Markdown, Java, Rider, JavaScript

Features

Columns

Departments

US subscriptions are US $29.99 for one year. Subscriptions outside the US pay US $44.99. Payments should be made in US dollars drawn on a US bank. American Express, MasterCard, Visa, and Discover credit cards are accepted. Bill me option is available only for US subscriptions. Back issues are available. For subscription information, send e-mail to subscriptions@codemag.com.

Subscribe online at codemag.com

CODE Component Developer Magazine (ISSN # 1547-5166) is published bimonthly by EPS Software Corporation, 6605 Cypresswood Drive, Suite 300, Spring, TX 77379 U.S.A. POSTMASTER: Send address changes to CODE Component Developer Magazine, 6605 Cypresswood Drive, Suite 300, Spring, TX 77379 U.S.A.

Canadian Subscriptions: Canada Post Agreement Number 7178957. Send change address information and blocks of undeliverable copies to IBC, 7485 Bath Road, Mississauga, ON L4T 4C1, Canada.

LEADTOOLS®
THE WORLD LEADER IN IMAGING SDKs

Multi-platform PDF SDK

Comprehensive PDF SDK technology for today's digital world. Develop multi-platform applications with support for extracting, editing, and writing text, hyperlinks, bookmarks, digital signatures, PDF forms, annotations, metadata, and pages from PDF files on any desktop, server, and mobile device.

Specific Features:

- View & Save on Desktop, Web & Mobile
- Extract, Delete, Insert & Replace Pages
- Convert Images to Searchable PDF
- Convert to PDF/A
- Load, Save & Convert Annotations
- PDF Forms

Microsoft .NET · iOS · macOS

Get Started Today
DOWNLOAD OUR FREE EVALUATION

LEADTOOLS.COM

Act 2: Full Steam Ahead

A few issues ago, I started talking about a project that my team was working on with a client. It's been nearly seven months since this project's initiation and a ton of progress has been made. The major change was in the size of the development team. When we started this project, the team was rather small: six developers and a project manager. The purpose of this small development team was to figure out the "how" aspect of the project. How the heck were we going to consume this elephant-sized project and scale up the team? After a few months of work, we made the jump and grew our team by around 10 developers. We did a formal three-day training session and after that, we went to work. In the last 60 days, we've extended the team even further and now our development team is approximately 30 people.

It's been a learning process, and this is where I "knock on wood:" This project is going well, beyond all my expectations. We've managed to complete the first major milestone of the project and are proceeding to move parts of the project into UAT (User Acceptance Testing). Throughout, we've learned a number of lessons. Here are some of them.

Playing to People's Strengths

In his book "The Mythical Man Month," Fred Brooks discusses the concept of organizing development teams like surgical teams. The surgeon performs the critical work while the remainder of the team supports the surgeon. Our team flipped this around. We chose small groups of developers to work on critical services of the system. These critical functions included things like frameworks, build management, help systems, paperless automation systems, user interface design, etc.

The developers of these features were considered support developers. Their job was two-fold: Build critical features while supporting other developers in their understanding and usage of these systems. Initially, these system-critical features were given to the more senior developers. As the development process progressed, we distributed the responsibility for these features to more developers as their depth of knowledge increased, which leads to my next concept, enabling developers.

Enabling Developers

One of the main goals of this project was to do a better job of enabling our developers. Every—and I mean EVERY—developer has good ideas that can help make the team more productive. We've tried to give developers the ability to try things and then to propagate those new things to the other developers when they make sense.

One of my favorite stories is how we managed the distribution of our stored procedure script files. Yes, this system is built on SQL Server and uses an extensive number of stored procedures. As a matter of fact, we have 1,000+ stored procedures that have been converted so far. When we started this project, we just ran all the scripts each time we updated our code from our master branch. This worked great when there were around 100 scripts. It only took a few seconds. It was when we added ten more developers that the flaws in this process became evident. It was taking too long to run this ever-growing number of script files. We had to do something.

Enabling developers was the answer. They created bash scripts to store the script names and dates most recently run in a local database table, and power shell scripts were attempted. Ultimately, a new script runner utility was created that took the best ideas from each developer and consolidated them into a single process. It came down to detecting changes in scripts by calculating a hash value against a stored hash value. If there were differences, the script was run and the stored hash updated. This process turned a long-running operation into a very quick run through 1000 files. This was ALL a result of enabling developers.

If there's one thing you can do to improve your development process, it's enabling developers. Enabled developers are happy developers. Each developer's ideas need to be considered carefully and, where they make sense, distributed to your team.

Don't Be a Blocker

At my core, I'm a programmer. I love writing code and after 20+ plus years, I still raise my fist in triumph when stuff works. Come on now, you know you do that too!

The sheer size of this project has greatly reduced the amount of time I could spend coding. As the project grew, I found myself getting increasingly frustrated as the features I was assigned to work on seemed to make little or no progress. This was because I was spending most of my day helping other developers and spending only an hour or two on my features. I was becoming a "blocker."

I eventually concluded that my job on this project was to help the other developers accomplish their tasks. As soon as possible, I transferred the features I was responsible for to other developers. Almost immediately, I felt better about what I was accomplishing each day. I was able to help other developers accomplish their goals. This gave me great satisfaction, as ultimately, what we do as software developers is help other people, be it users or other developers. Did I stop writing code? Nope! I still write code. I now focus my time coding features, tasks, or tooling that help the team accomplish their goals.

What's Next

This project is moving into a new phase. The UAT phase. This is where our code faces real users. I'll be back in a few months to let you know how that goes!

Rod Paddock
CODE

RAVENDB 4.1

Your Fully Transactional NoSQL Database

The amount of data your organization needs to handle is rising at an ever-increasing rate. We developed RavenDB 4.1 so you can handle this tougher challenge and do it while improving the performance of your application at the same time.

Enjoy top performance while maintaining ACID Guarantees

- 1 million reads/150,000 writes per second on a single node
- Native storage engine designed to soup up performance

All-in-One Database

- Map-reduce and full-text search are part of your database
- No need for plugins

Performs well on smaller servers and older machines

- Unprecedented hardware resource utilization
- Works well on Raspberry Pi, ARM Chips, VM, In-Memory solutions

Easy to Setup and Secure

- Setup wizard gets you started in minutes with top level data encryption for your data

Distributed Counters

- Increment a value without modifying the whole document
- Automatically handle concurrency even when running in a distributed system

Works with SQL Solutions

- Seamlessly send data to SQL databases
- Migrate easily from your current SQL solutions

Expand and Contract Your Cluster on the Fly

- Add new nodes in a click
- Your data cluster is fully transactional
- Assignment failover reassigns tasks from a downed node instantly

The DBAs Dream

- The most functional GUI on the market, manage your server directly from the browser without doing any complex configuration from the command line

Grab a FREE License

3-node database cluster with GUI interface, 3 cores and 6 GB RAM

www.ravendb.net/free

Docker for Developers

I've spent a large portion of my adult life in the Microsoft ecosystem. When I started doing a lot of SharePoint work, I discovered the magic of virtualization. The platform was wonky! (That's the technical term.) A lot of what I had to do involved some code, some IT pro, some business-user skills, some convincing, a little chicken blood, some luck, and viola, another day I was the hero.

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. You can find more about his training at http://www.winsmarts.com/training.aspx.

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

One of my best friends on my SharePoint journey was virtualization. As a consultant, it was invaluable for restoring a client's dev environment easily. I was so happy when I finally bought a laptop that could virtualize SharePoint with acceptable performance. Sure, the battery life sucked, but then I just got myself a bigger battery. I did realize at some point that saying "this laptop is da bomb" was not a great choice of words, along with saying "hi" to my friend Jack, at airports.

Let's be honest, virtualization rocked, but the heavy-duty laptop didn't. I was so jealous of the thin and slim computers my friends carried.

Could I have a thin computer, the fun and joy and speed of working on native, and all the benefits of virtualization? Yes! With Docker, I could.

You're not going to run SharePoint on Docker anytime soon, but as a developer, it opens up so many possibilities. I've found it really useful in my new wave of development, where I do a lot of development around .NET Core, NodeJS, Python, etc. Put simply, Docker rocks. It gives me all the benefits of virtualization, with none of the downsides. And although it doesn't address every single platform, it addresses enough for it to be my best friend in today's world.

What is Docker?

Docker is a computer program that performs operating system level virtualization, also known as containerization. It runs software packages as "containers." The word "container" is borrowed from the transport industry. The container you see on the back of a truck, on a train, or on ship, those are all the same container. Because these containers were standardized, it made transportation a whole lot easier. Cranes could be built to lift a container from a ship to a train. Imagine what you'd have to do before that? You'd have to open every container, and unpack and pack goods from ship to train, train to truck, and it was all so inefficient.

Despite this good example, in the software industry, we continue to make this mistake. Every single time we have an application we need to deliver, we go through the same old rigmarole of setting up a Web server, setting up a website, a database, a firewall—ugh!

The advent of the cloud made these tasks seem possible with efficiency and ease because we want to control the environment our application runs in. We don't want other people's applications on a shared infrastructure to interfere with ours. We want efficiency and reliability. We want to ship our application packaged up as a container, easily configurable by our customers, so they can set things up quickly and easily. Most of all, we want reliability and security!

Docker simplifies all that. It originated on Linux, but gradually this concept of containerization is making it to Windows also.

Put in very simple terms, using Docker, you can package an application along with all its dependencies in a virtual container and run it on any Linux server. This means that when you ship your application, you gain the advantages of virtualization, but you don't pay the cost of virtualizing the operating system.

> Using Docker, you can package an application along with all its dependencies in a virtual container and run it on any Linux server.

How much you virtualize is your choice and that's where the lines begin to muddy. Can you run Windows as a Docker image on Linux? Well, yes! But then you end up packing so much into the Docker image that the advantage of Docker begins to disappear. Still, you do have the concept of Windows containers. You can run Windows containers on Windows. And given that Linux is so lightweight, you can run Linux Docker images on Windows and they start faster than your usual Office application.

The State of Containerization

As much as I've been a Microsoft fan, I have to give props to Linux. Windows has always taken the approach of being flexible and supporting as much as they could out of the box. That's the "kitchen sink" approach. As a result, over the years, the operating system has gotten bigger and heavier. This wasn't an issue because CPUs kept up with the demand. We had an amazing capability called RDP (remote desktop protocol), that furthered our addiction to the kitchen sink approach. Over a remote network connection, we had the full OS, replete with a Start button.

Linux, on the other hand, didn't enjoy such a luxury. They had GUI, VNC, and similar things. But at its very heart, a Linux developer SSHed (secure socket shell) into a computer. The advantage this brought forth was that Linux always had a script-first mentality. And this really really shows when it comes to running Linux in the cloud. As a Linux developer, I don't care about the GUI; in fact, frequently the GUI gets in the way. I have a pretty good terminal, and I have SSH. Between SSH and a pretty good terminal, I'm just as productive on a remote computer on the moon with a very long ping time and a really poor bandwidth as I am on my local computer.

Although the gamble of a one-size-fits-all operating system right out of the box worked out well for Windows and Microsoft, the advent of the cloud challenged this approach. For one thing, running unnecessary code in the cloud started equating to real dollars in operational cost. But more than that, the lack of flexibility it lent meant that everything needing to be scriptable wasn't scriptable. As a result, set up became more expensive and complex when working over a wire. Now, surely, solutions exist, and Windows hasn't exactly been sitting on its thumbs either. Over the past many years, there have been fantastic innovations in Azure that let you work around most of these issues, and Windows itself has a flavor of containerization baked right in.

However, where containerization still really shines is on Linux and Docker. Although it may be tempting to think that containerization is a godsend for IT pros, the reality is that it's a godsend for developers too.

Let me explain.

As a developer, my work these days is no longer booting heavy-duty SharePoint VMs. In fact, even when I'm working with SharePoint, I'm writing a lot of TypeScript and JavaScript. When I'm working on AI, I'm doing lots of Python. The rest of the time, I'm either in .NET Core or some form of NodeJS.

The problem still remains: Working on multiple projects, I need to juggle various operating system configurations. And when I'm ready to ship my code, I want to quickly package everything and send it away in a reliable and efficient form.

As an example, one project I'm working on is still stuck in Node 6. I can't upgrade because the dependencies don't build on Node 8. I have no control on the dependencies. My choice is to use **npx** to switch node versions. Or, I could give myself a Node 6 dev environment running in Docker. I can SSH into this environment, expose it as a terminal window in VSCode, and at that point, for all practical purposes, I'm on an operating system that's Node 6-ready.

Here's another example. My Mac ships with Python 2x. But a lot of work I'm doing requires Python 3x. I don't want to risk breaking XCode by completely gutting my OS and forcing it to run Python 3 for everything. I know solutions exist and I know that many people use them. But I also know that as versions go forward, I'm not convinced that it's the most efficient use of my time.

Wouldn't it be nice if I could just have a VM that I could SSH into, and that was already configured with Python 3? And that I could launch this VM faster than MS word, and then almost not feel like I'm indeed in a virtualized environment. That's the problem Docker solves.

So you see, Docker is indeed quite valuable for developers. In the rest of the article, I'll break down how I built myself a dev environment using Docker. Let's get rolling.

Basic Set Up

Before I start, I'd like you to do some basic setup. First, go ahead and install Docker. For the purposes of this article, the community edition is fine. Installing Docker on Mac or Linux is a bit more straightforward than on Windows. You can find the instructions specific to your OS here: https://docs.docker.com/v17.12/install/. What's interesting is that all cloud vendors also support Docker. This means that whatever image you build, you can easily ship it to AWS, Azure, or IBM cloud. This is truly a big advantage of Docker.

> It's interesting that all cloud vendors already support Docker.

When it comes to running Docker in Windows, you'll have a couple of additional considerations. First, you'll need Hyper-V. Once you enable Hyper-V, things like VirtualBox and VMware workstation will no longer work. The other thing you'll have to consider is switching between Windows containers and Linux containers. All that aside, once you set up your Windows computer with Hyper-V, install Docker, and configure it to use Linux containers, you should be pretty much at the same spot as on a Mac or Linux computer. The instructions for Windows can be found here, https://docs.docker.com/docker-for-windows/install/.

There's one good take-away here. Windows can support both Windows and Linux containers. And although the toggling and initial set up may not be as convenient, you have one OS supporting both. Mac and Linux, on the other hand, require much more complex workarounds to run Windows containers. And when they do run, they run a lot heavier. In that sense, Windows, in a weird way, is preferable as a dev environment. Of course, a lot of dev work we do these days is cross-platform anyway.

The next thing you'll need is Visual Studio Code. And along with VSCode, go ahead and install the VSCode extension for Docker here https://marketplace.visualstudio.com/items?itemName=PeterJausovec.vscode-docker.

You can also optionally install Kitematic from https://kitematic.com/. Using Docker involves running a lot of commands, usually from a terminal. Once you get familiar with them, you can just use the terminal. But if you prefer a GUI to do basic Docker image and container management, you'll find Kitematic very useful.

With all this in place, let's start by building a Docker dev environment.

A Docker-based Dev Environment

The dev environment I wish to build will be based on Linux. Frequently, I wish to quickly spin up a Linux environment for fun and dev. Although a VM is awesome, I want something lighter, something that spins up instantly at almost zero cost. And I want something that, if I left it running, I wouldn't even notice.

Docker images always start from a base image. The reason I picked Linux is because Linux comes with the most

stripped-down bare-bones starter image. This means that all of the images I build on top of it will be light too, and I get to pick exactly what I want.

Set Up the Docker Image

To build a Docker image, you start by picking a base image. You can find the full code for this image here: https://github.com/maliksahil/docker-ubuntu-sahil. Let me explain how it's built.

First, create a folder where you'll build your Docker image. Here, create a new file and call it **Dockerfile**. Start by adding the following three lines:

```
FROM ubuntu:latest
LABEL maintainer=
"Sahil Malik <sahilmalik@winsmarts.com>"
```

In these lines, you're saying that your base image will be ubuntu:latest. Doing this tells Docker to use the Docker registry and find an image that matches this criterion. Specifically, the image you'll use is this: https://hub.docker.com/_/ubuntu/.

Right here, you can see another advantage of Docker over VMs. Because it's so cheap to build and delete these containers, you can always start with the latest and not worry about patches. Try doing that with VM snapshots. I used to spend half a day every week updating Windows VM snapshots. Not anymore! Sorry, I don't mean to harp on Windows, but this is the pain I've dealt with. Don't ask me to hold back.

The third line is pure documentation. I put my name there because I am maintaining that image.

Now this command gives me a bare-bones Ubuntu image. But I want more! I want my dev tools on it. Usually when I work, I like to have Git, Homebrew, a shell like Zsh, an editor like GNU nano, and Powerline fonts with Git integration, and the agnoster theme. Additionally, when this Docker Ubuntu image is set up, I'm essentially working in it as **root**. That's not ideal! I wish to work as a non-root user, like I usually do. So, I also want to create a sudo-able user called **devuser**.

Listing 1: Install the basic tools

```
RUN apt-get update && \
    apt-get install -y sudo curl git-core gnupg
   linuxbrew-wrapper locales nodejs zsh wget nano
    nodejs npm fonts-powerline && \
    locale-gen en_US.UTF-8 && \
    adduser --quiet --disabled-password
    --shell /bin/zsh --home /home/devuser
    --gecos "User" devuser && \
    echo "devuser:p@ssword1" |
    chpasswd &&  usermod -aG sudo devuser
```

Listing 2: Installing the theme

```
ADD scripts/installthemes.sh /home/devuser/installthemes.sh
USER devuser
ENV TERM xterm
ENV ZSH_THEME agnoster
CMD ["zsh"]
```

All of this is achieved through the next line of code I add in my Dockerfile, as can be seen in **Listing 1**. There's a lot going on in **Listing 1**, and I've added a bunch of line breaks to make it more readable. The RUN instruction in a Dockerfile executes any commands in a new layer on top of the current image and commits the results. The resulting committed image will be used for the next step in the Dockerfile.

Speaking of the next step. Next, I'd like to give the user of the image the choice of using agnoster, which is a great Zsh theme. This is achieved in the Dockerfile steps shown in **Listing 2**.

Build the Docker Image

If you were thinking this was going to be complex, you were wrong. My Docker image is done! All you need to do now is build it. Before you build it, ensure that Docker is running. There are two things you need to know about Docker: the concept of images and the concept of containers.

An **image** is what you just built above: It's a representation of everything you wish to have, but think of it as configuration. You've specified what you'd like to be in this image, and now, based on the image, you can create many containers.

A **container** is a running instance of a Docker image. Containers run the actual applications. A container includes an application and all of its dependencies. It shares the kernel with other containers and runs as an isolated process in user space on the host OS.

While we're at it, let's understand a few more terms.

A **Docker daemon** is a background service running on the host that manages the building, running and distributing Docker containers.

Docker client is a command-line tool you use to interact with the Docker daemon. You call it by using the command **docker** on a terminal. You can use Kitematic to get a GUI version of the Docker client.

A **Docker store** is a registry of Docker images. There is a public registry on Docker.com where you can set up private registries for your team's use. You can also easily create such a registry in Azure.

Your usual workflow will be as follows:

1. Pull a Docker image from a Docker registry.
2. Run a container based on the Docker image.
3. Start and stop this container as many times as you want, and you can easily revert it back to the original image definition.
4. Take a "snapshot," which I'll talk about later.

In this case, you need to first build an image out of the Dockerfile you just created. Creating an image is very easy. In the same directory as where the Dockerfile resides, issue the following command:

```
docker build --rm -f Dockerfile -t ubuntu:sahil .
```

Running this command creates the image for you. You can fire up Visual Studio Code, and if you've installed

the Docker extension, you should see the image shown in **Figure 1**.

The Docker image is ready, but how do you use it? Well, you have to run it, effectively creating a container. You can do so by issuing the following command:

```
docker run --rm -it ubuntu:sahil
```

Because the Docker image specifies Zsh as the shell, you should see a message prompting you to create a ~/.zshrc file. Choose **option 2** to populate using the default settings.

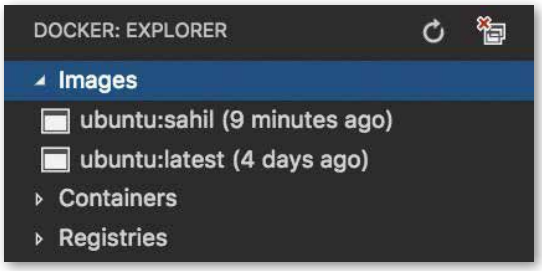


Figure 1: The newly created Docker image

```
--- Type one of the keys in parentheses --- 2
/home/devuser/.zshrc:15: scalar parameter HISTFILE created globally in function zsh-newuser-install
(eval):1: scalar parameter LS_COLORS created globally in function zsh-newuser-install
devuser@b57189f5d27f / %
```

Figure 2: My Docker container terminal

Almost immediately, you're landed on to the terminal, as shown in **Figure 2**.

Effectively, you ran the container, and now you're SSHed in. Congratulations! You have a dev computer running. You can verify in VSCode that a new container has been created for you. This can be seen in **Figure 3**.

The container is running, and you can start using the computer. It doesn't do much yet! Type **exit** to get out of the container. Notice that the container is now gone. What happened? Let's see if the container is hidden by issuing the following command:

```
docker container ls --all
```

It isn't hidden! It is indeed gone! What's going on? Where's all your work?

What happened is that when you exited the container, things reverted back to the image. This may sound strange if you're used to using virtual machines, but it does make sense. You always want to go back to a reliable clean version of your dev environment. The persistent work should ideally be on a volume mounted into the Docker container.

You can easily run a container and mount the current working directory into a folder called /developer using the following command:

```
docker run --rm -it -v `pwd`:/developer ubuntu:sahil
```

When you run this command, you effectively start the container and mount the current working directory in the /developer folder. This can be seen in **Figure 4**.

This is incredibly powerful because you can keep all of your work safe on your host computer, and effectively use Git integration on the host computer. Or perhaps you could launch a Docker container purely for Git integration. For instance, I use Git with two-factor auth for work, and I have a fun Git repo for other dev work. I keep the two separate without worrying about one-time passwords or SSH keys, by using—you guessed it—a Docker image.

But this brings me to another question. That terminal prompt you see in **Figure 2** is hardly attractive. I like Git integration, I like agnoster. I like those fancy arrows I see on my host computer in **Figure 4**. Or perhaps more generically, I want to install NodeJS and agnoster and use that, and I wish not to repeat those steps every single time.

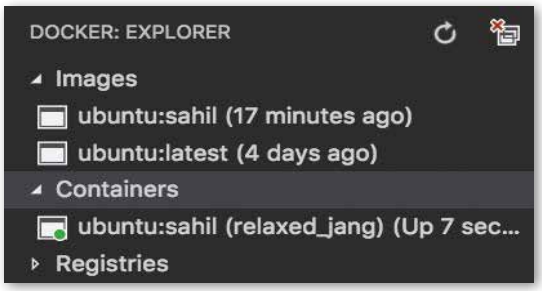


Figure 3: The container as shown in VSCode

```
devuser@2be4107173fd / % ls /developer
Dockerfile  README.md  build.sh  dependents.py  run.sh  scripts
devuser@2be4107173fd / % exit
 sahilmalik@ ~/Documents/Personal/Developer/docker/docker-ubuntu-sahil  master  ls
Dockerfile    README.md     build.sh      dependents.py run.sh        scripts
 sahilmalik@ ~/Documents/Personal/Developer/docker/docker-ubuntu-sahil  master
```

Figure 4: The container with a mounted volume

Effectively, I want to:

1. Launch an image.
1. Do some work.
2. Save that "snapshot" so next time I can treat that snapshot as my image.

Luckily, that's possible.

Snapshot Your Work

In order to snapshot your work, first let's put in some work! Specifically, let's install the agnoster theme and powerline fonts so within my Docker image, I can see Git integration on the terminal.

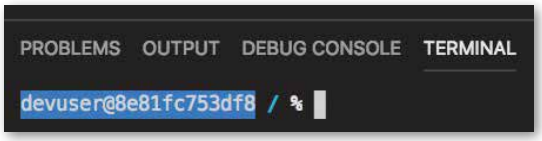


Figure 5: My newly minted container

Ensure that there are no Docker containers running and fire up a container using the following command:

```
docker run --rm -it ubuntu:sahil
```

As you can see, I can do this entirely in the VSCode shell, and my terminal now looks like **Figure 5**.

Now, go ahead and install Zsh themes, and change the theme to agnoster by issuing the following command on the Docker container terminal. Note that I've added line breaks for clarity.

```
wget
  https://github.com/robbyrussell/
  oh-my-zsh/raw/master/tools/install.sh -O -
  | zsh
```

Running this command shows you the message in **Figure 6**.

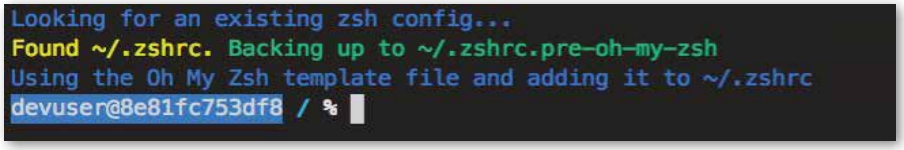


Figure 6: Message about the zshrc file

You need to edit the ~/.zshrc file to set the theme as agnoster. Because you installed nano earlier, you can simply type **nano ~/.zshrc** and modify the theme to agnoster. Now launch Zsh one more time. Your prompt should look like **Figure 7**.

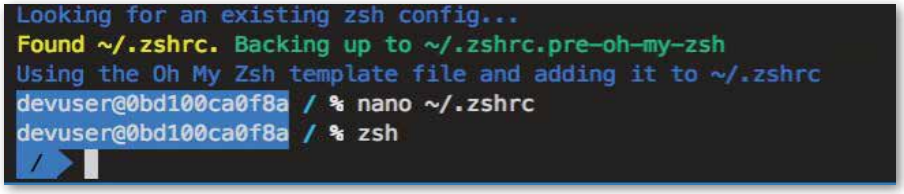


Figure 7: The agnoster prompt in zsh

Let's quickly test Git integration as well. You can do so by issuing the commands shown in **Figure 8**.

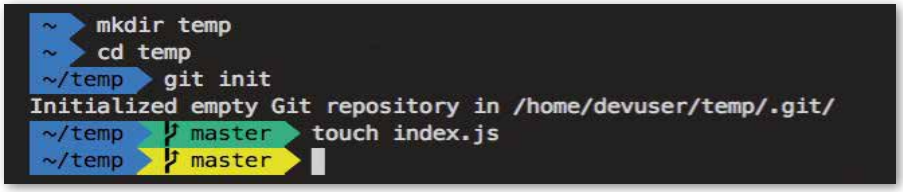


Figure 8: Git integration working

As can be seen in **Figure 8**, the arrows and the word "master" denote that you are on the master branch, and the yellow color denotes that there are changes.

This is simply fantastic. All you need to do now is remove that "temp" directory and snapshot your work. So go ahead and cd .. and remove the temp directory first. But don't close the Docker shell prompt.

In your host OS shell, first let's check what images you have. You can do so by running the following command:

```
docker image ls
```

Running this command shows the images in **Figure 9**. Note that your images will most likely be different. But you should have a common image of ubuntu:sahil

Figure 9: The images I have on my computer

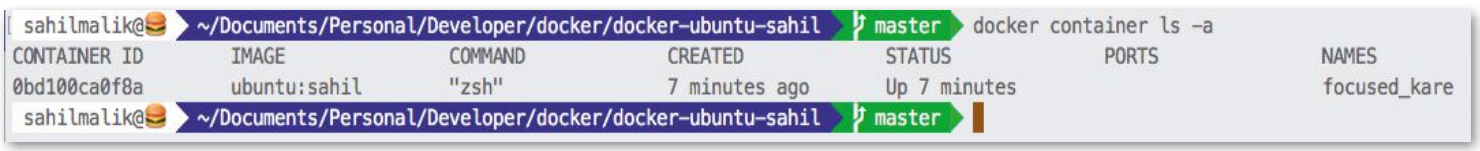


Figure 10: The containers that are running

Now let's check what containers I have, as can be seen in **Figure 10**.

That particular container, called "focused_kare" is where all your work is. You wish to save it and relaunch that state next time. You can do so by issuing the following command:

```
docker commit 0bd100ca0f8a ubuntu:sahil-zsh
```

Verify that this new image appears in VSCode Docker extension also, as you can see in **Figure 11**.

Now let's run this image using the following command:

```
docker run --rm -it ubuntu:sahil-zsh
```

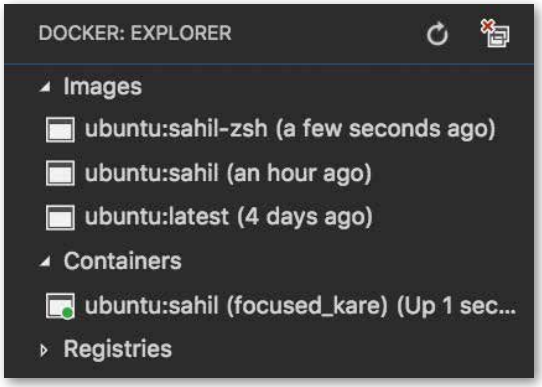


Figure 11: Your newly created image

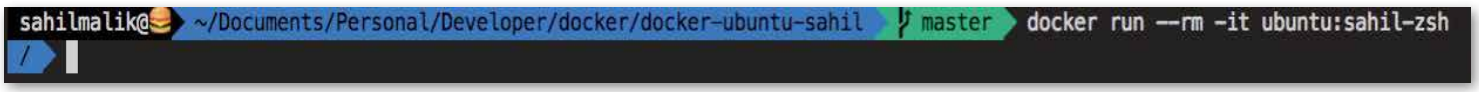


Figure 12: A new container with your saved work.

Running this command immediately launches you into a Zsh shell with agnoster theme running, as can be seen in **Figure 12**.

You have now effectively snapshotted your work. Imagine the power of this. You can now create a dev environment that surfaces up as a SSHed terminal in VSCode. This dev environment can have, say, Python3 with Git connected to a completely different repo. All of your code can remain on the host because you have a mounted folder.

Really, at this point, as far as Linux work is concerned, and as long as you don't care about the GUI, you have effectively snapshotted your VM. In the process, you've gained a few more things:

- Your dev computer configurations always remain reliable, because they always revert to an effectively immutable OS whenever you type "exit."
- They force you to keep all configurations as code externally, thereby promoting good practices in easily being able to reproduce the environments anywhere.
- They're so lightweight, it almost feels like launching notepad. In fact, on my main dev computer, which is a 2017 MacBook pro 15", I have dozens of Docker images running, and I almost don't even notice it. Of course, this depends on what a Docker image is doing. But for my purposes, my images are the usual dev stuff, NodeJS, Python, .NET Core, etc.

Summary

I can't thank virtualization enough. Over the years, it's saved me so much time and hair. All that configuration was so hard to capture otherwise. Despite all the talks and classes I have given, I would never have the confidence of things working if I didn't capture every single detail and then reproduce the environment with a "play" button. Or, consider all the clients I've worked for, with each of them requiring their own funky VPN solution that didn't play well with others. How could I have possibly worked without virtualization?

Virtualization has an added advantage. When the airline I was sitting in was being cheap with the heat and I was effectively freezing in my seat, I could boot up a couple of SharePoint virtual machines to keep my whole row warm. True story.

Docker gives me that same power but many times over. Because now I can build these images for dev work and ship them easily to any cloud provider or even on-premises solutions. They take so little power and there are so many images available to start from. You need a lightweight reverse proxy? Check out NGINX reverse proxy at https://hub.docker.com/r/jwilder/nginx-proxy/. You want a tensorflow image? Check out https://github.com/maliksahil/docker-ubuntu-tensorflow.

What's really cool is that this same image can run on your local computer, a remote server, or scale up to the cloud, with zero changes. And these Docker images are so lightweight, they can literally run for the price of a coffee per month.

Beat that, virtualization!

I could go on and on extolling the virtues of Docker. It's just like virtualization, with none of the downsides. Of course, with every solution come new problems.

There's the problem of orchestrating all these containers, for instance. Can I have a container with a Web server and another container with a database start together? And if one dies, does the other gracefully exit, or, for that matter, scale, and then restart in a sequence, or any number of such combinations across thousands of such containers?

Well that is what Kubernetes is for. I'll leave that for the next article.

But let me just say this, once you Docker, you are the rocker!

Sahil Malik

CODE

Security in Angular: Part 3

In my last two articles (CODE Magazine July/August and September/October 2018), you created a set of Angular classes to support user authentication and authorization. You also built a .NET Core Web API project to authenticate a user against a SQL Server table. An authorization object was created with individual properties for each item that you wished to secure in your application. In this article, you're going to build an array of claims and eliminate the use of single properties for each item that you wish to secure. Using an array of claims is a much more flexible approach for large applications.

Paul D. Sheriff
www.fairwaytech.com

Paul D. Sheriff is a Business Solutions Architect with Fairway Technologies, Inc. Fairway Technologies is a premier provider of expert technology consulting and software development services, helping leading firms convert requirements into top-quality results. Paul is also a Pluralsight author. Check out his videos at http://www.pluralsight.com/author/paul-sheriff

Download the Starting Application

Since the time I published the first article in this series, Angular 6 has released. I've migrated the application for this article to Angular 6. The only change that affected the sample application is the location of the RxJS classes.

Download the starting sample for this article from the CODE Magazine website page associated with this article, or from http://fairwaytech.com/downloads. Select "PDSA/Fairway Articles" from the Category drop-down, and choose "Security in Angular—Part 3." After you've downloaded the ZIP file, there are two folders within the ZIP entitled "Start" and "End." Extract all of the files and folders within the "Start" folder to a folder somewhere on your hard drive. You can then follow along with this article to build an array of claims from a UserClaim table.

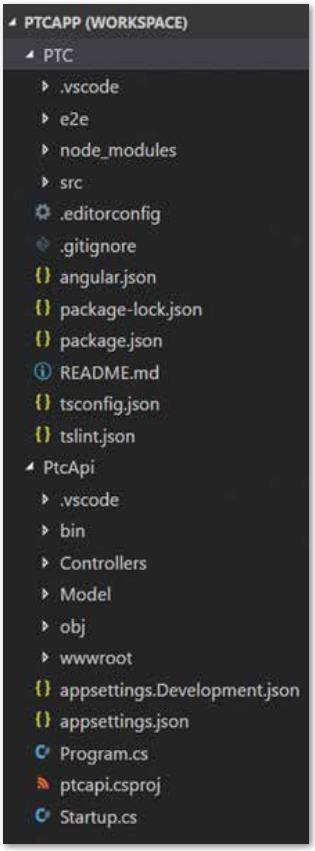


Figure 1: The starting application has two projects, the Angular project (PTC) and the .NET Core Web API project (PtcApi).

Load the Visual Studio Code Workspace

After extracting the "Start" folder from the ZIP file, double-click on the **PTCApp.code-workspace** file to load the two projects in this application. A workspace is loaded that looks like **Figure 1**. There are two projects: **PTC** is the Angular application and **PtcApi** is the ASP.NET Core Web API project.

The PTC Database

There's a SQL Server Express database named **PTC** included in the ZIP file. Open the **PtcDbContext.cs** file located in the **\PtcApi\Model** folder. Change the path in the connection string constant to point to the folder in which you installed the files from this ZIP file. If you don't have SQL Server Express installed, you can use the **PTC.sql** file located in the **\SqlData** folder to create the appropriate tables in your own SQL Server instance.

Security Tables Overview

The PTC database has two tables besides the product and category tables: User and UserClaim (**Figure 2**). These tables are like the ones you find in the ASP.NET Identity System from Microsoft. I've simplified the structure just to keep the code small for this sample application.

User Table

The user table generally contains information about a specific user such as their user name, password, first name, last name, etc. For the purposes of this article, I've simplified this table to just a unique ID (UserId), the name for the login (UserName), and the Password for the user. Please note that I'm using a plain-text password (**Figure 3**) in the sample for this application. In a production application, this password would be either encrypted or hashed.

User Claim Table

In the UserClaim table, there are four fields: ClaimId, UserId, ClaimType, and ClaimValue. The ClaimId is a unique identifier for the claim record. The UserId is a foreign key relation to the User table. The value in the ClaimType field is the one that's used in your Angular application to determine if the user has the appropriate authorization to perform some action. The value in the ClaimValue can be any value you want. I'm using a true or false value for this article.

You don't need to enter a record for a specific user and claim type if you don't wish to give the user that claim. For example, the CanAddProduct property (**Figure 4**) in the authorization object may either be eliminated for the user **bjones**, or you can enter a false value for the ClaimValue field. Later in this article, you learn how this process works.

Modify Web API Project

Now that you have the database tables configured for your users, you need to modify a few things in the Web API application to support an array of claims. In the previous article, the AppUserAuth class contained a Boolean property for each claim. You tested this Boolean using an Angular *ngIf directive to remove HTML elements from the DOM, thus eliminating the ability for a user to perform some action.

Using individual properties for each claim makes your AppUserAuth class become quite large and unmanageable when you have more than just a few claims. It also means that when you wish to add another claim, you must add a new record to your SQL Server, add a property to the AppUserAuth class in your Web API, add a property to the AppUserAuth class in your Angular application, and add a directive to any DOM element you wish to secure.

Using an array-based approach, you only need to add a record to your SQL Server and add a directive to a DOM element that you wish to secure. This means you have less code to modify, less testing to perform, and thus, your time deploying a new security change decreases.

Modify AppUserAuth Class

Open the **AppUserAuth.cs** file in the **\PtcApi\Model** folder and remove each of the individual claim properties you created in the last article. Add a generic list of AppUserClaim objects with the property name of Claims. You need to add a Using statement to import the **System.Collections.Generic** namespace. You should also initialize the Claims property to an empty list in the constructor of this class. After making these changes, the AppUserAuth class should look like **Listing 1**.

Modify Security Manager

The **SecurityManager.cs** file located in the **\PtcApi\Model** folder is responsible for interacting with the Entity Framework to retrieve security information from your SQL Server tables. Open the **SecurityManager.cs** file and remove the **for** loop in the BuildUserAuthObject() method that uses reflection to set property names. The following code snippet is what the BuildUserAuthObject() method should look like after you have made these changes:

```
protected AppUserAuth
  BuildUserAuthObject(AppUser authUser)
{
  AppUserAuth ret = new AppUserAuth();

  // Set User Properties
  ret.UserName = authUser.UserName;
  ret.IsAuthenticated = true;
  ret.BearerToken = BuildJwtToken(ret);

  // Get all claims for this user
  ret.Claims = GetUserClaims(authUser);

  return ret;
}
```

You also need to locate the BuildJWTToken() method and remove the individual properties being set. Each line where these properties are being set should present a syntax error in Visual Studio code because those properties no longer exist.

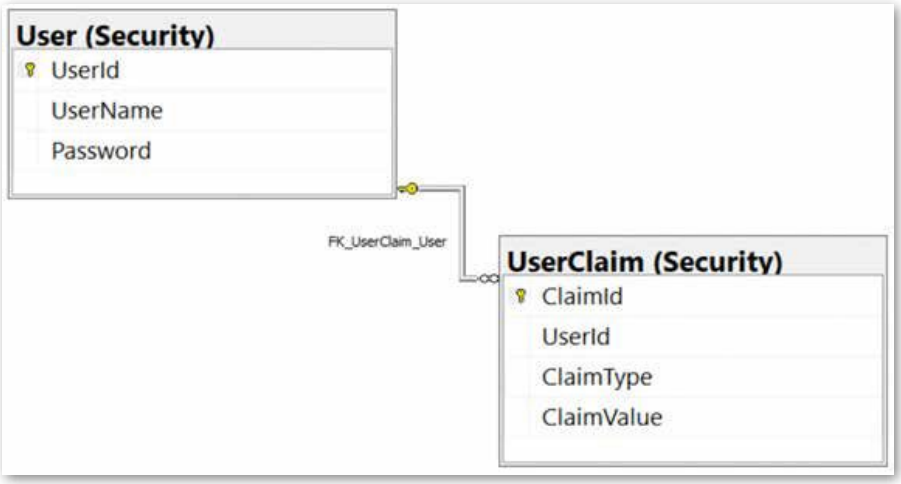


Figure 2: Two security tables are needed to authenticate and authorize a user.

Results Messages

	UserId	UserName	Password
1	4A1947EC-099C-4532-8105-64CF8C8B4B94	PSheriff	P@ssw0rd
2	898C9784-E31F-4F37-927F-A157EB7CA215	BJones	P@ssw0rd

Figure 3: Example data in the User table.

ClaimId	UserId	ClaimType	ClaimValue
cd98b0b88	4a1947ec-099c-453...	CanAccessCategories	true
cbb1b376-e...	4a1947ec-099c-453...	CanAccessProducts	true
408b006c-e...	4a1947ec-099c-453...	CanAddProduct	true
4eb3c52b-1...	4a1947ec-099c-453...	CanSaveProduct	true
ec624240-6...	4a1947ec-099c-453...	CanAddCategory	true
0d8004ca-a...	898c9784-e31f-4f37...	CanAccessCategories	true
71baa2ad-e...	898c9784-e31f-4f37...	CanAccessProducts	true
057f4ecc-82...	898c9784-e31f-4f37...	CanAddCategory	true

Figure 4: Example data in the UserClaim table.

Listing 1: Modify the AppUserAuth class to use an array of user claims

```
using System.Collections.Generic;

namespace PtcApi.Model
{
  public class AppUserAuth
  {
    public AppUserAuth()
    {
      UserName = "Not authorized";
      BearerToken = string.Empty;
      Claims = new List<AppUserClaim>();
    }

    public string UserName { get; set; }
    public string BearerToken { get; set; }
    public bool IsAuthenticated { get; set; }
    public List<AppUserClaim> Claims { get; set; }
  }
}
```

Listing 2: Check for a claim type and optionally a claim value in the isClaimValid() method

```
private isClaimValid(claimType: string):boolean {
  let ret: boolean = false;
  let auth: AppUserAuth = null;
  let claimValue: string = '';

  // Retrieve security object
  auth = this.securityObject;
  if (auth) {
    // See if the claim type has a value
    // *hasClaim="'claimType:value'"
    if (claimType.indexOf(":") >= 0) {
      let words: string[] = claimType.split(":");
      claimType = words[0].toLowerCase();
      claimValue = words[1];
    }
    else {
      claimType = claimType.toLowerCase();
      // Get the claim value, or assume 'true'
      claimValue = claimValue?claimValue:"true";
    }
    // Attempt to find the claim
    ret = auth.claims.find(
      c => c.claimType.toLowerCase()
             == claimType
           && c.claimValue == claimValue)
             != null;
  }

  return ret;
}
```

Modify the Angular Application

As is frequently the case with Angular applications, if you make changes in the Web API project, you need to make changes in the Angular application as well. Let's make those changes now.

Add an AppUserClaim Class

Because you're now going to be returning an array of AppUserClaim objects from the Web API, you need a class named AppUserClaim in your Angular application. Right mouse-click on the **\security** folder and add a new file named **app-user-claim.ts**. Add the following code in this file.

```
export class AppUserClaim  {
  claimId: string = "";
  userId: string = "";
  claimType: string = "";
  claimValue: string = "";
}
```

Modify the AppUserAuth Class

Open the **app-user-auth.ts** file and remove all of the individual Boolean claim properties. Just like you removed them from the Web API class, you need to remove them from your Angular application. Next, add an array of AppUserClaim objects to this class, as shown in the following code snippet:

```
import { AppUserClaim }
 from "./app-user-claim";

export class AppUserAuth {
  userName: string = "";
  bearerToken: string = "";
  isAuthenticated: boolean = false;
  claims: AppUserClaim[] = [];
}
```

Modify Security Service

Open the **security.service.ts** file located in the **\security** folder. Locate the resetSecurityObject() method and remove the individual Boolean properties. Add a line of code to reset the claims array to an empty array of claims, as shown in the following code snippet:

```
resetSecurityObject(): void {
  this.securityObject.userName = "";
  this.securityObject.bearerToken = "";
  this.securityObject.isAuthenticated = false;
  this.securityObject.claims = [];

  localStorage.removeItem("bearerToken");
}
```

Claim Validation

Now that you've made code changes on both the server and client sides, the Web API call returns the authorization class with an array of user claims. You now need to be able to check whether a user has a valid claim (authorization) to perform an action or to remove an HTML element from the DOM. You're eventually going to create a custom structural directive that you can use on a menu, as shown here:

```
<a routerLink="/products"
   *hasClaim="'canAccessProducts'">
  Products
</a>
```

To be able to do this, you need a method that takes the string passed to the ***hasClaim** directive and verifies that this claim exists in the array downloaded from the Web API. This method should also able to check for a claim value set with this claim type. Remember that the ClaimValue field in the SQL Server UserClaim table is of the type string. You can place any value you want into this field. This means that you also want to be able to pass in a value to check, as shown here:

```
<a routerLink="/products"
   *hasClaim="'canAccessProducts:false'">
  Products
</a>
```

Notice the use of a colon, and then the value you want to check for this claim; canAccessProducts:false. This string containing the claim type, a colon, and the claim value is passed to the hasClaim directive. The new method you're going to create should be able to parse this string and determine the claim type and the value (if any). Add this new method to the SecurityService class, and give it the name **isClaimValid()**, as shown in **Listing 2**.

The isClaimValid is declared as a private method in the SecurityService class, so you need a public method to call this one. Create a hasClaim() method that looks like the following:

```
hasClaim(claimType: any) : boolean {
  return this.isClaimValid(claimType);
}
```

Create Structural Directive to Check Claim

To add your own structural directive, **hasClaim**, open a terminal window in VS Code and type in the Angular CLI command in this next snippet. This command adds a new directive into the \security folder.

```
ng g d security/hasClaim --flat
```

Open the newly created **has-claim.directive.ts** file and modify the import statement to add a few more classes.

```
import { Directive, Input,
         TemplateRef, ViewContainerRef }
from '@angular/core';
```

Modify the selector property in the @Directive function to read **hasClaim**.

```
@Directive({ selector: '[hasClaim]' })
```

Modify the constructor to inject the TemplateRef, ViewContainerRef, and the SecurityService.

```
constructor(
  private templateRef: TemplateRef<any>,
  private viewContainer: ViewContainerRef,
  private securityService: SecurityService)
{ }
```

Just like when you bind properties from one element to another, you need to use the Input class to tell Angular to pass the data on the right-hand side of the equal sign in the directive to the hasClaim property in your directive class. Add the following code below the constructor:

```
@Input() set hasClaim(claimType: any) {
  if (this.securityService
        .hasClaim(claimType)) {
    // Add template to DOM
    this.viewContainer.
      createEmbeddedView(this.templateRef);
  } else {
    // Remove template from DOM
    this.viewContainer.clear();
  }
}
```

The @Input() decorator tells Angular to pass the value on the right-hand side of the equals sign to the **set** property named hasClaim(). The parameter to the hasClaim property is named claimType. Pass this parameter to the new hasClaim() method you created in the SecurityService class. If this method returns a true, which means that the claim exists, the UI element to which this directive is applied is displayed on the screen using the createEmbeddedView() method. If the claim doesn't exist, the UI element is removed by calling the clear() method on the viewContainer.

Modify Authorization Guard

Just because you remove a menu item doesn't mean that the user can't directly navigate to the path pointed to by the menu. In the first article, you created an Angular guard to stop a user from directly navigating to a route if they didn't have the appropriate claim. As you now verify claims using an array instead of Boolean properties, you need to modify the authorization guard you created.

Open the **auth.guard.ts** file, locate the canActivate() method, and change the **if** statement to look like the code shown below:

```
if (this.securityService
      .securityObject.isAuthenticated
        && this.securityService
          .hasClaim(claimName)) {
  return true;
}
```

Secure Menus

You're just about ready to try out all the changes you made. If you look at the Products and Categories menu items in the **app.component.html** file, you see that you're using an *ngIf directive to only display menu items if the securityObject property isn't null, and that the Boolean property, canAccessProducts, is set to a true value.

```
<li>
  <a routerLink="/products"
     *ngIf="securityObject.canAccessProducts">
    Products
  </a>
</li>
<li>
  <a routerLink="/categories"
     *ngIf="securityObject.canAccessCategories">
    Categories
  </a>
</li>
```

Because the *ngIf directive is bound to the securityObject using two-way data-binding if this property changes, the menus are redrawn. The structural directive you just created passes in a string to a **set** property that executes code, so there's no binding to an actual property. This means that the menus aren't redrawn if you add the *hasClaim structural as shown previously. Another problem is that you can't have two directives on a single HTML element. Not to worry: You may wrap the two anchor tags within an ng-container and use the *ngIf directive on them to bind to the isAuthenticated property of the securityObject. This property changes once a user logs in, so this allows you to control the visibility of the menus. Then you may use the *hasClaim on the anchor tags to control the visibility based on whether the user's claim is valid. Open the **app.component.html** file, locate the Products menu item and add the <ng-container> and the *hasClaim directive as shown below:

```
<li>
  <ng-container
    *ngIf="securityObject.isAuthenticated">
    <a routerLink="/products"
       *hasClaim="'canAccessProducts'">
      Products
    </a>
  </ng-container>
</li>
```

Next, locate the Categories menu item and add the <ng-container> and the *hasClaim directive as shown below:

```
<li>
  <ng-container
     *ngIf="securityObject.isAuthenticated">
    <a routerLink="/categories"
       *hasClaim="'canAccessCategories'">
      Categories
    </a>
  </ng-container>
</li>
```

Try it Out

You are finally ready to try out all your changes and verify that your menu items are turned off and on based on the user being authenticated, and that they have the appropriate claims in the UserClaim table. Save all the changes you've made in VS Code. Start the Web API and Angular projects and view the browser. The Products and Categories menus shouldn't be visible. Click on the Login menu and log in using a user name of **psheriff** and a password of **P@ssw0rd**. You should now see both menus appear.

Open the User table, locate the **bjones** user and remember the UserId for this user. Open the UserClaim table, locate the CanAccessCategories record for bjones, and change the value from a true to a false value. Back in the browser, log out as psheriff, and log back in as bjones. You should see the Products menu, but the Categories menu doesn't appear. Go back to the UserClaim table and set the CanAccessCategories claim value field back to a true for bjones.

Secure Add New Product Button

Add the *hasClaim directive to the "Add New Product" button located in the **product-list.component.html** file. Remove the *ngIf directive that was bound to the old canAddProduct property and use your new structural directive, as shown in the code below:

```
<button class="btn btn-primary"
        (click)="addProduct()"
        *hasClaim="'canAddProduct'">
  Add New Product
</button>
```

Don't forget to add the single quotes inside the double quotes. If you forget them, Angular is going to try to bind to a property in your component named *canAddProduct,* which doesn't exist.

Try It Out

Save all your changes and go back to the browser. Click on the Login menu and log in as psheriff. Click on the Products menu, and you should see the "Add New Product" button appear. Log out as psheriff and login as bjones. The "Add New Product" button should now be gone.

Remember that you added the capability to specify the claim value after the name of the claim. Add a colon after the claim type, then add **false** to the Add New Product button, as shown below:

```
<button class="btn btn-primary"
        (click)="addProduct()"
        *hasClaim="'canAddProduct:false'">
```

```
  Add New Product
</button>
```

If you now log in as psheriff, the Add New Product button is gone. Log in as bjones and it should appear. Remove the **:false** from the claim after you've tested this out.

Add Multiple Claims

Sometimes security requires that you need to secure a UI element using multiple claims. For example, you want to display a button for users that have one claim type and other users that have another claim type. To accomplish this, you need to pass an array of claims to the *hasClaim directive as shown below:

```
*hasClaim="['canAddProduct',
            'canAccessCategories']"
```

You need to modify the hasClaim() method in the SecurityService class to check to see if a single string value or an array is passed in. Open the **security.service.ts** file and modify the hasClaim() method to look like **Listing 3**.

As you now have two different data types that can be passed to the hasClaim() method, use the **typeof** operator to check whether the claimType parameter is a string. If it is, call the isClaimValid() method passing in the two parameters. If it isn't a string, assume it's an array. Cast the claimType parameter into a string array named claims. Verify that it's an array, then loop through each element of the array and pass each element to the isClaimValid() method. If even one claim matches, then return a true from this method so the UI element is displayed.

Secure Other Buttons

Open the **product-list.component.html** file and modify the "Add New Product" button to use an array, as shown in the following code snippet:

```
*hasClaim="['canAddProduct',
            'canAccessCategories']"
```

Try It Out

Save all the changes in your application and go back to your browser. Login as bjones and, because he has the canAccessCategories claim, bjones may view the "Add New Product" button.

Summary

In this final article on Angular security, you learned to build a security system that's more appropriate for enterprise type applications. Instead of individual properties for each item that you wish to secure, you return an array of claims from your Web API call. You built a custom structural directive to which you may pass one or more claims. This directive takes care of including or removing an HTML element based on the user's set of claims. This approach makes your code more flexible and it requires less coding changes should you wish to add or delete claims.

Paul D. Sheriff

Sample Code

You can download the sample code for this article by visiting www.CODEMag.com under the issue and article, or by visiting my website at http://www.fairwaytech.com/downloads. Select PDSA/Fairway Articles, and then select "CODE Magazine: Security in Angular—Part 3" from the drop-down list.

JavaScript Corner: Math and the Pitfalls of Floating Point Numbers

Depending on what mathematical requirements your application has, you may have to take special care to ensure that your operations return what you expect with respect to floating point (double) precision. For the JavaScript world at large, this isn't a new issue. In fact, for the programming world at large, this isn't a new issue. JavaScript often gets bashed because

John V. Petersen
johnvpetersen@gmail.com
about.me/johnvpetersen
@johnvpetersen

Based near Philadelphia, Pennsylvania, John has been an information technology developer, consultant, and author for over 25 years.

"It can't perform math correctly." If these same criteria were to be applied to C#, you'd reach the same conclusion. Later in this column, I'll illustrate how C# and JavaScript, as far as floating-point numbers are concerned, behave in the same way. In this edition's column, I take you through some of JavaScript's mathematical idiosyncrasies and how to work around them.

> JavaScript's only native numeric type is the double-precision (floating point) 64-bit type.

Context and Core Concepts

If you're a developer in a place where JavaScript is not the primary language and you're tasked with something that requires JavaScript, whether on the client or server, you may have some already baked assumptions based on your primary experience with C, C#, VB, Ruby, Python, etc. Your first assumption is probably that JavaScript has an integer type. Your second assumption may be that JavaScript handles mathematical operations just like your primary language. Both assumptions are incorrect. The next question to address is WHY this is the case. To begin to answer that question, let's see JavaScript in action with a simple task, adding two values and testing for equality. **Figure 1** shows a simple math problem.

In C, C#, and other similarly situated languages, the result of 0.1 + 0 .2 == 0.3 is, of course, true. But that's not the case in JavaScript. Your next assumption may be that JavaScript can't do math. This would also be an incorrect assumption. Before you can understand how JavaScript behaves, you must first address what JavaScript is and more specifically what JavaScript is in the mathematical context and what numerical data types JavaScript directly supports.

The answer is just one. JavaScript supports one mathematical type, 64-bit floating point numbers. Because I'm talking about a 64-bit (binary) system, base 2 applies. The question to ask is "how are 64-bit floating point numbers stored?" The answer is that 64-bit floating point storage is divided into three parts, as shown in **Figure 2**.

Reviewing the IEEE 754 standard, the largest integer that can safely be relied upon to be accurately stored is expressed as $(2^{53}) - 1$ or 9,007,199,254,740,991. Why do we need to subtract 1? The best way to explain that is to review the binary representation of 2^{53} or 9,007,199,254,740,992, which looks like this:

```
0
00000000010
0000000000000000000000000000000000000000000000000000
```

Only the 10th bit is flipped. This 10th bit however, under the storage scheme illustrated in **Figure 2** is in the bit range reserved for the exponent. Therefore, if you subtract 1 to get the largest safe integer value (9,007,199,254,740,991), the binary representation is as follows:

```
0
00000000001
1111111111111111111111111111111111111111111111111111
```

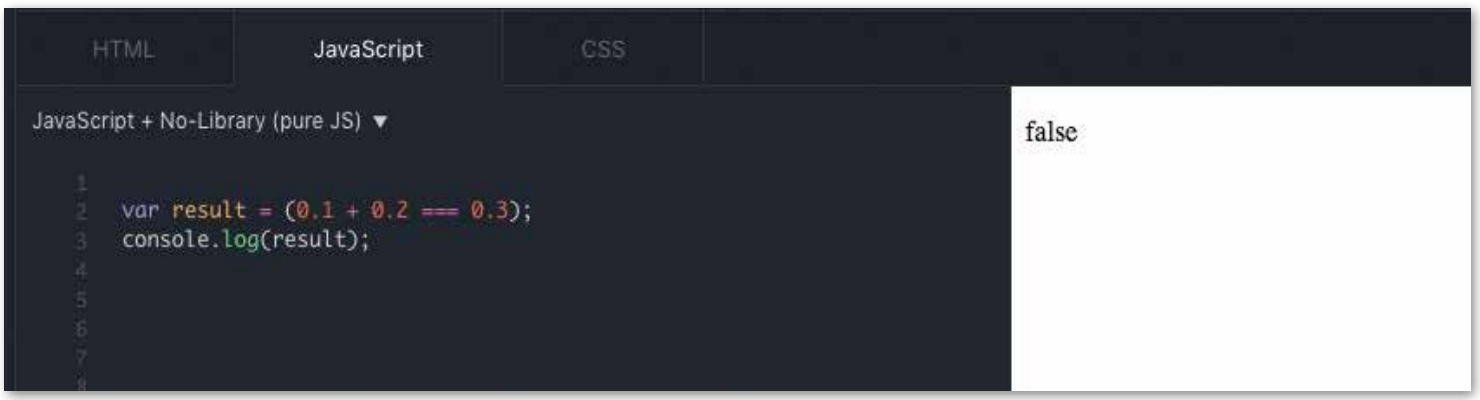


Figure 1: JSFiddle output illustrating that .1 + .2 does not equal .3.

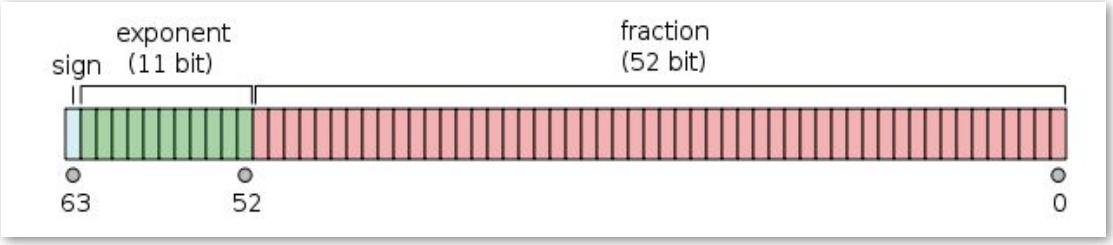


Figure 2: The IEEE 754 Standard for storage of 64-bit floating point numbers

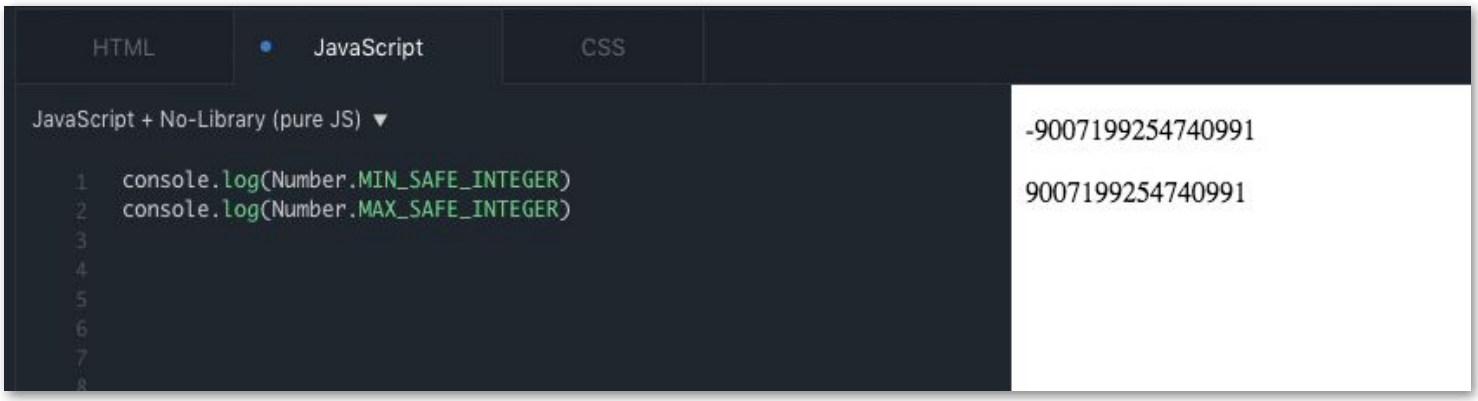


Figure 3: The Min and Max Safe Integer Constants provide quick access to the smallest and largest integers that can be safely relied upon in JavaScript.

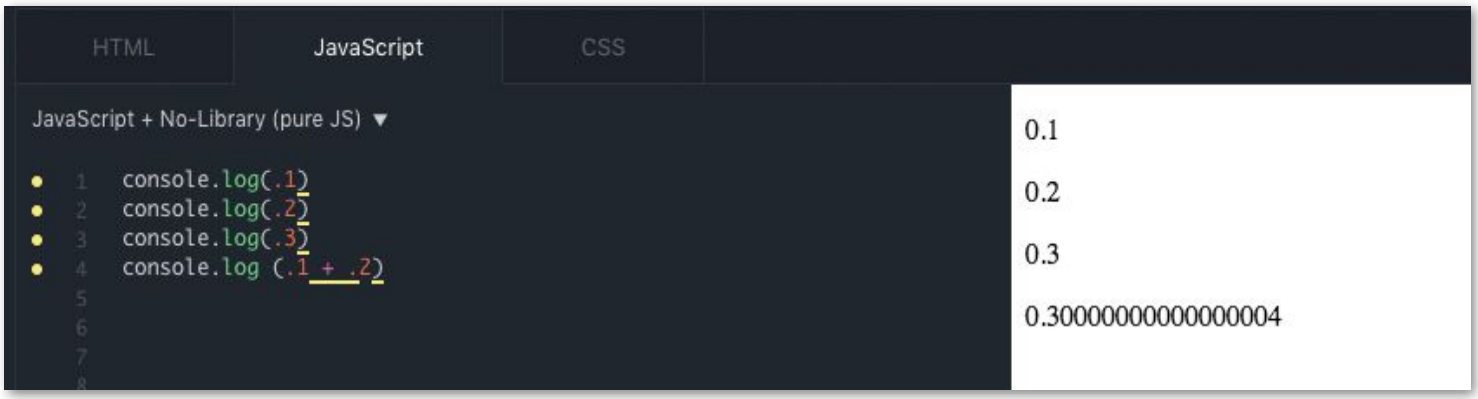


Figure 4: As far as JavaScript is concerned, the whole (.3) is not equal to the sum of the parts (.1 + .2).

Bits 0 through 52 are flipped. Accordingly, this translates to the value 9,007,199,254,740,991. In JavaScript, there's a simple way to access this number and its negative counterpart with the MAX_SAFE_INTEGER and MIN_SAFE_INTEGER constants hosted in the Number Prototype, as shown in **Figure 3**.

To round out the illustration, the binary representation of the smallest safe integer value (-9,007,199,254,740,991) is represented as follows:

```
1
11111111110
0000000000000000000000000000000000000000000000000000
```

In this case, you simply reverse how the bits are flipped with respect to the largest safe integer value.

Now that you know how JavaScript supports the single numerical type and how its storage works from the smallest and largest integer perspective, turn your attention to the initial problem with decimals: 0.1 + 0.2 === .3 (the result being false). Perhaps you've heard the statement that floating point values cannot be guaranteed to be accurate. You certainly know this to be the case because the expression 0.1 + 0.2 === 0.3 is false in JavaScript. The question is: Why is it false?

You must first recognize that a computer can only natively store integers. That's the essence of the binary (base 2) nature of computing. For example, you have the following binary/decimal value pairs:

- 0 : 0
- 0 : 1
- 10 : 2
- 11: 3
- 100 : 4
- 101 : 5
- ---

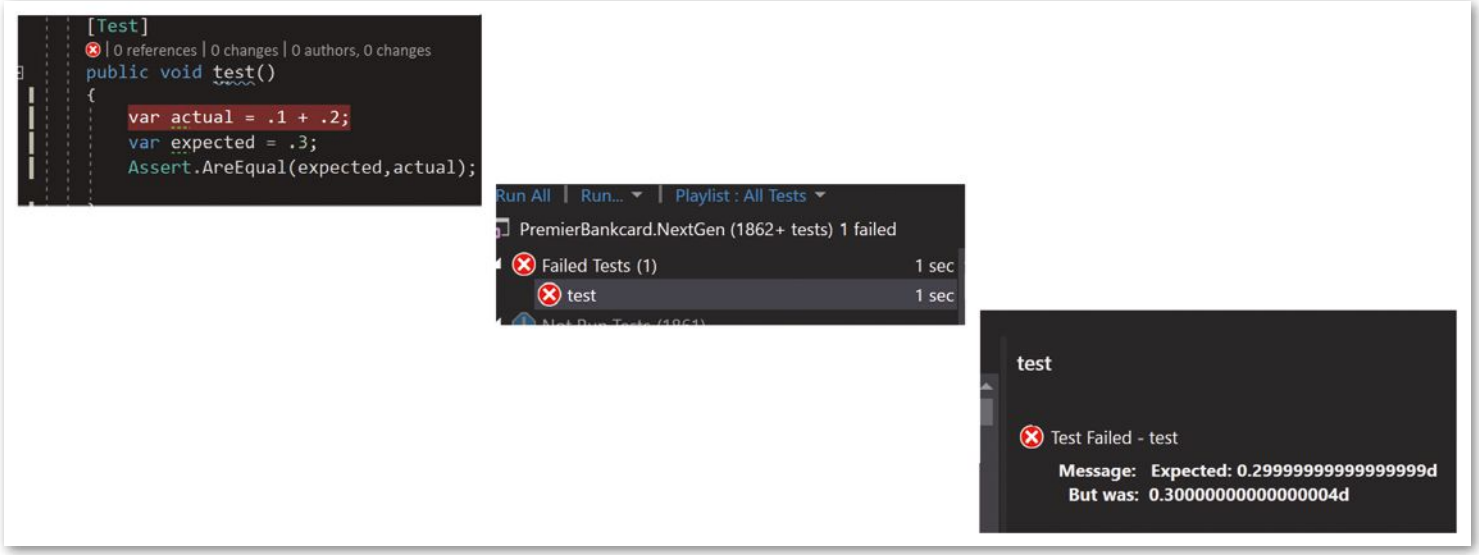


Figure 5: Whether in JavaScript or C#, the same pitfalls can exist when attempting to test for equality among floating-point numbers.

```
float actual = .1f + .2f;
float expected = .3f;
Assert.AreEqual(expected,actual);
```

Figure 6: In C#, if values are explicitly defined as floats, the comparison result will be true.

Outside of the computer, you routinely deal with numbers in base 10 system (every decimal point to the right is a reduction by a factor of 10). To get a non-repeating decimal result, your fraction must reduce to a denominator that is a prime factor of 10. The prime factors of 10 are 2 and 5. Using 1/2, 1/3, 1/4, and 1/5 as examples, when converting to decimal format, some are clean and terminating and some repeat. Accordingly, the preceding examples, converted to decimals, are as follows: .5, .33333333...., .25, and .2. In a base 10 system, as long as your divisor is evenly divisible by a prime factor of 10, you'll have a precise non-repeating result. Anything else repeats.

With a computer, there's the base 2 (binary) system of which there is only 1 factor, 2. This is the essence of a binary system; 0 or 1, on or off. There's no third option! This is why a computer can only natively store integers. Therefore, to get a finer level of precision via decimals, there must be a scheme within the operating bit range to represent such numbers. In the case with JavaScript, the scheme is 64 bit and **Figure 2** illustrates how the scheme is divided between the left- and right-hand portions of the decimal point as well as the sign (positive or negative).

Why then does 0.1 + 0.2 != 0.3 in JavaScript? To begin to answer that question, let's see JavaScript in action in **Figure 4**.

Figure 5 illustrates the same problem with C# .

In reality, the values illustrated in **Figure 5** are double precision types. If, in C#, you wish to treat the numbers as floats, you must use the "f" suffix, as in **Figure 6.**

I'm referencing C# for two reasons. First, to illustrate when it's a like-for-like comparison with C#, the same problem can exist. Second, because languages like C# support multiple numeric types, there are ways in languages other than JavaScript to leverage other types to counteract the problem that may result from comparing floating types.

Returning to the core question in JavaScript, why does (.1 + .2) != .3? This gets to how decimals are stored as binary. Let's look at 0.1. As a fraction, .1 can be represented at 1/10. Everything a computer does must be broken down to binary. So then, what is .1 in binary? To answer that question, you have to take the fraction 1/10 and make it binary. The number 10 expressed as binary is 1010 (the 2nd and 4th bits—values 2 and 8—are flipped: 2 + 8 == 10). Therefore, the binary division is as follows: 1010/1. Have a look at **Figure 7.**

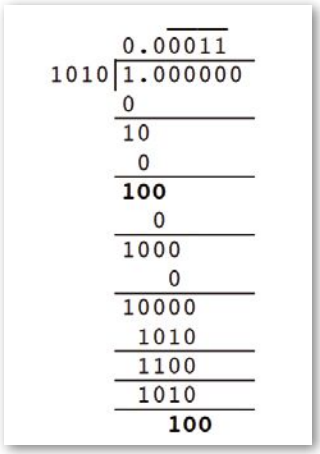


Figure 7: Binary long division—dividing 1 by the binary representation of 10: 1010.

> JavaScript performs arithmetic calculations with floating point numbers in a manner consistent with other programming languages.

As you can see, 0011 infinitely repeats. Because there's only a finite space to store the infinitely repeating decimal, some accommodation has to be made. In order to fit within the bit constraint, 64 in this case, rounding must be applied. A useful site to bookmark can be found here: https://www.h-schmidt.net/FloatConverter/IEEE754.html. If you enter values for .1 and .2, you'll see the binary representation for those numbers and error values due to conversion. In this case, you see rounded error values of 1E-9 and 3E-9 for .1 and .2 respectively. Accordingly, when you see the result of .1 + .2, it shouldn't be surprising to see the rounding error being carried forward; in this case for .1 + .2, you see a difference in the final result illustrated in **Figure 4**.

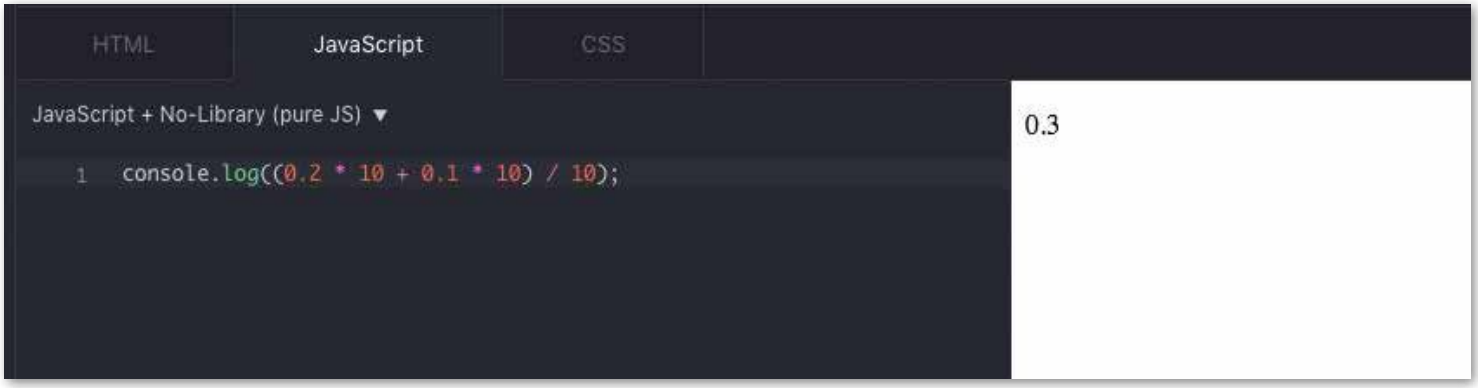


Figure 8: One method of avoiding the rounding errors that can result from floating point numbers being converted to and from binary format.

Application

What's a developer to do? Does this mean if you have mathematical operations to perform, JavaScript is useless? Of course not. The real issue is floating point/double precision numbers themselves and how errors can creep in when converted to and from binary format. One way to address the problem is to first take the floating-point aspect of things away and instead, work with derived integers and then dividing the result. **Figure 8** illustrates a way to resolve the problem.

If you really wish to geek-out on floating points, consider reviewing this article by David Goldberg entitled "What Every Computer Scientist Should Know About Floating Point Arithmetic" (https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html).

Another approach you may wish to consider is a service endpoint that's backed up by a language such as C, C#, VB, Ruby, Python, etc. Regardless of which approach you take, it's important to recognize, as early as possible, your application's potential sensitivity to precision problems with floating point numbers. In all cases, and especially this case, testing is your friend!

Key Takeaways

What's a developer to do? Does this mean if you have mathematical operations to perform, JavaScript is useless? Of course not. The real issue is the floating point/double precision numbers themselves and how errors can creep in when converted to and from binary. Regardless of the language environment, caution must be exercised when dealing with floating point numbers because not all real numbers can be exactly represented as floating point numbers. One solution is to simply not use floating point numbers. For JavaScript, that's difficult because the only number JavaScript has is the floating point. In cases where monetary operations are necessary, your best approach may very well be to let a back-end service handle your calculations and then pass data back to JavaScript as a string for display purposes. It all depends on what your specific application requires. If all you deal with are whole numbers that exist between the min and max safe integers in **Figure 3**, you shouldn't run into the issues described in this article. If that's not the case and instead you deal with decimals, consider using a library like decimal.js: https://github.com/MikeMcl/decimal.js/. It's important to understand and recognize that issues with floats exist. And if you think it isn't a matter of life or death, then consider what happened on February 25, 1991. On that day, there was a Patriot Missile failure. The Patriot Missile launch failed to intercept a Scud Missile, which lead to the death of 28 U.S. Army personnel. The cause? Lack of precision in floating point numbers stored in a 24-bit register: http://www-users.math.umn.edu/~arnold/disasters/patriot.html.

John V. Petersen

CODE

ECMAScript Standard (262)

ECMA (the European Computer Manufacturers Association) is the international standards-making organization that defines JavaScript (ECMAScript). The latest standard can be found here: https://www.ecma-international.org/ecma-262/9.0/index.html. Under section 4.3.20, Number Value, the standard clearly states that the only numeric value directly supported is the double-precision 64-bit binary format, as promulgated by IEEE 754.

Stages of Data #1: A New Beginning

Readers aren't that interested in personal history. They want (and rightly so) meaningful content and code samples they can use in their work. I was a reader for almost two decades before I wrote my first article. I could quote the articles and code samples written by Dr. Dobb's Journal columnist Al Stevens from memory because I nearly worshipped his offerings. But occasionally I was interested in a personal story from a columnist and I have one relating to my future plans.

Kevin S. Goff
kgoff@kevinsgoff.net
http://www.KevinSGoff.net
@KevinSGoff

Kevin S. Goff is a Microsoft Data Platform MVP. He has been a Microsoft MVP since 2005. He's a Database architect/developer/speaker/author, and has been writing for CODE Magazine since 2004. He's a frequent speaker at community events in the Mid-Atlantic region and also spoke regularly for the VS Live/Live 360 Conference brand from 2012 through 2015. He creates custom webcasts on SQL/BI topics on his website.

Exit Baker's Dozen…

In the summer of 2004, I penned my first article for CODE Magazine on the .NET DataGrid. I always wanted to write articles on productivity tips to help beginner- and intermediate-level developers. A fellow CODE Magazine author gave me a fantastic idea: "Here's what you do: Write 13 tips and call it a Baker's Dozen of Tips." That name/brand wound up being a great form of identity/recognition. One time at a conference, someone approached me and say, "Hey, Kevin Goff! Baker's Dozen! I read your stuff!" I try to minimize using marketing terms, but that was real brand recognition.

Over the years, I've received encouragement and feedback from readers, and I greatly appreciate all of it. When CODE Magazine featured my September/October 2016 cover article on 13 great features in SQL Server 2016 with a box of pastries, I felt honored and humbled and grateful all at once. Just like the picture of the Boston Cream Pie on the cover, that entire idea was sweet!

It was a big challenge to write 13 meaningful tips. True, some tips have more pop than others, but I always wanted each tip to carry some level of value. Over the years, balancing that effort with a busy career and a family life, trying to write 13 worthy items just became too much. CODE Magazine's Editor-in-Chief Rod Paddock always wondered when the burden of writing 13 items would become overwhelming, and of course, it took me a long time to get past my stubborn nature and admit that the time had come.

I've only written a few full Baker's Dozen articles in the last two years, providing further evidence of how difficult it's been: All of the other articles have been shorter pieces on various SQL Server topics. Despite "never say never," I believe I've written my last Baker's Dozen column. However, there's still plenty I want to write about. Where do I go from here?

"C'mon Batman, I Need a Name"

One habit I'll never drop is quoting movies (to the dismay of my editor). In "Batman Forever," when Dick Greyson's character eagerly wants to join Bruce Wayne/Batman, he presses Bruce: "C'mon Bruce, I need a name!" (I loved Bruce's reply: "How about Dick Greyson, college student?")

Similarly, although maybe with less exuberance, I wanted to start a new brand, one that wouldn't burden me, one where I could naturally leverage what I've already been speaking about (data warehousing activities) for a long time.

Enter, Stages of Data….

Nearly every application I've built has been intensely data driven. I've dealt with data in health care, insurance, manufacturing, steel-making, finance, and several other industries. In every application, data goes through several stages before the business side can use it for meaningful information. When companies acquire other companies, that serves to compound the impact of "stages of data" and mapping data to conform to one common set of business definitions.

As an example, when an executive sees that a sales margin has dropped 15% for Product XYZ in a certain market, the lineage behind that number can be very complex. Therefore, it stands to reason that you must manage all of the steps that make up that lineage.

I like to quote movies, but I've also been known to read and quote more serious literature. William Shakespeare's "As You Like It" talks about the Seven Ages of Man, and of course, that play also contains the famous line, "All the World's a Stage." The idea of the ages of man wasn't new: Aristotle wrote similarly about it. Likewise, Data Warehousing projects have life cycles, though, like snowflakes (no pun intended to star schema fans), no two life cycles are exactly the same. The more experience you have with the various life cycles, the more value you potentially can bring to future projects. In today's increasingly-competitive world, the more value you bring, the better!

That explains my new column name: "Stages of Data." No one person, not even the greatest DW professional on the planet, can claim to have seen it all, but certainly, there are repeated themes across projects. Good data warehouse systems collect data from one or more systems and make it presentable to business users. When an executive gets a dashboard report showing that production is up in certain plants/sections, mechanical-shop defects are up in other areas, and monthly margin percentages dropped on certain products, that data has passed through many stages. If you don't properly manage all of the stages, from extraction to validation to mapping to final aggregation, you don't serve the business.

Okay, Production, Quality, and Margin Numbers. Sounds Simple, Doesn't It? Not Always

A monthly chart of these types of metrics might seem like a mundane task, right? Not exactly. Each metric represents a process. For each process, a system collects data either manually or through some automation. That data has a particular grain/level of detail. Keep something in mind: This is 2018, and many companies are merging/buying other companies, etc.

So that means that there are multiple source systems with different rules and different levels of detail. And

even within one system, there can be sub-systems. Remember, the average turnover rate in this industry is about 18 months, which means that some teams aren't able to finish module/sub-system upgrades, which means any centralized solution might have to perform "double-reads" to get the full truth. Maybe your team agreed to do that for three months until new team members completed the upgrade. We all know how three months turns into six months, and what was once a short-term extra step has now become a (begrudgingly) accepted practice.

Then a new team comes in to build a new reporting system and no one formally documents the problem of having to do double-reads to get cost data. As a result, the new team believes that they only have to read X sub-systems to get a complete picture of the data, when it turns out that there are Y sub-systems.

Okay, convinced that it's not so simple? I'm just getting warmed up.

To Make Matters Worse….

There might be subject-matter data spread across sub-systems. Maybe cost-components A, B, and C are in sub-system 1, and cost-components C, D, and E are in sub-system 2. Perhaps sub-system 1 has some costs for component C that are not in sub-system 2, and vice-versa. It's possible that identifying the common values between sub-systems 1 and 2 for component C involve a mapping table that's maintained by some analyst in a spreadsheet.

Guess what? It could take days or even weeks of research to query (i.e., profile) that data to discover the whereabouts of that little detail.

Ever wonder why you read on LinkedIn that a high percentage of data warehouse projects fail? This is your future. Do you want it?

I've used a quote from a manager in the movie "A League of Their Own" many times: "It's supposed to be hard. The hard is what makes it great." In the last year, my team put together a Costing Module that showed costs and profitability breakdowns based on raw cost and production data that the company had been collecting but never previously reported on accurately.

The production data went through many processes and gyrations such that no one source system accurately reflected the true picture of profitability. Our data warehouse Costing Module represented a "Version of Truth," solving the proverbial Sphinx riddle by accounting for all the different stages of cost and production data in a methodical manner. The business was both stunned and happy. Acceptance by the business makes it GREAT.

At this point, some purists might say, "Okay, but that kind of mess is probably the exception." I would debate otherwise. Although I can't speak for every environment out there, I'd bet an expensive dinner that it's common enough to be a big factor in why so many data warehouse projects fail (or only get half-built).

That's why I'm formally defining my new column brand here: "Stages of Data." The more you know about the different stages that data can pass through, the different steps you need to look out for, and the more value you can bring to every step, the better.

I know others who have successfully built data warehouse functionality, and often these individuals have something in common: They are nearly "shadow-IT" in the business area, "shadow-business" in the IT area, and sometimes even competition for the company's formal research group. Because most shops either adopt (or pay huge lip service to) agile methodologies, a successful data warehouse professional needs to wear many hats in the course of a two-week development sprint. Versatility and, as I'll later discuss, preparation, are two of the many keys in a successful data warehouse project.

One last note: The concept of a "heads-down" developer is an oxymoron in the data warehousing world. Data warehouse professionals need to understand the proverbial big picture. If you hear a manager say that "heads-down" data warehouse developers can program from a complete spec, shoot them with a harmless water gun. If that manager also believes that someone can build complete data warehouse specs without tech knowledge and SQL skills, you have my permission to double-tap that water gun.

If your manager believes that someone can build complete data warehouse specs without tech knowledge and SQL skills, you have my permission to shoot them with a water gun.

High-level Topology, Revisited

Those who've read my recent articles should at least generally recognize the data warehouse topology diagram in **Figure 1**. I've modified it based on a recent speaking event where someone raised a question about a layer that I'd neglected to include: a report snapshot table layer.

Ironically, I've used this layer for years, but never thought to include it in my presentation. Here's an example where report snapshot tables can help. Suppose you have an end-of-quarter lockdown process, and you report profitability and other metrics to executives. So maybe 30-60 days after March 31 every year, you close out the first quarter. The reason for the delay is to adjust for late-arriving sales adjustments and other lagging data.

When you close out the quarter, you might want to write out and preserve the state of the quarterly reporting data to a sales and margin snapshot table. That makes it easier (and usually faster) for people to retrieve and report on the data years later. No matter how much people have archived and changed the structure of data, you still have snapshots of the report data. I highly recommend this process. These tables are usually a fraction of the source data, because they're summarized for the lowest level of detail for the report and only contain columns associated with the report.

Additionally, I know people who also save all of the underlying data that makes up the report data! That can be a little

Databases and Efficiency

The number one goal of a transactional system is to get data **INTO** the database as quickly and efficiently as possible. The number one goal of a data warehouse/analytic application is to get data **OUT** of the database as quickly as possible. That's always my leading statement when I explain to new data warehouse developers why data warehouse database models are usually flat and denormalized.

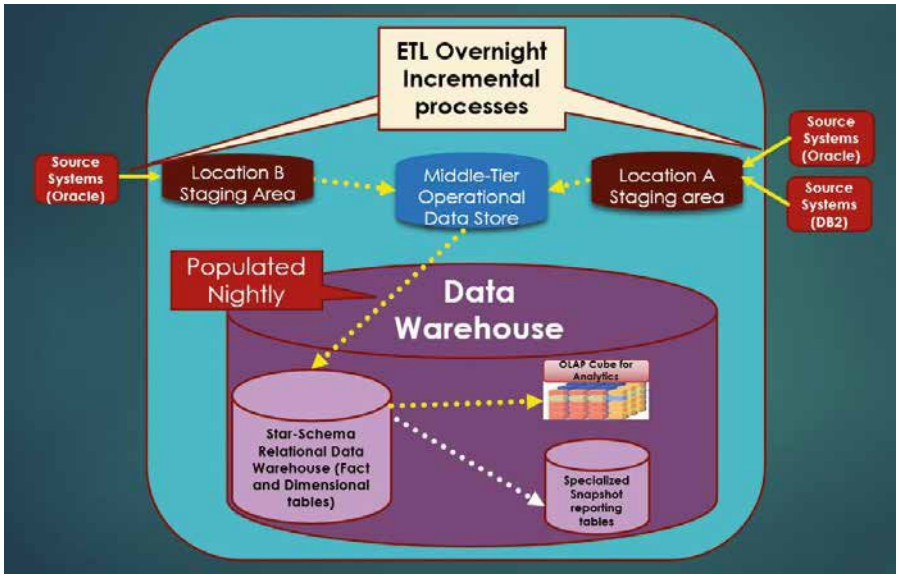


Figure 1: Modified topology, to account for a reporting layer based on snapshot tables

more complicated if you're saving several years, as it might require a large amount of space. However, if you're in an environment where you need to justify something you reported on three or four years ago (I've been in that situation more than once), having all the underlying details that rolled up to the report snapshot might help you. Not too long ago, I had to defend a summarization from early 2015 that looked "odd." Fortunately (it's a long story), I had a backup of the transactional data and was able to justify the summarization.

> Data warehouse storage isn't the place to start nickel-and-diming about resources, ESPECIALLY if you work in a place where the level of accountability regarding reported numbers is very high. Repeat that 100 times.

Lessons Learned

If you read enough data warehouse material, you'll see a health list of "don't do this" recommendations. Some people call them "anti-patterns you want to avoid." They're usually very good rules to live by. Here are some other random tips.

Random Tip #1: Avoid Spaghetti Reporting

The great thing about **Figure 1** is that you can use it as a launching point for so many discussions and "lessons learned." Here's a lesson to consider: the concept of spaghetti reporting. As a data warehouse environment grows and expands, users might demand reporting before the team has had the time to properly model the new data in the data warehouse. In some instances, the team might try to fast-track the process by building reports against both data in the data warehouse **and** data back in the ODS and even back in the staging area. The team might justify it as a short-term solution hack/workaround, but we all know how much short-term hacks wind up living in the proverbial basement long after they should have moved out and moved on!

I've faced the challenge the last few years, as business acquisitions have forced me to deal with reporting demands the very second I learned about new data I needed to load from the source area into the staging environment. If you're not careful, you can find yourself building a solution of 30 reports that read all over creation! That's not manageable over the long run.

Random Tip #2: Even If You Specialize, Don't Resist Wearing Many Hats

The second tip I'll tie back to **Figure 1** is the value of being able to work in many different layers of the data warehouse. Yes, there are some people who want to do nothing but work in the ETL (extract, transform, and load) layer. There's nothing at all wrong with that, as most data warehouse applications require more effort in the ETL layer than any other single layer. But don't discount the idea of working in the reporting area. Here's where you really get to see the business application.

But above all, you really can't be a strong data warehouse professional without strong SQL programming skills, whether it's T-SQL in the Microsoft world, PL-SQL in the Oracle world, etc. Although my claim to fame has always been the Microsoft platform, I can read PL-SQL code and DB2 SQL code and I'm familiar with many of the nuances because I frequently have to read from Oracle and DB2 systems.

Random Tip #3: Make Your Work Available and Accessible and Let Others Promote It

Very early in my career, I loved being a "heads-down" developer. I was fairly successful and won two awards by the time I was 25 years old. However, I was able to do so because I was working for some FANTASTIC project managers who made it possible for me to sit at my desk and crank out code and listen to music on my Walkman with very little in the way of interruptions.

In many ways, those days are gone. If you work in data warehousing, it's extremely likely that you'll need to interface with the business and understand business requirements. You'll need to communicate actively with the business and you'll need to "speak their language." That also means communicating aspects of the system in non-technical terms. I hate to use the term "political" here, but soft-skills are essential. If your data warehouse project is the newest project, you'll need the help of some key business people to champion your work. A single endorsement from a respected business user goes much further than ten developer-led presentations on new functionality that the business should be using. That's not to say that the latter doesn't have its place: but the former is how solutions grow in a company. I can't stress this enough.

Random Tip #4: The Many Pitfalls of TRUNCATE and LOAD

You might work in an environment where **Figure 1** seems like overkill. Maybe you have one primary data source and it's one where you have control. That data source might not be terribly large, maybe 10 million rows and that's it. Additionally, you might only need to update your data warehouse once a day, overnight, and it doesn't matter if it takes 30 minutes to rebuild it. You might decide to do what's called a **T&L**, for Truncate and Load.

In a T&L, you truncate the data warehouse every night and reload everything from the source system. No need to worry about incremental updates, just a full refresh of the data. I've done it, and so have many of you. Yes, it's primitive. It can work, but you should know the risks!

> A single endorsement from a respected business user can go further than ten developer-led presentations on new functionality.

You know about the first risk here: If the source data grows significantly larger and there's a need to refresh the data multiple times a day, the T&L approach becomes inefficient.

There's a second risk, one that's not as apparent. Perhaps you don't have control over those source systems and some of them wind up archiving or purging data. To make matters worse, maybe the purging follows a schedule that wreaks havoc on any relationships (or even assumptions of data) that you have in your data warehouse.

One day the data warehouse might seem fine, and the next day someone goes and runs a report for Q4 2017 and gets incomplete results, all because some of the source data used for that Q4 2017 result went through an archive process in the source system. Because you've been doing a T&L, your data warehouse winds up reflecting the archived state of the sourced system. That's NOT a good thing!

So be very careful about using T&L in an ETL strategy. Unless you have full control over your source data, you could be asking for trouble. At the very least, maintain a copy of what you extract, because the source system might wind up archiving it unexpectedly!

Random Tip #5: Don't Get Too Enamored with (Nor Too Frightened by) Emerging Technologies

As you go through the processes in **Figure 1**, there are many opportunities to use some of the newer technologies in SQL Server. Here are some examples:

- Use SQL Server In-Memory Optimized tables for the Staging area. As I wrote back in the September/October 2016 issue of CODE Magazine, In-Memory OLTP tables can speed read/write operations by two to three times or greater.
- Use SQL Server Temporal Tables or Change Data Capture to maintain historical versions of data in an operational data source.
- Use SQL Server columnstore indexes on significant Fact Tables for speeding up aggregations. I've written about columnstore Indexes several times in prior CODE Magazine articles.
- Use (or at least evaluate) newer reporting tools, such as Power BI.
- Use the SQL Server 2016 Query Store to monitor and manage query performance.

I know good developers who rarely (or never) use SQL constructs that Microsoft added after SQL 2005. I also know some people who quickly rush to use every new feature out of the box when Microsoft releases a new version or even interim release. Both sides might be well meaning but are potentially making mistakes.

Explore new database features, test them, study them, and use them judiciously. If you want to use In-Memory OLTP tables to speed your ETL, create a test environment and demonstrate over a period of time that In-Memory OLTP will help the performance of processing staging tables. Yes, creating and managing a test environment takes time, but it's the right thing to do.

> If new index feature XYZ doesn't give you better performance, you likely don't have a compelling reason to use it.

Random Tip #6: Don't Think You Can't Benefit from Inspiration

Last spring, my family and I took a trip to Disney in Orlando, Florida. I won't talk about the great rides, great food, or great fun we experienced there, although we certainly had a blast. What I will talk about is Disney's incredible attention to detail and their commitment to high quality in every service they provide. The hotels are run very professionally, all staff are highly knowledgeable, and the landscaping is immaculate. When you just stop and look around at all the sites in any of the theme parks, you really get a sense of the incredible effort that went into creating the entire Disney experience.

That trip not only helped me recharge my batteries after months of long work hours, it also re-dedicated me toward providing the highest quality of work possible for my clients.

Here's another story. My family and I had to go through a legal hassle last year. The specifics don't matter (they're actually quite silly), and fortunately the story had a happy ending. But something else came out of it. The attorney who represented my family was one of the most professionally tempered individuals I've ever met. I've never seen someone show so much patience and so much vision and confidence that we would come out fine in the end.

The "other side" of the legal issue often took measures that we found unfair and antagonistic, and the natural human response was to react or retaliate. Our attorney was the model of "let cooler heads prevail." He wound up being an inspiration for how I should deal with my clients. Most of the time I'm known as good-natured and flexible, but I'll admit to a few bouts of irritability when things get very rough. I learned some great lessons from this attorney and told him so.

If anyone tells me at the end of the year that they really liked working with me, I'll tell them I have Disney and a lawyer to thank <grin>. Don't forget that inspiration can come from many places.

Speak the Same Language

Seek common ground in communicating with the business side, even if you sincerely think the other side isn't doing the same. I've turned a few contentious relationships into workable ones over time, just by searching for common ground.

Random Tip #7: Don't Underestimate What It Takes to Be a Leader

A team needs leadership. I wrote in an editorial years ago that leadership is a verb more than a noun, in recognition of all the active work that goes into leading and mentoring. I'm at my best when I'm relaxed and walking my team through a design review in the same way a scout master might tell a campfire story. But I can only make that possible if I'm PREPARED.

Don't underestimate how much preparation ultimately factors into success. Always think forward about a module you might have to design or code next month. Always think forward about technical hurdles or walls your team might face. If you anticipate them and act in a timely manner, you'll gain the trust of your team. A good leader is earnest and humble about all the things he or she needs to be prepared for. I'll admit when I got my first leadership position in my 20s, I was perhaps too much in love with the idea of being a leader. Now I know, after decades of lessons, the love and passion needed for the process, and that the rest will (hopefully) take care of itself.

Random Tip #8: Don't Assume That Your Error Logging and Backup Recovery Will Work the Way You Expect

You've written what you think is a world-class error handling and error logging routine, but are you sure it works? Are you sure the error logging will always write out the necessary information so that you can diagnose the problem causing the error?

You run backups every day, but suppose you have to restore specific database objects? Do you have an idea how long the process might take? Are you certain that the restore will work properly?

You can see the theme here. Never assume. Yes, testing takes time. But I'll trade 100 hours of testing time over a sleepless night trying to determine why an ETL process terminated.

On the Lighter Side

Nearly everyone in the industry has a few anecdotal stories about interesting exchanges with the business side. (I'm sure it won't surprise developers to know that business execs have similar stories about technical people!)

We've all heard the expression, "There's no such thing as a bad question." My corollary is: "A bad assumption isn't necessarily bad, it just means someone has an honest gap in their knowledge." Take, for example, the following exchange:

Business manager: "I know you're working on the data warehouse, but I'd like to start talking about designing some dashboards with some important KPIs."

Me: "Great! Always happy to talk about that topic. What types of KPIs are you looking to show?"

Business Manager: "Well, I'd have to give it some thought, but certainly sales dollars are one."

Me: "OK, sure, financials are always a big part of a KPI scorecard. I actually don't recall seeing any sales goals in the data. Do you have that somewhere?"

Business Manager (pausing, with a bit of a frown): "Well, not at the moment, not officially, but we shouldn't need that to show sales KPIs." (Note: The manager's tone strongly indicates that the last part was a declaration, not a question, so I knew I might have a small challenge to face.)

I'll skip to the conclusion, although you can probably see where this was going. I explained to the manager that sales dollars constituted a measure, and that you'd need some other piece of information to evaluate and visualize actual performance (the P in KPI!). That other piece of information might be a rule that any monthly spike of 0.5% or more was very good, or it might be a goal table with sales objectives by salesman and month, to compare against the actual sales. By the end of the conversation, the manager realized there was more to it than he realized.

Yes, I've had that conversation many times over the years. Sure, it's human nature to be a little surprised when a conversation reveals that someone has an unexpected gap in their knowledge. But we're all human and no one is perfect, and even those who think "I always can tell when I don't know something" will someday find themselves in a room when he or she is the last to realize something that everyone else knows. It's a humbling world out there.

So Where Do We Go from Here?

If my handwriting weren't so bad, I'd take a picture of my whiteboard where I have a list of topics I plan to write about over the next year. I have an ETL example that I very much need to finish so that I can write about it. I'd like to talk about report architectures and also report delivery in a data warehouse environment. On that latter point, sometimes you have to tweak the best practices you think you're bringing to a company, in order to adjust to an organization's workflow. I'm thinking of naming my next Stages of Data article "Reporting for Duty," as a reflection about managers who mandate systems to push report content to users instead of users seeking out the data on a Web page.

I'd also like to talk about Agile and Sprint practices in data warehouse environments: I had a somewhat negative view of this a few years ago, and although I haven't done a full 180, I've learned a few tricks to help leverage Agile.

Finally, I plan to devote an article to a topic that many data warehouse environments face: handling version history and implementing slowly changing dimension logic. As a basic example, suppose you have a product that goes through three price changes in two years, and you want to track sales and return metrics for each price time period. Or, you have a sales manager who managed one sales territory for years, and then moved on to a new sales territory, and you want to compare sales metrics for the two periods. In other words, you care about the history associated with those data changes.

So, I have officially bid Baker's Dozen a fond adieu. I'm on to the next stage. Literally.

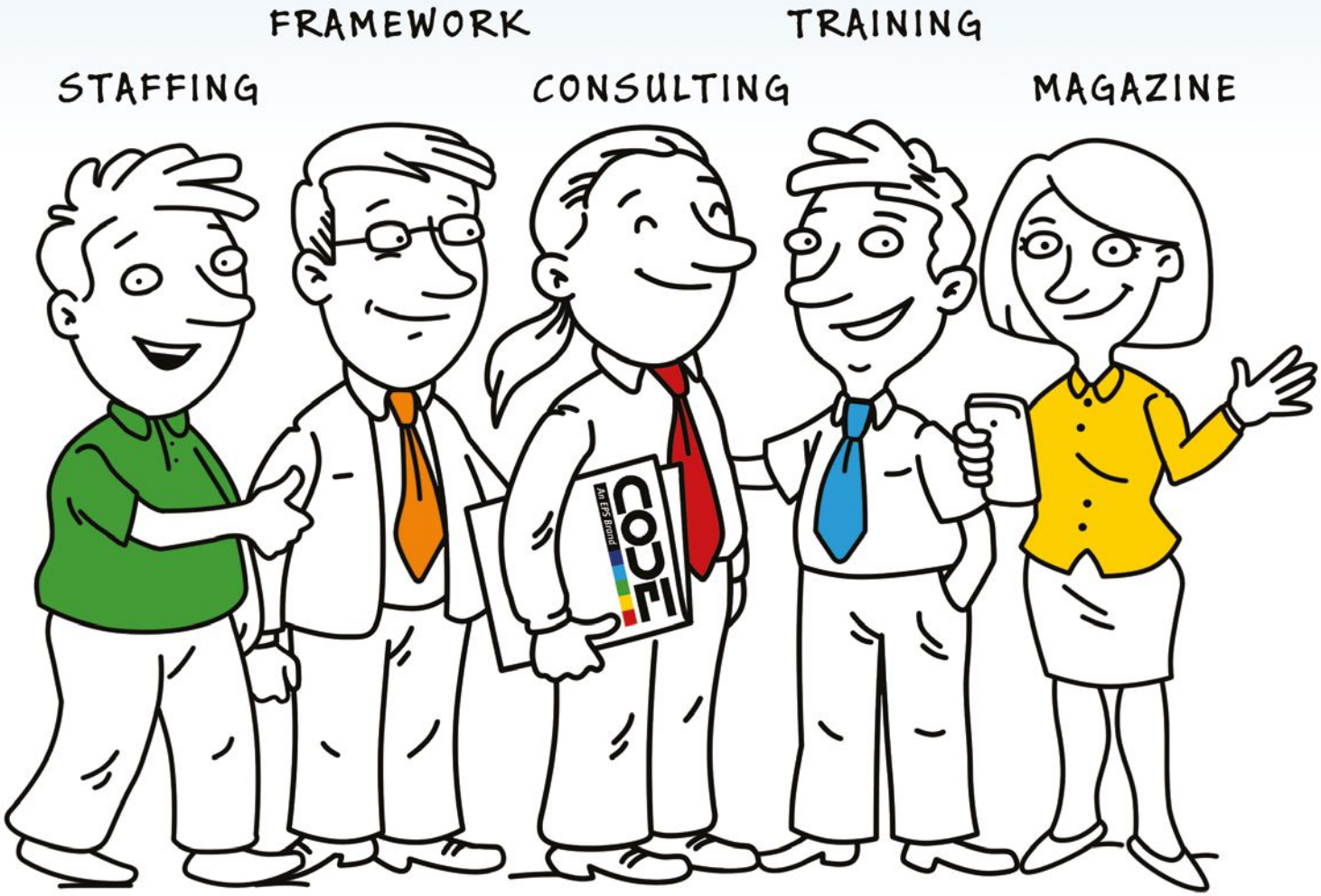
Kevin S. Goff

CODE

Don't Scoff at Simplicity

This is nearly a cliché, but the best solutions are the elegant ones. Recently I looked at two applications: Application A had 10 times more functionality than Application B, but B had some simple and straightforward user paths that A could have benefitted from.

CODE - More Than Just CODE Magazine



The CODE brand is widely-recognized for our ability to use modern technologies to help companies build better software. CODE is comprised of five divisions - CODE Consulting, CODE Staffing, CODE Framework, CODE Training, and CODE Magazine. With expert developers, a repeatable process, and a solid infrastructure, we will exceed your expectations. But don't just take our word for it - ask around the community and check our references. We know you'll be impressed.

Contact us for your free 1-hour consultation.

Helping Companies Build Better Software Since 1993

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

CODE
An EPS Brand

Angular and the Store

An Angular application requires managing some application state, including server-side data, user information, user input, UI state, and many other variables. Developers often make use of injectable services to provide this function (including communication with some back-end Web APIs) in one centralized place, so that the rest of the components in the application

Bilal Haidar
bhaidar@gmail.com
https://www.bilalhaidar.com
https://twitter.com/bhaidar

Bilal Haidar is an accomplished author, certified Microsoft MVP of 10 years, ASP.NET Insider and is internationally recognized as a PMP.

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

With 14 years of extensive experience in Web Development, Bilal is an expert in Enterprise Web Solutions and a consultant trained in teaching Domain Driven Design, CQRS, Event Sourcing, Microservices, ASP.NET Core, , Angular 2+, React JS, JavaScript and Typescript.

can access this shared data to process or update it. This works fine for small applications. In some cases, as the application grows in size and multiple components start issuing calls to update or read the application state, things become unstable. The application state may also become inconsistent or unreliable.

This article proposes a more mature and robust solution to application state management, including server-side data access and using the **ngrx/store** module.

Application State and the Need for a Store.

The **application state** is a collective set of slices of data that represent the state of an application at any given time. As mentioned earlier, the application state can be any of the slices of data that you need to share across the application:

- **Server-side response data:** The data requested and returned from the Web APIs
- **User information:** When the user logs into your application, store the username, email, and other user-related information. This makes for quick access when the user browses and navigates the application later.
- **User input:** On any search page, the user types in a subject or phrase for a search and the application displays the results as a list on their screen. When the user clicks on a result, the application navigates to another page to view the details of the selection. The user can view another search result by navigating back to the search page and choosing something else from the list.
- **UI state:** One of the popular application UI structuring patterns is to have a vertical left-side menu and a right-side contents area. You provide a button to minimize the left-side menu. The state of the left-side menu is stored so that when the user navigates to another screen or page, the left-side menu remains minimized.
- **Router/location state:** When navigating from one screen to another, the application keeps track of the routing details. The state needs to track which record the user selected so it knows what to render later on.

The options are endless. You may add or remove to the state depending on how the application evolves.

You need a state management library or a **Store**. A state lives in the context of a Store, but also, a Store provides at least the minimum information needed to help manage the state. You can:

- Model and store the application state
- Update the state to maintain its validity
- Read the state data
- Monitor and observe changes to the state and make relevant changes when required

The **ngrx/store** module is a state management library based on the concepts of Redux (React World) and a little of RxJS that gives the Store a reactive nature.

Introducing the NgRX Store Module

The NgRX Store module is a state management library made for Angular. It's derived from the popular Redux state management library for React. The NgRX Store imports the state management concepts from Redux and adds to them RxJS to provide an **observable** means of communication throughout the Store APIs, thus giving Angular developers a familiar experience in developing Angular apps.

Some of the major concepts or terminology of the NgRX Store, most of which also apply to Redux are in the following section.

Provide a Single Source of Truth

The NgRX Store models a state as a single and simple JavaScript object inside the Store. The state is immutable or read-only. This means that there is no direct Store API to change the state object inside the Store. There are other means of updating the state that I will be discussing soon. An example of such a state object can be represented as:

```
const state = {
  developer: []
};
```

Actions

In order to update the state inside a Store, the application needs to dispatch an **action**. A **reducer**, also known as a pure function, catches this action, performs the update on the state, and returns a new revised and immutable state object. There is more discussion of pure functions and reducers in the sections below. An example of an action could be:

```
const action = {
  type: 'ADD_DEVELOPER',
  payload: {
    name: 'Bilal',
    competency: ['ASP.NET', 'Angular']
  }
};
```

The **type** property above states the intention of the action. In this case, the **type** property is **ADD_DEVELOPER**, meaning that you are dispatching an action to add the new **Developer** object stored in the **payload** property of the action. The payload is merely the data related to the action type that the reducer adds to the new state returned to subscribers of the Store.

Pure Functions

The Store uses pure functions to update the state. By definition, a pure function doesn't modify the state of variables outside of its scope. Simply put, a pure function always returns the same result given the same parameters. You can read more about pure functions here: http://www.nicoespeon.com/en/2015/01/pure-functions-javascript/.

Reducer or Action Reducer

In terms of a state management library, **Action Reducer** or a **Reducer** are pure functions. Reducers respond to actions and return a new state object with all the changes incorporated inside the state, thus the immutable nature of the state. The reducer analyzes the action dispatched, reads in the payload of the action, performs the suitable action on the state inside the Store, and returns a brand new state object with all the changes inside. An example of a reducer could be:

```
function reducer(state: State, action: Action) {
  const actionType = action.type;
  const developer = action.payload;

  switch (actionType) {
    case 'ADD_DEVELOPER': {
      const developers = [...state.developers,
        developer];
      return { developers }
    }
    ...
  }
}
```

> An immutable object is an object whose state doesn't change after creation.

During the set up of **ngrx/store** module, as you will see later in this article, you configure the **StoreModule** class with a map among all available state segments in the application with their corresponding reducers. In other words, you're telling the Store that when you want to update this specific slice of the state, use this reducer.

Going back to the sample code above, a reducer is a pure function that accepts two input parameters. The first is the previous state (old values) and the second is the current action dispatched. Based on the action **type** and **payload** properties, the case statements take effect. The code snippet above spreads the previous array of developers of the state object into a new array and then adds the **payload** object into this new array. The payload is the new **Developer** object added onto the state array of **Developers**. The reducer returns a new state object encapsulating the latest data.

In the code snippet above, the last statement returns a new object with a single property of **developers** array. In this scenario, the state object tracks only a single property of type **array of developers**, hence the format of the return object of this reducer. The code uses a new feature of **ES6** which is the Shorthand property.

The result of dispatching an action and running the above reducer is having a new state object containing the new **Developer** record:

```
const state = {
  developers: [{
    name: 'Bilal',
    competency: ['ASP.NET', 'Angular']
  }]
};
```

> The reducer always returns a new state object with all the changes incorporated and never returns the previous state amended.

Store

The application state resides inside a Store. The Store acts like a container for the state in the **ngrx/store** module. In addition, Angular components inject the Store into their constructors to establish the communication channel. The Store exposes two methods used by the Angular components. By injecting the Store, a component can have access to the following functions:

- **select()**. The Store uses this method to return a slice of state data from the state contained in the Store. It returns the Store object itself, which is an Observable, so that components can hook into the **select()** method in order to monitor changes of the state when the Store composes a new state object.
- **dispatch()**. The Store uses this method to allow components to dispatch actions to the Store. An action could, with an option, contain a payload. The Store handles the action dispatched via a reducer.

To summarize, a component dispatches an action on the Store. The Store responds to the action by executing the reducers and ends up composing a new state object. Store now has a new state and notifies subscribers (components) of the new update.

Selectors

Selectors are pure functions that take slices of the state as input arguments and return slices of state data that components can consume. Just as databases have their own SQL query language, **ngrx/store** module has its own query tools that are the **Selectors**.

In an Angular application, each Feature module is responsible for injecting its own state into the root application state. Hence, the state is a tree of properties that has sub properties, etc. You define selectors at different levels of the state tree to avoid manually traversing the state tree over and over whenever the Store composes a new state and notifies components.

Based on the state defined above in code, you want to query and return the array of Developers from the state using a selector:

Read on!

If you're interested in learning more about the ES6 Spread feature, you can visit this link: https://goo.gl/BfDhdg

Read more on the ES6 Shorthand property here: https://goo.gl/VpEX2U

Angular Change Detection is explained here: https://goo.gl/3bYCcu

Angular Visualize Change Detection is explained here: https://goo.gl/ioiDMq

Angular Change Detection and the OnPush Strategy is explained here: https://goo.gl/LprwQ6

For more on actions like HeroesAction, you can visit the following link: https://goo.gl/Te69ET

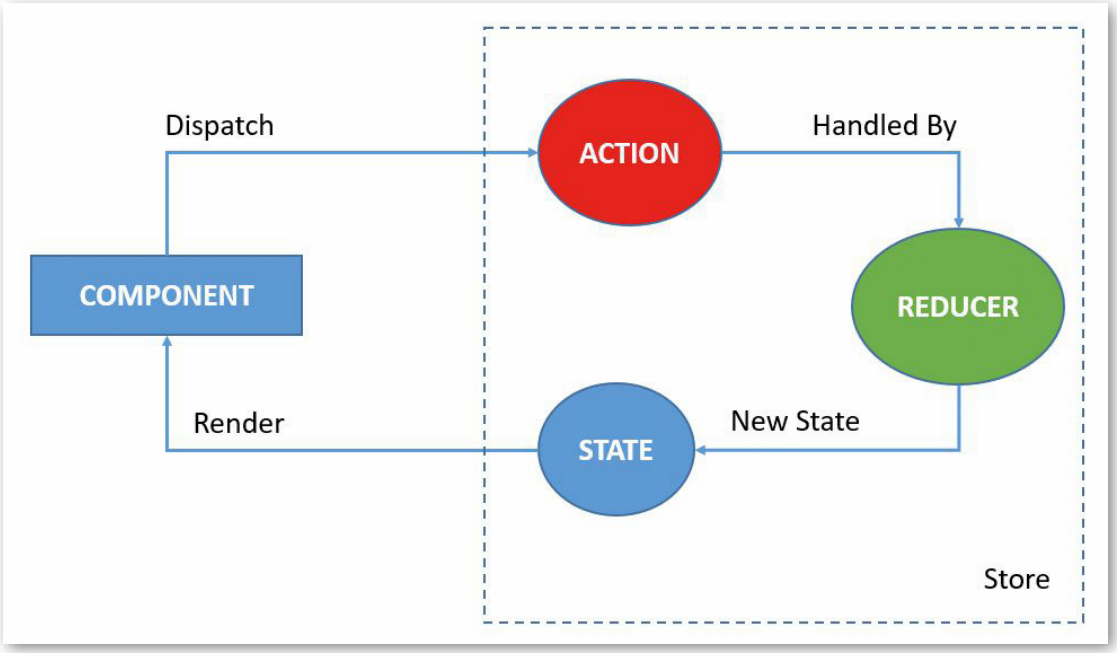


Figure 1: Components and Store communication lifecycle

```
export const getAllDevelopers =
  createSelector(getState,
  (state): Developer[] => {
    return state && state.developers;
  }
);
```

This selector returns a typed array of Developers. Now, any component can make use of the Store.select() method to subscribe to changes of the Developers data (a slice of state data).

```
this.store.select<Developer[]>(getAllDevelopers).subscribe(
    developers => console.log(developers)
);
```

Figure 1 depicts the communication among the different **ngrx/store** elements (Store, State, Reducer, Action, and Component):

The cycle starts by having an Angular component dispatching an action to the Store. The Store captures the action and executes the reducers. The result is a new composed state saved inside the Store. Only then does the Store notify all the components associated with the changes in the stat, that a new state is available and ready for use. Notice that the flow among the components happens in a one-way data flow, simplifying and solving the chaos that results from not using a Store and relying only on components communicating with services to read and update the application state.

That was a brief summary of the NgRX Store, its definition, structure, and major components. Let's move on to the benefits of using the NgRX Store.

Benefits of NgRX Store Module

In general, there are a number of benefits when using a Store in Angular to manage the application state. For the **ngrx/store** module, it's clear that the benefits listed below apply perfectly and are virtually flawless in Angular applications:

- The Store maintains the state, giving the developer a single source of truth to query the application state and even update the state in a consistent manner.
- Testability is first class in the **ngrx/store** module. Reducers and selectors are pure functions, which makes the process of testing them easy. In addition, actions are simple JavaScript objects that you can mock to provide testing scenarios.
- The **ngrx/store** module supports both Root and Feature module state management. With the **ngrx/store** module, you continue to build your Angular app with either Lazy Loading or Eager Loading modules. The **ngrx/store** module allows you to define a slice of the root application state per each Feature module. The result is a single Root application state object, containing all the slice of states collected from all loaded Feature modules in the application.
- The performance benefits are remarkable in Angular application once you use the **ngrx/store** module. Change Detection in Angular is a huge topic and there isn't room for the details in this article. There are great articles listed in the sidebar.

In brief, Angular represents an application as a tree of components in a form of a hierarchy starting from the root component and going down to the leaf children components. At the same time, Angular creates a shadow tree for change detectors in a sense that every component has its own shadow change detector in the tree. Whenever a component's model changes, Angular runs the change detection process by traversing the tree of change detectors, compares each component's **@Input()** parameters and their properties to the state they had before the process ran, and makes sure all components in the application that are associated with the change are notified so that they can render a new HTML if necessary.

The default change detection strategy is not an efficient process in terms of performance as every component's **@Input()** parameter property is checked for any possible change. Angular provides an alternative change detection strategy which **OnPush** accommodates for. The performance is better, with its own strict conditions and requirements.

The **OnPush** strategy assumes that all components' **@Input()** parameters are immutable and unchangeable objects.

The Store uses the reducers to return new immutable composed slices of state. Such immutable data returned from the Store can be used to feed in the components' **@Input()** parameters. By using the Store, you're sure that all components' **@Input()** parameters are immutable and that the **OnPush** change detection strategy is applicable. Angular runs in the best optimized mode of performance when using this type of change detection process.

Angular triggers the change detection process only when the Store publishes a new state object. When the components' **@Input()** parameters are replaced, their references are changed and any change to a components' **@Input()** parameters' properties are ignored.

Angular Component Architecture

Angular Component Architecture can benefit Angular applications by taking full advantage of the Angular (**@Input()** and **@Output()**) and **ngrx/store** (**dispatch()** and **select()** methods) intrinsic features.

Figure 2 describes this pattern in detail.

This pattern defines two types of components:

- Smart or Container
- Dummy or Presentational

The container component is the only component that's aware of the existence of the Store. The ngrx/store module intrinsic feature facilitates the communication between this component and the Store. The following explains the communication:

- The component subscribes to the Store via the **select()** method to receive the stream of data requested whenever it's available in the Store.
- The component dispatches an action to the Store via the **dispatch()** method to signal the need to update the state.

Although the container component knows and communicates directly with the Store, the presentational component is not aware of the store. It simply uses Angular's intrinsic features to communicate with the container component. The container component acts like the middle man between the two while communicating with the Store. Any interaction between the container component and the presentational component filters through to the Store this way.

This is how the communication between the presentational component and the container component works:

- The presentational component defines **@Input()** parameters to receive any slices of data coming over from the state via the container component's subscription to the Store. It's the responsibility of the container component to provide the proper data required by the presentational component. Remember, those **@Input()** parameters are immutable objects!
- The presentational component uses **@Output()** Event Emitters to request any update on the state of the application. The container component handles the event of the presentation component, which, in turn, dispatches an action directly to the Store.

You can clearly see the differentiation of responsibility for each type of the components. Although the presentational component solely uses the Angular intrinsic features to render any HTML, the container component depends fully on the **ngrx/store** module intrinsic features.

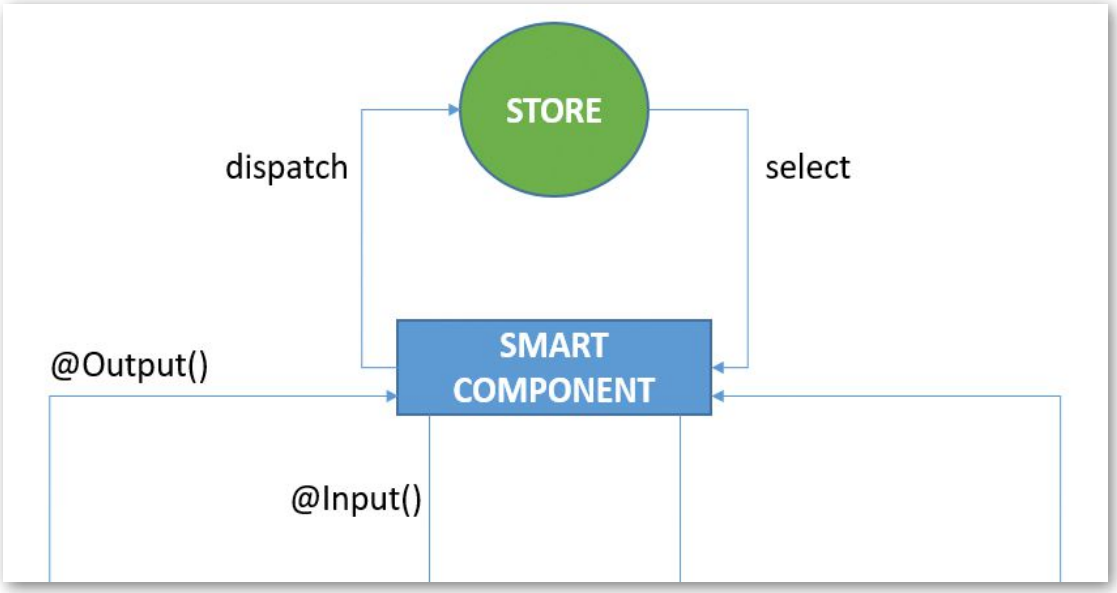


Figure 2: The Angular component architecture

By using this pattern, you're adapting one of the main concepts of software development, which is the Separation of Concerns. Only one layer of components in your application is aware of the Store, and the rest, that is, the building blocks of presentational components, behave, as if there's no store in the application. This pattern promotes composing your application of small, concise, and single responsibility presentational components to build up your application.

Migrate an Existing Angular Application to Use the NgRX Store Module

Now that you have some background information about what an application state is and how the NgRX Store helps with managing states, it's time to shift gears and move on to explore how to add and implement the **ngrx/store** module to an existing application.

Most applications you develop are pure vanilla Angular applications. Here, it might be more interesting to learn how an existing Angular application can be migrated to use the **ngrx/store** module. Therefore, I've chosen as a starting application the official Heroes Angular application provided by the Angular team to demonstrate Angular features.

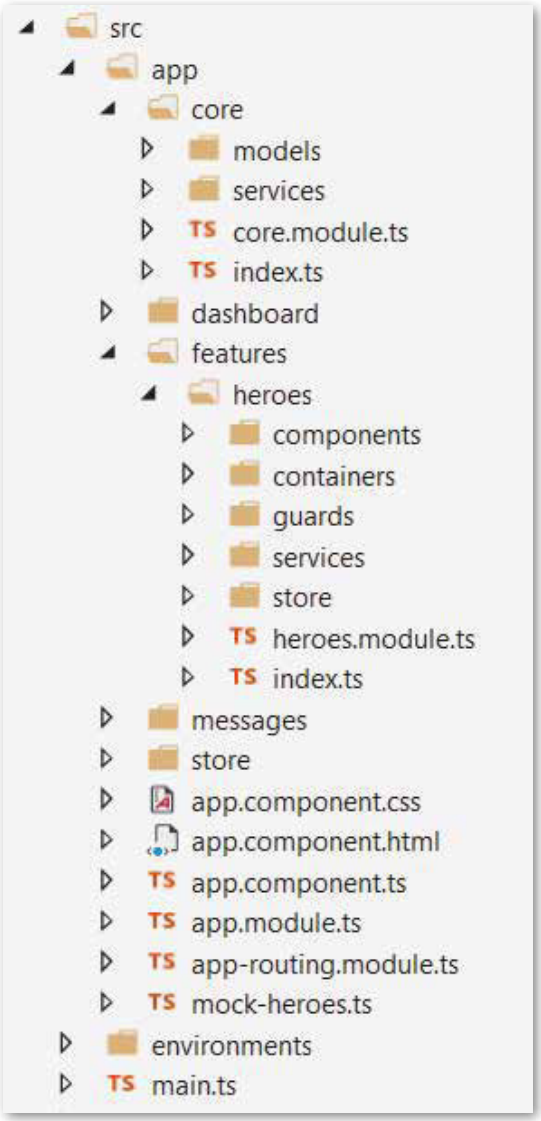


Figure 3: Store folder structure

> You're adapting one of the main concepts of software development, the Separation of Concerns, when you use this pattern.

Demonstrating the power of **ngrx/store** module in handling Angular Feature modules, I had to refactor a bit of the Heroes application by introducing a new Feature module called Heroes. This module now contains all Hero-related code and components. You can check the final Heroes application integrated with the **ngrx/store** module by following this link: https://stackblitz.com/edit/tour-of-heroes-store.

Figure 3 shows the new application structure.

Add the NgRX Store Module into an Existing Angular Application

First things first though, let's start by installing the following Node packages into the application.

- **@ngrx/store**. The **@ngrx/store** package represents the main NgRX Store package.
- **@ngrx/store-devtools.** The **@ngrx/store-devtools** package helps in instrumenting the Store to let you debug the application with time-travel debuggers like the famous Redux DevTools Chrome Extension (https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmbekcpmknklioeibfkpmmfibljd?hl=en).
- **ngrx-store-freeze.** The **ngrx-store-freeze** package represents a meta-reducer function of type **MetaReducer** used to prevent mutating the state and is used only during development.
- **@ngrx/effects.** The **@ngrx/effects** package handles the side effects of using **ngrx/store** module in an app. I haven't addressed the side effects yet, but will do so a little further down the track.
- **@ngrx/router-store.** The **@ngrx/router-store** package integrates the Angular Router with the **ngrx/store** module. The Store represents the single source of truth of an app and therefore, with the help of this Node package, the Store accesses Router-related information

Next up, how to add and configure the **ngrx/store** module to the application. (Refer to **Listing 1.**)

Start by importing the necessary modules from their corresponding Node packages, and then define an array of a single meta-reducer function. Once the application runs in Development mode, it activates and executes this meta-reducer function.

```
export const metaReducers: MetaReducer<any>[] =
  !environment.production ? [storeFreeze] : [];
```

Listing 1: Add ngrx/store module to App Module

```
import { StoreModule, MetaReducer } from '@ngrx/store';
import { EffectsModule } from '@ngrx/effects';
import { StoreDevtoolsModule } from '@ngrx/store-devtools';
import { storeFreeze} from 'ngrx-store-freeze';

import {
  StoreRouterConnectingModule,
  RouterStateSerializer
} from '@ngrx/router-store';

import {
  AppState,
  reducers,
  CustomSerializer,
  effects
} from './store';

export const metaReducers: MetaReducer<any>[] =
  !environment.production ? [storeFreeze] : [];

export const storeDevTools: ModuleWithProviders[] =
  !environment.production ? [StoreDevtoolsModule.instrument()] : [];

@NgModule({
  imports: [
    …
    StoreModule.forRoot((reducers) as any,
      { metaReducers }),
    EffectsModule.forRoot(effects),
    storeDevTools,
    …
  ],

…
```

A **MetaReducer** type represents a higher order reducer. A reducer acts as a pure function, so a **MetaReducer** represents a higher order function. By definition, a higher order function represents a function that takes an input parameter, a parameter that is itself a function, or a higher order function that returns a value of type function. A **MetaReducer** type accepts a reducer as an input parameter and returns a function with the exact same signature of a reducer. The **ngrx/store** module internally composes all of the provided reducers and wraps them with the provided meta-reducers. The **ngrx/store** module guarantees that the meta-reducer functions run first before the actual reducers.

A logging meta-reducer function represents a typical example of a useful meta-reducer. A logging meta-reducer function adds some logging messages before a reducer executes:

```
export function logger(reducer:
 ActionReducer<AppState>):
    ActionReducer<AppState> {
  return (state: AppState, action: any):
AppState => {
    console.log('state', state);
    console.log('action', action);

    return reducer(state, action);
  };
}
```

The **logger()** meta-reducer function shown accepts an input parameter of type **ActionReducer<AppState>** (i.e., a pure function or reducer) and also returns a function of type **ActionReducer<AppState>**. The function returned logs the **state** and **action** variables to the console before returning the wrapped reducer.

The **ngrx/store** module defines the **ActionReducer<T>** type as follows:

```
export interface ActionReducer<T,
   V extends Action = Action> {
    (state: T | undefined, action: V): T;
}
```

A meta-reducer function defined with such a signature makes coding all reducers and meta-reducers much easier to compose and wrap.

Going back to the App Module code, you define an array of **ModuleWithProviders** classes to wrap the **StoreDevtoolsModule.instrument()** method so you can import it later on the App Module.

```
export const storeDevTools:
ModuleWithProviders[] =
  !environment.production ?
     [StoreDevtoolsModule.instrument()] : [];
```

Finally, import the modules into the App Module as follows:

```
StoreModule.forRoot((reducers) as any,
      { metaReducers }),

EffectsModule.forRoot(effects),

storeDevTools,
```

For now, the **StoreModule.forRoot()** method accepts an empty object. Later on, you'll provide some reducers.

In addition, the **StoreModule.forRoot()** method accepts a second argument of type **StoreConfig** interface.

```
export declare type StoreConfig<T,
   V extends Action = Action> = {
    initialState?: InitialState<T>;
    reducerFactory?: ActionReducerFactory<T, V>;
    metaReducers?: MetaReducer<T, V>[];
};
```

In this case, you only provide an array of meta-reducer functions, as defined above.

Right after importing the **StoreModule** class, you also import the **storeDevTools** array to provide a better debugging experience.

Lastly, on the App Module, import the **EffectsModule** class with an empty array. Later on, when you create the

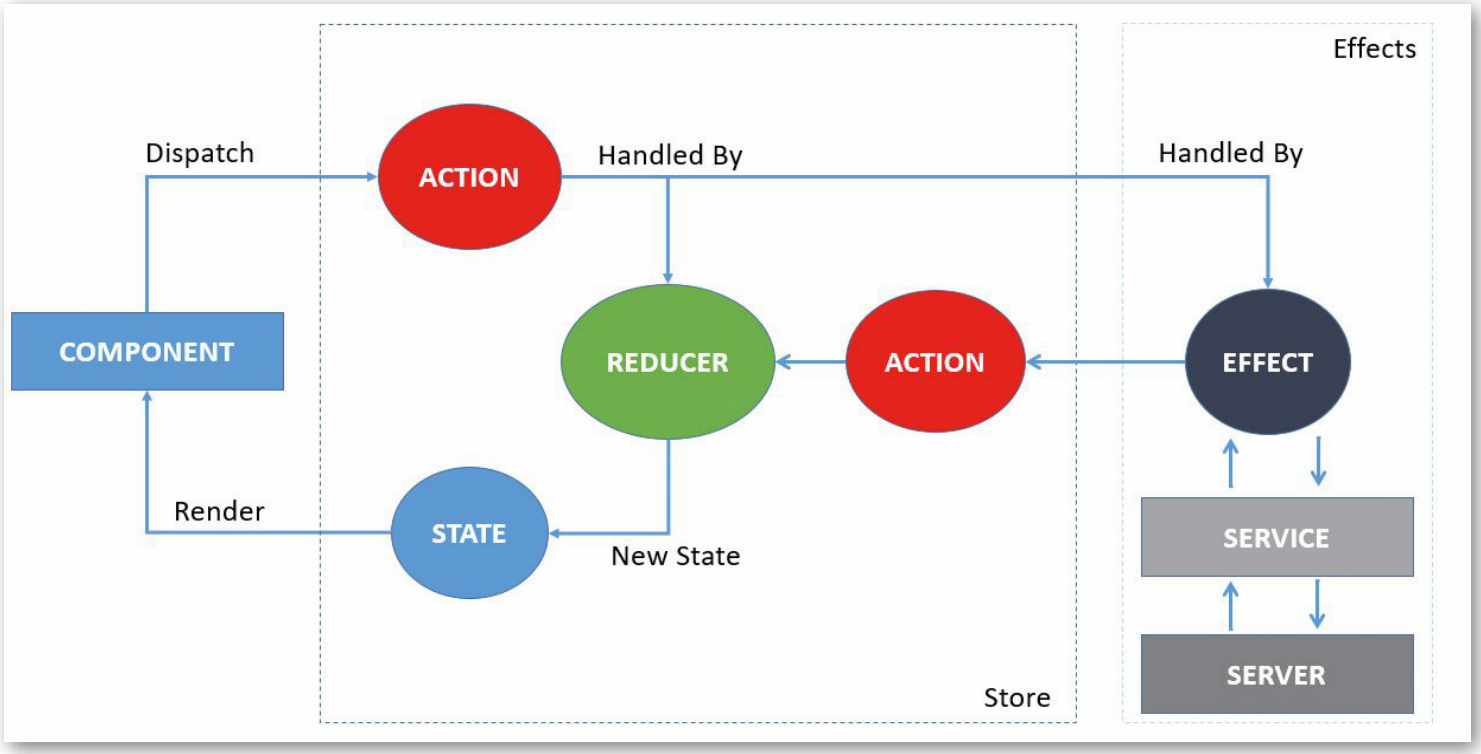


Figure 4: NgRX/store, ngrx/effects, and container components communication lifecycle

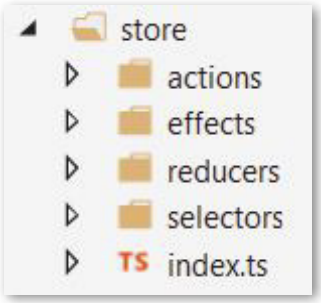


Figure 5: Store folder structure

Effects, I will revisit the App Module and fill in the Effects. I haven't touched on the **Effects** or side-effects thus far, but I will do so in the section to come.

Before you configure the **Heroes** module with the NgRX Store modules, let's proceed by defining the Store first and then returning back to the **Heroes** module to configure it for the NgRX Store features.

Store Side-Effects

A code that dispatches an action to the Store can sometimes lead to some side-effects. For instance, you dispatch an action to load all Heroes data into the application. Such an action leads to a side-effect of having to communicate, via an Angular Service, with a Web API hosted on the server-side (or cloud) to return the Heroes data. The direct side-effects of the code dispatching an action to the Store results in the communication with an external Service.

The **ngrx/effects** module isolates the side-effects from the container components and provides a neat and concise solution to handle them inside an NgRX Store-enabled Angular application. Later on, the **ngrx/store** module dispatches an action to the Store to pass on any results or data retrieved from the server.

Figure 4 summarizes the communication path when using the **ngrx/effects** module.

Here's what's happening:

1. The **ngrx/effects** module listens to the actions dispatched to the Store.
2. If there's an Effect hooked to listen to a dispatched action, the Effect runs and executes.
3. Once the Effect receives the data from the server-side, it dispatches a new action to the Store so that the reducer kicks in and handles the data received from the Effect and updates the state accordingly.

Let's go back to the application and define a workable Store folder structure.

Define Store Folder Structure

Now that you've configured the **ngrx/store** module in the application, the next obvious step is to add the Store folder structure into the application. You may organize your folder structure the way you find suitable. In my case, I follow the basic folder structuring as shown in **Figure 5.**

Remember that we need this Store folder structure in two places:

- The root of the application
- Inside the Heroes (Feature) module

Figure 5 shows the Store folder structure.

The structure is self-explanatory.

- The **actions** folder contains all Store actions.
- The **effects** folder contains all the Store effects.
- The **reducers** folder contains the Feature module state together with the reducers.
- The **selectors** folder contains all the selectors.
- The **index.ts** file is a barrel file to export everything from the Store (to make referencing the Store easier in the application via short paths).

Let's move on to define the actions.

Define Actions

When you first approach the **ngrx/store** module, you have to decide on the actions needed to run your application. Start by analyzing the process of loading Heroes data from the server-side and decide on the actions.

- You want to load all Heroes data from the server-side (Load Heroes Action).
- An Effect kicks in and retrieves the Heroes data from the server-side. The Effect also needs to inform the Store of the Heroes data retrieved, so it needs to dispatch an action (Load Heroes Success Action). In case the communication with the server-side fails or for any other reason, the Effect dispatches another action (Load Heroes Fail).

You can conclude that you need the following actions:

- Load Heroes action
- Load Heroes Success action
- Load Heroes Fail action

The **ngrx/store** module defines the **Action** interface as follows:

```
export interface Action {
    type: string;
}
```

You start the process of adding actions by defining some **const** variables that will serve as action types later on.

```
export const LOAD_HEROES =
   "[Heroes] Load Heroes";

export const LOAD_HEROES_FAIL =
   "[Heroes] Load Heroes Fail";

export const LOAD_HEROES_SUCCESS =
   "[Heroes] Load Heroes Success";
```

Always use a meaningful description for an action type that usually starts by specifying a category. In this case, the **Heroes** category.

Right after that, you define the custom action classes as follows:

```
export class LoadHeroes implements Action {
  readonly type = LOAD_HEROES;
}

export class LoadHeroesFail implements Action {
  readonly type = LOAD_HEROES_FAIL;

  // can pass any error from server-side
  constructor(public payload: any) {}
}

export class LoadHeroesSuccess
 implements Action {
  readonly type = LOAD_HEROES_SUCCESS;
  constructor(public payload: fromCore.Hero[])
{}
}
```

Every action class defines a **type** property because it implements the **Action** interface. In addition, in some cases, you need to define an optional **payload** property on the custom action classes so that the code that dispatches this action can pass in some additional data that the reducer uses later to compose a new state.

The **LoadHeroesFail** action class defines a **payload** property to hold the body of an **Exception** thrown during the communication with the server-side.

Similarly, the **LoadHeroesSuccess** action class defines a **payload** property to hold the actual Heroes data retrieved from the server-side and passes it to the reducer.

Finally, because you use TypeScript to develop Angular applications, you can add some action type checking by defining a new TypeScript type, called the **HeroesAction** type, to hold all of the action classes defined above. This way, a reducer handles only actions defined on this type.

```
export type HeroesAction =
  | LoadHeroes
  | LoadHeroesFail
  | LoadHeroesSuccess;
```

It's always a good idea to export all of the action classes defined in the Actions folder inside the **/actions/index.ts** barrel file. This makes it easier for referencing them later.

Now that you've defined the actions, let's move on to define the Feature module states and reducers.

Define Action Reducers

An action reducer or, simply, a reducer, is a pure function that the ngrx/store defines as an **ActionReducer** TypeScript interface. Reducers are the brains of the **ngrx/store** module. Whenever the reducer executes, it's responsible for creating and returning a new state.

First of all, you start by defining the **HeroesState** interface:

```
export interface HeroesState {
    entities: {[id: number]: fromCore.Hero},
    loaded: boolean,
    loading: boolean,
}
```

With this **HeroesState** interface in hand, you're tracking the following state information:

- **Entities:** This property is defined as an array-like object with a key of type **number** and a value of type **Hero** class. The Hero class is a simple class with only two properties: **id** of type **number** and **name** of type **string**.
- **Loaded and loading:** These Boolean properties track the status of loading the data from the server-side. On one hand, the **loading** property is set to **true** when the Service initiates the communication with the server-side to retrieve the Heroes data. On the other hand, it's set to **false** when the data arrives from the server-side. Also, the **loaded** property is set to **true** only when the data arrives from the server-side.

Listing 2: Define Effects for Heroes State

```
@Injectable()
export class HeroesEffects {
  constructor(
    private actions$: Actions,
    private service: fromCore.HeroService
  ) {
  }

  @Effect()
  loadHeroes$ =
    this.actions$.ofType(fromActions.LOAD_HEROES).pipe(
      switchMap(() =>
        this.service.getHeroes().pipe(
          map((heroes: fromCore.Hero[]) =>
            new fromActions.LoadHeroesSuccess(heroes)),

          /* return an observable of the error */
          catchError(error =>
            of(new fromActions.LoadHeroesFail(error)))
        )
      )
    );
}
```

Right after that, you define the **initialState** variable with some default values. This variable defines the default value of the **state** parameter on the **reducer()** function, as you will see soon.

```
export const initialState: HeroesState = {
    entities: {},
    loaded: false,
    loading: false,
};
```

You define the **entities** property as an empty object and set both the **loaded** and **loading** properties to **false**.

Then you define the reducer function:

```
export function reducer(
  state: HeroesState = initialState,
  action: fromActions.HeroesAction
): HeroesState {

  switch (action.type) { }

  return state;
}
```

The reducer is a function that accepts two parameters:

- **state:** The **ngrx/store** module calls the **reducer()** function by passing the previous state object saved in the Store. You give the **state** input parameter an initial value so as not to have an **undefined** value.
- **action:** The **action** parameter represents the action that is currently dispatched to the Store.

The **reducer()** function simply returns a new immutable state.

The **reducer()** function checks the action's **type** property and performs an action accordingly to return a new immutable state. To illustrate more on this idea, assume action type value is **LOAD_HEROES_SUCCESS**:

```
case fromActions.LOAD_HEROES_SUCCESS:
{
  const heroes = action.payload;

  const entities = heroes.reduce(
    (accEntities: {
        [id: number]: fromCore.Hero
    },
    hero) => {
    return {
      ...accEntities,
      [hero.id]: hero
    };
  },
  {
    ...state.entities // initial value
  }
);

  return {
    ...state,
    loading: false,
    loaded: true,
    entities
  };
}
```

An Effect defined somewhere in the application dispatches this action. It creates a new **LoadHeroesSuccess** action object and populates its **payload** with the data retrieved from the server-side.

You reduce the array of Heroes into an **array-like** object (this format makes it easy later on to add/update/remove from the **entities** property) and return a new immutable state object. You start by **Spreading** (an ES6 feature) the previous **state** properties and then include the new **entities** property (containing the array-like object of Heroes data) so the new **entities** property overrides the previous value.

You provide more action type cases to handle more action types. You can check the rest of states and reducers here: https://goo.gl/EABAAbe, https://goo.gl/EKyuT7.

Define each specific state together with its reducer inside their own code file. Use the **reducers/index.ts** barrel file to group all the states and reducers that you've defined separately into a single Feature state and reducer as follows:

```
import * as fromHeroes from './heroes.reducer';
import * as fromSearch from
 './heroes-search.reducer';

/**
 * Prepare feature module state
 */
export interface HeroesFeatureState {
```

```
    heroes: fromHeroes.HeroesState;
    search: fromSearch.SearchHeroesState
}
```

You compose the **HeroesFeatureState** interface by including all other states as properties on this interface.

```
/**
 * Register the reducers for the
 * HeroesFeatureState
 */
export const reducers:
   ActionReducerMap<HeroesFeatureState> = {
    heroes: fromHeroes.reducer,
    search: fromSearch.reducer
}
```

Then merge all reducers defined in this Feature module into a single **reducer** variable of type **ActionReducer <HeroesFeatureState>**, which is used later on to initialize the **StoreModule** class with reducers inside the Heroes module, as shown here:

```
StoreModule.forFeature('heroes', reducers),
```

Finally, define a selector to query the entire Feature module state inside the **reducers/index.ts** barrel file. Use this selector later on to query subsequent slices of the top-level Feature state, as you'll discover in the Define Selectors section below.

Define Effects

After you define the reducers, proceed to explore the Effects and define of few of them. (Refer to **Listing 2.**)

The first Effect to define is responsible for communicating with the server-side via an Angular Service to retrieve the Heroes data.

In the constructor of the **HeroesEffects** class, you inject two main things:

- **actions$:** An observable of all actions dispatched to the Store. The **ngrx/effects** module provides this observable.
- **service:** A custom Service wrapping all Heroes-related Web API calls (POST, GET, etc.)

Then, you define your first Effect block that you decorate with **@Effect()** decorator to signal to the **ngrx/effects** module that you want to register a new Effect.

You filter the **actions$** observable searching for an action of type **LOAD_HEROES**. In other words, you're handling a dispatched action of type **LOAD_HEROES**. Then you use the **switchMap()** operator coming from the RxJS Library in order to call the Heroes Service to get all the Heroes from the server-side. Once the data is ready, you dispatch the **LoadHeroesSuccess** action.

Another interesting Effect to go over is the one defined below:

```
@Effect()
createHeroSuccess$ =
this.actions$.ofType(
fromActions.CREATE_HERO_SUCCESS)
pipe(
      map(
 (action: fromActions.CreateHeroSuccess) =>
        action.payload),
      map(
 (hero: fromCore.Hero) =>
        new fromRoot.Go({
          path: ["/heroes/detail", hero.id]
        })
      )
);
```

You filter the **action$** observable searching for an action of type **CREATE_HERO_SUCCESS**. When the application creates a new **Hero** object, it dispatches this action with a payload of the specific **Hero** object created. This Effect listens for this action to be dispatched and navigates the user to a new component to view all the details of this new **Hero** object. It navigates the user by dispatching a special action called **Go** action. This is a Route action that I will cover further down in the article.

Because you usually have more than one Effect, go on and group all Effect classes into a single array inside the **effects/index.ts** barrel file. Later, you'll configure the ngrx/effects module inside the Heroes module by injecting this array to the **EffectsModule** class:

```
import { HeroesEffects } from
 './heroes.effects';
export const effects: any[] = [HeroesEffects];
```

And then inside the Heroes module:

```
EffectsModule.forFeature(effects)
```

You're finished configuring the **ngrx/store** and **ngrx/effects** modules inside the Heroes Feature module. Yay! Let's go on and define some selectors that you're going to use inside the container components later on.

Define Selectors

Actions, Reducers, and Effects are now in place, making the Store a workable environment. What you still need to do is figure out how the container components will respond to changes in the state.

You achieve this by defining selectors. Think of selectors as queries you execute against the Store to retrieve slices of the state.

You start first by defining a selector to retrieve the top-level Feature module state.

```
export const getHeroesFeatureState =
   createFeatureSelector
     <HeroesFeatureState>('heroes');
```

Use the **createFeatureSelector()** function to create a query for the state defined for the entire feature module. Provide a **string key** representing the exact same name that you used to configure the reducers inside the **StoreModule** class in the Heroes module.

More Info

To learn more about other kinds of Effects, click: https://goo.gl/yKU5s1

Selectors are defined here: https://goo.gl/AS1vNS, https://goo.gl/kpfRdm

To see all variations of the **select()** method, click: https://goo.gl/46Tn2F

For an example application of getHeroesLoaded demonstration by the ngrx/store team look here: https://goo.gl/7tkD1G

There's more on meta-reducers here: https://github.com/brandonroberts/ngrx-store-freeze.

Listing 3: HeroListComponent

```
@Component({
  selector: 'hero-list',
  changeDetection: ChangeDetectionStrategy.OnPush,
  template: `...`,
  styles: [`...`],
})
export class HeroListComponent {
  heroes$: Observable<fromCore.Hero[]>;

  constructor(private store:
   Store<fromStore.HeroesFeatureState>) { }

  ngOnInit() {
    this.heroes$ = this.store.pipe(select(fromStore.getAllHeroes));
    this.store.dispatch(new fromStore.LoadHeroes());
  }

  onDeleteHero(hero: fromCore.Hero): void {
    const remove =
     window.confirm('Are you sure you want to remove this hero?');
    if (remove) {
      this.store.dispatch(new fromStore.RemoveHero(hero));
    }
  }

}
```

Because the Store represents the state as a tree of properties inside the Store, use the selectors to traverse down the tree to retrieve slices of the state.

```
export const getHeroesState =
   createSelector(
fromFeature.getHeroesFeatureState,
(state: fromFeature.HeroesFeatureState) =>
      state.heroes);
```

Use the **createSelector()** function to create a query to retrieve the **HeroesState**. This function accepts one or more selectors. The last input parameter for this method accepts a **projector()** function. The projector function accepts all of the selectors defined on the **createSelector()** function as input.

Simply provide a single selector for the **createSelector()** function and for the projector() function, a single parameter which, in this case, is the **HeroesFeatureState** object. As a result, the **projector()** function you return is the **HeroesState** slice defined on the **HeroesFeatureState** object.

To get the entities out of the Heroes state object, define a new selector as follows:

```
export const getHeroesEntities =
  createSelector(getHeroesState,
     fromReducer.getHeroesEntities);
```

In this case, you call the **createSelector()** function passing in a single selector, the getHeroesState. The **projector()** function is already defined inside the reducer file and you pass it onto the **createSelector()** function to eventually return the **entities** property defined on the **HeroesState** object.

Another interesting feature of the Selectors is that you can compose selectors from various other selectors defined on different slices of the State. The selector below demonstrates this.

```
export const getSelectedHero =
  createSelector(
     getHeroesEntities, fromRoot.getRouterState,
    (entities, router):
      fromCore.Hero => {
      return router.state &&
         entities[router.state.params.id];
    }
);
```

One component of the Heroes module lists all the Heroes data available. Clicking any of the Heroes navigates you to the Details component to show all the details of the selected **Hero** object. Therefore, the component uses the selector above to retrieve the selected **Hero**.

This selector expects as input the **getHeroesEntities** and the **getRouterState** selectors. The **getRouterState** queries the **Router** object for information about the current **Route** (more on Router state in a moment). The **projector()** function expects the **entities** property and the current **Router** state as input parameters. Finally, given a Hero **id** (retrieved from the **router.state.params.id**) you query the **entities** property to return the **Hero** object intended.

Check the **HeroListComponent** to see an example of consuming those selectors. (Refer to **Listing 3.**)

You specify the **ChangeDetectionStragey.OnPush** at the component level. Having the **ngrx/store** module configured in an Angular application, you must always make use of the **OnPush** strategy.

> With the ngrx/store module configured in an Angular application, you must always make use of the OnPush strategy.

You inject the Store object into the component's **constructor()**. You will use the Store object later to execute selectors and dispatch actions to the Store.

Now define the **heroes$** observable variable to hold a reference for an observable stream of **Hero** objects. In the **ngOnInit()** method, you subscribe to the **getAllHeroes** selector by using a predefined method on the Store object, the **select()** method.

```
this.heroes$ =
    this.store.pipe(
      select(fromStore.getAllHeroes));
```

The Store itself is an RxJS **Subject** and you can **pipe()** in some selectors to retrieve the slices of data you're in-

terested in. The **select()** method allows you to execute a selector or access a top-level state object via its name.

Here, you are passing the **getAllHeroes** selector to the **select()** method. The **select()** method returns an observable of Hero array. The **heroes$** variable is automatically bound to any changes in this observable. When the reducer composes a new state object with new set of Heroes data, the **select()** method executes automatically and updates the **heroes$** variable. The **heroes$** variable is bound to the HTML markup with the use of Angular Async Pipes. The Async Pipe automatically subscribes to the **heroes$** observable so that the HTML always receives the latest and newest set of Heroes data.

```
<div *ngFor="let hero of heroes$ | async">
```

Next, dispatch an action to the Store to load the Heroes data:

```
this.store.dispatch(new fromStore.LoadHeroes());
```

Now that you've dispatched the action, internally, the Store calls on the Effect defined before to handle such an action. Right after that, the Store calls the reducer to compose a new state by populating the **entities** property with whatever data that the Effect sends in. Finally, because the Store is an RxJS **Subject**, it emits the **next()** call to inform all the components linked to respond to the selectors that a new state is available. In this example, the **heroes$** variable is automatically updated to reflect the new Heroes data.

> Every time you visit the HeroListComponent you dispatch an action to the Store to retrieve the Heroes data once again from the server-side.

A pitfall with this method is that every time you visit the **HeroListComponent** you dispatch an action to the Store to retrieve the Heroes data once again from the server-side. You're better off loading the data once and when needed. For that you need to define a new Route Guard.

```
{
   path: '',
   canActivate: [fromGuards.HeroesGuard],
   component: fromContainers.HeroListComponent
}
```

Use a Route Guard to dispatch an action to the Store to load the Heroes data. You do this once in the **Guard** and that's all!

Define a new Route Guard and implement the **canActivate()** function with your own logic to communicate with the Store to dispatch any action.

Implement the **canActivate()** function in such a way that if the Heroes data is not yet loaded, it asks to load them by dispatching an action to the Store. Otherwise, use whatever's in the Store in that moment of time.

```
@Injectable()
export class HeroesGuard
 implements CanActivate {
  constructor(
     private store: Store<fromStore.HeroesFeatureState>) { }

  canActivate(): Observable<boolean> {
    return fromStore.checkStore(this.store)
     .pipe(
      switchMap(() => of(true)),
      catchError(() => of(false))
    );
  }

}
```

The **canActivate()** function is defined inside a **HeroesGuard** class returning an observable of Boolean. It's either **true** or **false**. To activate or not to activate the route, that's the question! It then calls the **checkStore()** function that would return an observable of **true** only when the data has been successfully loaded.

The implementation of the **checkStore()** function is as follows:

```
export const checkStore =
  (store:
   Store<fromFeature.HeroesFeatureState>):
    Observable<boolean> => {

  return store.select(getHeroesLoaded).pipe(
    tap(loaded => {
      if (!loaded) {
        store.dispatch(
          new fromActions.LoadHeroes());
      }
    }),
    // wait here
    filter( (loaded: boolean) => loaded),
    take(1)
    );
 }
```

You execute the **getHeroesLoaded** selector and tap into the results. If the data is not loaded, dispatch an action to the Store to load the data. Otherwise, the RxJS **filter()** operator executes. The RxJS **filter()** operator only returns a value of **true**. Therefore, if the data hasn't been loaded yet, the processing of the above code pauses and waits for the data to be loaded. Because you dispatched an action inside the RxJS **tap()** operator, once the data is available in the Store, the **store.select()** method re-runs, causing the **getHeroesLoaded** selector to return true and therefore, the **filter()** operator returns with a value of **true**. Finally, you take a single value from the stream and return the results.

You have successfully implemented the **ngrx/store** module inside the Heroes Feature module. It's time to explore how to implement the **ngrx/store** module on the App Module and how to define the Router state.

Define the Router State

There is no harm in directly accessing the **Router** object from within the container components. However, given that the application treats the Store as a single source of truth, it's wise to integrate the **ngrx/store** module with the Router object.

Earlier, you installed the **@ngrx/router-store** Node package. This package is responsible for hooking up the Store with the Router object. To properly configure the **ngrx/router-state** module in the App Module, refer to **Listing 4**.

First, import the **StoreRouterConnectingModule** module. Then override the **RouterStateSerializer** class with a custom implementation. By default, when you import the **ngrx/router-store** module as-is, it registers a default **RouterStateSerializer** class that brings in the entire router state into the **ngrx/store** module. If you want to track only some properties of the router state in the Store, you must define a custom **RouterStateSerializer** class to extract only those properties from the router state.

Therefore, you add to the **reducer/index.ts** barrel file of the Store folder defined at the root of the application the following:

```
export interface RouterStateUrl {
    url: string;
    queryParams: Params;
    params: Params;
}

export interface AppState {
  router:
   fromRouter.RouterReducerState<RouterStateUrl>
}
```

You start by defining the custom **RouterStateUrl** interface. In this case, you're only tracking the **URL** of the **Route**, Query Parameters, and all Parameters passed to the **Route** itself.

Right after that, you define the root state of the application with a single sub-state that's the router state. The router state makes use of the built-in wrapper state, the **RouterReducerState<RouterStateUrl>**.

Then define the application-level reducer as follows:

```
export const reducers:
   ActionReducerMap<AppState> = {
    router:
       fromRouter.routerReducer
}
```

Here, you're using the built-in **routerReducer** class from the **ngrx/store** module. This reducer takes as input the previous or current initial state of type **RouterReducerState<RouterStateUrl>** and creates a new state.

Next, you want to configure the **StoreModule** class in the App Module with the new reducer.

```
    StoreModule.forRoot((reducers) as any,
      { metaReducers }),
```

Within the reducer file above, you define the **CustomSerializer** class, as shown in **Listing 5**.

Given a **RouterStateSnapshot** object, you extract the information required from the router state and return a new immutable object of type **RouterStateUrl**.

In summary, the **StoreRouterConnectingModule** class hooks into the Angular Router and dispatches a **ROUTER_NAVIGATION** action to the Store every time the Angular Router navigates to a new route. The **StoreRouterConnectingModule** class uses the **CustomSerializer** class to extract the needed information from the router state in order to populate the custom **RouterStateUrl** object.

After you configure the router state property, start using the selectors like the one in the Define Selectors section above, mainly the **getRouterState** selector.

Listing 4: Configure Router State

```
@NgModule({
  imports: [
    …
    StoreRouterConnectingModule,
  ],
  declarations: [
    …
  ],
  providers: [
    {
      provide: RouterStateSerializer,
      useClass: CustomSerializer
    }
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

Listing 5: CustomSerializer class

```
export class CustomSerializer
  implements fromRouter.RouterStateSerializer<RouterStateUrl> {
    serialize(routerState: RouterStateSnapshot): RouterStateUrl {
        const { url } = routerState;
        const { queryParams } = routerState.root;

        let state: ActivatedRouteSnapshot = routerState.root;
        while (state.firstChild) {
            state = state.firstChild;
        }
        const { params } = state;

        return { url, queryParams, params };
    }
}
```

Listing 6: Custom Router Actions

```
import { Action } from '@ngrx/store';
import { NavigationExtras } from '@angular/router';

export const GO = '[Router] Go';
export const BACK = '[Router] Back';
export const FORWARD = '[Router] Forward';

export class Go implements Action {
    readonly type = GO;
    constructor(public payload: {
        path: any[];
        query?: object;
        extras?: NavigationExtras;
    }) {}
}

export class Back implements Action {
    readonly type = BACK;
}

export class Forward implements Action {
    readonly type = FORWARD;
}

export type Actions = Go | Back | Forward;
```

Listing 7: Router Effects

```
@Injectable()
export class RouterEffects {
  constructor(
    private actions$: Actions,
    private router: Router,
    private location: Location
  ) {}

  @Effect({ dispatch: false })
  navigate$ =
    this.actions$.ofType(RouterActions.GO).pipe(
      map((action: RouterActions.Go) => action.payload),
      tap(({ path, query: queryParams, extras }) => {
        this.router.navigate(path, { queryParams, ...extras });
      })
    );

  @Effect({ dispatch: false }) // no need to dispatch an action
  navigateBack$ =
    this.actions$.ofType(RouterActions.BACK).pipe(
      tap(() => this.location.back()));

  @Effect({ dispatch: false }) // no need to dispatch an action
  navigateForward$ = this.actions$
    .ofType(RouterActions.FORWARD)
    .pipe(tap(() => this.location.forward()));
}
```

Go ahead now and introduce some **ngrx/store** actions to communicate with the **Router** 1 and the **Location** service provided by Angular.

Define Router Actions and Effects

Older versions of **@ngrx/router-store** Node package defined some actions and Effects to serve as a wrapper on top of the **Router** object. With the latest version, those actions were dropped in favor of accessing the **Router** directly.

I prefer using those action wrappers and I tend to include them in my Angular applications. I brought in most of the code for those action wrappers from the older version of **ngrx/router-store** package as you can see here. https://github.com/ngrx/platform/blob/master/MIGRATION.md.

To start with, navigate to **/app/store/actions/router.actions.ts** and explore the file contents, as shown in **Listing 6**. You have just added basic NgRX Store routing actions.

The **/app/store/effects/router.effects.ts** file contains the **RouterEffects** class that implements the Effects corresponding to the actions above. (Refer to **Listing 7.**)

The **Go** action is handled by an Effect that performs a navigation using the **Router** object **navigate()** method.

Notice how the **@Effect()** decorator specifies a value of **false** for the **dispatch** property. This signals to the **ngrx/effects** module that this Effect won't dispatch any action to the Store.

The Effects for both **Back** and **Forward** actions use the **Location** Service.

Once you have the router actions and Effects in place, you can easily make use of those actions in your container components as follows:

```
onGoBack(): void {
  this.store.dispatch(new fromRoot.Back());
}
```

Conclusion

With the intention of providing some insight into the NgRX Store capabilities, this introduction sheds light on the application state concepts, on how to use the Store in both Angular Root and Feature modules, and demonstrates how an existing Angular application can be migrated to use the NgRX Store to manage its state.

Application requirements sometimes gets more complicated to handle with a basic Store structure. In an upcoming article, join me as I delve further into using the NgRX Store to model an Angular application. This requires dynamically embedding multiple instances of a container component inside a parent component, in a way that the application uses a single state structure to manage the state of all dynamically embedded container components. This is a requirement that's rather rare and a little complicated but needed in developing Enterprise and Portal applications.

Bilal Haidar

CODE

Marking up the Web with ASP.NET Core and Markdown

Over the last few years, Markdown has become a ubiquitous text-entry model for HTML text. It's creeping up in more and more places and has become a standard for documents that are shared and edited for documentation purposes on the Web. If you're using Git and GitHub in particular, you're already using Markdown for formatting your README.md files and likely all other documentation-related documents that you write for your software development projects. Most developer-related documentation you see online today, whether it's commercial documentation from companies like Microsoft, Google, and so on, or generic documentation solutions like ReadTheDocs or KavaDocs, they're all created and maintained with Markdown.

Rick Strahl
www.west-wind.com
rstrahl@west-wind.com

Rick Strahl is president of West Wind Technologies in Maui, Hawaii. The company specializes in Web and distributed application development and tools, with focus on Windows Server Products, .NET, Visual Studio, and Visual FoxPro. Rick is the author of West Wind Web Connection, West Wind Web Store, and West Wind HTML Help Builder. He's also a C# MVP, a frequent contributor to magazines and books, a frequent speaker at international developer conferences, and the co-publisher of CoDe Magazine.

Markdown All the Things

Because Markdown is text based, it's easy and very fast to write in. It's easy to compare and merge for change tracking using built in merge features in source control tools like Git, and it doesn't require a special editor create it—it's plain text and works with a textbox.

> Markdown has quickly become a widely used standard for documentation and Web-based extended text entry.

Although Markdown provides only a limited set of HTML features, it provides the most common features that are needed for most writing purposes. Unlike WYSIWYG HTML editors that are usually difficult to write efficiently in due to their messy formatting and lagging performance, Markdown is only text and can be entered in a simple text box. Markdown has no support for layout features, but almost exclusively focuses on inline markup for text. Headers, text bolding, italics, underlines, links, and images, and slightly more complex structures like lists and tables, can all be expressed with very short and terse but easily remembered markup characters. The reason Markdown has become so popular is that it's very simple and can be easily learned within a few minutes of reviewing the basics. It's mostly logically flowing text with a little bit of markup.

Markdown is addictive too. Once you start writing with Markdown, you'll want to use it everywhere. So much so that I often find myself typing Markdown text in places where I wish it worked but doesn't—in emails, Skype text, or inputs on various websites.

Markdown is also getting more mainstream—you see Markdown cropping up in all sorts of document solutions these days and if you're building documentation of any kind, it's very likely that Markdown is part of that process. But Markdown extends much further than just documentation: You can also use it in your own applications to hold rich text content for things like product descriptions, interactive discussions, notes, and any other "memo" type of text that can benefit from a little bit of formatting to make it easier to read.

Putting Markdown to Work in Your Applications

In this article, I'll introduce a number of Markdown features that you can add to your ASP.NET Core applications. I'll start with what you need to **parse Markdown text into HTML** using a simple library that lets you use Markdown easily in code and in your ASP.NET Core MVC Razor pages. This is all you need to render your own Markdown text from application data into your applications. I'll also show a couple of useful content helpers that allow you to embed static **Markdown Islands** text into Razor pages using a Markdown TagHelper, and a handy Markdown middleware component that allows you to easily **serve Markdown pages as HTML content** in the context of your application's Web UI.

If you're impatient, you can jump straight to the NuGet package or the GitHub repository to get going. Otherwise read on.

- Install-Package Westwind.AspnetCore.Markdown
- Westwind.AspnetCore Repository on GitHub (https://bit.ly/2G4VfhM)

Markdown Parsing in .NET

The first thing needed is a way to parse Markdown to HTML in .NET. This is surprisingly easy as there are a number of Markdown parsers available for .NET. The one I like to use is called **MarkDig** (https://github.com/lunet-io/markdig), a relative newcomer that's very fast and provides a nice extensibility pipeline, making it possible to build custom Markdown extensions.

MarkDig is open source and available as a NuGet package to add to your .NET Core or full framework projects:

```
Install-Package Markdig
```

Using MarkDig in its simplest form, you can do the following:

```
public static class Markdown
{
  public static string Parse(string markdown)
  {
    var pipeline = new MarkdownPipelineBuilder()
        .UseAdvancedExtensions()
```

```
        .Build();
    return Markdown.ToHtml(markdown, pipeline);
  }
}
```

MarkDig uses a configuration pipeline of support features that you can add on top of the base parser. The **.UseAdvancedExtensions()** method adds a number of common extensions (like GitHub-Flavored Markdown, List Extensions, etc.), but you can also add each of the components you want and customize exactly how you want Markdown to be parsed.

The code works, but it's not very efficient as the pipeline is recreated for each parse operation. It's much better to build a small abstraction layer around the Markdown parser, so the parser can be cached for better performance. You can check out the code to create a **MarkdownParserFactory** and a customized **MarkdownParser** implementation on GitHub (https://bit.ly/2KNV8e4) that includes an **IMarkdownParser** interface containing a **.Parse(markdown)** method that's ultimately used to handle the rendering.

To make this functionality easily accessible from anywhere, a static **Markdown** class wraps the factory and parses functions like this:

```
public static class Markdown
{
  public static string Parse(string markdown,
                  bool usePragmaLines = false,
                  bool forceReload = false)
  {
     if (string.IsNullOrEmpty(markdown))
       return "";

     var parser = MarkdownParserFactory
        .GetParser(usePragmaLines, forceReload);
     return parser.Parse(markdown);
  }

  public static HtmlString ParseHtmlString(
                string markdown,
                bool usePragmaLines = false,
                bool forceReload = false)
  {
     return new HtmlString(Parse(markdown,
                  usePragmaLines, forceReload));
  }
}
```

This Markdown class can then be used in the application and the components I describe later to access Markdown functionality. The Markdown middleware configuration also allows using dependency injection for accessing these components that I'll describe later.

With this class in hand, you can now easily parse Markdown to HTML. To parse Markdown in code and retrieve the string, you can use:

```
string html = Markdown.Parse(markdownText)
```

To parse Markdown and get back a Razor embeddable HTML string, you can use the **.ParseHtmlString()** method:

```
<div>
  @Markdown.ParseHtmlString(
          Model.ProductInfoMarkdown)
</div>
```

These simple helpers make it easy to turn stored Markdown into HTML in your applications.

> Markdown is addictive: Once you start writing with it, you'll want to use it everywhere.

Markdown as Static Content in a Dynamic Website

When building dynamic Web applications, we often don't think about static content much. Because there's usually not a lot of static content, we go ahead and code up pages like About, Contact us, Privacy Policy, etc., using plain old HTML. Most of these pages are completely static and usually don't contain anything more than basic text with a few headers and lists and other simple paragraph formatting.

Wouldn't it be nice to replace these plain text portions of a Web page with a block of static Markdown text that's embedded in the content? Or, even better, have a very simple way to serve Markdown files directly inside your website just like other static files?

To make this happen, I created two additional components:

- **Markdown Islands: A TagHelper to embed Markdown blocks into Razor views.** The TagHelper supports embedding static or model-bound Markdown islands into any Razor View or Page. Using the TagHelper lets you replace longer HTML text blocks with easier-to-edit Markdown blocks in a natural way.
- **Middleware to serve Markdown files as full content pages.** The middleware allows you to configure a folder or the entire site to serve .md files as self-contained content pages. The middleware works by converting .md file content on disk and merging it into a configured Markdown Page template that you create. The Markdown gets rendered into this template, producing a page that matches the UI of the rest of your site.

> Markdown Islands are blocks of Markdown text embedded in a larger HTML document.

Let's take a look how to build both of these components and, in the process, see how two important ASP.NET Core concepts, **TagHelpers** and **Middleware**, work and how you can create your own implementations.

Markdown and HTML Styling

It's important to understand that when you use Markdown and you see an HTML preview, that preview is a particular interpretation of that HTML. Likewise, when you embed rendered Markdown into your own site, your CSS styling may make the HTML look different from what you saw in a Markdown preview.

In other words, what rendered Markdown looks like depends entirely on the host application that renders and displays it.

Using a TagHelper to Embed Markdown into a View

The Markdown TagHelper's main purpose is to take a bit of Markdown text either from its embedded content or from a Model, turn it into HTML, and embed it into the Razor output. The Markdown TagHelper allows you to embed static Markdown text into a Razor view or page using a **<markdown>** tag:

```
<h3>Markdown TagHelper Block</h3>

<markdown normalize-whitespace="true">
    ## Markdown Text

    * Item 1
    * Item 2

    The current Time is:
    **@DateTime.Now.ToString("HH:mm:ss")**
</markdown>
```

The text between the tags is evaluated and turned from Markdown into HTML, which looks something like **Figure 1**. Notice that the Markdown snippet contains an embedded Razor expression that's evaluated and embedded **before** the Markdown parsing occurs. This is quite powerful, as you can use Razor logic inside of the Markdown content, which effectively gives you a way to automate Markdown text. This embedded Markdown parsing is free, courtesy of the Razor parser in ASP.NET Core, and it opens up a lot of opportunities for making it easier to create text-based formatting for things like merges and form letters.

The tag helper also supports Model binding using a **markdown** attribute on the tag where **MarkdownText** is a property on the passed-in Model object:

```
@model MarkdownModel

<markdown markdown="MarkdownText" />
```

Before you can use the tag helper, you have to register it. You do this in the **__ViewImports.cshtml** special Razor View or Page so that the TagHelper is available in all pages of the application. Please note that both MVC Views and Razor Pages use separate sets of pages and if you mix and match, you need to add the following to both locations.

```
@addTagHelper *, Westwind.AspNetCore.Markdown
```

Creating the Markdown TagHelper

Let's take a look and see how to create this TagHelper. The interface to create a TagHelper is primarily a single method process interface that takes a TagHelper **Context** to hold the element, tag, and content information that you can then use to generate an output string to replace the TagHelper tag and embed into the Razor content. It's a very simple interface that's easy to understand and work with and often doesn't require very much code. Case in point: This Markdown TagHelper is very small and most of the code is helper code that has nothing to do with the actual Markdown logic.

A TagHelper encapsulates rendering logic via a very simple **ProcessAsync()** method interface that renders a chunk of HTML content into the page at the location where the TagHelper is defined. The **ProcessAsync()** method takes a TagHelper Context as input to let you get at the element and attributes for input, and provides an output to which you can write string output and generate your embedded content.

To use TagHelpers, they have to be registered with MVC, either in the page or, more likely, in the **_ViewImports.cshtml** page of the project.

Markdown and HTML Sanitation

Markdown is effectively a superset of HTML in that you can embed raw HTML into a Markdown document. Because Markdown parses into HTML **you should treat Markdown captured from users the same way you treat raw HTML: It's potentially unsafe and open to XSS attacks**.

The Westwind.AspnetCore.Markdown library includes basic support for HTML Sanitation via optional SanitizeHtml parameters and attributes in the Parse() and ParseHtml() methods, the TagHelper and the Markdown configuration for the Middleware Page Handler.

For more info on concerns and implementation, check out my detailed blog post on this topic: https://bit.ly/2znFRwM

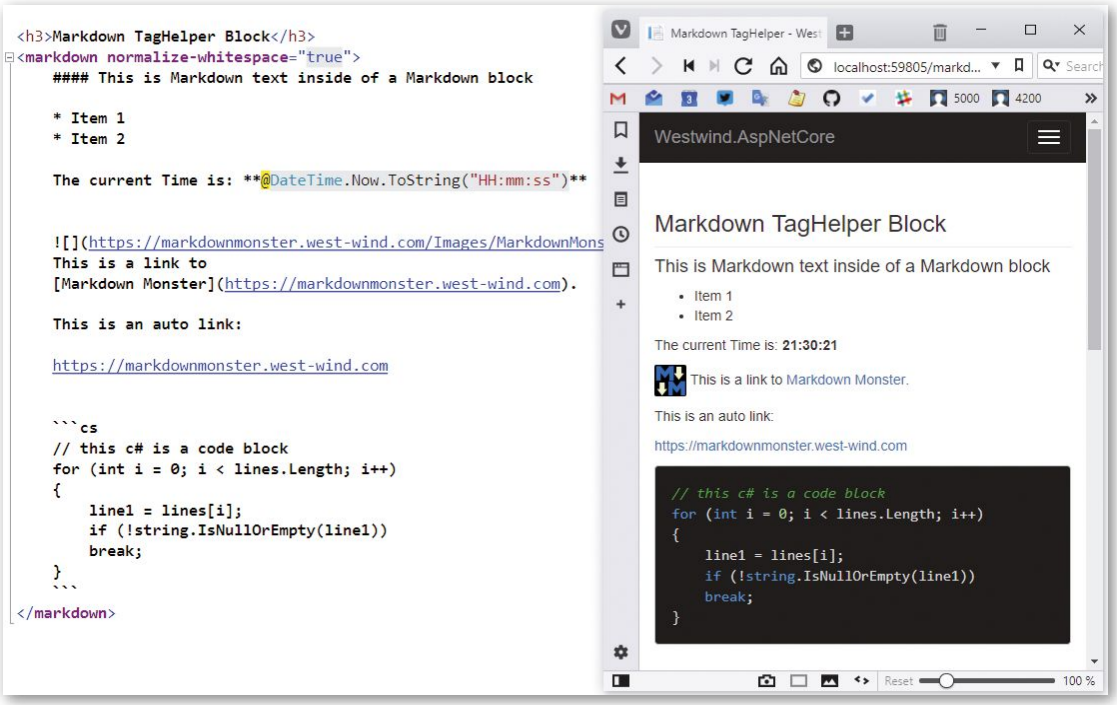


Figure 1: Output from the Markdown TagHelper rendered into the stock ASP.NET Core templates

Listing 1: A Markdown TagHelper to parse embeded Markdown Content

```
[HtmlTargetElement("markdown")]
public class MarkdownTagHelper : TagHelper
{

[HtmlAttributeName("normalize-whitespace")]
public bool NormalizeWhitespace { get; set; } = true;

[HtmlAttributeName("markdown")]
public ModelExpression Markdown { get; set; }


public override async Task ProcessAsync(TagHelperContext context,
                                        TagHelperOutput output)
{
    string content = null;
    if (Markdown != null)
        content = Markdown.Model?.ToString();
    if (content == null)
        content = (await output
                    .GetChildContentAsync(NullHtmlEncoder.Default))
                    .GetContent(NullHtmlEncoder.Default);

    if (string.IsNullOrEmpty(content))
        return;

    content = content.Trim('\n', '\r');

    string markdown = NormalizeWhiteSpaceText(content);

    var html = Markdown.Parse(markdown);

    output.TagName = null;  // Remove the <markdown> element
    output.Content.SetHtmlContent(html);
} }
```

To create a Tag Helper:

- Create a new Class and Inherit from **TagHelper**
- Create your TagHelper implementation via **ProcessAsync()** or **Process()**

To use the TagHelper in your application:

- Register your TagHelper in **_ViewImports.cshtml**
- Embed **<markdown></markdown>** blocks into your pages
- Rock on!

For the **<markdown>** TagHelper, I need to create a content control whose content can be retrieved as Markdown and then be converted into HTML. Optionally, you can also use a **Markdown** property to bind Markdown text for rendering, so if you have Markdown as part of data in your model, you can bind it to this property/attribute in lieu of static content you provide.

Listing 1 shows the base code for the **Markdown TagHelper** class that accomplishes these tasks.

Note that the Markdown property is not specified as a string but as a **ModelExpression,** which means it expects a Model value that corresponds to the name of a property on the Model. To grab the value, the expression can use the **Model** property, so **Markdown.Model** retrieves whatever the Model's value is. If that value has something in it, it will be used because the value takes precedence over content.

If content is specified instead, the content is retrieved with the **GetChildContentAsync().GetContent()**, which retrieves the content contained between the **<markdown></markdown>** tags. These routines retrieve the content and also execute any embedded Razor expressions. The **NullHtmlEncoder.Default** is passed to ensure that any Razor expressions within the tags are **not HTML-encoded**. This is important because I don't want my Markdown text passed into the Markdown Parser to be encoded by Razor. There are two reasons: HtmlEncoding might mess up some Markdown Tags in evaluated expressions, and the Markdown parser will do its own HTML encoding as part of the Markdown parsing.

Once I have the raw tag markdown content, I have to normalize the white space, which means stripping out leading spaces applied to a block of text. In Markdown, leading white space is important and four spaces or a tab signify a code block. Unless you left-justify the Markdown, it renders all of it as code. There's a **normalize-whitespace** attribute that can be set on the TagHelper and it's **true** by default. You can explicitly set the attribute to **false** if you explicitly justify left or if you really need the leading white space in place.

After normalization, I now have the normalized Markdown to parse and I can simply call **Markdown.Parse(markdown)** to retrieve the HTML. All that's left is to set the output on **output.Content**. Here, I use **SetHtmlContent()** to set a string result that's the result HTML.

> Most dynamic sites have some content pages that are mostly static and are easier to create using Markdown rather than HTML Tag Soup.

There you have it: HTML output generated from static or model-bound Markdown. You can check out the full source code for the TagHelper including the white space normalization on GitHub (https://bit.ly/2rwW6Dg).

Serving Markdown Pages as HTML in Any ASP.NET Core Site

When you're building a website, wouldn't it be nice to have an easy way to add in a documentation section or a blog area on the website simply by being able to drop Markdown files into a folder and then have those Markdown files served as content pages with your site's brand identity rendered around it? This idea is nothing new—most CMS systems and content generators do exactly that. But these tend to be dedicated tools that are separate from your Web application. Wouldn't it be nice to have a simple way to just add Markdown page serving capabilities into your **existing Web application**?

BabelMark: Validating Markdown

Although overall Markdown is fairly standard, there are slight differences among Markdown engines. It's possible that your previewer may render slightly differently than what your site renders.

If you want to check out how various engines render Markdown, you can take a look at BabelMark, which lets you enter some Markdown text and see the HTML result for a huge number of Markdown parsers: https://johnmacfarlane.net/babelmark2/

Let's take a look at a ASP.NET Core middleware component that makes it easy to set up folders to serve Markdown files as content from your website.

What Do You Need to Serve Markdown Pages?

Here are the requirements for serving static Markdown pages:

- A "wrapper" page that provides the site UI around your rendered Markdown
- A content area into which the rendered HTML gets dropped
- The rendered Markdown text from the file
- Optional YAML parsing for title and headers
- Optional title parsing based on a header or the file name
- Folder-level configuration for the template and options

The idea for all of this is pretty simple: For each configured folder hierarchy, you can specify a Razor template that acts as a Markdown content host page. This page is very simple—in fact, for most sites, I expect this page to have no content other than the embedded markdown and a reference to the site's **_layout.cshtml** page.

Inside of the Markdown page template, a model is passed in that holds the rendered markdown text, which I can then embed into the template at the desired location. I can create a template any way I like, either as a standalone HTML page or simply referring back to the layout page and embedding the **@Model.RenderedMarkdown** property into the page.

I also need to create a configuration to hook up the middleware that specifies which folders (or the root) to work with and whether I want to handle extensionless URLs in addition to handling the **.md** extension.

Then I can simply drop files with a **.md** extension into my site's **wwwroot** folder and the configured path(s) just as I would with static HTML files.

Getting Started

If you want to try out the middleware I describe in this post, you can install the from here:

```
PM> Install-Package Westwind.AspNetCore.Markdown
```

Set Up the Markdown Middleware

Once the NuGet package is installed, you can now use the middleware. This is done by hooking up the middleware configuration:

- Use **AddMarkdown()** to **configure** the page processing.
- Use **UseMarkdown()** to **hook up** the middleware.
- Create a Markdown View Template (the default is: **~/Views/__MarkdownPageTemplate.cshtml**).
- Create **.md** files for your content.

The following configures a **/posts/** folder in your application to allow for Markdown document serving:

```
public void ConfigureServices(
                    IServiceCollection services)
{
  services.AddMarkdown(config =>
  {
    config.AddMarkdownProcessingFolder(
       "/posts/",
       "~/Pages/__MarkdownPageTemplate.cshtml");
  });

  services.AddMvc();
}
```

For this basic configuration, I provide a path of a base folder for which Markdown documents are handled. You can configure multiple folders and also use the root path (/). You can also specify a Razor template that's used as the container for the markdown content.

The code above is the simplest thing possible, but there are additional options you can use for further configuration. You can configure multiple paths, specify how files are processed, provide a pre-process handler that provides you with a code hook to run before the markdown document is rendered, and you can customize the MarkDig Markdown pipeline configuration.

Next, I need to hook the middleware into the middleware pipeline in the Startup's **Configure()** method using **app.UseMarkdown()**:

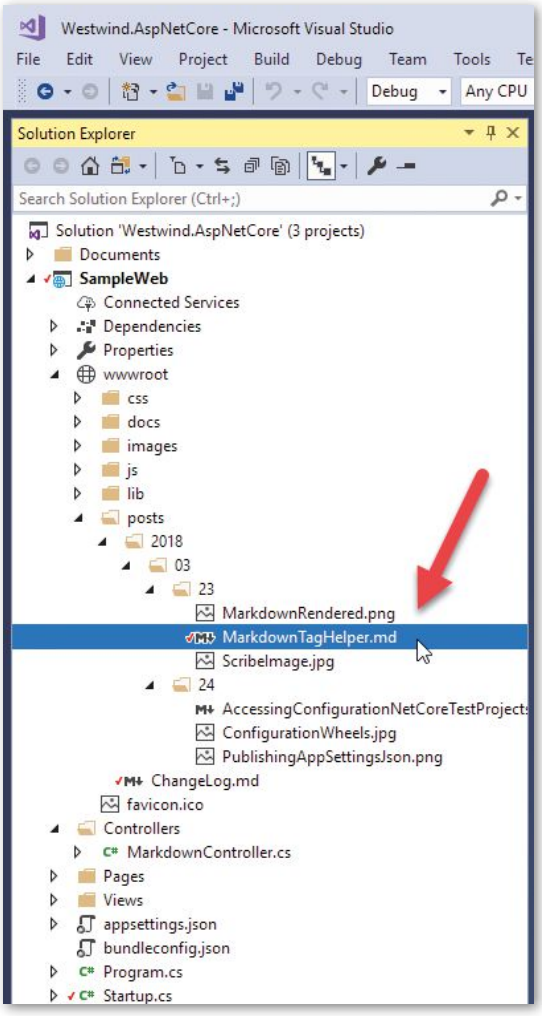


Figure 2: Dropping a markdown file with content into a folder for rendering

Code Syntax Highlighting

When you render Markdown that includes code blocks to HTML, your previewer may show that code using syntax coloring. But that coloring isn't native to the HTML output that was generated. Code is simply rendered with a **<pre><code>** that also includes a language code. It's the responsibility of the host application that renders your HTML to display your code with highlighting.

There are a number of JavaScript highlighters available and I like **highlight.js** (https://highlightjs.org/), which requires only a tiny code snippet to embed code. For more info, please see this blog post: https://bit.ly/2KMOPaq

```
public void Configure(IApplicationBuilder app,
                      IHostingEnvironment env)
{
    app.UseMarkdown();
    app.UseStaticFiles();
    // MVC required for Razor template rendering
    app.UseMvc();
}
```

Create a Markdown Container Razor Page

The Markdown middleware relies on MVC to render the Markdown content by essentially rewriting the request path and pushing the current request path into a custom controller that the component provides. The controller then renders the Razor template that I specified in the **AddMarkdown()** configuration shown earlier.

The simplest possible template looks like this:

```
@modelWestwind.AspNetCore.Markdown.MarkdownModel
@{
    ViewBag.Title = Model.Title;
    Layout = "_Layout";
}
<div style="margin-top: 40px;">
    @Model.RenderedMarkdown
</div>
```

The model contains two values of interest, the **RenderedMarkdown** and the **Title**, which you can embed into the template. **Title** is inferred from a YAML header's **title** property, if present, or the first **#** header tag in the document. If neither of these are present, no title is set. If you want to push other values into your view, the configuration also supports a preprocessing hook that hooks into a Markdown request

```
folderConfig.PreProcess = (model, controller) =>
{
    // controller.ViewBag.Model =
     new MyCustomModel();
};
```

The model also has relative and physical path properties and provides access to the active folder configuration so your pre-processing code can run complex logic based on which file is activated, and it can pass that data into your view via the **ViewBag** or **ViewData** properties on the controller.

At this point, I can start dropping Markdown files into my **/wwwroot/posts/** folder. I'm going to create a folder hierarchy that matches a common Blog post structure that includes a date and post name, as shown in **Figure 2**, and drop in one of my weblog posts along with some of its related image resources.

That's all that's needed, and I can now access this page with the following URL (it's broken for layout reasons):

```
http://localhost:59805/posts/2018/03/23/_
  MarkdownTagHelper.md
```

Or the extensionless version:

```
http://localhost:59805/posts/2018/03/23/_
  MarkdownTagHelper
```

The default configuration works both with an .md extension or no extension. When no extension is specified, the middleware looks at each extensionless request, tries to append .md, and checks whether a file exists and renders it. The result is shown in **Figure 3.**

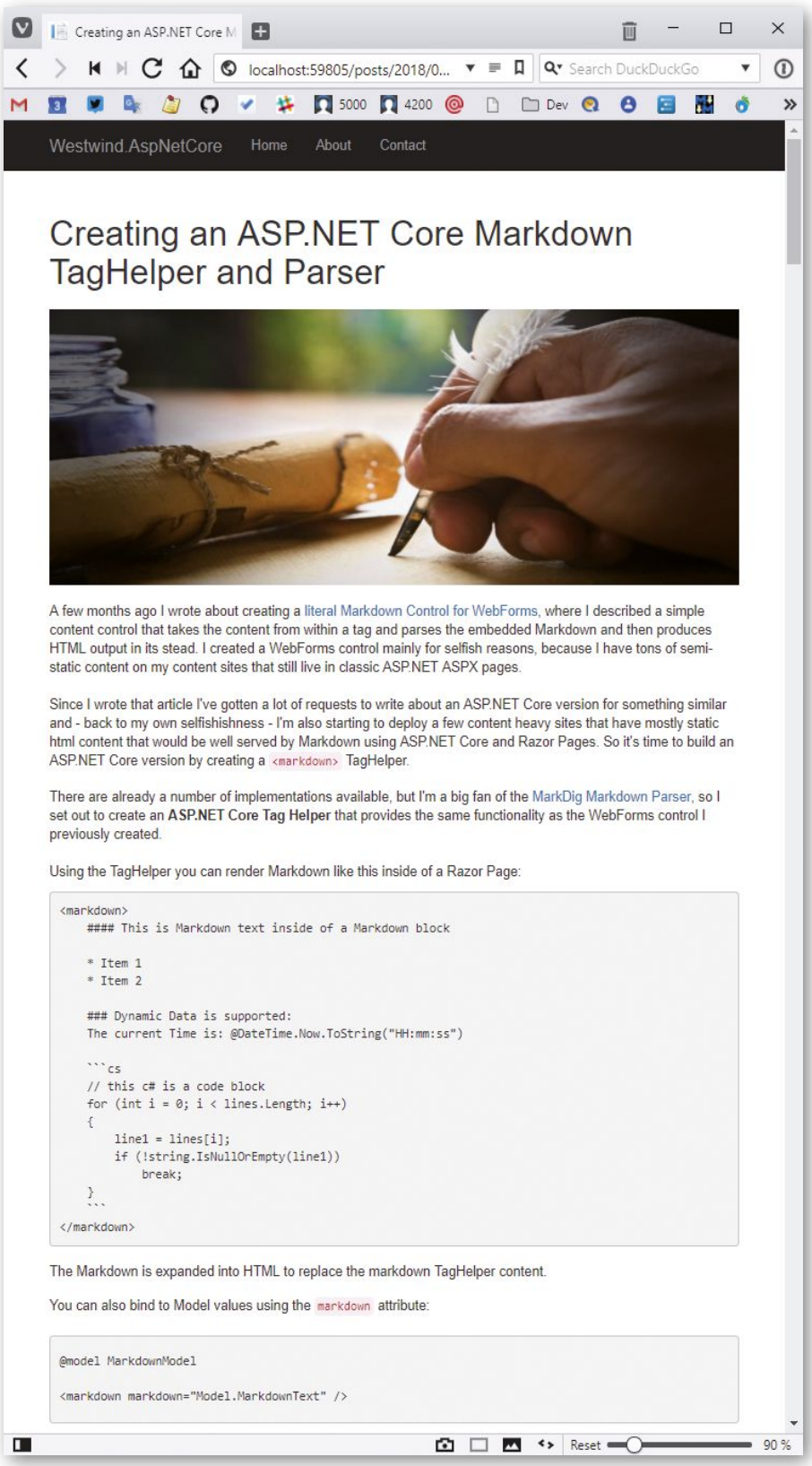


Figure 3: The rendered Markdown document in a stock ASP.NET Core site

The Markdown content is properly displayed as HTML and, as expected, I get the appropriate site UI from the stock ASP.NET Core Web site template. I haven't made any changes to the stock New Project template, but even so, this markdown text renders nicely into the page maintaining the site's UI context.

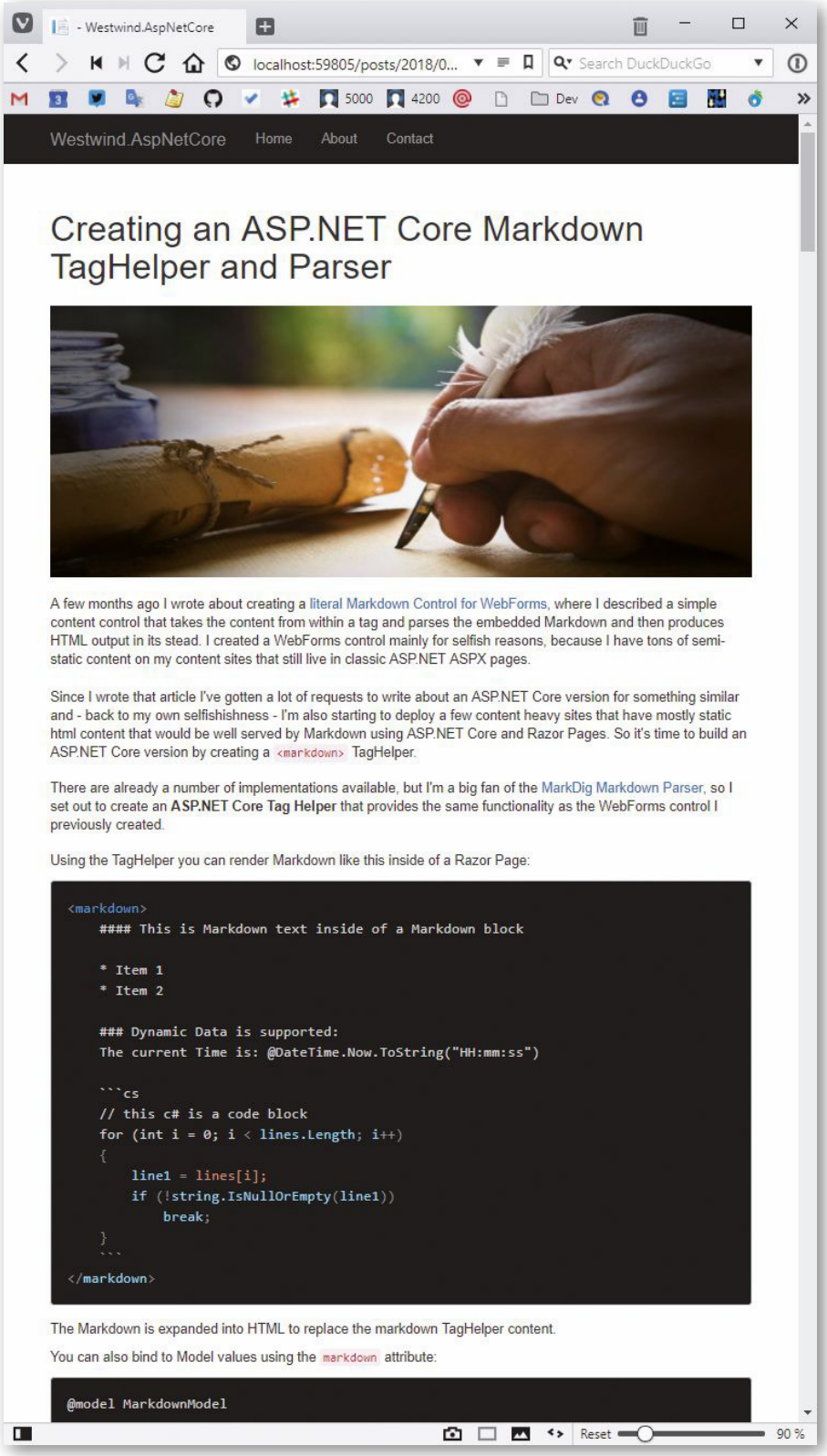


Figure 4: The new and improved page

More Cowbell

There's one thing that definitely could use improving: The sample code shows up in boring monochrome text rather than syntax highlighted text. To fix this, I can now modify the Razor template and add in some JavaScript to enable syntax coloring and add some Bootstrap styling fix-up to make it look nicer. **Listing 2** shows this updated template.

Figure 4 shows what the rendered page looks like now.

This code uses the **highlightJS** JavaScript library to format syntax-colored code. This library requires a link to the library, a theme (VS2015, which is a Visual Studio Code Dark-like theme) and a little bit of script that finds a **pre>code** elements in the page and applies syntax coloring. The rendered Markdown also includes a language attribute that the highlightJS understands and uses to pick the appropriate supported language. HighlightJS is customizable and you can create custom packages that include the languages that you care about. I've created a custom package that includes most common .NET, Windows, and Web-related languages that you can find in the library's GitHub repository.

At this point, I can just drop Markdown files into my **wwwroot/posts/** folder and they'll just render as self-contained pages. Sweet!

Creating the Markdown File Middleware

So how does all of this work? As you might expect, the process of creating this isn't very difficult, but it does involve quite a few moving pieces, as is standard when you're creating a piece of middleware.

Here's what is required:

- Middleware implementation to handle the request routing
- Middleware extensions that configure and hook up the middleware
- MVC Controller that handles the render request
- The Razor template used to render the rendered HTML

A Quick Review of Middleware

ASP.NET Core Middleware is a class that implements an **InvokeAsync(HttpContext context)** method. Alternately, Middleware can also be implemented directly in the **Startup** class or as part of a Middleware Extension using **app.Use()** or for terminating middleware using **app.Run()**.

The idea behind middleware is quite simple: You implement a middleware handler that receives a context object and calls a **next(context)** that passes the context forward to the next middleware defined in the chain, and it calls the next, and so on, until all the middleware components have been called. The chain reverses and each of those calls return their task status back up the chain. Effectively, middleware handlers can intercept inbound and outbound requests all with the same implementation. **Figure 5** illustrates the flow of this chained interaction of middleware components. Note that order of the pipeline is significant.

Listing 2: A Razor template with Syntax highlighting for HighlightJs

```
@model Westwind.AspNetCore.Markdown.MarkdownModel
@{
    ViewBag.Title = Model.Title;
    Layout = "_Layout";
}
@section Headers {
    <style>
        h3 {
            margin-top: 50px;
            padding-bottom: 10px;
            border-bottom: 1px solid #eee;
        }
        /* vs2015 theme specific*/
        pre {
            background: #1E1E1E;
            color: #eee;
            padding: 0.5em !important;
            overflow-x: auto;
            white-space: pre;
            word-break: normal;
            word-wrap: normal;
        }
        pre > code {
            white-space: pre;
        }
    </style>
}
<div style="margin-top: 40px;">
    @Model.RenderedMarkdown
</div>

@section Scripts {
   <script src="~/lib/highlightjs/highlight.pack.js"></script>
   <link href="~/lib/highlightjs/styles/vs2015.css" />
    <script>
        setTimeout(function () {
            var pres = document.querySelectorAll("pre>code");
            for (var i = 0; i < pres.length; i++) {
                hljs.highlightBlock(pres[i]);
            }
        });

    </script>
}
```

Listing 3: A middleware component to serve Markdown pages

```
public class MarkdownPageProcessorMiddleware
{
private readonly RequestDelegate _next;
private readonly MarkdownConfiguration _configuration;
private readonly IHostingEnvironment _env;

public MarkdownPageProcessorMiddleware(RequestDelegate next,
                      MarkdownConfiguration configuration,
                                  IHostingEnvironment env)
{
    _next = next;
    _configuration = configuration;
    _env = env;
}

public Task InvokeAsync(HttpContext context)
{
    var path = context.Request.Path.Value;
    if (path == null)
        return _next(context);

    bool hasExtension = !string.IsNullOrEmpty(Path.GetExtension(path));
    bool hasMdExtension = path.EndsWith(".md");
    bool isRoot = path == "/";
    bool processAsMarkdown = false;

    var basePath = _env.WebRootPath;
    var relativePath = path;
    relativePath = PathHelper
                      .NormalizePath(relativePath)
                      .Substring(1);
    var pageFile = Path.Combine(basePath, relativePath);

    // process any file WITH .md extension explicitly
    foreach (var folder in
              _configuration.MarkdownProcessingFolders)
    {
        if (!path.StartsWith(folder.RelativePath,
            StringComparison.InvariantCultureIgnoreCase))
            continue;

        if (isRoot && folder.RelativePath != "/")
            continue;

        if (context.Request.Path.Value.EndsWith(".md",
            StringComparison.InvariantCultureIgnoreCase))
        {
            processAsMarkdown = true;
        }
        else if (path.StartsWith(folder.RelativePath,
             StringComparison.InvariantCultureIgnoreCase) &&
             (folder.ProcessExtensionlessUrls && !hasExtension ||
              hasMdExtension && folder.ProcessMdFiles))
        {
            if (!hasExtension && Directory.Exists(pageFile))
                continue;

            if (!hasExtension)
                pageFile += ".md";

            if (!File.Exists(pageFile))
                continue;

            processAsMarkdown = true;
        }

        if (processAsMarkdown)
        {
            // push values we can pick up in the controller
            context.Items["MarkdownPath_PageFile"] = pageFile;
            context.Items["MarkdownPath_OriginalPath"] = path;
            context.Items["MarkdownPath_FolderConfiguration"] =
                                                    folder;

            // rewrite path to our controller
            context.Request.Path =
                          "/markdownprocessor/markdownpage";
            break;
        }
    }
    return _next(context);
} }
```

Listing 4: The Controller method that handles serving Markdown content

```
[Route("markdownprocessor/markdownpage")]
public async Task<IActionResult> MarkdownPage()
{
    var basePath = hostingEnvironment.WebRootPath;
    var relativePath =
                    HttpContext.Items["MarkdownPath_OriginalPath"]
                    as string;
    if (relativePath == null)
        return NotFound();

    var folderConfig =
            HttpContext.Items["MarkdownPath_FolderConfiguration"]
            as MarkdownProcessingFolder;
    var pageFile = HttpContext.Items["MarkdownPath_PageFile"]
                    as string;
    if (!System.IO.File.Exists(pageFile))
        return NotFound();

    // string markdown = await File.ReadAllTextAsync(pageFile);
    string markdown;
    using (var fs = new FileStream(pageFile, FileMode.Open,
                                   FileAccess.Read))
    using (StreamReader sr = new StreamReader(fs))
        markdown = await sr.ReadToEndAsync();

    if (string.IsNullOrEmpty(markdown))
        return NotFound();

    var model = ParseMarkdownToModel(markdown);
    model.RelativePath = relativePath;
    model.PhysicalPath = pageFile;

    if (folderConfig != null)
    {
        model.FolderConfiguration = folderConfig;
        folderConfig.PreProcess?.Invoke(model, this);
        return View(folderConfig.ViewTemplate, model);
    }

    return View(
        MarkdownConfiguration.DefaultMarkdownViewTemplate,
        model);
}
```

Listing 5: Configuring and hooking up the Middleware to the Pipeline

```
public static class MarkdownMiddlewareExtensions
{

public static IServiceCollection AddMarkdown(
          this IServiceCollection services,
           Action<MarkdownConfiguration> configAction = null)
{
    var config = new MarkdownConfiguration();

    configAction?.Invoke(config);

    if (config.ConfigureMarkdigPipeline != null)
        MarkdownParserMarkdig.ConfigurePipelineBuilder =
                              config.ConfigureMarkdigPipeline;
    config.MarkdownProcessingFolders =
        config.MarkdownProcessingFolders
            .OrderBy(f => f.RelativePath)
            .ToList();

    services.AddSingleton(config);

    return services;
}

public static IApplicationBuilder UseMarkdown(
                                      this IApplicationBuilder builder)
{
    return builder.UseMiddleware<MarkdownPageProcessorMiddleware>();
} }
```

So a pre-processing feature like Authentication or CORS needs to happen **before** a page serving feature like MVC or StaticFile servicing in order to work properly.

Implementing a dedicated middleware component usually involves creating the middleware component as well as a couple of middleware extensions used to configure and hook up the middle using **app.Add<Middleware>()** and **app.Use<middleware>()**, which is the pattern that most of ASP.NET Core's built-in middleware components use.

Implementing Markdown Page Handling as Middleware

The primary job of the Markdown page handling middleware is to figure out whether an incoming request is asking for a Markdown document by checking the URL. If the request points at an **.md** Markdown file, the middleware effectively rewrites the request URL and routes it to a custom, well-known Controller endpoint that's provided as part of this component library. **Listing 3** shows what the middleware looks like.

The key in this middleware is **Context.Path.Value**, which holds the current request path. Based on this path, the component checks to see if it points to either the **.md** file directly, or if the URL is an extensionless URL to which it adds the **.md** extension and checks for the file.

> Rewriting a request path in middleware in ASP.NET Core is as easy as changing the HttpContext.Path property.

If the path points at a Markdown file, the middleware sets a flag, and stores the original path and the path to the file into a few Context items. More importantly, it rewrites the current path to point at a well-known **MarkdownPageProcessorController** that has a fixed route of **/markdownprocessor/markdownpage**.

The Generic Markdown Controller

This effectively reroutes the request to my custom controller, which can then render the physical file's content using the configured Razor template.

```
[Route("markdownprocessor/markdownpage")]
public async Task<IActionResult> MarkdownPage()
```

This hard-coded Attribute Route is found even though it lives in a separate NuGet-provided library. Note that this route only works in combination with the middleware because it depends on preset Context.Items values that were stored by the middleware earlier in the request. **Listing 4** shows the action method that's responsible for serving the Markdown file.

The code starts by first picking up the Context variables that were set in the middleware when the request was forwarded, checking for the file, and, if found, reading it from disk and rendering it to HTML. A model is created and built up with the important title and rendered HTML that are used directly by the view, along with some of the file information and configuration information that can be used either inside the view or in the pre-process hook. When it's all said and done, the model is sent to the view for rendering.

Middleware Extensions

Once you've created a middleware component, it still needs to be hooked up and added to the pipeline. ASP.NET Core provides generic functions for adding typed middleware, but these functions aren't easily discoverable. A better way is to do what the native middleware components do, which is providing extension methods that extend the **IServiceCollection** and **IApplicationBuilder. Listing 5** shows the code for the extensions.

The **AddMarkdown()** method provides the service configuration by creating a default configuration object, and then taking an optional **Action<MarkdownConfiguration>()** that you provide to configure the middleware. This is a common pattern which delay-invokes the configuration when the first request comes in. The method's final action is to explicitly add the configuration to the Dependency Injection container so that it can be retrieved in the middleware and the controller via Dependency injection.

UseMarkdown() is very simple and simply delegates to **builder.UseMiddleware<MarkdownPageProcessor>()**. The sole purpose of the wrapper method is to provide a discoverable method in the **Startup.Configure()** code that's consistent with how ASP.NET Core built in middleware behaves.

Et Voila!

I now have a totally reusable Markdown page-rendering engine that can be easily plugged into **any** ASP.NET Core application with a few lines of configuration code. From then on, I can just drop Markdown files and related resources into my **wwwroot** folder and I'm good to go.

In this article, I've shown you a number of different approaches that you can now use for getting Markdown into your applications. Whether you need to render inline Markdown from your application's data using the **Markdown.Parse()** and **@Markdown.ParseHtml()** methods, whether you need to create a few small Markdown islands inside of existing pages, or whether you want to create free-standing Markdown files that are served as if they were static pages, you're covered with all of these approaches.

So, go ahead. Markdown all the things!

Rick Strahl

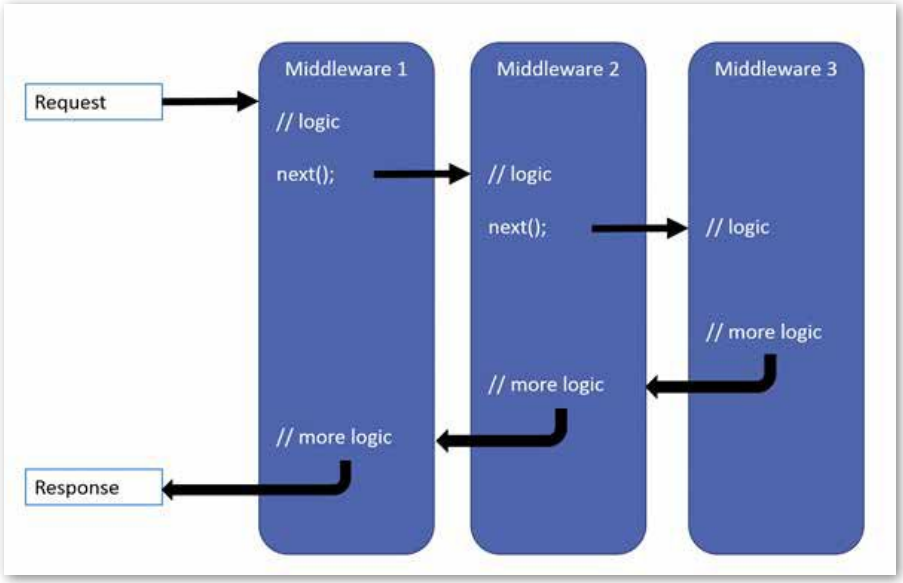


Figure 5: Middleware components plug into inbound and outbound request processing

Java

With the thaw in the Cold War that erupted between Sun and Microsoft in 2000, and Microsoft's embracing Java as a tool for building applications on top of Azure, attitudes in the .NET-leaning camp have changed somewhat significantly toward Java. When combined with the simple fact that Java has been a staple of the enterprise world for close to two decades,

Ted Neward

Ted Neward is the CTO of iTrellis, a software development consultancy located in Seattle. He's an internationally recognized speaker, instructor, consultant, and mentor and spends a great deal of time these days focusing on languages and execution engines like the JVM and CLR.

it seemed time to pull out the ol' language passport, dust it off, and wander on over to the land of Java for a while.

Before I get too deep into this, however, let's be clear about one thing; given the Java language's syntactic and semantic similarity to C#, the purpose of this article is not to spend a ton of time on the Java language itself. As a progenitor and influence on C#'s design, as well as an extremely popular (in terms of books and articles) language, it seems a reasonable assumption that most readers of this magazine have some degree of familiarity with the language already; barring that, Java and C# are so similar, even readers who've never set one foot in the Java world will find the language easy to pick up. (As one friend of mine put it, "Java's just like C#... if you took all the interesting parts out.")

What also needs to be clear is that the Java ecosystem is easily as large (or larger, by some measurements) as the .NET one, and to try to cover the entirety of the Java ecosystem in one article is laughably impossible. The goal here is to get some major points laid down: Get Java onto your system, build some simple applications to show off a bit of the language, and then build a simple HTTP API endpoint using Spring, a deeply popular open-source set of frameworks and associated tools that have been a staple in "enterprise" Java development for over a decade. (Spring currently is owned and maintained by Pivotal, the cloud company that recently joined the .NET Foundation.)

Along the way, I'll talk about some of the Java platform's history. More so than any other contemporary language/platform, and thanks to the complete lack of a strong central benevolent dictator figure (like Guido for Python, Matz for Ruby, or Microsoft for .NET), Java wasn't "designed" as much as it "evolved" into what you see today, making it largely defined by its history: Decisions that today seem strange and almost irresponsible look far different when viewed from the timeframe in which they were made.

Most of all, however, the goal here is the same as any week-long excursion within a foreign land: Get a taste of the culture, hit the tourist highlights, maybe pick up a tacky T-shirt or two, and head home exhausted, making promises to return for a longer visit next time.

Shall we?

Installing

The first step toward "writing once, running anywhere" is to get a Java Development Kit installed on your system of choice.

Historically, the best way to do this was to download the JDK from Sun (now Oracle); however, thanks to some odd choices on Sun's part, that's never been an open-source friendly option. In more recent years, an effort to create an open-source-friendly version of Java has yielded an almost-the-same-thing-JDK called the OpenJDK. For the most part, these two implementations are identical, the differences lying in those parts of the Sun/Oracle JDK codebase that still use non-OSS-licensed code (mostly fonts and audio/video codecs). If the open-source distinction matters, or if you're not really sure why the license should matter and you just want to explore and play, pull down an OpenJDK installation from http://jdk.java.net/ and unpack. Otherwise, wander on up to the Oracle Java website (http://www.oracle.com/technetwork/java/javase/downloads/index.html) and download the installer for your OS of choice. As of this writing, the latest release version of Java is Java 10, with Java 11 under development, but that brings up the first little diversion.

In the early days, Java and the JDK had some slightly different naming conventions; the first version of Java was called, of course, 1.0, and the one after it became 1.1. After that, however, the next version was called Java 2, owing to a hasty Sun Marketing rebrand; scuttlebutt suggests that the name was supposed to be Java 2000, to compete with the forthcoming Windows 2000, but Sun employees reacted so negatively that marketing trimmed off the last three zeroes to call it Java 2. Three different editions of Java bear that Java2 nomenclature as Sun started to diversify Java from the core bits (Java2 Standard Edition, or J2SE) into the enterprise (Java2 Enterprise Edition, or J2EE) and mobile (Java2 Mobile Edition or J2ME). Unfortunately, this created a ton of confusion for the next decade, as each version of the Java2 software development kit was given its own number: the initial release of the Java2 SDK was called J2SE 1.2; the one after that, J2SE 1.3, and then J2SE 1.4. It wasn't until JDK 1.5 that the name was finally bumped to Java 5, but the JDKs still had their own version numbers; Java 5 was JDK 1.5, and then JDK 1.6, then Java7, Java8, and so on.

Meanwhile, the 2 was dropped from the various names, so Java2 Enterprise Edition became simply Java Enterprise Edition, or JEE; habit being a hard thing to break, however, many Java developers continue to call it J2EE, so don't be too surprised if you run across that name while out here in the Java islands. Complicating matters even further, J2EE is essentially nothing more than a pile of specifications to which J2EE-compliant implementations conform, and, just to complete the hat trick, J2EE and JavaEE versions were done separately from JavaSE, so the current version of JavaEE is JavaEE 8, even though JavaSE is at version 10.

Why all this versioning nonsense? In part, it was because Sun was trying very hard (some say too hard) to set up an ecosystem in which everybody had a level playing field, and partly because Sun was a company run by engineers. At the time, the distinction between JavaSE and JavaEE was touted as a strength—compared against .NET in particular and its monolithic installation practices—but the

price has been a little steep for those getting started. Simply think of it as a charming anachronism carried over from the good old days.

If you are fond of package managers, there are a couple of options, depending on your operating system of choice. For either macOS or Linux, use Homebrew or the Linux package manager (apt on Ubuntu, for example) to install the OpenJDK, because that's "open-source friendly" and doesn't require manual acceptance of the Oracle license agreement. For Windows, there are several options, including Scoop (https://scoop.sh/) or Chocolatey (https://chocolatey.org/), but my current favorite option is to run SDKMAN (for SDK Manager) under a Bash shell in the Windows Subsystem for Linux (WSL) on Windows 10. Getting WSL up and running is a little out of scope to this article; see (https://docs.microsoft.com/en-us/windows/wsl/install-win10) for details.

Once WSL is up and running, fire up a Bash shell, visit the SDKMAN website (https://sdkman.io/), run the indicated command in the Bash shell and use SDKMAN to install Java. SDKMAN also works quite well inside of a Cygwin shell, or MinGW, if you're a fan of those tools. Or, if you really just don't want to put anything extra on your computer, there's running Java in a Docker container (docker run -it openjdk:10, for example), or even creating a virtual machine in the cloud; both Amazon and Microsoft have Java-based cloud virtual machine images that are easily accessible, installable, and runnable within their respective cloud environments.

Note that any of the JDK installations only contain command-line tools (javac, java, jar, etc.). It has been a staple practice of the Java ecosystem for two decades now that the tools should remain entirely command-line, and any sort of extra functionality (like, you know, an editor) should be the developers' personal choice. In fact, this is a common theme throughout the Java ecosystem: a marked preference for standards and conventions, rather than an umbrella installation including IDE, project templates, and so on (such as what you get when you install Visual Studio). Whether this is a pro or a con is certainly open to debate (hint: it's both), but if you're looking for the closest to the Visual Studio experience, you'll want to visit the JetBrains website and download IntelliJ IDEA, Community Edition (https://www.jetbrains.com/idea/download/) and install that. Note that many of the Java tools and environments—particularly when you get closer toward deploying code—expect a command-line environment, or at least familiarity with where the various JDK pieces are installed but installing the IDE can be an easier way to get up-and-running quickly.

Hello, Java

Once Java is installed, you should be able to run the Java compiler from a command-line terminal of your choice with:

```
javac --version
```

It responds with the latest version of Java. In many cases, particularly on Windows, it may be necessary to add the **bin** directory of the JDK installation to the PATH, but before doing that the simple way, take note: Many Java tools (which are also often command-line driven) expect an environment variable, **JAVA_HOME**, to be defined and pointing to the root of the JDK. Thus, it's often a better idea to set up a simple batch file that defines JAVA_HOME, then adds Java to the PATH by use of JAVA_HOME, like so:

```
@ECHO OFF
title JavaSE10 Prompt
set JAVA_HOME=C:\Prg\openjdk10
set PATH=%JAVA_HOME%\bin;%PATH%
java -version
```

This way, you have complete control over the environment variables. Note that on Windows, if you use the Oracle installer, historically it installed two things on your system: the Java Runtime Environment (JRE), which is all the bits necessary to run a Java application, and the JDK, which is all the bits necessary to build a Java app. The Oracle installer puts the JRE under **C:\Program Files**, and that's often what's installed on the PATH for you, which means you can run Java, but you can't build it. Taking an extra second to make sure you know where things are installed and what's in what location can save a ton of time later.

Basics

Let's take a second to play with the command-line tooling. Create a **project home** directory somewhere on your filesystem, call it **hello**, and put a subdirectory called **src** in it. In the src directory, create a file **Hello.java**, and put the following into it:

```
package com.newardassociates.demo;

public class Hello {
    public static void main(String... args) {
        System.out.println("Hello, world!");
    }
}
```

You compile this using the **javac** utility, so while in the **hello** directory, enter the following into your terminal window of choice:

```
javac -d classes src/*.java
```

The **-d** parameter indicates a target subdirectory into which the compiled code will go; unlike .NET, Java doesn't create a single output artifact. Instead, owing to almost two decades' worth of history, Java produces a collection of **.class** files, on a (nearly) one-to-one basis with the source files. However—and this is where things get really interesting or awkward, depending on your point of view—the generated class files are placed into subdirectories corresponding to the **package** declaration at the top of the source file. Because the Hello code is in the **com.newardassociates.demo** package, the **classes** directory contains a subdirectory called **com**, which contains **newardassociates**, which contains **demo**, which contains the **Hello.class** file. For this reason, source files typically follow the same directory convention; this means that were this a traditional Java project, the Hello.java file would've been in **src/com/newardassociates/demo**, in direct correlation to the package statement.

To run this, you need to point Java at the location of the compiled code, commonly known as the **class path**. In the earliest releases of Java, this was defined using the environment variable **CLASSPATH** but this quickly became

untenable, and instead you choose to pass it in when you run the code. Running Java code is done using the **java** launcher, which bootstraps the JVM into memory and then runs the **main** method of the class name (not the file name) passed in at the command-line, like so:

```
java -classpath classes com.newardassociates.demo.Hello
```

[Note: Due to the constraints of the printed page, many lines of code that aren't normally broken might be broken in odd places in this article.]

This, of course, produces the traditional greeting to the console. Take note, again, that the entry point is specified by the classname passed on the command-line; for a time, it was common for every Java class to have its own **main** method, which was used to unit-test the code in that class until more sophisticated unit-testing tools (like JUnit) came along.

The deployment story here is a mess—copying a whole directory tree of .class files around. For this reason, Java developed the JAR (Java ARchive) file as a single-file artifact. Creating one requires the use of the **jar** command-line tool, and a **manifest** file to specify the entry point (the **Main-Class**). Assuming the **manifest.txt** file contains the following name/value pair:

```
Main-Class: com.newardassociates.demo.Hello
```

Then you run the following from the command-line:

```
jar -c -v -m manifest.txt -f hello.jar -C classes/ .
```

This produces the **hello.jar** file, which you can then run from the command-line using Java's **-jar** parameter:

```
java -jar hello.jar
```

This produces the expected greeting.

Build Tools

If all this sounds ridiculously complicated, welcome to the world of Java in 1997. In the beginning, when Java was first thought to be something for building embedded systems (such as the cable set-top box for which it was originally designed), the whole process was relatively acceptable, but as Java moved into the world of the Web, this was obviously not going to fly for long.

> If all this sounds ridiculously complicated, welcome to the world of Java in 1997.

Thus, it should've been no surprise when the nascent Java open-source community created tools to solve this problem.

The original tool was called Ant (short for Another Nifty Tool), built to make it easier to build Tomcat (which I'll get to in a second), and it used XML as a descriptor format to describe the structure of the project and specify which steps should be run in which order and which depended on each other. (Remember, back in the late 90s, we were all excited about XML for, well, just about everything.) The originator of Ant, James Duncan Davidson, has since admitted that the main reason for using XML was to save himself the hassle of writing a custom parser. Fortunately, Ant came with a ton of plug-ins that could be repurposed, and the build-tool explosion had begun.

The next build tool, Maven, went one step better than Ant in that it created on online repository of build artifacts, including both libraries and stereotypes (project templates) that could be downloaded automatically rather than by hand. The Maven build structure was all XML again, and if anything, it was more verbose than Ant's. Around this time, some of the alternative languages for the JVM (Groovy, Scala, Clojure, to name a few) began to emerge, and each began to build their own build tools. One, incorporating Groovy into the build language itself, called Gradle, has built a following, and has since been adopted by Google to be the build tool of choice for building Android applications.

If you drop a Gradle file to do all that I've described (compile source, assemble into a JAR file) into the project root directory, it looks like this:

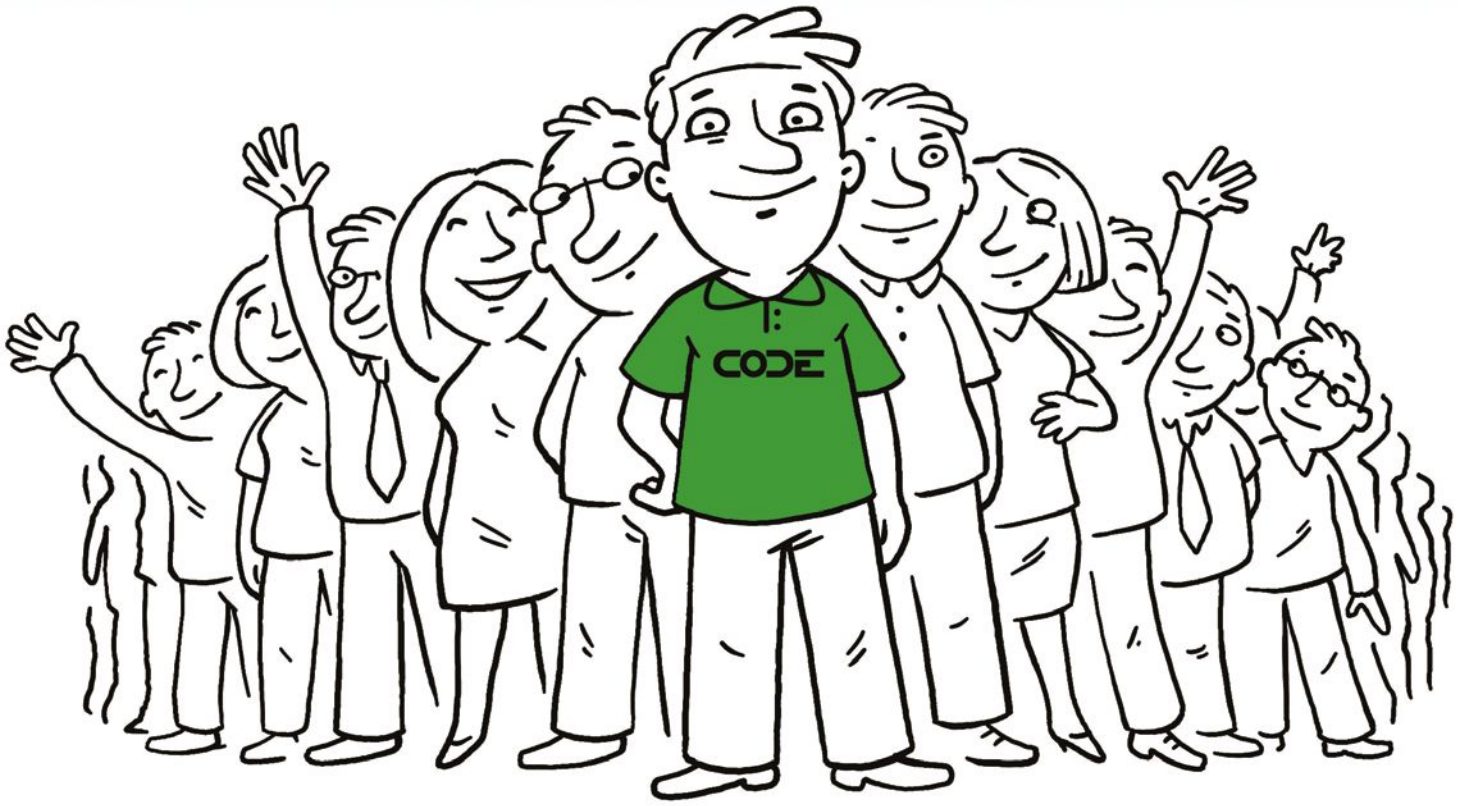
```
plugins {
    id 'java'
}

group 'com.newardassociates.demo'
version '1.0-SNAPSHOT'
jar {
    manifest {
        attributes 'Main-Class':
            'com.newardassociates.demo.Hello'
    }
}

sourceCompatibility = 1.8

repositories {
    mavenCentral()
}
```

By running **gradle jar**, all of the necessary dependencies (if any) are downloaded from Maven Central (the online repository of Java libraries), the code compiled, and the JAR file assembled in the **build/libs** directory off the project home. If you're following this at home, by the way, you may need to download Gradle before this will work; package managers make this easy, so (for example) in SDKMAN grabbing Gradle is one command: **sdk install gradle**. Once installed, however, Gradle can generate a **wrapper** script that bootstraps the Gradle download so any future developers won't need to do that initial step.

Enterprise

What put Java on the map wasn't its impressive array of build tools but its place in the enterprise. Widely regarded to be the king of the enterprise today, Java certainly didn't start that way. It largely earned that distinction because it made building Web applications easier than its competitors of the time—Microsoft ASP, ColdFusion,

Qualified, Professional Staffing When You Need It



CODE Staffing provides professional software developers to augment your development team, using your technological requirements and goals. Whether on-site or remote, we match our extensive network of developers to your specific requirements. With CODE Staffing, you not only get the resources you need, you also gain a direct pipeline to our entire team of CODE experts for questions and advice. Trust our proven vetting process, and let us put our CODE Developer Network to work, for you!

Contact CODE Staffing today for your free Needs Analysis.

Helping Companies Build Better Software Since 1993

www.codemag.com/staffing
832-717-4445 ext. 9 ● info@codemag.com

CODE STAFFING

or (shudder) CGI-bin Perl scripts—and had better access to the relational databases that define the core of what it means to be enterprise.

Bear with me for a moment while I fast-forward through about a decade of history.

Databases

For data access, the engineers at Sun realized that getting Java to talk to a relational database was clearly a high priority, and as far back as Java 1.1, they introduced a standard API for accessing the RDBMS called JDBC. In many ways, JDBC was constructed in the same manner as its Microsoft cousin ODBC, in that JDBC itself only specified interfaces, and driver vendors provided implementations of those interfaces that knew how to speak to the RDBMS in question. One of the first provided drivers was a reference implementation that allowed JDBC to make use of an ODBC driver, which effectively allowed Sun to bootstrap the enterprise Java story, because almost every database then had an ODBC driver, thereby allowing Java developers to access data. Because the JDBC-ODBC driver wasn't all that fast, it wasn't long before database vendors were peppered with requests for better drivers, and before long, all the major database vendors had JDBC drivers available, sans one: Microsoft SQL Server. (Go figure.)

The JDBC story, however, was the first successful salvo in vendor neutrality**.** This was the idea that Java developers could write code that was somehow vendor-agnostic so that a developer's code would never be "hostage" to a (database) vendor seeking to exploit the cost of rewriting code by moving to a new vendor. Should Oracle show up one day and demand a million (new) dollars just to hook their database up to the Internet (true story), Java developers could simply pivot the JDBC driver they were using and start using MySQL. Or Postgres. Or any of a dozen other options (in theory). It was largely for this reason that Java developers constantly eschew the use of stored procedures in the database, regardless of whatever benefits might be had from doing so.

Naturally, over time, developers started experimenting with how to represent objects in a relational database, with one of the more popular efforts being the Hibernate project. Various groups sought to standardize the object-relational mapping effort, and eventually settled in on a specification called JPA (Java Persistence API), although it's nowhere close to being dominant. Several different data access layers—all of which use JDBC as their principal means to connect to the database in question—are all currently in widespread use.

But where Java truly solidified its hold in the enterprise was on the Web.

Servlets

The core of the Web story rested on an idea that Sun called servlets—Java classes that could "plug in" to a Web server, respond to an HTTP request, and hand back a formatted response. This wasn't revolutionary, per se, but Java offered a nice sweet spot between performance and development speed, and the Java language syntax was more approachable to the C++ developers to whom it was pitched. On the backs of this humble beginning was Java's enterprise empire built.

Servlets are designed to run inside servlet containers, which originally meant a plug-in to an existing commercial Web server like IIS or Apache. Sun needed to produce a reference implementation of the servlet specification and quickly built one that was code-named Tomcat. Over time, as people began to realize that a Web server really wasn't all that big an implementation problem to solve, the roles flipped, until Tomcat became one of the open-source Web servers of choice and the staple of Java enterprise server farms.

One important development in the Java Web space was the growth of several open-source Web presentation-layer technologies, including the later-standardized Java Server Pages (JSP), and the growth of Model 2 applications, later called MVC: a servlet took an incoming request and processed it (Controller), interacting with a set of domain objects that usually knew how to be persisted to a database (Model), and then the request was forwarded to a JSP page to display the results (View).

Naturally—this is the Java space and there's always a competing opinion—several other flavors of MVC-based frameworks emerged, such as Struts, Tapestry, and a few others to boot. Although there was never anything by way of a consensus, the JSP/servlet combination powered a great number of applications for quite some time. More importantly, all of these competing frameworks could all be deployed into the same servlet container because at their heart, they all rested on top of the servlet specification, making Tomcat a common server of choice and simplifying the deployment story by standardizing on a WAR (Web ARchive) file format for deploying Java Web applications. By creating a common infrastructure story, the Java community ensured that opinions could diverge quite a bit without having to start over on the infrastructure, servers, and tools.

The servlet specification itself is pretty simple: Extend a base class, override one of the various HTTP-verb methods (doGet, doPost, and so on), each of which take a request object and a response object, and write the results to the response object's buffer, and off you go. In and of itself, it's not particularly interesting, or, to be more accurate about that, not particularly interesting to those who've seen ASP.NET or Node's ExpressJS framework. (In many ways, the design of the servlet concept stood as a template for others to follow—and improve.) For those who're seriously interested in seeing what a Hello World servlet looks like, have a look at the Tomcat documentation, in the section labeled "Application Developer's Guide" (https://tomcat.apache.org/tomcat-9.0-doc/appdev/index.html).

What's important to realize from the Web discussion is that Java grew to embrace the idea of standards over implementations, at least to start. Over time, as we struggled to build enterprise applications, we discovered that embracing standards over implementations carried with it a development-time cost too high for the corresponding benefits (if any) that doing so granted us. What was worse, applications written to the standards often weren't nearly as implementation-portable as we'd thought they would be, leaving us to drown in a sea of complexity for little added benefit.

To a lot of developers, winter wasn't just coming, it was here. Enter Spring.

Spring

Originally, the Spring framework was a singular framework, designed to act as a container of other objects, called bean (as in, Java beans, get it? No, really, that's the truth, that's why they're called that. Don't hate the player, hate the game). Via some simple method interception tricks, the Spring container could provide certain kinds of behavior as calls came in to the beans and went back out again, similar in concept to what the commercial implementations of EJB were doing, but with far fewer restrictions and with far better portability. In essence, developers wrote their code to be Spring beans, and the Spring developers worried about how Spring would live inside of a commercial container.

(It was around this timeframe that aspects and aspect-oriented programming, or AOP, took the Java world by storm, as that was the heart of what and how Spring was able to provide much of its behavior, and Spring doubled down on AOP when they acquired the principal AOP language, AspectJ. Although some of the aspect-oriented mentality would later reach the .NET community, aspects were never as widely adopted in .NET as they were in Java. Whether this was a good thing or a bad thing is still hotly debated within both communities.)

Spring quickly established itself as something of the *de facto* standard for Java. The principal maintainers of the Spring codebase quickly formed a consulting company (SpringSource) and began working on a number of different related frameworks, until Spring reached a level of complexity on par with the very JavaEE standards it sought to replace. The core Spring framework begat SpringMVC (an MVC framework on top of servlets), SpringData (for doing data access), SpringBatch (for batch-style processing, typically asynchronous in nature), SpringSecurity (just what it sounds like), and so on.

As Spring got more complicated, configuring a Spring application became even more so. Trying to track which libraries and which tools were in use or were necessary on compilation command-lines became almost as much of a nightmare as using the original JavaEE stack. The Spring team, therefore, decided to look for a way to get all of Spring under one (conceptual) umbrella, and the Spring Boot project was born, and (as of this writing) serves as the best way to get started using Spring.

Spring Boot

Let's assume that your interest is in getting a Spring Boot project up and running, and because this is a greenfield project effort, you want to build an HTTP API endpoint that will do the classic HTTP API endpoint thing: provide CRUD access in the form of JSON input and output against a relational database. (This is, largely, the hello world example of the enterprise world, but it's a useful way to get started working with any enterprise full-stack Web stack, whether Java's or anybody else's. Once this works, it's usually not difficult to customize and expand a server-side API from here.)

To begin, you'd like to File|New a project, but Spring has a different way to get that started; rather than try to maintain three separate IDE project templates (one for IntelliJ IDEA, one for Eclipse, and one for NetBeans), the Spring team decided to go with a slightly different approach, one that they call Spring Initializr. (Yes, the "e" is missing.) This is an online service that can be used to help create an initial project template with all the dependencies already in place, generating a build script that will mostly never need to be touched again once configured, assuming all the dependencies are known ahead of time.

Using Spring Initializr takes one of two forms: either install the Spring CLI via the package manager (sdk install springboot) or head up to the website http://start.spring.io and fill in the missing fields. Because the website makes it a little easier to visualize, use it; if you're not already looking at it in a browser, bring it up.

To keep things simple (or at least as simple as you can, anyway), choose to build a Gradle Project, with Java, and using Spring Boot 2.0.3 (which, as of this writing, is the latest non-beta version). The Project Metadata is critical to making sure configuration values and source code gets defined and declared correctly; the Group is the package prefix you want to use for all the source code generated, and typically it's a reverse-DNS name, like com.newardassociates or something similar. The Artifact is going to be the name of the application itself, which in this case, let's call helloworld.

Note that the Dependencies section is just a text-edit field, which is great if you already know what you need to add but doesn't help much if you're not sure what the options are. Fortunately, underneath the big green Generate Project button (don't push it yet!), there's a hyperlink to Switch to the full version, and clicking it yields a long list of all the possible starter dependencies for Spring Boot for easy selection.

It's a really, really long list.

Before you finish with Spring Initializr, take a look at what's available. There's support for just about every database you can imagine (plus a few you haven't heard of before), both relational and non-relational, there's a variety of different Web framework support (depending on whether you want a traditional or reactive style app), there's support for four different templating engines (for defining views used in a Web app), and there's some cloud support—including easy Azure support, as well as AWS or GCP.

I can't explore them all in these pages, so let's stick with the basics for now: check Web (for basic Web support, which includes JSON), JDBC (so you can make use of a JDBC driver to access relational data) and H2 (which is an embedded database written in Java, frequently used for demos because it requires pretty much zero initialization). Click the Generate Project button, and Spring Initializr sends you a helloworld.zip file containing the scaffolding for this new Web app.

Once it's finished downloading, explode the ZIP file somewhere on your hard drive and take a look at what's inside.

Project Contents

To start with, you'll see a build.gradle file, which, if you'll recall, is the core build file using the Gradle build

tool. It's actually not that much more complex than the one I showed you earlier, but in truth, you don't really need to look at it; in addition to masking and hiding a whole bunch of options you don't need to worry about, it doesn't list all of the possible things you can do with it. The best way to see that is to use Gradle to list all of the tasks it can find (both explicitly and implicitly defined), which is done by running gradle tasks.

(Notice the presence of the gradle.bat and gradle shell script files? Those are the wrapper scripts I mentioned earlier, which pull down and install Gradle onto your computer if it's not already present. These should never be modified by hand but should be checked in to source control simply on the grounds that it helps new developers get up-and-running more quickly.)

The gradle tasks output lists about a dozen or so possible options (and that's not the full list; run gradle tasks --all if you really want to be overwhelmed), but the one you care most about is the one at the very top of the list, called bootRun. (Note that there's also a bootJar option; that's different.) The bootRun task is particular to Spring Boot and runs the Spring Boot application locally; in other words, this is our F5 option outside of the IDE.

The rest of this directory is mostly configuration and your principal concern lies elsewhere, in the src directory. Within that, you see an immediate split between main and test; as might be guessed, the test directory is where unit-test code should reside, so that the build can compile and execute—but not include in the final deployment artifact—the test code for this project.

Unit-testing a Java project is generally considered a bare-minimum-hygiene sort of practice, and unit-testing frameworks in Java are similar enough to their .NET cousins that if you're a unit-test aficionado, you'll have no problem adjusting; if you aren't, then you probably wouldn't be convinced to start here, either. (And you should feel bad. Like, really, really bad.)

Within each of the main and test directories lies one more level of distinction, and that's the language in question in which the source is written. Because you chose a Java project to be generated, main will have a java subdirectory; had you chosen Groovy or Kotlin (two other languages for the JVM), those directories would have appeared here. In many respects, the Java community has done an excellent job of reconciling different languages to first-class-citizen status on the JVM, and Spring wants to be equally accessible to any of them, even inside the same project.

Sitting next to the Java directory is another one called Resources. In here, you can drop non-code assets, like configuration files, graphics files, or other resources. Spring Initializr has already created two placeholder (empty) directories within resources, one for static assets and another for templates (which are usually views). In the root of the Resources directory, you'll find an application.properties file, which is the standard base configuration file for this application.

(Another bit of history: Since Java 1.0, Java has had a Properties class, extending the Hashtable class, that knows how to read and write itself from and to disk. It's been the staple for Java configuration since that time, and by convention, those text files are always suffixed as .properties.)

Keep a virtual thumb on the application.properties file; you'll be coming back to it when you add a few configuration options later. For now, though, let's get some code going—just a simple HTTP endpoint that hands back an incrementing counter and says (what else?) "hello" to whomever is invoking the endpoint.

Code

In src/main/java/com/newardassociates/helloworld/HelloworldApplication.java, there's a main() method that turns right around and asks Spring to execute the application. This is largely a placeholder for when you want to run the Spring app with an embedded HTTP server, as opposed to building the app to run inside of a standalone Web server (such as Tomcat). To keep things simple, run it standalone, but if you ever want to experiment with Tomcat, it's pretty trivial to get Gradle to produce a WAR file that can be just dropped in to a running Tomcat server and deployed.

You need to define an HTTP endpoint, so in the same directory, create a new file, GreetingController.java, and fill it with the following:

```
package com.newardassociates.helloworld;

import java.util.concurrent.atomic.AtomicLong;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class GreetingController {

    private static final String template =
        "Hello, %s!";
    private final AtomicLong counter =
        new AtomicLong();

    @RequestMapping("/greeting")
    public String greeting(@RequestParam(value="name",
            defaultValue="World") String name) {
        return String.format(
            template, counter.incrementAndGet());
    }
}
```

Much of this is conceptually similar to ASP.NET MVC (or similar frameworks from Ruby, Python or Node); you use Java annotations (the @-prefixed names) to provide some custom metadata that Spring consumes to know how to map this class against an HTTP endpoint (/greeting), in this case. Rather than holding the counter in a standard int, you protect against concurrency by using the java.util.concurrent.atomic.AtomicLong class, which guarantees thread-safety when incrementing a long value, even when being done simultaneously. (Only one instance of GreetingController is instantiated to process multiple simultaneous requests, so this is of some concern.)

Kick off gradle runBoot and Spring starts listening on port 8080. Note that your terminal window fills with a ton of log messages, many of the form **2018-07-25 00:19:16.800**

INFO 3373 --- [main] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [//favicon.ico] onto handler of type [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler]".** This is standard Java logging, and the o.s.w.s. that you see above is a shortened package name. Because Java packages can often nest five or six levels deep, logging frameworks often reduce the first parts of the full package name down to just the first letters. (The reverse-domain-name convention was a good idea for a while, but once it became commonplace to have packages nested eight levels deep, it lost some of its luster.)

The reverse-domain-name convention was a good idea for a while, but once it became commonplace to have packages nested eight levels deep, it lost some of its luster.

If you browse to http://localhost:8080/greeting, you'll see the generic greeting; if you tack **?name=Fred** onto the end of the URL, you'll see a custom greeting to Fred, and the counter increments on every invocation. Yay!

But this is still pretty trivial, and something that could've been pretty easily done with a simple Java servlet. Where does Spring add value?

Data Access: JDBC

As with most frameworks of its type, an application doesn't really benefit from the power of Spring until it reaches a certain level of complexity; simple things are usually simple to do and don't require any additional help. Data access—that is, retrieving and storing data in a database—is one area where Spring can help keep the cognitive load lighter, particularly as the application scales up.

As an example, let's assume that I'm deciding to take my college class with me on a trip to a foreign country, perhaps to a conference like Devoxx Poland (http://www.devoxx.pl, which is a great Java show, and that's before I even *begin* to talk about Polish vodka). I obviously need a database! I need to store student names and passport numbers, so that I can make sure that they all have passports and...stuff. (I'll make sure to send out a change to my privacy policy at the end of the trip.)

For starters, I'll need a domain class to track a Student; in Spring parlance, this is a bean, and it's a POJO: Plain Old Java Object. As you can tell from the code in the next snippet, Java lacks several language features that would've made this code shorter (like C# auto-generated properties, for example):

```
package com.newardassociates.helloworld;

public class Student {
    private Long id;
    private String name;
    private String passport;

    public Student() {
        this(0, "", "");
    }
    public Student(String name, String passport) {
        this(0, name, passport);
    }
    public Student(Long id, String name,
            String passport) {
        this.id = id;
        this.name = name;
        this.passport = passport;
    }

    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }

    public String getName() { return name; }
    public void setName(String name) {
        this.name = name; }

    public String getPassport() { return passport; }
    public void setPassport(String passport) {
        this.passport = passport; }

    @Override
    public String toString() {
        return String.format(
            "Student [id=%s, name=%s, passport=%s]",
            id, name, passport);
    }
}
```

Oh, that reminds me. You'll need a database schema to go along with this. Here's where Spring starts to earn its keep. Drop a file named schema.sql into src/main/resources, that looks like this:

```
create table student
(
   id integer not null,
   name varchar(255) not null,
   passport_number varchar(255) not null,
   primary key(id)
);
```

And then drop another file, data.sql, to prepopulate the database with some values, like this:

```
insert into student
values(10001,'Amy', 'E1234567');

insert into student
values(10002,'Ruby', 'A1234568');
```

Now re-run the application with gradle bootRun. Somewhere, in the middle of the log messages, you'll see that Spring picks up on the two SQL files as resources, and automatically executes them. (Because why else would they be there?) The log messages will look something like this:

```
2018-07-25 02:47:48.379  INFO 4304 --- [
main] o.s.jdbc.datasource.init.ScriptUtils     :
Executing SQL script from URL [file:/mnt/d/Projects/
helloworld/build/resources/main/schema.sql]
2018-07-25 02:47:48.388  INFO 4304 ---
```

Listing 1: The StudentRepository

```
package com.newardassociates.helloworld;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.BeanPropertyRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.stereotype.Repository;

@Repository
public class StudentRepository {
    @Autowired
    JdbcTemplate jdbcTemplate;
    class StudentRowMapper implements RowMapper<Student> {
        @Override
        public Student mapRow(ResultSet rs, int rowNum)
            throws SQLException {
            Student student = new Student();
            student.setId(rs.getLong("id"));
            student.setName(rs.getString("name"));
            student.setPassport(rs.getString("passport_number"));
            return student;
        }
    }
    public List<Student> findAll() {
        return jdbcTemplate.query("select * from student",
            new StudentRowMapper());
    }

    public Student findById(long id) {
        return jdbcTemplate.queryForObject(
            "select * from student where id=?",
            new Object[]{ id },
            new BeanPropertyRowMapper<Student> (Student.class));
    }

    public int deleteById(long id) {
        return jdbcTemplate.update("delete from student where id=?",
            new Object[]{ id });
    }
    public int insert(Student student) {
        return jdbcTemplate.update(
            "insert into student (id, name, passport_number) " +
            "values(?,  ?, ?)",
            new Object[] { student.getId(), student.getName(),
                student.getPassport() });
    }

    public int update(Student student) {
        return jdbcTemplate.update("update student " +
                                   "set name=?,passport_number=? " +
                                   "where id = ?",
            new Object[]{ student.getName(),
                student.getPassport(), student.getId() });
    }
}
```

```
[           main] o.s.jdbc.datasource.init.ScriptUtils
: Executed SQL script from URL [file:/mnt/d/Projects/
helloworld/build/resources/main/schema.sql] in 8 ms.
2018-07-25 02:47:48.390  INFO 4304 --- [
main] o.s.jdbc.datasource.init.ScriptUtils     :
Executing SQL script from URL [file:/mnt/d/Projects/
helloworld/build/resources/main/data.sql]
2018-07-25 02:47:48.391  INFO 4304 ---
[           main] o.s.jdbc.datasource.init.ScriptUtils
: Executed SQL script from URL [file:/mnt/d/Projects/
helloworld/build/resources/main/data.sql] in 1 ms.
```

Of course, a reasonable question to ask at this point is "With what database?!?" Such is the power of the H2 database. It's an in-memory, mostly-SQL-92 compliant relational database that frequently serves as the database of choice when doing demos and early prototyping.

As a matter of fact, quit the running app, then add the following to the contents of the application.properties file from earlier:

```
# Enabling H2 Console
spring.h2.console.enabled=true
```

The **#** character starts a comment and the line after it sets the spring.h2.console.enabled property to **true**. As the name implies, this turns on an in-memory console for interacting with the H2 database, which you can reach by pointing the browser to http://localhost:8080/h2-console . Behold! A tiny SQL console for interacting with the in-memory database. However, the console needs to know how to connect to the H2 database (there could be several), so you use the correct form of JDBC URL to describe the in-memory test database: jdbc:h2:mem:testdb.

This is, in all ways, shapes, and forms, a legitimate URL: jdbc is the scheme of the resource described, and will always be present in any JDBC URL, just as http or https is always present in any HTTP URL. The second portion describes which JDBC driver should be used to do the connection, and h2 corresponds to the H2 driver. (Microsoft SQL Server's driver descriptor is mssql or mssqlserver, MySQL's is mysql, and so on.) After that, the contents of the URL are entirely driver-specific, and in the case of H2, mem says it's an in-memory database (as opposed to being written to disk in plain text files), and testdb is the name of the database in question.

If you use the H2 console to connect to jdbc:h2:mem:testdb, it'll show the STUDENTS table in the left-hand drop-down, which proves not only that you're connected to the correct database, but also proves that Spring executed the DML in schema.sql from the resources. Nifty.

Accessing the database from the code should be pretty easy, too, right?

Data Access: Repository

In keeping with the Java philosophy that there should never only be one way to accomplish something, Spring offers several different persistence libraries to support whichever approach makes the most sense. Personally, I'm a fan of doing just straight SQL against the database, usually through a class designed to encapsulate details away (the Repository pattern), but this usually suffers from a certain amount of overhead: a Connection object has to be forged out of somewhere, and from there a Statement object has to be created with the SQL in question,

Listing 2: The StudentController

```
package com.newardassociates.helloworld;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class StudentController {
    @Autowired
    StudentRepository repository;

    @RequestMapping(value="/students", method=RequestMethod.GET)
    public List<Student> getAll() {
        return repository.findAll();
    }

    @RequestMapping("/students")
    public Student getOne(@RequestParam(value="id") Long id) {
        return repository.findById(id);
    }

    @RequestMapping("/students")
    public int delete(@RequestParam(value="id") Long id) {
        return repository.deleteById(id);
    }
}
```

then executed, then the results harvested out of the ResultSet that's returned, and so on. All of this is fraught with the knowledge that the database could (if the network gods are feeling capricious) disappear at any time and therefore you need to handle connection errors. That's a ton of code to have to write for each SQL statement.

Spring to the rescue! Spring provides a JDBC template class that can handle all of the non-schema-specific parts, so long as I supply the SQL to execute and a class designed to map columns out of the ResultSet into an object instance. Given that this is to fetch and store Student instances, call this class StudentRepository and sit it next to the other Java code, as shown in **Listing 1**.

Notice the placeholder question marks (**?**) in the SQL syntax. These are parameterized statements, and they're your first line of defense against SQL injection attacks. Not optional.

The thrust of the StudentRepository here is mostly providing domain-specific elements for the JdbcTemplate class. A StudentRowMapper understands how to map a SQL ResultSet into a Student object. The Object arrays being instantiated on the fly provide the parameters to the parameterized SQL queries. And so on. All of the rest of the work, dealing with the JDBC API, are buried inside the Spring plumbing, where common-sense defaults (in case of errors) take place to alert developers and clients when things go wrong.

This makes writing the StudentController, which will use the StudentRepository to do all the CRUD work you need, look like **Listing 2**.

The **@Autowired** annotation, by the way, tells Spring that it needs to instantiate one of these objects and initialize it as appropriate (from either configuration settings or by using common-sense defaults). And, like most other frameworks of its nature, Spring takes the Student objects handed back from the controller and magically transforms them into a JSON representation suitable for consumption by whatever invoked this HTTP endpoint in the first place.

Voila! You now have a fairly simple CRUD endpoint, wired up against a relational database. Not bad for an afternoon's work.

Wrapping Up

As I mentioned at the beginning, this is just the very tip of a very large iceberg; the Spring collection of libraries, tools, and frameworks is itself a huge collection of material to wade through, not to mention all of the Java ecosystem. Like the .NET ecosystem, the Java ecosystem is both commercial—Pivotal, the current caretakers of the Spring repositories, being a big player in that space, but obviously also Oracle, Google, and a few other players too—and communal. In fact, in large measure, the open-source world cut its collective teeth on the Java ecosystem, and much of the lessons learned about how to run a large community project were learned the hard way when open source and Java were both coming of age.

> Circles within circles, cycles upon cycles, each feeds the other with interesting insights and innovation, and the industry as a whole just keeps getting better.

Even if Java isn't something directly in your future, knowing the Java ecosystem provides a tremendous amount of insight and information into .NET and the .NET ecosystem—seeing some of the differences, and seeing some of what's missing. Case in point: when .NET came out, it introduced the notion of custom attributes, which provide a way for code to provide custom metadata for consumption by a variety of sources (such as ASP.NET frameworks). Having seen custom attributes, and recognizing its usefulness, Java turned around and introduced the same concept, the @-prefixed annotations you saw earlier. The JUnit unit-testing framework then turned around and adopted annotations to indicate methods requiring execution as part of a test suite, which the NUnit framework maintainers saw, liked, and adopted themselves, in turn. Circles within circles, cycles upon cycles, each feeds the other with interesting insights and innovation, and the industry as a whole just keeps getting better.

I hope you've enjoyed your trip to the Java islands this afternoon, and I hope to see you on a future CODE Magazine flight.

Ted Neward

Building a .NET IDE with JetBrains Rider

JetBrains Rider is a cross-platform IDE that supports .NET, Mono, and .NET Core and technologies that use frameworks such as ASP.NET, ASP.NET Core, Xamarin, and WPF. Rider supports many languages, such as C#, VB.NET, F#, JavaScript, and TypeScript to build console apps, libraries, Unity games, Xamarin mobile apps, ASP.NET MVC, and other Web application types

Chris Woodruff
chris.woodruff@jetbrains.com
http://chriswoodruff.com
https://twitter.com/cwoodruff

Chris Woodruff (or Woody, as he is commonly known) has been developing and architecting software solutions for over 20 years in many different platforms and tools. A Microsoft MVP in Visual C#, Data Platform, and SQL Server, Woody was recognized in 2010 as one of the top 20 MVPs worldwide. Woody is a Developer Advocate for JetBrains and evangelizes .NET, .NET Core and JetBrains' products in North America.

Maarten Balliauw
maarten.balliauw@jetbrains.com
https://blog.maartenballiauw.be/
twitter.com/maartenballiauw

Maarten Balliauw's main interests are in .NET Web technologies, C#, Microsoft Azure, and application performance. A Developer Advocate at JetBrains and an ASP Insider and former Microsoft MVP, he's a frequent speaker at national and international events and organizes Azure User Group events in Belgium.

such as Angular, React, and Vue.js. Rider has many features to aid .NET developers in their daily work. It has support for supported languages through code completion, code generation, a large number of refactorings, navigation, over 2,300 code inspections, and much more. In addition to these coding features, Rider also has everything that .NET developers expect in their IDEs: a debugger, a unit test runner, a (fast!) NuGet client, database tooling, a WPF XAML preview window, version control, and integration with tools like Docker and Unity Editor is there as well. Even with all of these capabilities and features, Rider is fast, responsive, and memory efficient.

A Bit of History

Before we get into the technology and architecture of Rider, we must look at where this IDE came from. As far back as 2004, JetBrains was looking at a stand-alone application for the Visual Studio add-in ReSharper. It was never released, but a fully functional prototype was around at that time. As you can see in the snapshot of the ReSharper 2.0 UI in **Figure 1**, it provided a solution explorer, an editor, find usages, code completion, and refactorings. And although not a modern user interface, it's quite spectacular that the editor was able to provide adornments for displaying documentation in-line, given that it was built on .NET WinForms and Windows Presentation Foundation (WPF) wasn't around yet.

The project was halted, but the work didn't go to waste. A few valuable features for both future versions of ReSharper as well as what would become Rider were born, including the action system, text control implementation, several tool windows and toolbar controls, the unit test runner, and the ReSharper command line tools (CLI).

One of the design choices for the ReSharper 2.0 IDE was to keep functionality separate from the actual IDE. This approach helped future versions of ReSharper: It could support Visual Studio 2010, 2013, 2015, and 2017, all using the same core with an IDE interoperability layer on top. This design also proved key in being able to develop Rider. Roughly speaking, all that was needed was to plug another integration layer on top of ReSharper's core.

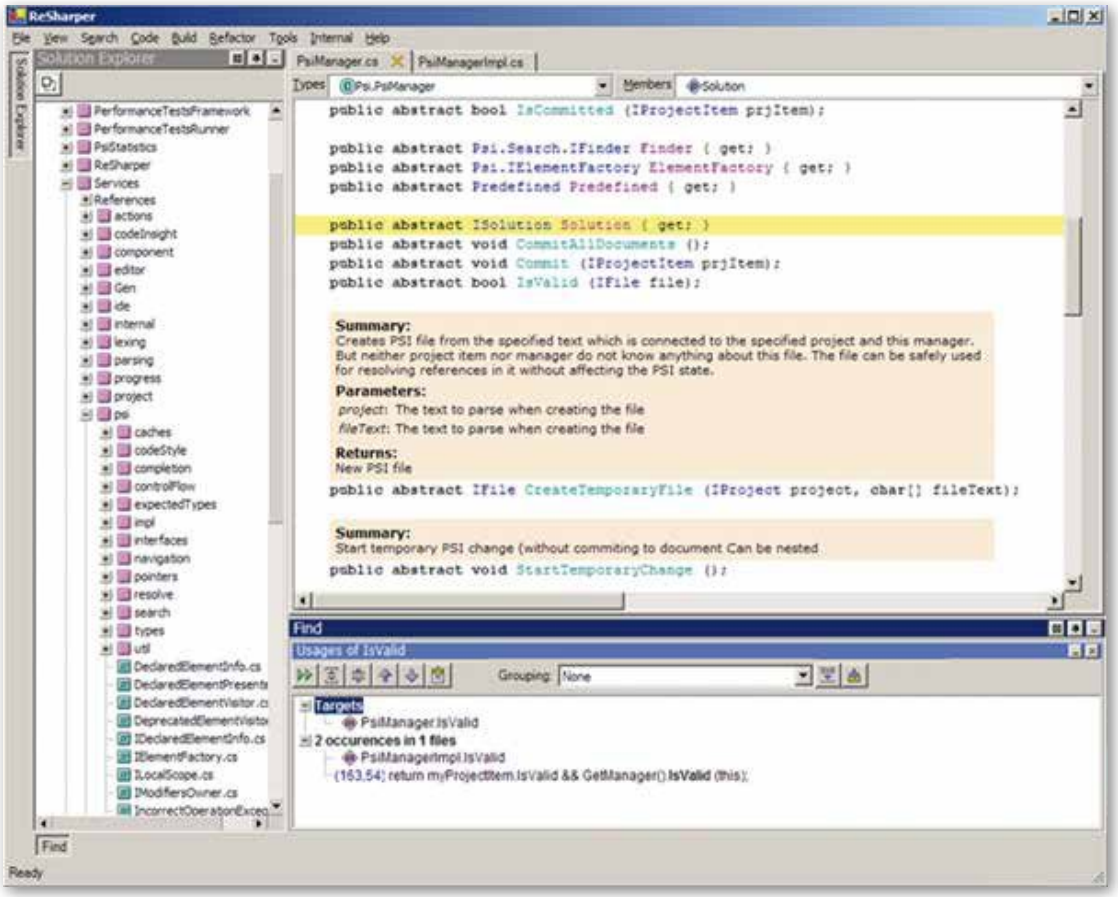
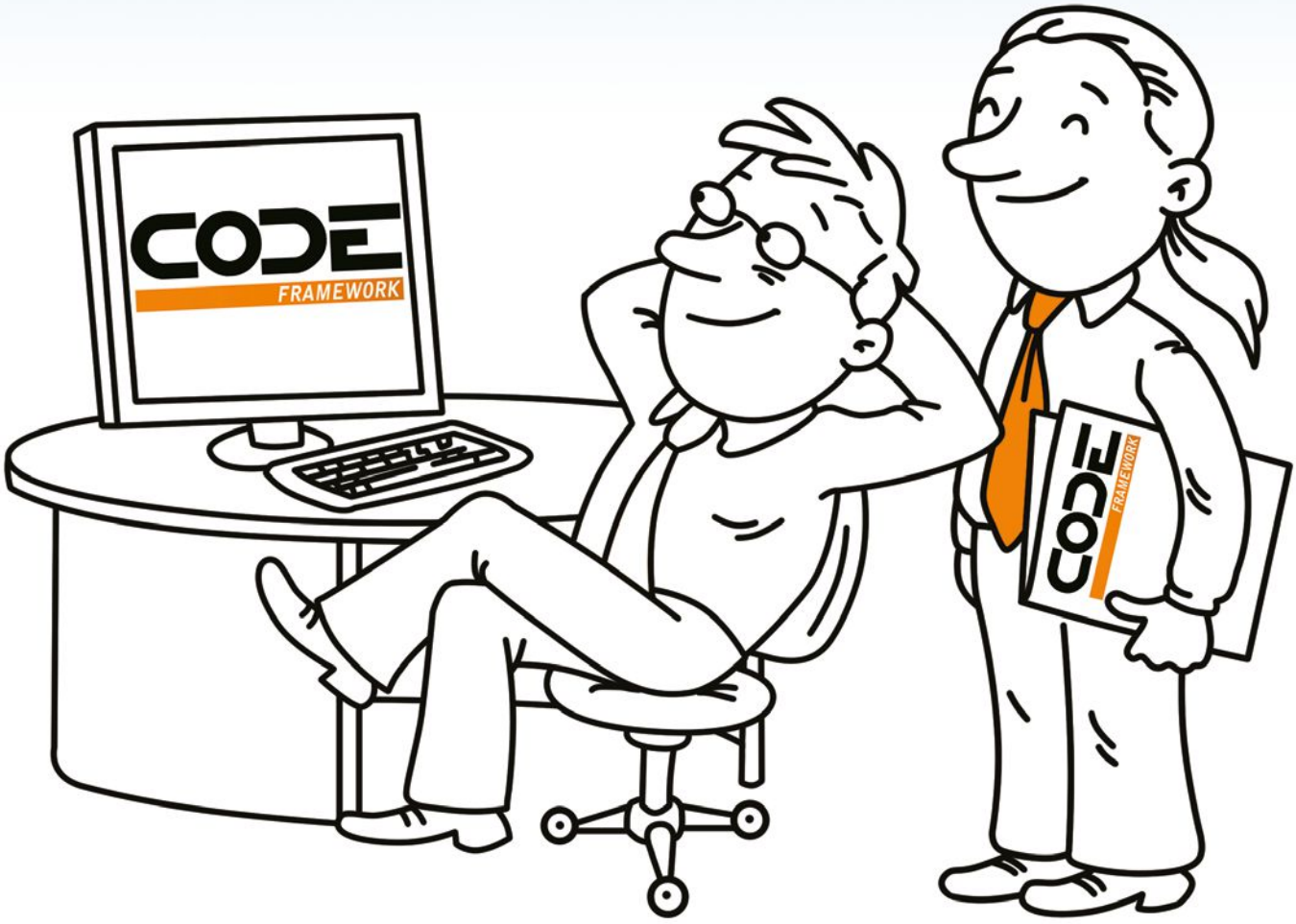


Figure 1: ReSharper 2.0 UI

CODE Framework:
Business Applications Made Easy



Architected by Markus Egger and the experts at CODE Magazine, CODE Framework is the world's most productive, maintainable, and reusable business application development framework for today's developers. CODE Framework supports existing application interoperability or can be used as a foundation for ground-up development. Best of all, it's free, open source, and professionally supported.

Download CODE Framework at www.codemag.com/framework

Helping Companies Build Better Software Since 1993

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

CODE
FRAMEWORK

Much like with the ReSharper 2.0 IDE, JetBrains wanted to reuse as much of the existing technology and tools as possible (full disclosure: we both work for JetBrains). A logical step was to re-use the IDE platform that JetBrains had been building for years: IntelliJ IDEA. It supports many development scenarios and is the foundation of the other JetBrains IDEs, like WebStorm and IntelliJ IDEA Ultimate (https://www.jetbrains.com/webstorm/ and https://www.jetbrains.com/idea). And then there's ReSharper (https://www.jetbrains.com/resharper), which understands C# and VB.NET like no other IDE.

JetBrains set out on an adventure of marrying IntelliJ and ReSharper. One could provide a rich front-end and its existing tools, like version control and many more. The other could provide .NET tooling. Each is built on a different technology stack. IntelliJ is built on the Java Virtual Machine (JVM), and ReSharper is all .NET-based. We needed a way for both processes to be able to work together.

A Thin but Smart UI Layer

Rider runs IntelliJ as a thin layer on top of ReSharper. IntelliJ provides the user interface, displaying an editor and a text caret in a source file. Some languages, like JavaScript and TypeScript, have a full language implementation and Rider gets that, too. For other languages, like C#, VB.NET, and F#, the front-end has no knowledge about the language itself: This comes from the ReSharper back-end process.

When editing a source file, IntelliJ tracks what you're doing. For example, when you want to complete a statement, IntelliJ asks the current language service for completion items. C#, VB.NET, F#, and several other languages have no real implementation in Rider's front-end, and instead call into a facade language service that requests the information from the ReSharper back-end. This information then flows back into the front-end where a list of potential completions can be displayed.

Information flows in the opposite direction as well. Rider's IntelliJ front-end has the infrastructure to show code inspections ("squigglies"), but again has no notion of inspections for C#. When a file is opened, it notifies the ReSharper process and waits for it to analyze the file, run inspections, and gather highlights that should be displayed. ReSharper publishes a list of document ranges, inspection severity, and a tooltip text, and the front-end simply displays these.

In the end, Rider's IntelliJ front-end knows about some languages, providing several tool windows and things like version-control integration. For the .NET languages, it's really a thin UI layer that provides editing and other infrastructure, and it gets information from the back-end process when it's needed.

The Need for a Custom Protocol

Rider's IntelliJ front-end comes with a lot of infrastructure for editing, code completion, showing menu entries, code inspection results, and so on. The ReSharper back-end process has similar concepts of providing code completion, inspections and running refactorings.

Because of these similarities, JetBrains realized that we could work with a simplified model that's shared between the front-end and the back-end. When editing code, we could pass around small bits of text and deltas. Surfacing a code inspection meant providing a document range, a severity, and a description. If we can surface one code inspection from ReSharper in IntelliJ, we can surface all of them. If we can change chunks of code in the editor and make one refactoring work by passing around small chunks of data to update the model, we can make all refactorings work.

JetBrains experimented with various ways of passing around actions and small chunks of data between both front-end and back-end, such as:

- Re-using the Language Server Protocol (LSP), an idea we discarded. LSP is great and does many things, but it's also a lowest common denominator. Some refactorings in ReSharper are impossible to implement in LSP without bolting on many customizations. It's also unclear how LSP should handle languages like Razor, which typically mix C#/VB.NET and HTML, CSS, and JavaScript. There would have to be an LSP component for the separate languages, but also one for the combined languages. LSP would introduce many complexities and provide little benefit for our particular use case.
- Building a custom REST-like protocol where each process could call into the other. JetBrains experimented with various transport mechanisms and serializers, such as JSON and ProtoBuf. Unfortunately, this protocol proved slow, hard to customize, and hard to work with during development.

We realized that the main downside with both approaches was that they use a Remote Procedure Call (RPC) type of protocol. We would need to always think in a "request-action-response" flow, rather than having a more natural flow between front-end and back-end that are built on the same concepts. Instead of the front-end sending a "add file to solution message" to the back-end and then waiting for acknowledgement of updated state, we'd rather write a **solution.Add(filename)** in one place and have both processes in sync automatically, without having to think much about the conflict resolution or what would happen in case of an exception at that point.

Another realization came in the form of the type of data passed around. Coming back to inspections, this would be only a document range, a severity, and a description. Except in an RPC style, we need more details: Which solution is this inspection for? Which project in the solution? And which file? Every call would need such contextual information, making the RPC calls bulky and adding a lot of overhead for developers.

Our aha moment came when we tried modeling this protocol as a Model-View-ViewModel (MVVM) pattern. Its definition on Wikipedia states that MVVM "facilitates separation of development of the user interface (view) from development of the back-end (model) using a view model as a value converter." IntelliJ is our view, ReSharper provides the model. And our protocol is the view model that shares lightweight data and data required for UI components.

Instead of having state in both processes and trying to keep them in sync, we have a shared model that both can

contribute to. In essence, exactly what we wanted in the first place! When a developer adds a new file to a project in the front-end, the model is updated and both processes react on it. When a refactoring running in ReSharper adds a new file, that same model is updated and again, both processes can react on it. The model becomes our state, and both sides can subscribe and react to changes in the model.

Conflict Resolution

MVVM in inter-process communication doesn't solve conflicts. Changes can come from either the front- or back-end and may conflict. For example, the front-end may delete a file even as the back-end is running a refactoring and reports and updates to that deleted file in the model. When typing code while running a refactoring in the ReSharper back-end, how would you solve conflicting changes? Does the human who's writing code win? Does the back-end process that just rewrote code for our human developer win? How do you keep things synchronized?

One solution could be locking and preventing conflicts like this from happening. Except that nobody would like to see the IDE stop responding while a refactoring is running. Developers want to be able to delete that file, create a new one, and update the model even if some longer-running task hasn't completed yet. We want things snappy!

JetBrains decided on some basic concepts that prevent conflict resolution:

- There's a client and a server. For Rider, the client is IntelliJ and the server is ReSharper.
- Each value stored in the view model has a version.
- Every update of a value by the client increments the version.
- Server-side updates do not increment the version.
- Only accept changed values if the version is the same or newer.

When a value is changed by the client, the view model accepts it. If the server comes back with a change and the version number is lower, that change isn't accepted by the view model. This helps ensure that the client-side of the protocol always wins when there are conflicts.

Rider Protocol

JetBrains built and open-sourced the Rider Protocol using the MVVM approach and the rules for conflict resolution. But we didn't want to bother our developers with too many details of the protocol. Instead, we wanted the protocol to be more like **Figure 2**.

The lowest layer of the protocol provides the communication itself and works with sockets. Rider uses a binary wire protocol that sends across deltas. It provides batching, if needed, and supports logging to dump all data that goes over the wire. All of that is still too detailed to work with this protocol on a day-to-day basis! So JetBrains built a framework on top.

Having the IntelliJ front-end running on the JVM and the ReSharper back-end running on .NET, we created both a Kotlin (a JVM language) library and a C# library, both supporting several primitives. We also created a (Kotlin-based) code generator that allows us to define our view model in a domain-specific language using these primitives, and then generate code that can be used in the front-end and in the back-end.

JetBrains development teams only needed to know about these few primitives to work with the protocol, instead of having to learn all of the other magic that sits behind it. An added bonus here is that when using this framework, the protocol layer can be generated instead of having to work with reflection and introspection, helping in making it performant.

The protocol also supports the concept of **lifetime**, JetBrains' approach to managing object hierarchies. Instead of having to know when and where to dispose an object from memory, objects are attached to a lifetime and are disposed when the lifetime disappears. For example, a code inspection can be attached to the lifetime of an editor tab, so that when the editor tab is closed (and its lifetime ends), the inspection is also removed from memory. And that editor tab itself can be attached to the solution lifetime, so that when the solution closes (and the parent lifetime ends), all underlying objects are

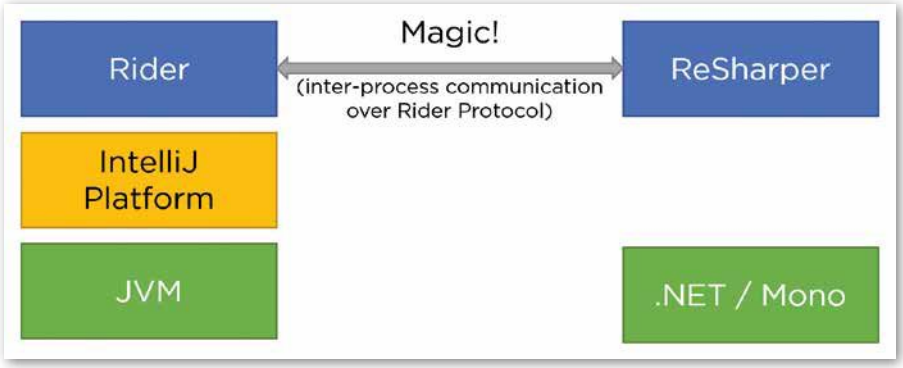


Figure 2: Rider's Protocol with ReSharper back-end

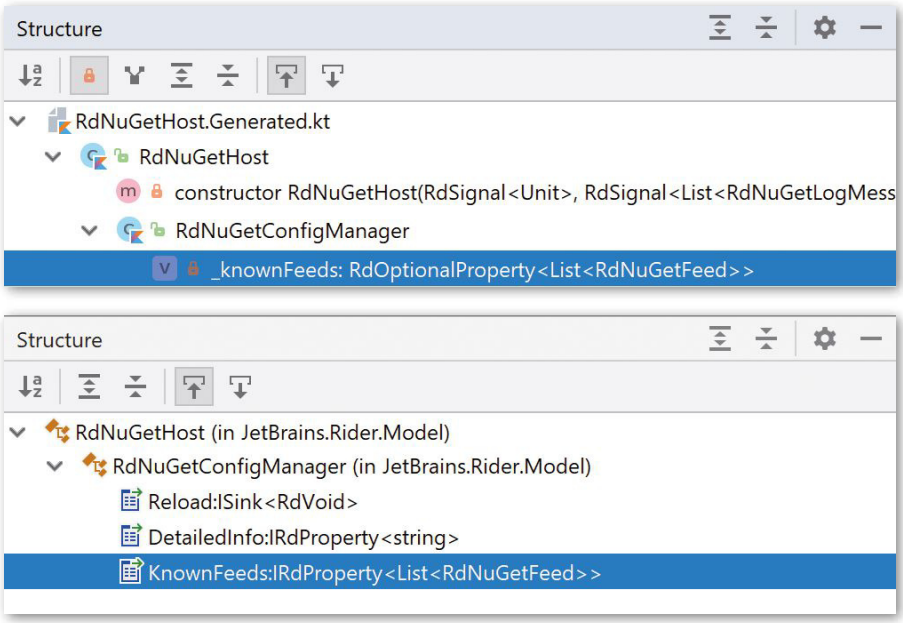


Figure 3: Code generation in Rider

disposed of as well. Both IntelliJ and ReSharper make use of this approach, and the view model in the protocol should partake in this.

Next to lifetime, the protocol supports:

- **Signals:** events that are fired when something happens
- **Properties:** observable values
- **Maps:** observable collections
- **Fields:** immutable values
- **Calls:** RPC-style calls, something which is needed from time to time

All of these can pass around data types that are **string**, **int**, **enum**, **classdef** or **aggregatedef** (a node in the view-model), and **structdef** (object holding data).

This may all sound vague, so let's look at a simplified example. Rider's NuGet client supports working with various package sources. These all have a name and a URL.

In the protocol, you define a view model node that holds NuGet client information (**RdNuGetHost**), which itself has a node that manages NuGet configuration (**configManager**) and has a property holding NuGet source names and URLs (**knownFeeds**).

```
object RdNuGetHost :
  Ext(SolutionModel.Solution,
    extName = "nuGetHost")
{
  field("configManager",
    ("RdNuGetConfigManager")
    {
      property("knownFeeds",
        immutableList(
          structdef("RdNuGetFeed")
          {
            field("name", string)
            field("url", string)
          }))
    })
}
```

After running code generation, you get a class definition in Kotlin, which the IntelliJ front-end can use, and a class definition in C#, which the ReSharper back-end can use, as seen in **Figure 3**.

You can then work against the generated protocol. For example, on the front-end side, you can set the list of feeds in the Kotlin programming language:

```
configManager.knownFeeds.set(arrayListOf(
    RdNuGetFeed("NuGet.org",
    "https://api.nuget.org/v3/index.json")
))
```

You can subscribe to this list in the back-end .NET code and react to changes. Or you can enumerate the values in the list at a given point in time. You even get code completion, thanks to the protocol code being generated, as seen in **Figure 4**.

In essence, thanks to code generation for the protocol, you can work on the same model without having to worry about communication and state maintenance between the IntelliJ front-end and ReSharper back-end.

Microservices

We already discussed that Rider consists of two processes: the IntelliJ front-end and the ReSharper back-end. Because both are running on a different technology stack (JVM vs. .NET), Rider has to run multiple processes that communicate with each other using the Rider protocol. There are a few up sides to running multiple processes: each has its own 64-bit memory space, and on multi-core machines, there's a good chance that these processes may run on their own CPU cores, providing better performance.

There is another, equally interesting upside: isolation. Multiple processes run independently from each other (apart from communicating via the protocol), so they can run garbage collection independently, as well as start and stop independently. This last one is very interesting! Imagine you could start and stop processes as needed, with their own memory space, their own garbage collector, and potentially on their own CPU core?

Rider may run more than two processes, depending on the type of solution you're working with. When working on a Windows Presentation Foundation (WPF) project, there may be three processes running, as seen in **Figure 5**:

The editor itself is Rider's IntelliJ front-end, and code completion, code analysis, quick-fixes, and so on, are powered by Rider's ReSharper back-end. The XAML Preview window at the bottom is a third process: ReSharper's back-end will detect that you're working on a WPF application and spin up a rendering process that communicates a bitmap representation with ReSharper over another instance of the protocol (**Figure 6**).

Rider also supports running Roslyn analyzers. Many development teams write their own analyzers to provide additional tooling for the frameworks they build. The way this works in Rider is that the ReSharper back-end spins up a separate process analyzing code using Roslyn, and then communicates the results to the IntelliJ front-end (**Figure 7**).

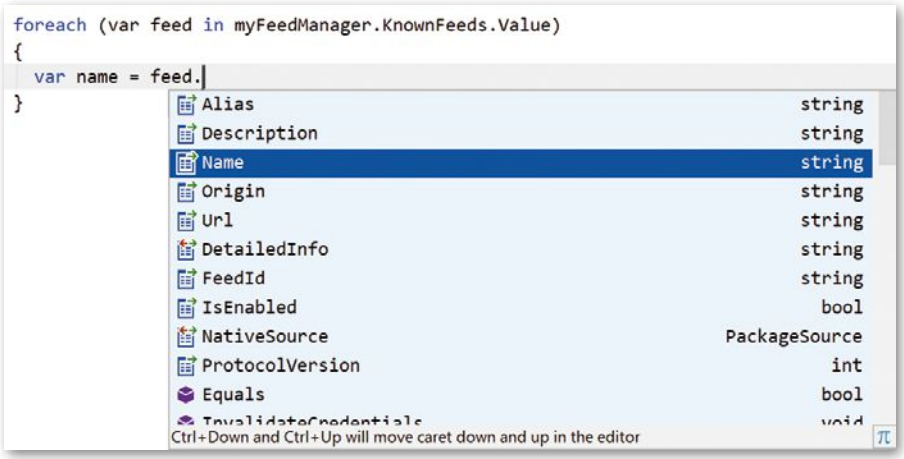


Figure 4: Code Completion example in Rider

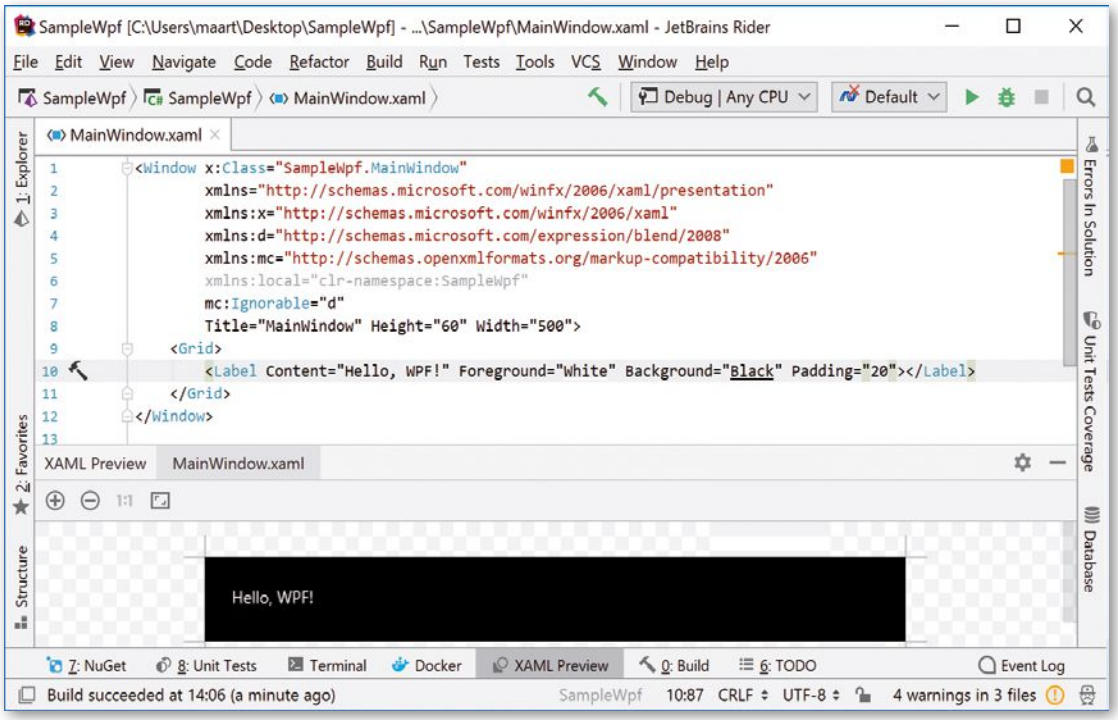


Figure 5: A WPF XAML preview inside Rider

The debugger also runs in a separate process, spawned by the ReSharper back-end and communicating using the Rider protocol. So when you have the XAML Preview tool window open in a project that uses Roslyn analyzers and you happen to be debugging that one, there will be even more processes running (**Figure 8**).

With an architecture where you can start and stop processes on demand, you can achieve a number of things:

- Separate memory space and independent garbage collection
- Start/stop functionality on demand, loading only the full feature set when needed by a solution
- Independent crashes. This becomes important in processes like the XAML Preview renderer, where you may have to render faulty user data. In those cases, you can fail gracefully and just stop one process instead of the entire IDE.

There is one additional benefit that we haven't discussed yet. Because the Rider protocol is socket-based, processes do not necessarily have to run on the same computer. For example, in the Docker debugging support, the debugger process runs in a Docker container (essentially a different computer), and communicates with the Rider's ReSharper back-end process. This opens the door to many possibilities, now and in the future.

In almost all of the examples we've looked at so far, Rider owns the process that is started/stopped. For the integration with the Unity Editor, Rider doesn't own the process. Rider and Unity Editor can be started independently, but this doesn't prevent both processes from looking for each other's Rider Protocol connection. When both are launched and the connection is allowed, Rider can share its view model with Unity Editor and vice-versa. This lets Rider

Figure 6: Rider's processes with XAML Preview renderer

Figure 7: Rider's processes with Roslyn

control play, pause, and stop buttons in Unity, and also provides the ability to debug code in Rider, even though it's running in the Unity Editor (**Figure 9**)!

In case you're interested in the internals of this plug-in, we have open sourced it (https://github.com/JetBrains/resharper-unity) so you can have a look at its IntelliJ front-end, its ReSharper back-end, and the Unity Editor plugin. Microservices!

What about the UI…

So far, we've focused on sharing the view model between Rider's IntelliJ front-end and the ReSharper back-end. What about the view itself? Surely, Rider must have a number of user interface elements and tool windows developed in the view side as well?

That's entirely correct. Surfacing code inspections and quick-fixes are really sharing a document range, severity, and description, and do not require any changes in the front-end as they re-use what is already in IntelliJ. In

other places, for example, the NuGet tool window, you have to implement every feature on top of the protocol you've built.

While developing more recent versions of Rider, JetBrains did come to the realization that many user interfaces and user interface elements are similar in nature. For example, all inspection settings are roughly a list of inspections, their severity, and a Boolean that switches them on or off. The settings for these all share the same user control that binds that list to a grid view.

For many other settings, the user interface is often a header, followed by one or more pieces of text, followed by a checkbox, a textbox, or a dropdown. Instead of rebuilding those all the time, we've started building some standard views that you can populate using the protocol. For example, the C# Interactive settings in Rider are defined in the ReSharper back-end, using the following snippet of code:

```
AddHeader("Tool settings");
AddToolPathFileChooserOption(lifetime,
    commonFileDialogs);
AddEmptyLine();
AddStringOption(
    (CSharpInteractiveOptions s)
        => s.ToolArguments,
    "Tool arguments:")
```

```
AddHeader("Tool window behavior");
AddBoolOption(
    (CSharpInteractiveOptions s)
        => s.FocusOnOpenToolWindow,
    "Focus tool window on open");
AddBoolOption(
    (CSharpInteractiveOptions s)
        => s.FocusOnSendLineText,
    "Focus tool window on Send Line");
AddBoolOption(
    (CSharpInteractiveOptions s)
        => s.MoveCaretOnSendLineText,
    "Move caret down on Send Line");
```

The front-end can then render these, and in Rider's IntelliJ front-end, you'll see a proper settings pane (**Figure 10**):

The benefit of defining user interfaces on the ReSharper side is two-fold. First, you no longer have to build the UI manually and can instead generate it from the proto-

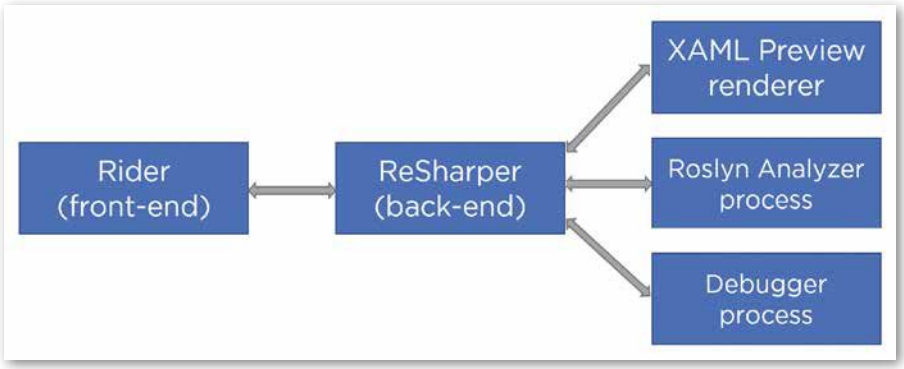


Figure 8: Rider's processes with debugger

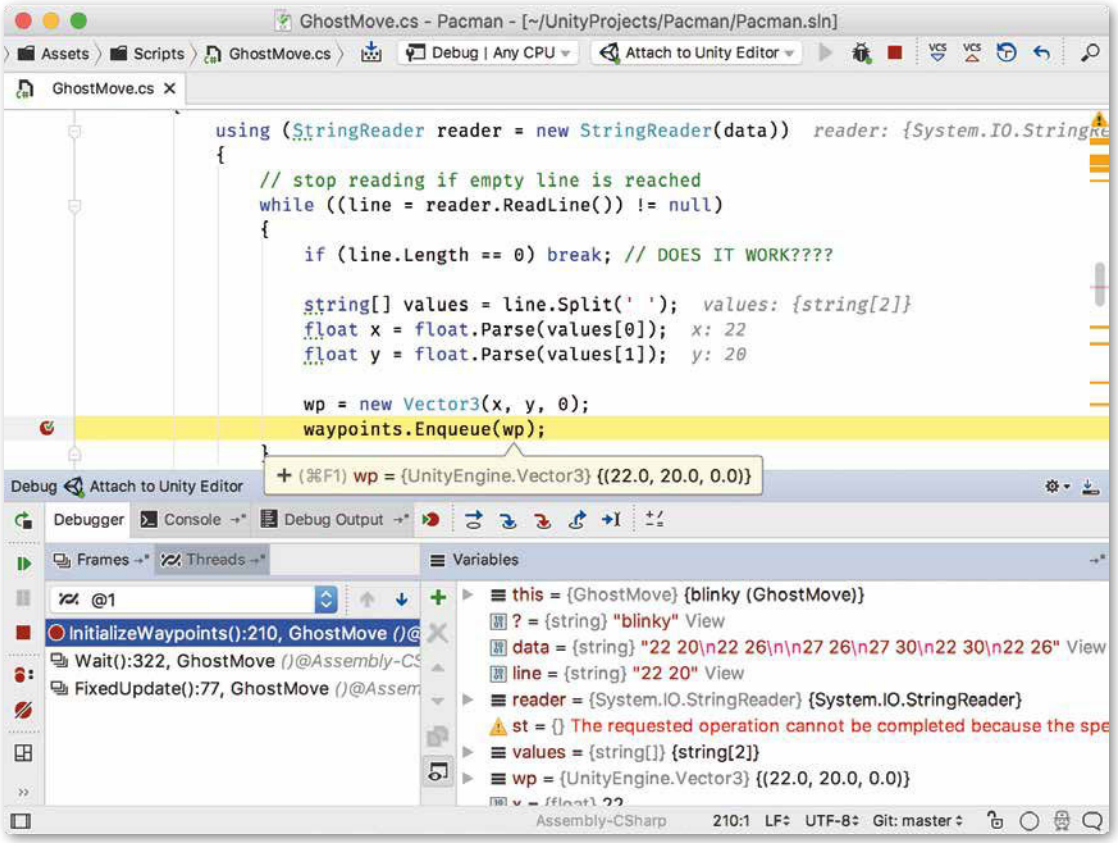


Figure 9: Rider's Unity Debugging in action on Mac OS X

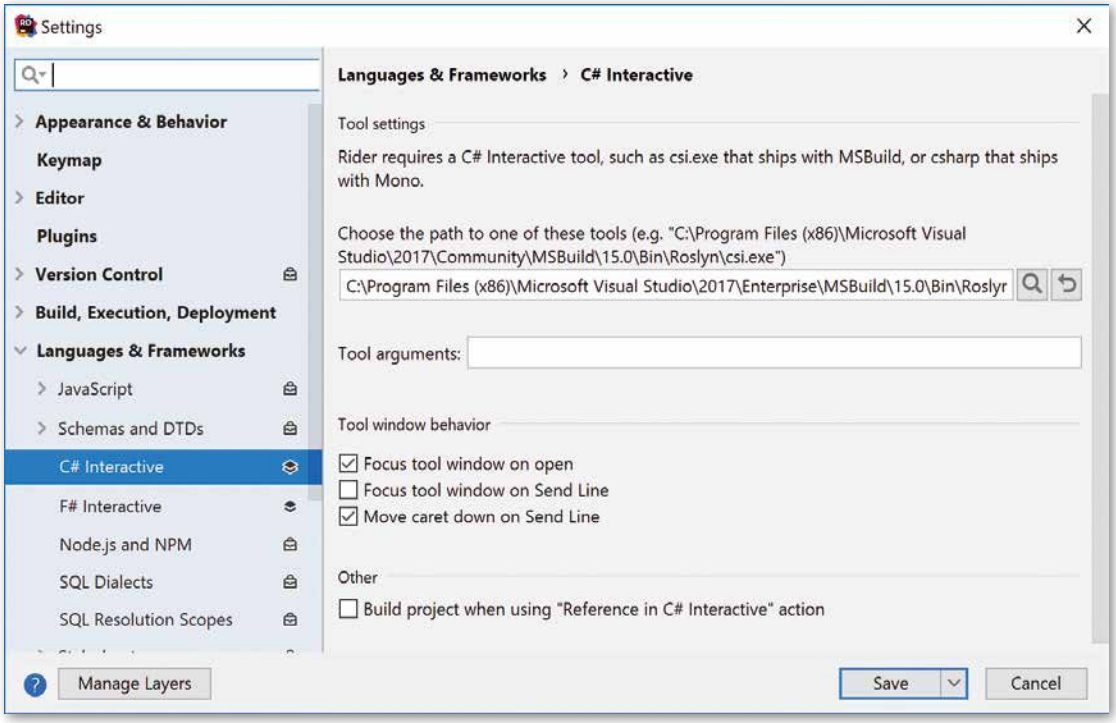


Figure 10: Rider's Settings Pane

col model. Second, because Rider and ReSharper are the same codebase, future versions of ReSharper can render these as well!

How Does Rider Make ReSharper Better?

We've touched on the fact that Rider re-uses as much as possible from ReSharper. The benefit of that is that features JetBrains adds to Rider are also available in ReSharper, and features JetBrains adds to ReSharper flow into Rider.

This also means that performance optimizations flow from one to the other. If JetBrains optimizes the speed of solution loading in ReSharper, Rider will benefit from that. If JetBrains optimizes loading of IDE components in Rider, ReSharper will benefit from that as well.

For Rider, JetBrains is focusing on getting the ReSharper back-end to run on top of the .NET Core CLR instead of requiring the full .NET Framework (on Windows) or Mono (on Mac O X and Linux). This effort will benefit Rider in that the runtime environment on all platforms will be .NET Core, instead of having two runtime environments today. Rider will also benefit from the many performance enhancements that were made in .NET Core, as will ReSharper.

Having Rider run ReSharper as a separate process also opens the door to running Visual Studio with ReSharper as a separate process. JetBrains is actively working on this, and this will provide the benefits outlined in this article: both Visual Studio and ReSharper will run in their own isolated processes and have their own memory space and, potentially, their own CPU cores. Together with the changes to support running ReSharper on the .NET Core CLR, this will greatly improve performance of ReSharper in Visual Studio!

Bringing in More Features: DataGrip, WebStorm, and dotTools

As you've learned from the history of Rider, the strength of the IDE comes from having the functionality from other IDEs in the IntelliJ family in the Rider IDE. For .NET and ASP.NET developers, JetBrains has integrated many of the features of the DataGrip and WebStorm IDEs into Rider, essentially by bundling existing plug-ins with Rider.

By bundling the database tools from DataGrip, Rider provides a rich experience for working with data and databases, supporting Oracle, MySQL, Microsoft SQL Server, Azure SQL Database, and many more. Rider users can write SQL code using the intelligence of the SQL text editor, execute SQL statements, view result sets, and navigate quickly through the database schema to look at the tables, views, and procedures with the database you're using.

JetBrains does want to go beyond simply bundling existing plug-ins and functionality. Because we feel that one and one should equal three, we're actively looking to extending functionality. How nice would it be if we could provide SQL code completion inside a string in a C# file? Or provide an **Alt+Enter** action to run that query? This is exactly what we're working on. Because we get the intelligence from both processes (and perhaps other contributing microservices), the IDE can become even smarter.

Other examples are in the bundled WebStorm features. With the rise of Web client-based development in the last decade, most ASP.NET and ASP.NET Core developers have seen the need to strengthen their use of JavaScript

in their projects and solutions. Rider works and understands not only JavaScript in the editor but also the most popular Web frameworks today including Angular, React, and Vue.js. Rider lets you take advantage of checking your Web code using its own code analysis functionalities, combined with the most popular open-sourced linters available today. You also get to debug and test all of your JavaScript code independently from your .NET code using built-in debuggers and unit testing functionality from WebStorm. There's even a REST client to test your Web APIs built using ASP.NET and ASP.NET Core.

Right now, when working in a TypeScript file, all related functionality is provided by the WebStorm plug-in on the front-end. However, ReSharper also has many features related to JavaScript, TypeScript, HTML, and so on. We're working on combining the best of both worlds here by having both the front-end and the back-end contribute code analysis, completion, code generation, and all that.

In Rider 2018.2, code coverage and continuous testing is now integrated. This is powered by another back-end process: dotCover. Additional features will come from dotTrace (JetBrains' performance profiler) and dotMemory (JetBrains' memory profiler). Right now, these only run on Windows. The goal is to have these tools run on Mac OS X and Linux as well, bringing rich profiling features in Rider to those platforms, too.

Conclusion

The future of .NET and ASP.NET development is very exciting, and Rider wants to be a part of it. JetBrains has plans to add and improve support for many of the technologies that .NET developers use today such as Azure, WinForms, WPF, Xamarin, and .NET Core. JetBrains is also working to improve the performance of Rider by improving start-up time when opening .NET solutions, as well as adding zero-latency typing in the editors.

Some other things coming in the future include the ability to work with C++ code within solutions and projects, and opening up the plug-in ecosystem by providing an easier SDK that lets plug-in developers extend Rider's IntelliJ front-end, its ReSharper back-end, or both.

We hope this article provided some insights into the Rider IDE, as well as exposed the challenge faced when combining existing products that run on different technology stacks. Rider is a very rich and rewarding .NET IDE that you can use on Windows, Mac OS X, and Linux. Do try it out for yourself—there's a free trial available at https://www.jetbrains.com/rider.

Chris Woodruff
CODE

Maarten Ballauw
CODE

(Continued from 74)

adjust or correct it. Figure out what the next roadblock will likely be, before the team hits it, and start working on getting it out of the way. If the team wants to move to a more DevOps-ish approach, talk to the Ops team and get their perspective on the idea. Talk to Ops folks about why you want your team to go down this path, and what benefits it provides to them. And be honest! If you don't know something, ask. Most importantly, don't be afraid to be wrong. In fact, you encourage a great deal more faith within the team by admitting that your ideas could be flawed. Encourage dissent, and react to it—change the plan in response to feedback. Nothing will encourage the team's trust in you more than when you take their thoughts and opinions seriously and incorporate that feedback.

Summary

Above all else, be honest with yourself: It can be painful to accept feedback, and it'll be particularly hard to take what you thought would be a "quick win" and discover that it's not anywhere close to "quick" or "win". Don't panic! At no point will your character and competence be more on display than when things start to go wrong, and it's a golden opportunity to show the team that you value feedback and encourage correction than when you take that feedback and correct yourself. Your personal and professional ethics will be on display every day as a manager, and although that can feel pretty vulnerable at times, at others, you'll be surprised—pleasantly so—when the team responds positively to those ethics and starts incorporating it into their own decision-making process.

Trust is a hard thing to earn, but once given, it can make all the difference between being the butt of the Dilbert cartoon and the team that everybody wants to be on.

Ted Neward
CODE

CODE

Nov/Dec 2018
Volume 19 Issue 6

Group Publisher
Markus Egger

Associate Publisher
Rick Strahl

Editor-in-Chief
Rod Paddock

Managing Editor
Ellen Whitney

Content Editor
Melanie Spiller

Writers In This Issue
Maarten Ballauw
Bilal Haidar
Ted Neward
Paul D. Sheriff
Chris Woodruff
Kevin S. Goff
Sahil Malik
John V. Petersen
Rick Strahl

Technical Reviewers
Markus Egger
Rod Paddock

Production
Franz Wimmer
King Laurin GmbH
39057 St. Michael/Eppan, Italy

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.
International Bonded Couriers (IBC)
Newsstand: Ingram Periodicals, Inc.
Media Solutions

Subscriptions
Subscription Manager
Colleen Cade
ccade@codemag.com

US subscriptions are US $29.99 for one year. Subscriptions outside the US are US $44.99. Payments should be made in US dollars drawn on a US bank. American Express, MasterCard, Visa, and Discover credit cards accepted. Bill me option is available only for US subscriptions. 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 300, Spring, Texas 77379
Phone: 832-717-4445
Fax: 832-717-4460

On Trust

For an industry that prides itself on its analytical ability and abstract mental processing, we often don't do a great job applying that mental skill to the most important element of the programmer's tool chest—ourselves. Do you trust your management? No, this is a serious question.

Without trust in management, it's extremely hard to get anything accomplished. If individuals don't trust management, there's no way for individuals to raise issues that management needs to know about and take action on. Management learns quickly not to trust the individuals, including their estimates and assessments, and before long, the entire office resembles a Dilbert cartoon on a bad day.

Being trustworthy as an individual is more of a moral, ethical and philosophical discussion, and often involves questions of concepts like "personal honor" or honesty, and so on. For many folks, that's between themselves, the mirror, and maybe a religious figure. In some ways, the calculus is simple: being trustworthy on an individual level is about not breaking the trust others (including your manager) place in you.

But what about when you become the manager? Particularly when you're now in charge of your former team?

Awkwardness

Let's face the elephant in the room head-on: It can be exceedingly awkward to be placed in charge of the people that used to be peers. Where before you could sit in the break room and complain about the silly decisions and clear cluelessness about those above you on the org chart, now, well, now you're one of them. And the rest of your peers aren't. In fact, you're now the one directly above them on the org chart, and that means if the team is going to go on sitting in the break room complaining, they're going to be complaining about you. (And that doesn't even consider the possibility that there may be team members who competed for the position that you now hold, and who now may, at best, think that the company made a mistake hiring you instead of them, and at worst, resent you for it.)

It's in your best interest to gain the trust of your team as quickly as possible, because without that trust, things could spiral out of control.

But trust is one of those things that's easier discussed than done. How, exactly, does one create trust?

According to a Linda Hill (a Harvard Business School professor) and Kent Lineback (an executive coach), trust is made up of two principal components.

Trust: Character

Character is a measurement of your values as a person and as a manager. Are you out for your own gain or are you out to make things better for your team? Are you there to improve the fortunes of the company at the expense of your team? Do you genuinely care about the team and the people on it? If the team suspects that your interests are more selfish than selfless, they'll quickly figure out that they can't trust you.

Your character will be on display through your actions—more specifically, how well your actions align with your stated values. Ask questions. Be respectful. Invite feedback. Keep your emotions in check no matter the situation. Give others the spotlight when credit and kudos are going around.

For example, one of my subordinates managing a team in my group came to me with a proposal. We're in the home stretch of a big software release, and he thought the team might be more successful if we set aside our two-week sprint cadence in favor of a more Kanban-style approach until we hit the ship date. Hmm. Interesting proposal: We don't really need the sprint opening and closing ceremonies because we've locked in everything on the backlog, and bug fixes don't really fit well into a story-based format. On the other hand, this represents a new change, and changing things up right before a period of stress is not always a good idea. Hmm. Which way do I decide to go?

One of the values that I've stressed to my managers and the teams they manage is that of autonomy (one of the ARC triple, from the "On Motivating" article a while back). I want my teams to have the authority, with my support, to figure out what works best for themselves, and make their own decisions. I'm also aware that it can be easy to stay silent when presented with a plan that everybody thinks is a good one, and I didn't want anyone to feel steamrollered into going along with a plan that they didn't believe was viable. It's important, too, that the team have a good working rapport with their manager, and if this plan doesn't work, I want the blame, if any, to fall on me.

Given all that, my response was this: "Let's ask the team." I sent an email to the team, saying that the manager and I had been discussing this plan (notice, I give no credit to whose idea this is—if it's good, it was my subordinate's idea, and if it's not, it was mine), and I wanted the team to give me explicit reasons why it wouldn't work. Note that I don't "ask for their thoughts." I specifically want dissent on the plan. This way, if anybody offers up such, it's not because they don't believe in it, it's because I asked for it.

For the record, we're still in the home stretch, but the team took to the decision quickly and easily, and several of them have commented that they find this approach to be working well for us. The jury's still out, but I think my subordinate's plan was a good one.

Trust and Competence

The other thing the team will be watching is how well you understand their business. For a software development team, this is obviously the part where they will be judging how well you understand the practice of writing software, but it's going to be more than just knowing how to use the language or the platform. It's going to be a measure of how well you understand the processes involved, and how the project fits within the "big picture."

It's also going to be about how well you can deliver as a manager, too. How well can you eliminate the obstacles that the team faces? If the team wants to get desktop computers on which to do development because the laptops are too wimpy to handle all the simultaneous services that need to be running, can you get them? If the team wants to reorganize their workspace, can you get the office team to go along with the idea? When the team scores a win, can you get upper management to recognize the win, so your team can get the kudos they deserve?

That all seems like a tall order and it is. But like any sports team coming from behind, you don't have to score big entirely on one play; start with some "quick wins" with the team. Find a policy that's entirely under your control that the team dislikes, and

(Continued on page 73)

AI & BIG DATA EXPO NORTH AMERICA 2018

BLOCKCHAIN EXPO NORTH AMERICA 2018

CYBER SECURITY & CLOUD EXPO NORTH AMERICA 2018

IoT TECH EXPO NORTH AMERICA 2018

THE FUTURE OF ENTERPRISE TECHNOLOGY

NORTH AMERICA
28-29 NOVEMBER 2018
SANTA CLARA CC,
SILICON VALLEY

GLOBAL
25-26 APRIL 2019
OLYMPIA GRAND,
LONDON

EUROPE
19-20 JUNE 2019
RAI, AMSTERDAM

REGISTER FREE

+44 (0) 117 980 9023
enquiries@iottechexpo.com

www.ai-expo.net
www.iottechexpo.com
www.blockchain-expo.com
www.cybersecuritycloudexpo.com