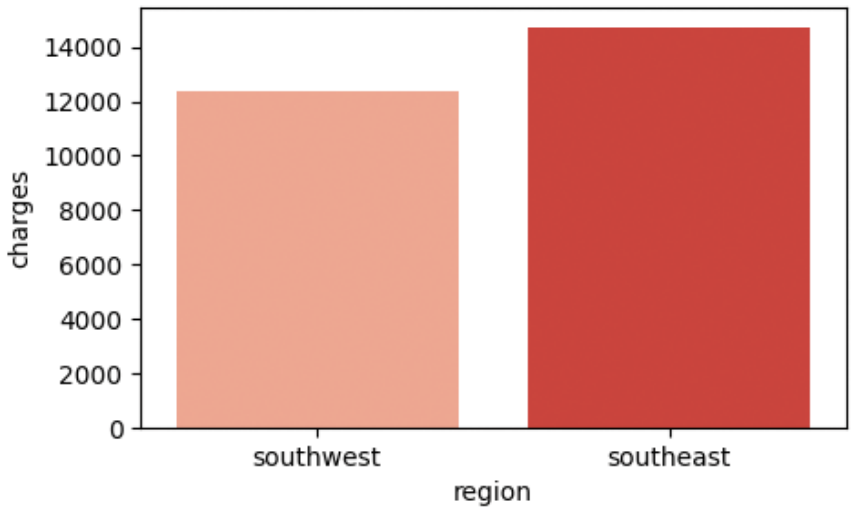


Figure 9: The charges for customers in the southwest and southeast regions

So let's add a function named **fmt()** to customize the labels displayed on each pie on the pie chart:

```
# sum up total number of people
total = df['Count'].sum()
```

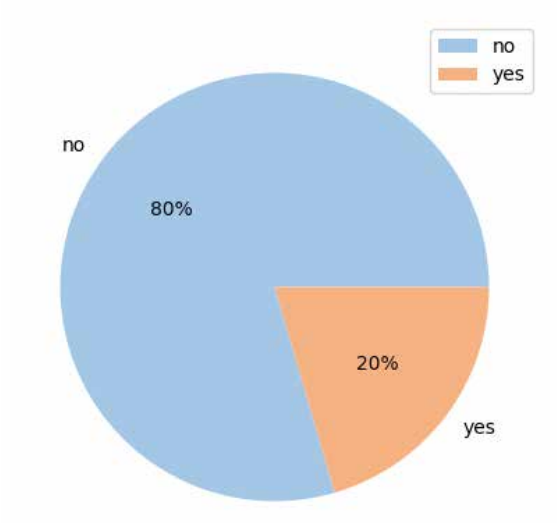


Figure 10: The proportion of smokers vs. non-smokers

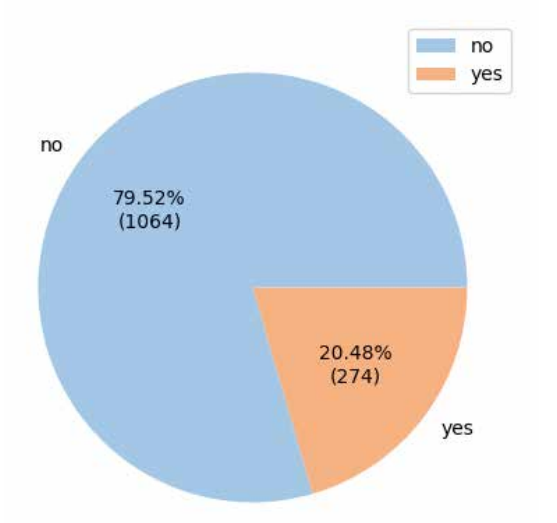


Figure 11: The updated pie chart with the numbers on each pie

```
def fmt(x):
    # display percentage followed by number
    return '{:.2f}%\n({:.0f})'.format(
        x, total * x / 100)

palette_color = \
    seaborn.color_palette('pastel')
plt.figure(figsize = (5, 5))

plt.pie('Count',
        labels = 'smoker',
        colors = palette_color,
        data = df,
        autopct = fmt)  # call fmt()

plt.legend(df['smoker'], loc="best")
```

Figure 11 shows the updated pie chart.

JSON Ingestion

One of the new features announced in the recent DuckDB release (version 0.7) is the support for JSON Ingestion. Basically, this means that you can now directly load JSON files into your DuckDB databases. In this section, I'll show you how to use this new feature by showing you some examples.

For the first example, suppose you have a JSON file named **json0.json** with the following content:

```
[
  {
    "id": 1,
    "name": "Abigail",
    "address": "711-2880 Nulla St.
               Mankato Mississippi 96522"
  },
  {
    "id": 2,
    "name": "Diana",
    "address": "P.O. Box 283 8562
      Fusce Rd. Frederick Nebraska 20620"
  },
  {
    "id": 3,
    "name": "Jason",
    "address": "606-3727 Ullamcorper.
      Street Roseville NH 11523"
  }
]
```

This JSON file contains an array of objects, with each object containing three key/value pairs. This is a very simple JSON file and the easiest way to read it into DuckDB is to use the **read_json_auto()** function:

```
import duckdb
conn = duckdb.connect()

conn.execute('''
    SELECT
        *
    FROM read_json_auto('json0.json')
''').df()
```

The output of the above code snippet is as shown in **Figure 12**.

If you only want the **id** and **name** fields, modify the SQL statement as follows:

```
import duckdb
conn = duckdb.connect()

conn.execute('''
    SELECT
        id, name
    FROM read_json_auto('json0.json')
''').df()
```

	id	name	address
0	1	Abigail	711-2880 Nulla St. Mankato Mississippi 96522
1	2	Diana	P.O. Box 283 8562 Fusce Rd. Frederick Nebraska...
2	3	Jason	606-3727 Ullamcorper. Street Roseville NH 11523

Figure 12: The JSON file reads as a Pandas DataFrame

You'll now only get the **id** and **name** columns (see **Figure 13**)

The **read_json_auto()** function automatically reads the entire JSON file into DuckDB. If you only want to selectively read specific keys in the JSON file, use the **read_json()** function and specify the **json_format** and **columns** attributes as follows:

```
import duckdb
conn = duckdb.connect()

conn.execute('''
    SELECT
        *
    FROM read_json('json0.json',
    json_format = 'array_of_records',
    columns = {
        id:'INTEGER',
        name:'STRING'
    })
''').df()
```

The output is the same as shown in **Figure 13**.

Suppose now the content of **json0.json** is slightly changed and now saved in another file named **json1.json**:

```
[
  {
    "id": 1,
    "name": "Abigail",
    "address": {
      "line1":"711-2880 Nulla St.
               Mankato",
      "state":"Mississippi",
      "zip":96522
    }
  },
  {
    "id": 2,
    "name": "Diana",
    "address": {
      "line1":"P.O. Box 283 8562",
      "line2":"Fusce Rd. Frederick",
      "state":"Nebraska",
      "zip":20620
    }
  },
  {
    "id": 3,
    "name": "Jason",
    "address": {
      "line1": "606-3727 Ullamcorper",
      "line2":"Street Roseville",
      "state":"NH",
      "zip": 11523
    }
  }
]
```

As you can see, the address of each person is now split into four different key/value pairs: **line1**, **line2**, **state**, and **zip**. Notice that the first person does not have line2 attribute in its address.

As usual, you can use the **read_json_auto()** function to read the JSON file:

	id	name
0	1	Abigail
1	2	Diana
2	3	Jason

Figure 13: The DataFrame now only contains the ID and name columns.

	id	name	address
0	1	Abigail	{'line1': '711-2880 Nulla St. Mankato', 'state...
1	2	Diana	{'line1': 'P.O. Box 283 8562 ', 'state': 'Nebr...
2	3	Jason	{'line1': '606-3727 Ullamcorper', 'state': 'NH...

Figure 14: The various parts of an address are all squeezed into a single column.

```
import duckdb
conn = duckdb.connect()

conn.execute('''
    SELECT
        *
    FROM read_json_auto('json1.json')
''').df()
```

Observe that the address field is now a string containing all the four key/value pairs (see **Figure 14**).

So how do you extract the key value pairs of the addresses as individual columns? Fortunately you can do so via the SQL statement:

```
import duckdb
conn = duckdb.connect()

conn.execute('''
    SELECT
        id,
        name,
        address['line1'] as line1,
        address['line2'] as line2,
        address['state'] as state,
        address['zip'] as zip,
    FROM read_json_auto('json1.json')
''').df()
```

Figure 15 now shows the key/value pairs for the addresses extracted as individual columns.

As the first person doesn't have the **line2** attribute, it has a **NaN** value in the DataFrame.

Consider another example (json2.json) where all the information you've seen in the previous examples are encapsulated within the **people** key:

```
{
  "people": [
    {
      "id": 1,
      "name": "Abigail",
      "address": {
        "line1":"711-2880 Nulla St. Mankato",
        "state":"Mississippi",
        "zip":96522
      }
    },
    {
      "id": 2,
      "name": "Diana",
```

	id	name	line1	line2	state	zip
0	1	Abigail	711-2880 Nulla St. Mankato	NaN	Mississippi	96522
1	2	Diana	P.O. Box 283 8562	Fusce Rd. Frederick	Nebraska	20620
2	3	Jason	606-3727 Ullamcorper	Street Roseville	NH	11523

Figure 15: The addresses are now split into multiple columns.

	people
0	[{'id': 1, 'name': 'Abigail', 'address': {'lin...

Figure 16: All of the details are stored in the people column.

```
    "address": {
      "line1":"P.O. Box 283 8562 ",
      "line2":"Fusce Rd. Frederick",
      "state":"Nebraska",
      "zip":20620
    }
  },
  {
    "id": 3,
    "name": "Jason",
    "address": {
      "line1": "606-3727 Ullamcorper",
      "line2":"Street Roseville",
      "state":"NH",
      "zip": 11523
    }
  }
 ]
}
```

If you try to load it using **read_json_auto()**, the people key will be read as a single column (see **Figure 16**).

```
import duckdb
conn = duckdb.connect()

import json

conn.execute('''
    SELECT
        *
    FROM read_json_auto('json2.json')
''').df()
```

To properly read it into a Pandas DataFrame, you can use the **unnest()** function in SQL:

```
import duckdb
conn = duckdb.connect()

import json

conn.execute('''
    SELECT
        p.id,
        p.name,
        p.address['line1'] as line1,
        p.address['line2'] as line2,
        p.address['state'] as state,
        p.address['zip'] as zip,
    FROM (
        SELECT unnest(people) p
        FROM read_json_auto('json2.json')
    )
''').df()
```

The JSON file is now loaded correctly, just like **Figure 15**.

Summary

If you're a hardcore SQL developer, using DuckDB is a godsend when performing data analytics. Instead of manipulating DataFrames, you could now use your SQL knowledge to manipulate the data in whatever form you want. Of course, you still need a good working knowledge of DataFrames, but the bulk of the analytics part can be performed using SQL. In addition, JSON support in the latest version of DuckDB is definitely useful. Even so, loading the JSON properly into the required shape definitely takes some getting used to. Things are still rapidly evolving and hopefully, the next version will make things even easier!

Wei-Meng Lee

CUSTOM SOFTWARE DEVELOPMENT

STAFFING TRAINING/MENTORING SECURITY

MORE THAN JUST A MAGAZINE!

Does your development team lack skills or time to complete all your business-critical software projects? CODE Consulting has top-tier developers available with in-depth experience in .NET, web development, desktop development (WPF), Blazor, Azure, mobile apps, IoT and more.

Contact us today for a complimentary one hour tech consultation. No strings. No commitment. Just CODE.

codemag.com/code

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

Authentication in Laravel: Part 1

Laravel 9 provides a built-in authentication system for securing your application. There's always a restricted area in your application that requires you to grant access to allocated users. To do so, you need to identify the user trying to access this area and check whether they're allowed access. This is called Authentication and is a general component in Laravel and web

Bilal Haidar
bhaidar@gmail.com
https://www.bhaidar.dev
@bhaidar

Bilal Haidar is an accomplished author, Microsoft MVP of 10 years, ASP.NET Insider, and has been writing for CODE Magazine since 2007.

With 15 years of extensive experience in Web development, Bilal is an expert in providing enterprise Web solutions.

He works at Consolidated Contractors Company in Athens, Greece as a full-stack senior developer.

Bilal offers technical consultancy for a variety of technologies including Nest JS, Angular, Vue JS, JavaScript and TypeScript.

applications on the internet. Once you've identified the user and allowed access, you need to check what actions this user can perform. For instance, perhaps the user can create new records but not delete existing ones. This is also called Authorization and is a standard practice in Laravel and most web applications.

Authentication in Laravel

How does Laravel handle authentication and authorization? The answer is in this first article in a series on building MVC applications in Laravel.

Laravel uses guards to determine how a user is authenticated for each request. Guards essentially serve as the gatekeeper for your application, granting or denying access accordingly. Laravel provides several built-in guards, including.

- **Session Guard**: This guard uses session data to authenticate users. A session is created when a user logs in and their session ID is stored in a cookie. On subsequent requests, the session guard retrieves the session ID from the cookie, retrieves the corresponding session data, and uses it to authenticate the user.
- **Token Guard**: This guard uses API tokens to authenticate users. Instead of storing user data in a session, a token guard stores a unique API token in a database table and sends it to the client in an HTTP header. On subsequent requests, the token guard retrieves the token from the header to authenticate the user.

With both guards, the user provides the credentials as an email address and password.

In the Session Guard, Laravel matches the provided credentials with the records stored in the database. If there's a match, it stores the User ID inside the Session Data. The next time the user visits the application, Laravel checks whether an Auth Cookie holds a Session ID. If one is found, Laravel extracts the User ID from the Session Data and authenticates the user accordingly.

In the Token Guard, Laravel matches the provided credentials with the records stored in the database. Laravel generates a token representing the authenticated user's session if a match exists. This token is usually stored in a JSON Web Token (JWT) and contains the following information:

- **iss (Issuer)**: Identifies the issuing entity of the token, usually the name or identifier of the Laravel application.
- **sub (Subject)**: Identifies the token's subject, which is the user who's being authenticated. This field typically contains an identifier for the user, such as their unique ID in the database.
- **iat (Issued At)**: When the token was issued.
- **exp (Expiration Time)**: When the token expires.

In addition to these standard fields, you can add custom claims to the token to store additional information about the user, such as their permissions, roles, etc.

Laravel then returns this token to the front-end. It can store the token inside Local Storage or a cookie (recommended). On subsequent requests to the server, the client passes the JWT in the HTTP headers, and the server uses the information in the JWT to identify and authenticate the user for that request.

Login, Logout, and Auth Middleware

In this section, let's explore how Laravel handles authenticating users using the User model, Login View, Login Controller, Logout Controller, Routes, and Auth middleware.

I started by creating a new Laravel app. There are several methods for creating a new Laravel app. I chose the Docker-based one using the Laravel Sail package. You can read more about Laravel installation by following this URL (https://laravel.com/docs/10.x/installation).

Choose the method that best suits you. Before you start, make sure you have Docker running on your computer.

I'm using a MacBook Pro. I start by running this command:

```
curl -s \
"https://laravel.build/laravel-authentication" \
| bash
```

This creates a new Laravel application on your computer under the directory named **laravel-authentication**.

After the installer finishes, run the following commands:

1. Build up the Docker container.

```
./vendor/bin/sail up
```

2. Install the NPM packages.

```
./vendor/bin/sail npm install
```

3. Serve the application.

```
./vendor/bin/sail run dev
```

The application is now accessible at http://localhost. Open the URL in the browser, and you'll see the same view as in **Figure 1**.

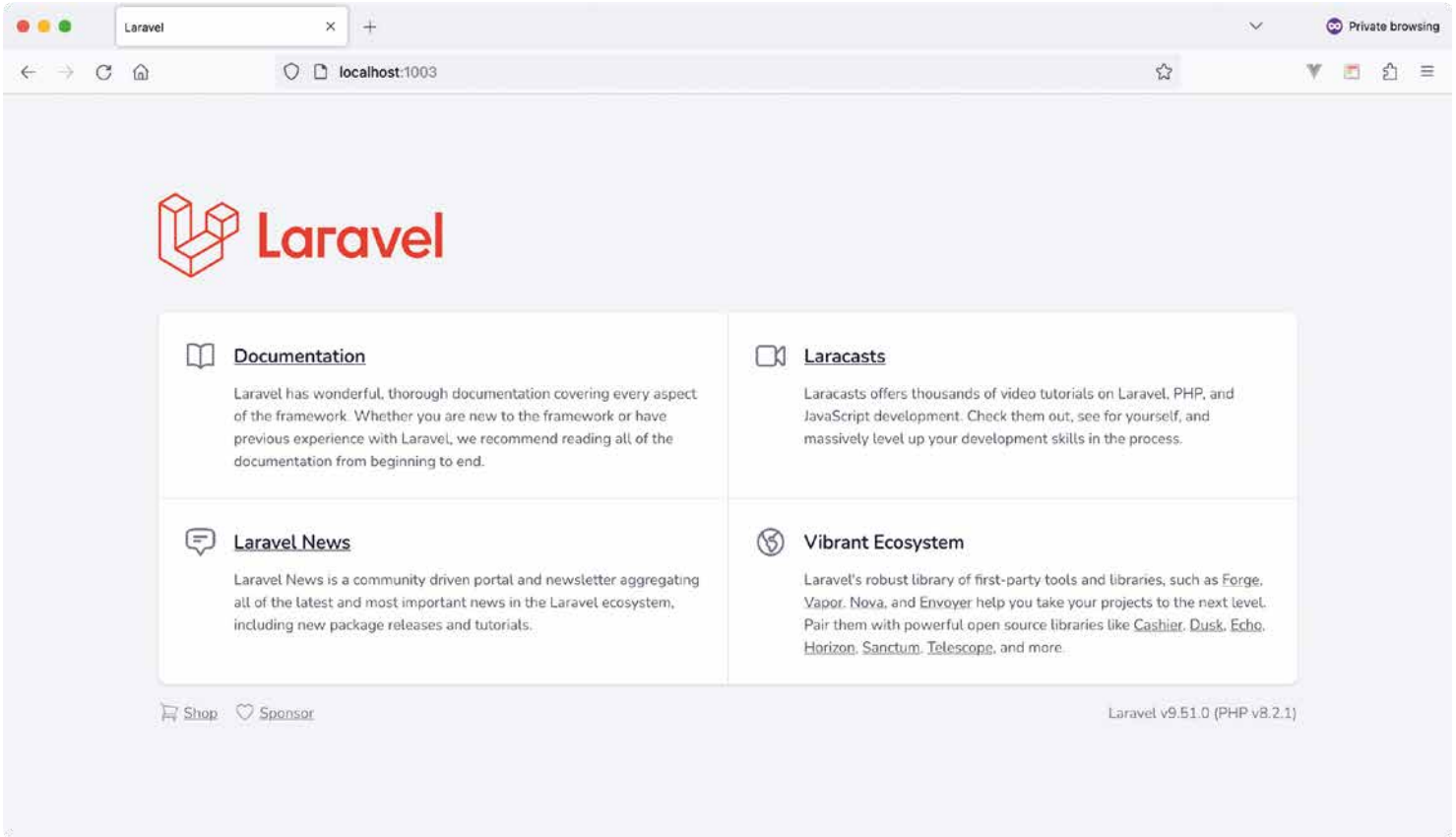


Figure 1: The Laravel 9 landing page

Next, let's install the Laravel Breeze starter kit. The Laravel team provides this starter kit for scaffolding Laravel authentication and Profile management. This is my ultimate choice when starting a new Laravel project. Trust me: It saves you a lot of time! You can read more about Laravel Breeze here (https://laravel.com/docs/10.x/starter-kits#laravel-breeze).

Laravel Breeze Installation

Laravel Breeze comes in three flavors:

- Breeze & Blade
- Breeze & React
- Breeze & Vue
- Breeze & Next.js/API

I'll use Breeze and Vue for this article.

Run the following commands to install Laravel Breeze, VueJS, and InertiaJS together.

```
./vendor/bin/sail composer require \
                laravel/breeze -dev
./vendor/bin/sail artisan breeze:install vue
./vendor/bin/sail artisan migrate
./vendor/bin/sail npm install
./vendor/bin/sail npm run dev
```

By installing Laravel Breeze, you've installed a ton of new things in your application. Let me list them briefly, but you can always check this repository commit titled **(install Laravel Breeze package)** to check every new file and configuration added by the package.

> In addition to Breeze Starter Kit, Laravel also offers the JetStream Starter Kit, which is a more advanced option.

More importantly, Laravel Breeze does the following:

1. Adds Vue, InertiaJS, and Tailwindcss
2. Configures InertiaJS
3. Configures Tailwindcss
4. Scaffolds a front-end application using Vue/InertiaJS
5. Adds Laravel Authentication controllers, views, and routes:
    a. Login
    b. Logout
    c. Register new user
    d. Forgot password
    e. Reset password
    f. Email verification
    g. Profile management
    h. Auth routes

Let's explore each of these sections in more detail.

Users in Laravel

Laravel represents a user in the application through the User model (app\Models\User.php). It backs up this model in the database by storing user information inside the

#	column_name	data_type	character_set	collation	is_nullable	column_default	extra	foreign_key	comment
1	id	bigint unsigned	NULL	NULL	NO	NULL	auto_increment	EMPTY	EMPTY
2	name	varchar(255)	utf8mb4	utf8mb4_unicode_ci	NO	NULL	EMPTY	EMPTY	EMPTY
3	email	varchar(255)	utf8mb4	utf8mb4_unicode_ci	NO	NULL	EMPTY	EMPTY	EMPTY
4	email_verified_at	timestamp	NULL	NULL	YES	NULL	EMPTY	EMPTY	EMPTY
5	password	varchar(255)	utf8mb4	utf8mb4_unicode_ci	NO	NULL	EMPTY	EMPTY	EMPTY
6	remember_token	varchar(100)	utf8mb4	utf8mb4_unicode_ci	YES	NULL	EMPTY	EMPTY	EMPTY
7	created_at	timestamp	NULL	NULL	YES	NULL	EMPTY	EMPTY	EMPTY
8	updated_at	timestamp	NULL	NULL	YES	NULL	EMPTY	EMPTY	EMPTY

Figure 2: The Users' table structure

users' database table. **Figure 2**. Shows the users' table structure.

When I first created the Laravel application, it automatically added a database/factories/UserFactory.php. Laravel uses the factory class to generate test data for the application. **Listing 1** has the entire UserFactory.php file.

Notice how Laravel hashes the password and never stores plain text passwords. When the user tries to log in, it hashes the incoming password and compares it to the one stored in the database.

In Laravel, the Factory and Seeder classes work together to generate test data for your application.

- **Factory:** A factory class defines the data structure you want to generate. You can specify the attributes and values for each field in the model. The factory class uses Faker, a library for generating fake data, to fill in the values for each field.
- **Seeder:** A seeder class calls the factory class and generates the data. You can specify how many records you want to create and what data type to generate. The seeder class inserts the data into the database.

Listing 1: UserFactory.php file

```
<?php
namespace Database\Factories;
use Illuminate\Database\Eloquent\Factories\Factory;
use Illuminate\Support\Str;

class UserFactory extends Factory
{
    public function definition()
    {
        return [
            'name' => fake()->name(),
            'email' => fake()->unique()->safeEmail(),
            'email_verified_at' => now(),
            'password' => '$2y$10.../igi',
            'remember_token' => Str::random(10),
        ];
    }
    public function unverified()
    {
        return $this->state(fn (array $attributes) => [
            'email_verified_at' => null,
        ]);
    }
}
```

Locate the database/seeders/DatabaseSeeder.php file, and inside the run() method, add the following line of code:

```
\App\Models\User::factory(1)->create();
```

This line inserts a new record into the users' database table.

Run this command to seed the database with a testing user record:

```
./vender/bin/sail db:seed
```

You can use factories and seeders to feed any testing data you want in your application. Generally, it's a best practice to provide factories and seeders to test your application locally.

User Registration

Laravel Breeze places the Register view inside the resources/js/Pages/Auth/Register.vue Vue component. **Listing 2** shows a simplified version of the Register.vue component.

The script section uses the useForm composable that InertiaJS offers to create a form object with five fields: name, email, password, password_confirmation, and terms.

The submit function is then defined to submit the form data to the server. The form is posted to the Register POST route with the form.post() method. The onFinish() option is set, which specifies a function to be executed after the request has finished. In this case, the function **form.reset('password', 'password_confirmation')** is called, which resets the password and password_confirmation fields in the form.

The template section defines the Register form. It's defined with a submit.prevent() event listener, which prevents the default form submission behavior and calls the submit function instead. The form contains five input fields bound to a corresponding field in the form state object using the v-model directive. The ID attribute specifies a unique identifier for each field. Finally, the PrimaryButton component displays a button for submitting the form.

Let's check the Register routes Laravel Breeze defines inside the routes/auth.php file.

Listing 2: Register.vue component

```
<script setup>
const form = useForm({
    name: '',
    email: '',
    password: '',
    password_confirmation: '',
    terms: false,
});

const submit = () => {
    form.post(route('register'), {
        onFinish: () => form.reset(
         'password',
         'password_confirmation'
        ),
    });
};
</script>

<template>
<form @submit.prevent="submit">
  <div>
     <TextInput
         id="name"
         type="text"
         v-model="form.name"
         required
     />
     <InputError :message="form.errors.name" />
  </div>

  <div>
     <TextInput
         id="email"
         type="email"
         v-model="form.email"
         required
     />
     <InputError :message="form.errors.email" />
  </div>

  <div>
     <TextInput
         id="password"
         type="password"
         v-model="form.password"
         required
     />
     <InputError :message="form.errors.password" />
  </div>

  <div>
     <TextInput
         id="password_confirmation"
         type="password"
         v-model="form.password_confirmation"
         required
     />
     <InputError
           :message="form.errors.password_confirmation" />
  </div>

  <div>
     <PrimaryButton>
         Register
     </PrimaryButton>
  </div>
</form>
</template>
```

```
Route::get('register',
    [RegisteredUserController::class, 'create']
)->name('register');

Route::post('register',
    [RegisteredUserController::class, 'store'
);
```

The first route is a GET route for the /register URL, and it maps to the create() method of the RegisteredUserController class. The name() method is used to specify a name for the route, which can be used later in the application to generate URLs or to refer to the route by name.

The second route is a POST route for the /register URL, and it maps to the store() method of the RegisteredUserController class. This route is intended to receive form submissions from the login form.

> Laravel Breeze stores all Auth routes in a separate file located at routes/auth.php

Listing 3 shows the source code for the store() method Laravel Breeze defines on the RegisteredUserController class. This code authenticates the user into the application.

The store() method takes a Request object as an argument, which contains the user input data. The method

Listing 3: RegisteredUserController store() method

```
public function store(Request $request): RedirectResponse
{
     $request->validate([
      'name' => 'required|string|max:255',
      'email' => [
        'required',
        'string',
        'email',
        'max:255',
        'unique:'.User::class
      ],
      'password' => [
        'required',
        'confirmed',
        Rules\Password::defaults()
      ],
     ]);
    $user = User::create([
        'name' => $request->name,
        'email' => $request->email,
        'password' => Hash::make($request->password),
    ]);
    event(new Registered($user));
    Auth::login($user);
    return redirect(RouteServiceProvider::HOME);
}
```

first validates the user input data using the validate() method on the $request object. The validation rules specify that the name field is required and must be a string with a maximum length of 255 characters. The email field is also required. It must be a string, must be a valid email address, and must be unique in the users' table. The password field is required and must be confirmed. The Confirmed validation rule expects a matching field of password_confirmation to exist in the $request object.

Finally, the **Rules\Password::defaults()** method is used to specify default validation rules for password fields in Laravel.

You can read more about Laravel Validation here (https://laravel.com/docs/10.x/validation).

If the validation passes, a new user is created by calling the create method on the User model and passing an array of data that includes the user's name, email, and hashed password. The **Hash::make()** method is used to hash the password.

An event Registered is then fired with the newly created user as its argument. Locate the app\Providers\EventServiceProvider.php file and notice the $listen variable.

```
protected $listen = [
    Registered::class => [
        SendEmailVerificationNotification::class,
    ],
];
```

Laravel adds an event listener for the Registered event. The SendEmailVerificationNotification class sends an email verification to the registered user if the email verification feature is enabled. I'll look at the email verification feature in the coming sections.

Back to the store() method, the **Auth::login()** method is called to log the user in, and finally, the method returns a redirect to the home page using the redirect method and passing the **RouteServiceProvider::HOME** constant as its argument.

User Login

Laravel Breeze places the Login view inside the resources/js/Pages/Auth/Login.vue Vue component. **Listing 4** shows a simplified version of the Login.vue component.

The script section uses the useForm composable that InertiaJS offers to create a form object with the properties email, password, and remember me. The properties are initialized with the default values of an empty string, an empty string, and a false string, respectively.

The submit function is then defined to submit the form data to the server. The form is posted to the Login POST route with the form.post() method and the onFinish() option used to reset the password field after completing the form submission.

The template section defines the Login form's structure, which comprises text inputs for the email and password fields, an error message component for displaying validation errors, a checkbox for remembering the user, and a primary button for submitting the form. The form is bound to the submit event using the **@submit.prevent** Vue directive, and the submit function defined in the script section is called when the form is submitted. The text inputs and the checkbox are bound to the properties of the form object using the **v-model** Vue directive. The error message components display the error messages from the form object using the message prop.

Let's check the Login routes Laravel Breeze defines inside the routes/auth.php file.

```
Route::get('login',
    [AuthenticatedSessionController::class, 'create']
)->name('login');

Route::post('login',
    [AuthenticatedSessionController::class, 'store'
);
```

The first route is a GET route for the /login URL, and it maps to the create() method of the AuthenticatedSessionController class. The name() method is used to specify a name for the route, which can be used later in the application to generate URLs or to refer to the route by name.

The second route is a POST route for the /login URL, and it maps to the store() method of the AuthenticatedSessionController class. This route is intended to receive form submissions from the login form.

Listing 5 shows the source code for the store() method Laravel Breeze defines on the AuthenticatedSessionController class. This code authenticates the user into the application.

Listing 4: Login.vue component

```
<script setup>
const form = useForm({
   email: '',
   password: '',
   remember: false,
});

const submit = () => {
   form.post(route('login'), {
       onFinish: () => form.reset('password'),
   });
};
</script>

<template>
    <form @submit.prevent="submit">
      <div>
        <TextInput
          id="email"
          type="email"
          v-model="form.email"
          required
        />
        <InputError :message="form.errors.email" />
      </div>

      <div>
        <TextInput
         id="password"
         type="password"
         v-model="form.password"
         required
       />
       <InputError :message="form.errors.password" />
      </div>

      <div>
        <Checkbox name="remember" v-model:checked="form.remember" />
        <span class="ml-2 text-sm text-gray-600">Remember me</span>
      </div>

      <PrimaryButton>
        Log in
      </PrimaryButton>
    </form>
</template>
```

The store() method takes a LoginRequest FormRequest object as an argument and returns a RedirectResponse object. The method starts by calling the authenticate() method on the LoginRequest object, which is used to validate the incoming login request and verify that the user-provided credentials are valid.

The next line calls the regenerate() method on the session object, which regenerates the user's session ID to increase the security of the session.

Finally, the method returns a redirect response to the intended destination. Laravel tracks the intended destination when the user tries to visit a private page before logging in to the application. If there is no such intended destination, Laravel redirects the user to the URL specified in the constant **RouteServicePovider::HOME**, which is, in this case, the /dashboard URL.

Let's inspect the **LoginRequest::authentiate()** method closely. **Listing 6** shows the entire source code for this method.

The authenticate() function accepts no parameters and returns nothing (void method).

The first line calls the ensureIsNotRateLimited() method, which checks if the user is rate limited (e.g., if they've tried to log in too many times in a short period). If the user is rate limited, the method throws an exception.

The second line uses the Auth facade's attempt() method to attempt to log the user in with the email and password provided in the request. The method also accepts a Boolean argument, determining whether the user's session should be remembered.

If the authentication attempt fails, the method calls the hit() method on the RateLimiter class to increase the user's failed attempts. The method then throws a ValidationException with a message indicating that the authentication has failed.

If the authentication attempt is successful, the method calls the clear() method on the RateLimiter class to clear the number of failed attempts for the user.

That's how Laravel Breeze validates the user's credentials and attempts a login to the application.

You must have noticed that the attempt() method accepts as a third parameter whether it should remember the user's login session. If Laravel resolves this parameter to a Boolean True value, it generates a **remember-me token**.

In Laravel, the **remember token** is a random string generated and stored in the database when the user checks the "Remember Me" checkbox during login. This token is then used to identify the user on subsequent visits.

Typically, the remember me token is a hash of a combination of the user's ID, a random value, and the current timestamp. This hash is designed to be unique and secure, so the resulting hash will be different even if two users have the same combination of ID and timestamp. The hashed token is then stored in the database, linked to the user's account, and used to identify the user on subsequent visits.

Listing 5: AuthenticatedSessionController store() method

```
public function store(LoginRequest $request):
    RedirectResponse
{
    $request->authenticate();
    $request->session()->regenerate();
    return redirect()->intended(RouteServiceProvider::HOME);
}
```

Listing 6: LoginRequest::authenticate() method

```
public function authenticate(): void
{
    $this->ensureIsNotRateLimited();
    if (
        !Auth::attempt(
            $this->only('email', 'password'),
            $this->boolean('remember')
        )
    ) {
            RateLimiter::hit($this->throttleKey());
            throw ValidationException::withMessages([
                'email' => trans('auth.failed'),
            ]);
    }
    RateLimiter::clear($this->throttleKey());
}
```

Generating a remember token involves the following steps:

1. When a user logs into the application and selects the "Remember me" option, Laravel generates a unique token using a secure random number generator.
2. The token is combined with additional information, such as the user's ID and a time stamp, to create a signed token that the application can verify.
3. The signed token is stored in the user's session, cookie, and database record. This allows the application to maintain the user's login state and easily verify the user's authentication status on subsequent requests.

In this way, the **remember token** serves as a secure and convenient way for Laravel to remember the user's identity and allow them to remain logged in across multiple visits.

This section summarizes the entire process that Laravel uses to log users into the application.

Email Verification

Laravel supports email verification at the core. Email verification is a process you use to confirm the validity and authenticity of an email address provided by a user during registration.

To enable email verification in your application, ensure that the User model implements the Illuminate\Contracts\Auth\MustVerifyEmail contract. That's all!

Assuming that you've enabled email verification in your application, right after a successful registration process, Laravel fires the Registered event, as you've seen in the User Registration section previously.

Laravel hooks into the Registered event inside the **AppServiceProvider::class** by listening to this event and fires the SendEmailVerificationNotification event listener.

```
protected $listen = [
    Registered::class => [
        SendEmailVerificationNotification::class,
    ],
];
```

Inside this listener, Laravel sends the user a verification email to the email address provided. The email contains a URL generated by **URL::temporarySignedRoute()** method with a default expiration date of 60 seconds and a hashed string of the User ID and User Email.

The user must click this URL, and the Laravel application validates and checks the parameters and verifies the user in the database.

Now that you know how the email verification process works, let's discover all aspects of enabling and making this feature usable in Laravel.

Listing 7: SendEmailNotification class

```
class SendEmailVerificationNotification
{
     public function handle(Registered $event)
     {
       if (
         $event->user instanceof MustVerifyEmail
         &&
         !$event->user->hasVerifiedEmail()
       )
       {
         $event->user->sendEmailVerificationNotification();
       }
     }
}
```

Listing 8: MustVerifyEmail trait

```
<?php
namespace Illuminate\Auth;
use Illuminate\Auth\Notifications\VerifyEmail;
trait MustVerifyEmail
{
    // …
    public function sendEmailVerificationNotification()
    {
        $this->notify(new VerifyEmail);
    }
    // …
}
```

Listing 9: Temporary signed route code

```
URL::temporarySignedRoute(
   'verification.verify',
   Carbon::now()
      ->addMinutes(
        Config::get('auth.verification.expire', 60)
      ),
   [
     'id' => $notifiable->getKey(),
     'hash' => sha1($notifiable->getEmailForVerification()),
   ]
);
```

Start by navigating to the app/Models/User.php model file. The User model must implement the MustVerifyEmail contract as follows:

```
class User extends
      Authenticatable implements MustVerifyEmail
```

Next, let's revisit the **SendEmailVerificationNotification::class** and see how Laravel prepares and sends the verification email. **Listing 7** shows the class implementation.

The handle() method of the class sends the user an email only when:

- The User model implements the MustVerifyEmail contract.
- The user is not yet verified.

Let's inspect the **sendEmailVerificationNotification()** method Laravel defines inside the **Illuminate\Auth\MustVerifyEmail.php** trait. **Listing 8** shows the related code in this trait.

Laravel applies this trait on the User Model. Hence, the **sendEmailVerificationNotification()** method is accessible on the User model instance.

Laravel issues a new notification using the VerifyEmail notification class inside the method. Once again, because Laravel uses the Notifiable trait on the User model, it's safe to call the **$this->notify()** method to send out a notification.

The **VerifyEmail::class** is just a Laravel notification found in the **Illuminate\Auth\Notifications\VerifyEmail.php** file.

I'll dedicate a full article to using Laravel Notifications in the future. For now, it's enough to understand that this VerifyEmail notification prepares an email with the temporary signed route and sends it to the user. **Listing 9** shows the code to prepare the signed route.

The method takes three parameters:

- The name of the route, which is **verification.verify**
- The expiration time for the signed route, which is set to the current time plus the number of minutes specified in the auth.verification.expire configuration setting (with a default value of 60 minutes)
- An array of parameters passed to the route. In this case, the array contains two parameters: id, the User ID, and hash, is a hash of the user's email address.

This is an example of a temporary signed route in Laravel:

```
http://localhost:1003/verify
email/3/374a976d6e8288d2633ed8ad94b6d4bdd8487d50?expir
s=1676289142&signature=b2fd6e7f2def3e2cfeb9aedab9b58c8
40e8263a20975bc85d1ecdc78d6441d3
```

Figure 3. Shows the email sent to the user to verify the email address.

You can read more about the signed URLs in Laravel here (https://laravel.com/docs/10.x/urls#signed-urls).

Once the user clicks the Verify Email Address button, Laravel loads the route verification.verify in the browser. This is handled by the **VerifyEmailController::class __invoke()** method, as shown in **Listing 10**.

> Email verification is an optional feature. It's recommended to use it, though.

The code in **Listing 10** checks if the user is already verified otherwise, it verifies the user by calling the markEmailAsVerified() method and finally redirects the user to the intended route, if it exists; otherwise, it redirects the user to the route defined in the **RouteServiceProvider::HOME constant**.

Laravel provides the **\Illuminate\Auth\Middleware\EnsureEmailIsVerified::class** middleware defined inside the app/Http/Kernel.php file under the route middleware array. You can use this route middleware to prevent access to the application (even if the user has successfully signed in) until the user verifies the email address. Here's an example of limiting users to verified only users when accessing the /dashboard URL in the application:

```
Route::get('/dashboard', function () {
    return Inertia::render('Dashboard')
})->middleware(['verified'])
->name('dashboard');
```

The user has to verify the email address before accessing the /dashboard URL. **Listing 11** shows a simplified version of the source code for the **EnsureEmailIsVerified::class**.

The middleware redirects the user to the **verification.notice** route only when the user has not yet verified the email address. Otherwise, it passes over the request to the next middleware in the middleware pipes.

To get more insight on the topic of Laravel middleware and how it handles the request flow, you can grab a free copy of my e-book on this topic: "Mastering Laravel's Request Flow" (https://tinyurl.com/2pbddgf5).

> "Master Laravel's Request Flow" delves deep inside the Laravel Framework and explains all needed details.

Laravel defines the verification.notice route in the routes/auth.php file as follows:

```
Route::get('verify-email',
   [
        EmailVerificationPromptController::class,
        '__invoke'
    ]
)->name('verification.notice');
```

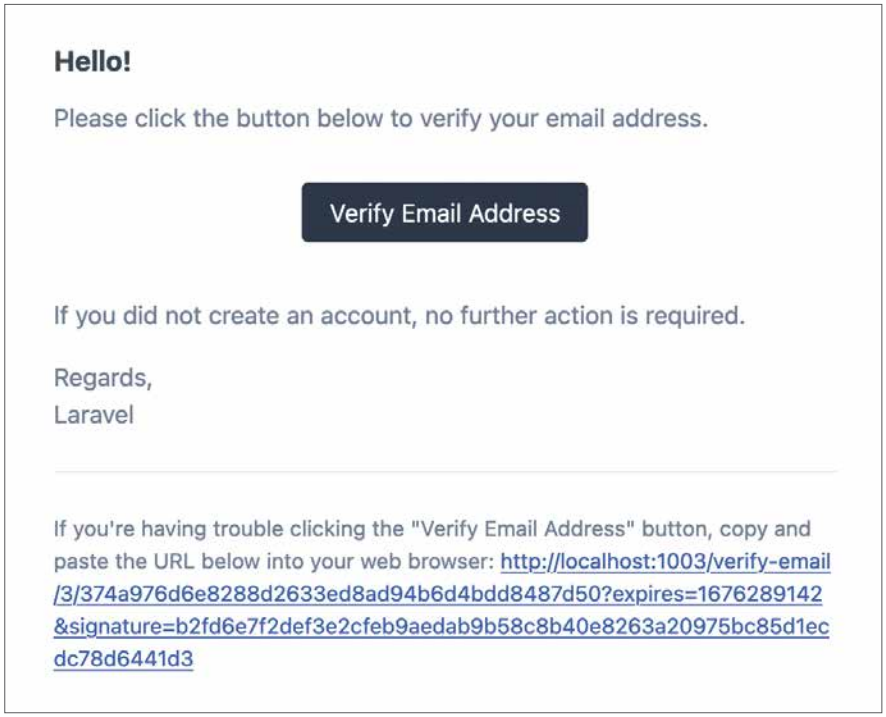


Figure 3: The email address verification email in Laravel

Listing 10: VerifyEmailController __invoke() method

```
class VerifyEmailController extends Controller
{
    public function __invoke(
        EmailVerificationRequest $request
    ): RedirectResponse
    {
       if ($request->user()->hasVerifiedEmail())
       {
         return redirect()
 ->intended(RouteServiceProvider::HOME.'?verified=1');
     }
     if ($request->user()->markEmailAsVerified())
        {
            event(new Verified($request->user()));
     }
     return redirect()
 ->intended(RouteServiceProvider::HOME.'?verified=1');
    }
}
```

Listing 11: EnsureEmailsVerified middleware class

```
class EnsureEmailIsVerified
{
    public function handle(
        $request,
        Closure $next,
        $redirectToRoute = null
    )
    {
        if (! $request->user() ||
           ($request->user() instanceof MustVerifyEmail &&
           ! $request->user()->hasVerifiedEmail())) {
               return $request->Redirect::guest(
                URL::route(
                 $redirectToRoute ?: 'verification.notice'
                )
               );
     }
     return $next($request);
    }
}
```

Listing 12: ForgotPassword.vue page

```
<script setup>
defineProps({
    status: String,
});
const form = useForm({
    email: '',
});
const submit = () => {
    form.post(route('password.email'));
};
</script>
<template>
    <div v-if="status">
        {{ status }}
    </div>
    <form @submit.prevent="submit">
        <div>
            <TextInput
              id="email"
              type="email"
              v-model="form.email"
              required
            />
        </div>
        <PrimaryButton>
            Email Password Reset Link
        </PrimaryButton>
    </form>
</template>
```

The __invoke() method defined on the **EmailVerificationPromptController::class** redirects the user to the resources/js/Pages/Auth/VerifyEmail.vue page. The page allows the user to resend the verification email, as shown in **Figure 4**.

That's all you need to know, at this stage, regarding email verification in Laravel. Let's move on and study how Password Reset works in Laravel.

Forgot Password and Reset

On the Login page, you can retrieve your forgotten password by clicking the link named **Forgot your Password?**

Laravel uses the following markup to define this link:

```
<Link :href="route('password.request')">
    Forgot your password?
</Link>
```

You can find the route **password.request** inside the routes/auth.php file defined as follows:

```
Route::get('forgot-password',
    [PasswordResetLinkController::class, 'create']
)->name('password.request');
```

Let's explore the create() method.

```
public function create(): Response
{
    return Inertia::render('Auth/ForgotPassword', [
        'status' => session('status'),
    ]);
}
```

The create() method returns the resources/js/Pages/Auth/ForgotPassword.vue page. **Listing 12** shows a simplified version of the ForgotPassword.vue page.

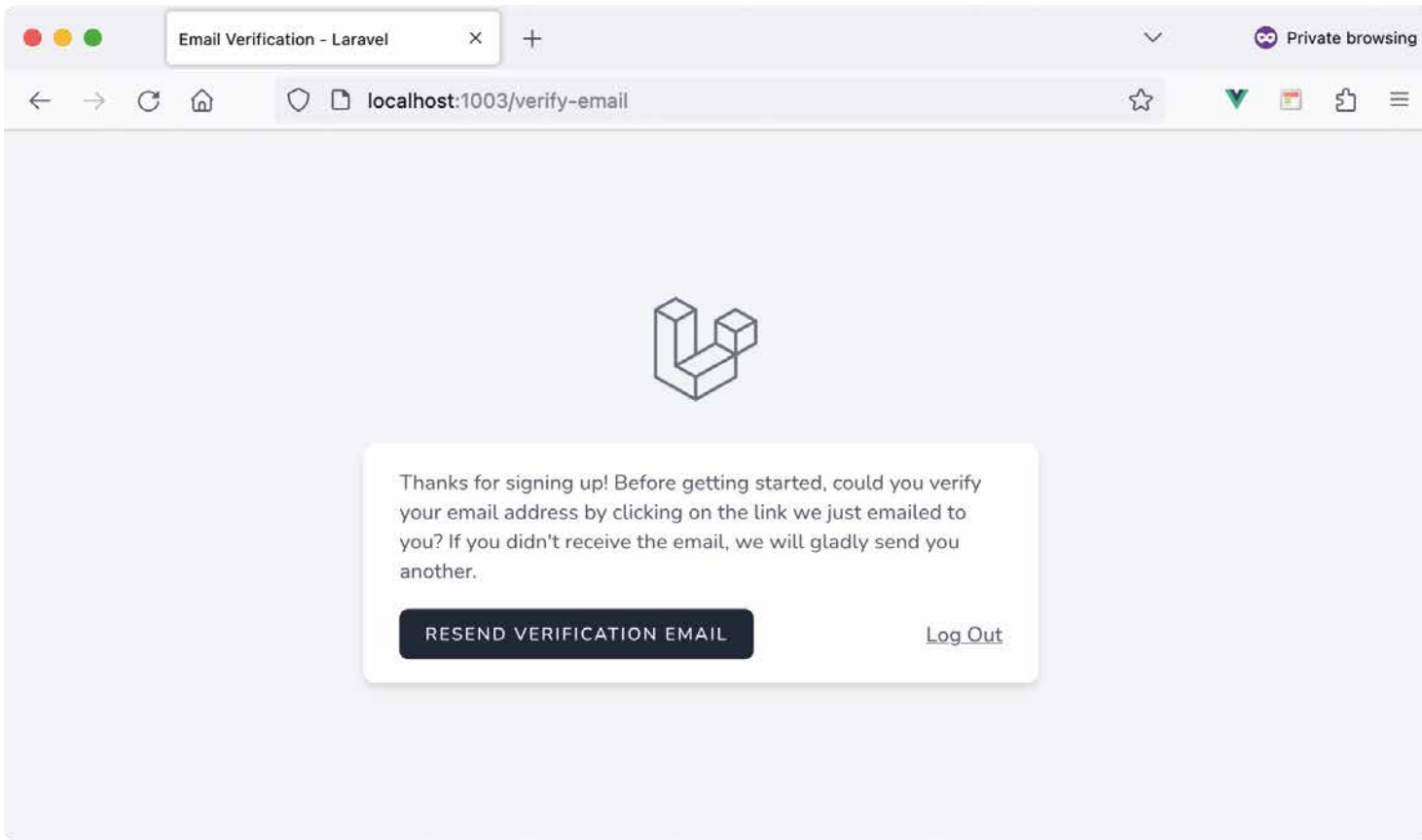


Figure 4: Resend the email address verification email

The <script> section defines the component's logic and data.

- **defineProps()** is a function that defines the component's props. In this case, the component has a single prop named status, which is of type String.
- The **useForm** hook manages the component's form data. It creates an object named form with a single property email set to an empty string.
- The submit() function is called when the form is submitted. It uses the form.post() method to send a POST request to the URL defined by the **password.email** route.

The <template> section defines the component's HTML structure and visual appearance.

- A div to display the status of sending a Forgot password reset link.
- The form element contains a single input field to allow the user to enter the email address. The v-model directive is used to bind the value of the input to the form.email property.
- The PrimaryButton component is a button the user can click to submit the form and trigger the password reset process.

You can find the route password.email inside the routes/auth.php file defined as follows:

```
Route::post('forgot-password',
    [PasswordResetLinkController::class, 'store']
)->name('password.email');
```

Let's explore the store() method. **Listing 13** shows the source code for the store() method.

The function takes an instance of the Request class as its input. This class represents the HTTP request that the user submitted. The method validates the incoming email address to make sure it exists and that it's an email address.

The sendResetLink() method on the Password Facade class is called, passing the user's email address entered in the form. This method sends a password reset email to the user.

The result of the sendResetLink() method is stored in the $status variable. If $status equals **Password::RESET_LINK_SENT**, the password reset email was successfully sent, and the user is redirected to the previous page (ForgotPassword.vue) with a success status message.

If the value of $status is not equal to **Password::RESET_LINK_SENT**, an error occurs, and a ValidationException is thrown with an error message.

Digging more inside the Laravel framework, I discovered that the Illuminate\Auth\Notifications\ResetPassword.php is the notification that executes when it's time to send the user a password reset link.

Let's track the most important part of this notification class, the protected resetUrl() function. **Listing 14** shows the source code for the resetUrl() function.

Listing 13: PasswordResetLinkController::class store() method

```
public function store(Request $request): RedirectResponse
{
    $request->validate([
        'email' => 'required|email',
    ]);
    $status = Password::sendResetLink(
        $request->only('email')
    );
    if ($status == Password::RESET_LINK_SENT)
    {
        return back()->with('status', __($status));
    }
    throw ValidationException::withMessages([
        'email' => [trans($status)],
    ]);
}
```

Listing 14: ResetPassword::class resetUrl() function

```
protected function resetUrl($notifiable)
{
    // …
    return url(route('password.reset', [
        'token' => $this->token,
        'email' => $notifiable->getEmailForPasswordReset(),
    ], false));
}
```

Listing 15: NewPasswordController::class create() method

```
public function create(Request $request): Response
{
    return Inertia::render('Auth/ResetPassword', [
        'email' => $request->email,
        'token' => $request->route('token'),
    ]);
}
```

The method takes an instance of the $notifiable object as its input, which represents a recipient of the password reset email.

It returns a password reset URL generated by calling the url() function and passing it the result of the route function.

The route() function generates a URL for the password reset page represented by the **password.reset** route. It takes as parameters the token and the email address of the recipient. Laravel generates a random token and stores it in the database inside the **password_resets** table.

The password.reset route is defined inside the routes/auth.php file as follows:

```
Route::get('reset-password/{token}',
    [NewPasswordController::class, 'create']
)->name('password.reset');
```

This route handles clicking the password reset link that was sent in the email. Let's explore the create() method. **Listing 15** shows the source code for the **NewPasswordController::class** create() method.

The method returns the resources/js/Pages/Auth/ResetPassword.vue page passing down to the page the email and token parameters. **Listing 16** shows the ResetPassword.vue page.

Listing 16: ResetPassword.vue page

```
<script setup>

const props = defineProps({
    email: String,
    token: String,
});

const form = useForm({
    token: props.token,
    email: props.email,
    password: '',
    password_confirmation: '',
});

const submit = () => {
    form.post(route('password.store'), {
        onFinish: () => form.reset(
            'password',
            'password_confirmation'
        ),
    });
};
</script>

<template>
    <form @submit.prevent="submit">
        <div>
            <TextInput
              id="email"
              type="email"
              v-model="form.email"
              required
            />
        </div>

        <div>
            <TextInput
              id="password"
              type="password"
              v-model="form.password"
              required
            />
        </div>

        <div>
            <TextInput
              id="password_confirmation"
              type="password"
              v-model="form.password_confirmation"
              required
             />
        </div>

        <PrimaryButton>
           Reset Password
        </PrimaryButton>
    </form>
</template>
```

The page prompts the user to submit their email, new password, and confirmed password. It then submits the form to the **password.store** route. Let's check this route in the routes/auth.php file.

Listing 17: NewPasswordController::class store() method

```
public function store(Request $request): RedirectResponse
{
    $request->validate([
      'token' => 'required',
      'email' => 'required|email',
      'password' => [
        'required',
        'confirmed',
         Rules\Password::defaults()
       ],
    ]);

    $status = Password::reset(
        $request->only(
           'email',
           'password',
           'password_confirmation',
           'token'
        ),

        function ($user) use ($request) {
            $user->forceFill([
              'password' => Hash::make($request->password),
              'remember_token' => Str::random(60),
            ])->save();

            event(new PasswordReset($user));
        }
    );
    if ($status == Password::PASSWORD_RESET)
    {
        return redirect()->route('login')
                ->with('status', __($status));
    }
    throw ValidationException::withMessages([
        'email' => [trans($status)],
    ]);
}
```

```
Route::post('reset-password',
    [NewPasswordController::class, 'store']
)->name('password.store');
```

This is the last step in retrieving and resetting the user password. Let's explore the store() method. **Listing 17** shows a simplified store() method version on the **NewPasswordController::class**.

The store() method handles the form submission for resetting the password. The method first validates the incoming request using the validate() method, which ensures that the request contains a token, an email, and a password that matches the confirmation and meets the default password rules.

The **Password::reset()** method is then called to reset the user's password. It takes an array of parameters, including the user's email, the new password, the confirmation of the new password, and the reset token. It also takes a closure responsible for updating the user's password and firing an event to notify the system of the password reset upon a successful password reset by Laravel.

The reset() method matches the incoming token and email with those stored in the password_resets database table before authorizing a password reset action.

If the password reset is successful, the method redirects the user to the login page with a status message indicating that the password has been reset. If the password reset fails, a ValidationException is thrown with a message indicating the reason for the failure. **Figure 5** shows the password reset page.

That's it! Now you can retrieve and reset your forgotten password in Laravel.

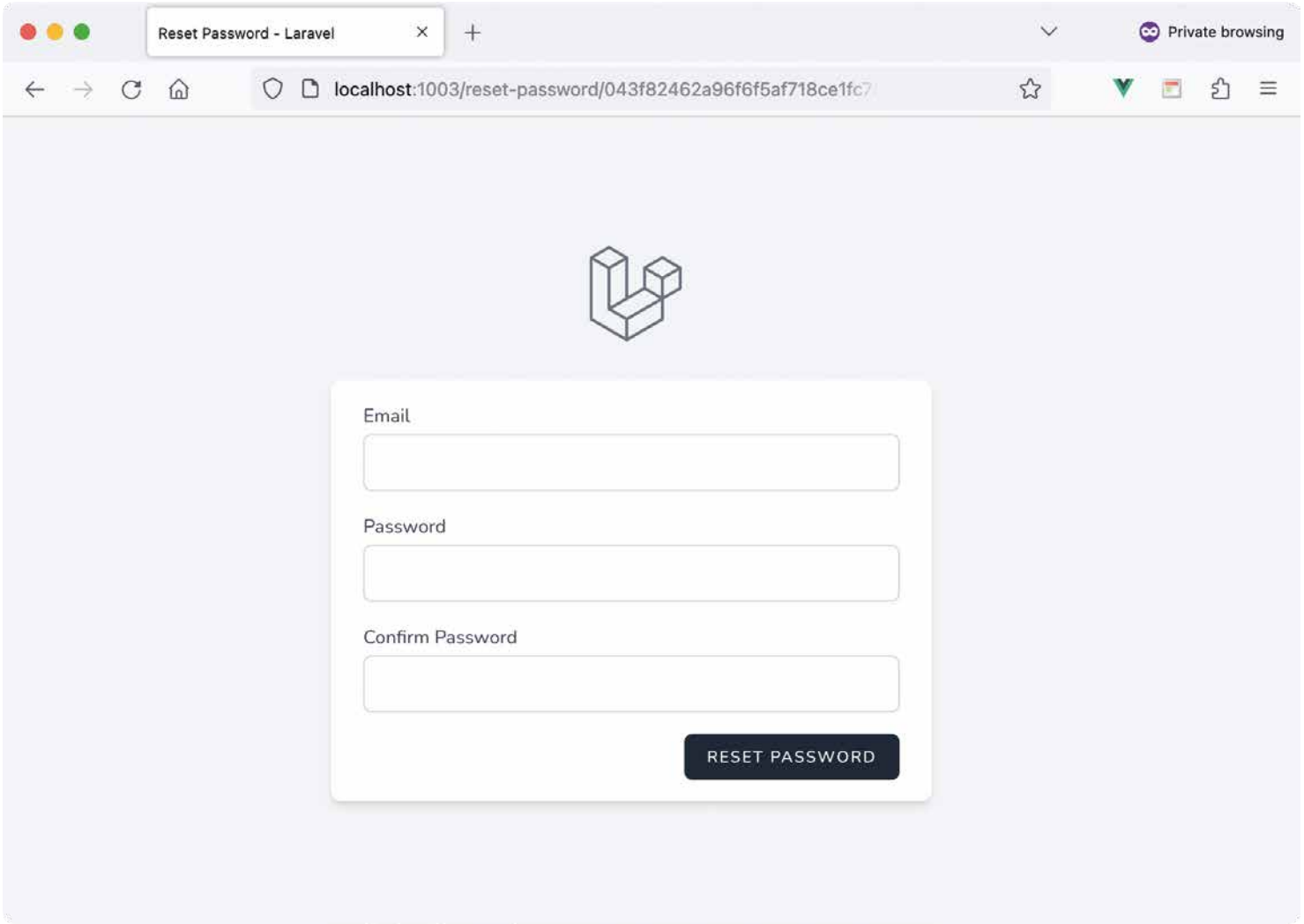


Figure 5: The password reset page

User Logout

Logging out of the Laravel application is straightforward. You'll notice a link to log out from the application on the Welcome page. This link posts to a route named logout.

Let's explore the logout route in the routes/auth.php file.

```
Route::post('logout',
    [
     AuthenticatedSessionController::class,
     'destroy'
    ]
)->name('logout');
```

The destroy() method defined on the **AuthenticatedSessionController::class** is responsible for logging the user out. Let's check it out. **Listing 18** shows the destory() method.

The destroy() method handles the request to log out the authenticated user.

The **Auth::guard('web')->logout()** method is called to log out the user by invalidating the user's session on the web guard, which is the default authentication guard for web applications in Laravel (SessionGuard.php).

Listing 18: AuthenticatedSessionController::class destory() method

```
public function destroy(Request $request): RedirectResponse
{
    Auth::guard('web')->logout();

    $request->session()->invalidate();
    $request->session()->regenerateToken();
    return redirect('/');
}
```

The session()->invalidate() method is called to invalidate the user's session, which ensures that any data associated with the user's session is cleared from the storage and generates a new session ID.

The session()->regenerateToken() method is called to regenerate the CSRF token, which is a security feature that helps to prevent session hijacking. I didn't cover the CSRF protection in Laravel as Laravel automatically handles it. However, if you're interested in learning more about it, you can read about it here: https://laravel.com/docs/10.x/csrf.

Finally, the method redirects the user to the home page. The user is now fully logged out of the application.

Listing 19: Authenticate middleware

```
<?php
namespace App\Http\Middleware;
use Illuminate\Auth\Middleware\Authenticate as Middleware;
class Authenticate extends Middleware
{
    protected function redirectTo($request)
    {
        if (! $request->expectsJson()) {
            return route('login');
     }
    }
}
```

Auth Middleware and Protecting Routes

In the last episode of the series of building MVC apps with PHP Laravel, I've explained routing and middleware in Laravel. If you need a quick refresh, here's the link: https://www.codemag.com/Article/2301041/Mastering-Routing-and-Middleware-in-PHP-Laravel.

In the context of today's article, let's look at the Authentication route middleware Laravel provides to ensure that an authenticated user can access a secure area in your application.

Let's start by limiting routes to only authenticated users. The routes/auth.php file defines a route for the /dashboard URL that limits it to only authenticated users.

```
Route::get('/dashboard', function () {
    return Inertia::render('Dashboard');
})->middleware(['auth'])->name('dashboard');
```

The route defines the auth middleware to protect the /dashboard URL. Where's this auth middleware defined? Go to the app/Http/Kernel.php file and look at the $routeMiddleware array.

```
protected $routeMiddleware = [
 'auth' => \App\Http\Middleware\Authenticate::class,
];
```

The name auth is an alias for the authentication middleware at **app/Http/Middleware/Authentication::class**. **Listing 19** shows the source code for the **Authenticate::class** middleware. The **Authenticate::class** middleware extends the **Illuminate\Auth\Middleware\Authenticate::class** middleware class provided by Laravel framework.

The redirectTo() function is used internally by the **Illuminate\Auth\Middleware\Authenticate::class** to redirect the user to the /login page when authentication fails.

Let's have a look at the **Illuminate\Auth\Middleware\Authenticate::class**.

```
public function handle(
   $request,
   Closure $next,
   ...$guards)
{
    $this->authenticate($request, $guards);
    return $next($request);
}
```

The handle() function is straightforward. It tries to authenticate the user and then hands off the processing of the current request to the remaining middleware in the pipeline.

You can add your own logic to handle user authentication for other scenarios you might have in your application.

Conclusion

This article provided a comprehensive overview of Session authentication, which is widely used by numerous web applications in Laravel. However, authentication in Laravel extends beyond just Session authentication, as there are other providers, such as token-based authentication and authentication with social providers like Google and Twitter.

Subsequent installments of this series will explore token-based authentication, a popular method for authenticating API requests, and will also delve into authenticating with social providers.

I aim to equip you with the knowledge and skills to implement robust and secure Laravel applications.

Happy Laraveling!

Bilal Haidar

to. You can be annoyed at your neighbor all you want. But under the circumstances, absent fraud or some other actionable untoward conduct, what did your neighbor do to you that you didn't do to yourself?

Multi-team projects share some of the same issues. For example, there may be new integrations to implement. Some might have third-party services and some might have internal legacy applications that must integrated in ways not foreseen in its design. How this legacy application comes to the party is an important consideration. The app's arrival also means the arrival of a new team. Fences between teams are about clearly defined responsibilities, some shared and some individual. Fences are not about siloed activity. There still needs to be cooperative behavior. No team on a common project can or should work in a vacuum.

This is why the **Swimlane Diagram** is useful for matching process to the software and processes (with clear boundaries between) tasked with the work. With clear boundaries, we know **where** they are and **when** they are encountered. That's a moment for **inspection and adaption** because an artifact is passed between boundaries that reference entities that are accountable to one another for collective success. So long as there are clear, reasonable, and compatible expectations, necessary accountability doesn't devolve into unproductive, destructive, and avoidable conflict. The less conflict, the more neighborly and enjoyable and successful your project is likely to be.

John V. Petersen
CODE

CODE

May/Jun 2023
Volume 24 Issue 3

Group Publisher
Markus Egger

Associate Publisher
Rick Strahl

Editor-in-Chief
Rod Paddock

Managing Editor
Ellen Whitney

Contributing Editor
John V. Petersen

Content Editor
Melanie Spiller

Editorial Contributors
Otto Dobretsberger
Jim Duffy
Jeff Etter
Mike Yeager

Writers In This Issue
Jason Bock
Vassili Kaplan
Sahil Malik
John Petersen
Dan Spear
Bilal Haidar
Wei-Meng Lee
Eliot Moule
Paul D. Sheriff

Technical Reviewers
Markus Egger
Rod Paddock

Production
Friedl Raffeiner Grafik Studio
www.frigraf.it

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

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

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

Circulation & Distribution
General Circulation: EPS Software Corp.
Newsstand: American News Company (ANC)

Subscriptions
Subscription Manager
Colleen Cade
ccade@codemag.com

US subscriptions are US $29.99 for one year. Subscriptions outside the US are US $50.99. 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, e-mail subscriptions@codemag.com.

Subscribe online at
www.codemag.com

CODE Developer Magazine
6605 Cypresswood Drive, Ste 425, Spring, Texas 77379
Phone: 832-717-4445

CODA: Good Fences, Good Neighbors

Is the implication that bad fences make for bad neighbors? Not any more than good fences making good neighbors. As a rule, constraints by way of clear boundaries are valuable guidance because they remove variables from consideration. The fewer variables you need to consider, the simpler the problem

domain. Where should those fence posts be set? How should they be set? For such guidance, I'll turn away from the usual technical context toward some other, broader alternative context: the law. There's no aspect of your personal and professional life that isn't subject to the law. The law, like other rules you're subject to, self-imposed or not, are constraints.

Perhaps if we understand how those broader constraints are considered, defined, and imposed, we can gain a better understanding of how constraints can be best applied in our shared business and technical context. How can "good fences" make for "good neighbors" and how does that analogy broadly and specifically apply? I'll ponder that and a few other, deeper questions here.

There are many kinds of fences, all serving different purposes based on their physical characteristics. Some are purely decorative. Some are electrified and deadly. Most others fall somewhere in between. But what does it mean to fence-out versus fence-in? Why one or the other? As the names imply, when it comes to fencing, the context may be to keep nuisances out to prevent damage to my property. Or keep nuisances in to prevent damage to others' property. Almost all municipalities have rules and regulations concerning fencing. As an example of how extensive and non-trivial these rules and regulations may be, consider the state of Oregon: https://nationalaglawcenter.org/wp-content/uploads/assets/fencelaw/oregon.pdf. Those 16 pages of referenced statutory provisions also contain other referenced statutory provisions! A "fence" may indeed be more complicated than first thought because much of what a fence is and represents is intangible. Therefore, I should rephrase an earlier statement: fences may be physical in this context.

It turns out that what fences physically are versus what they represent in a legal sense are very different things. In other words, what things are **as implemented** is distinct from what things are **as defined.** It's no different with Agile. There's how it's defined and then there's how it's implemented. You have to consider both because the latter is the fusion of the former, plus people and process. The question is whether something implemented is compatible with what it's supposed to be, which entails what it means and how it's designed. A practical example is implementing Agile as Waterfall. These are not compatible things. Like diesel fuel in a gas engine, it's not a combination conducive to success. Therefore, understanding what things are, in relation to their intended context, is an immutable success factor. In this case, to have some understanding of what a fence is, you need to have some understanding of the rules that define what the default rule is, whether you're fencing in or fencing out, and, more importantly, **why** the default rule may be constructed as it is. Often, the rationale for a rule may be quite simple and pragmatic.

A good rule is one that considers fairness "equity," meaning that obligations are assigned on who or what can **best** bear the burden. In the United States, as we travel from east to west, the land is more open. Consider Montana and cattle ranching. Perhaps you've watched the popular Kevin Kostner series Yellowstone. Montana, like many western states, is considered "open range," meaning that adjacent property owners must "fence out." Imagine if all cattle ranchers had to physically "fence-in" their cattle. Such fencing would need to be robust and very expansive to the point that it would be impracticable to keep cattle fenced-in. Therefore, the burden of fencing is placed on those that would need to keep cattle "out" of their property. Therefore, such property owners must "fence-out." There are other governing constitutional rules that require uniform application.

Another important context is that these rules were enacted in one form or another well over 150 years ago. Sometimes these rules were more formal early on. In many cases, rules begin as and evolve from customs and norms. It can be a very difficult thing indeed to distill customs to a generic set of adopted rules to be applied uniformly. I emphasize **uniformity** sarcastically because there is nothing that prevents rule application being "uniformly" arbitrary and discriminatory! In our shared technology context, consider the never-ending posts on LinkedIn concerning another failed Agile/Scrum/DevOps. Those are all cases where **something new came to a pre-existing environment**, implying that a **new neighbor is moving in across the street!**

The other implication is that equitable and fairness principles may not result in an outcome to your liking. One simple reason is that in any project involving multiple teams, each team distinct in terms of its lifecycle and personnel, and each must cede something for the good of the order. There must be a cooperative effort toward a common objective. In the case of fencing duties in the open range, one objective is to set proper expectations among prospective members of a community.

Imagine that you live in the eastern U.S and that you decide that it's time to leave the city, get back to nature, and live a simple life. You end up moving west and building your homestead on a small 20-acre semi-wooded parcel in the middle of what appears to be something characterized to you as the open range. As it turns out, you were in too a big hurry to move! That's why you didn't know that the "open range," for purposes of your new locale, is a noun not an adjective. It also turns out that it isn't just a noun, it's a proper noun, meaning that it's a thing, with a legal definition that has bearing on the citizenry, even the new ones! Your neighbor, as it turns out, is a cattle rancher. If peace and quiet is the objective, your new locale and neighbor situation sounds like just what the doctor ordered! It's a shame when that peace and quiet was disturbed due to some errant cattle causing significant damage to your property. Suddenly, living on the open range becomes living on the "Open Range," with the fencing-out duties in tow. When we come to the party, as far as property law is concerned, we're on notice. Nobody forced you to move to the country. In a more populated, less agricultural sphere, you would represent the norm, perhaps with an opposite fencing rule. But in your new context, you're the outlier. That may not be a problem. It all depends on understanding what being the outlier really means. Not being a cattle rancher may be fine, so long as your objectives are compatible with your cattle rancher neighbor in the context of the rules you both are subject

(Continued on page 73)

CODE SECURITY

DREADING SHARING THAT YOUR APPLICATION CAUSED A DATA BREACH?

CODE SECURITY'S APPLICATION TESTING SERVICES CAN HELP YOU PREVENT THAT BREACH.

- My development team practices secure coding techniques ☐ Yes ☒ No
- My applications have been tested for security vulnerabilities ☐ Yes ☒ No
- Code Audits and Penetration Testing have been performed by a third party ☐ Yes ☒ No

CODE Security experts can help you identify and correct security vulnerabilities in your products, services and your application's architecture. Additionally, CODE Training's hands-on **Secure Coding for Developers** training courses educates developers on how to write secure and robust applications.

Contact us today for a free consultation and details about our services.

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

CODE
LEGACY

OLD TECH HOLDING YOU BACK?

Are you being held back by a legacy application that needs to be modernized? We can help. We specialize in converting legacy applications to modern technologies. Whether your application is currently written in Visual Basic, FoxPro, Access, ASP Classic, .NET 1.0, PHP, Delphi… or something else, we can help.

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