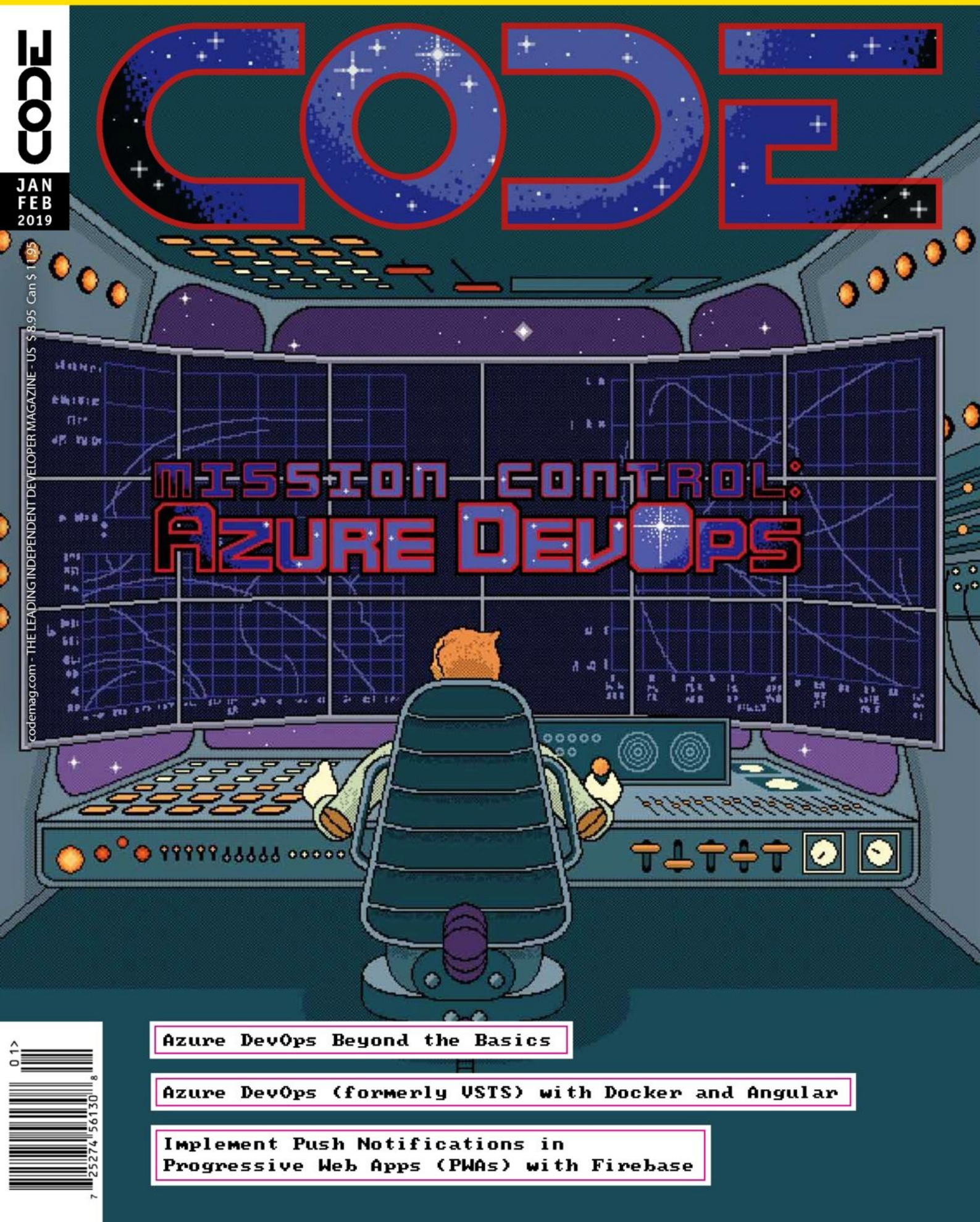
Angular, Unit Testing, Chocolatey on Windows, Docker

CODE

JAN
FEB
2019

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



MISSION CONTROL: AZURE DEVOPS

Azure DevOps Beyond the Basics

Azure DevOps (formerly VSTS) with Docker and Angular

Implement Push Notifications in Progressive Web Apps (PWAs) with Firebase

01>
7 25274 56130 8

Empower your CODE * COMPANY * CAREER

DEVintersection
SQLintersection

CO-LOCATED
<anglebrackets/>

Microsoft Azure + AI Conference

CO-PRODUCED BY
Microsoft & DEVintersection

June 10–13, 2019

Walt Disney World Swan and Dolphin Resort
Orlando, FL

Powered by Microsoft NeXtGen SQLskills .NET Rocks!

DEVintersection.com anglebrackets.org 203-264-8220, 9-4, M-F

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.

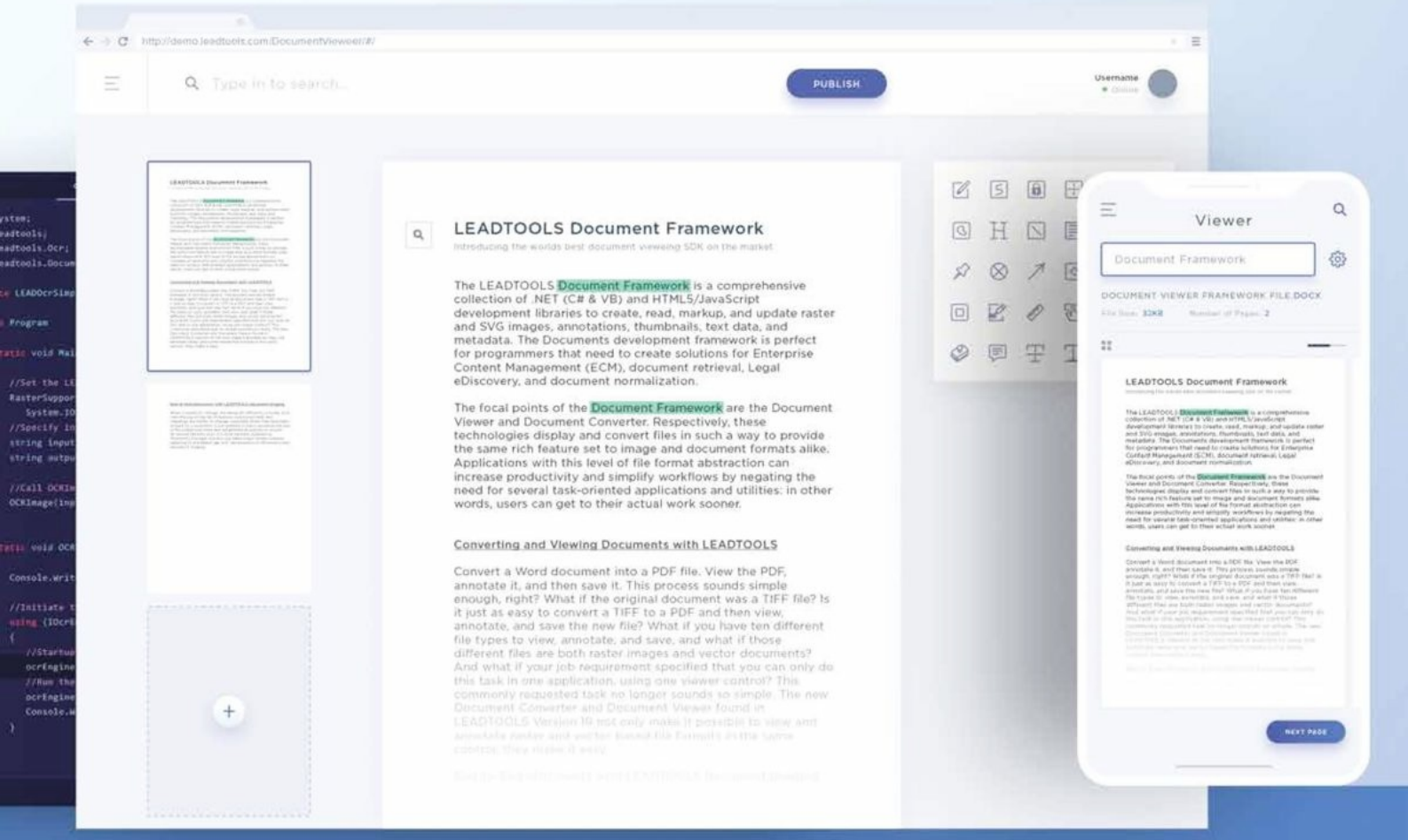
Subscribe online at codemag.com

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

Multi-platform Document Viewer SDK



With only a few lines of code, developers can use the LEADTOOLS Document Viewer SDK to add rich document viewing features to any project, including text search, annotation, memory-efficient paging, inertial scrolling, and vector display.

- VIEW, CREATE, ORGANIZE & EXPORT A DOCUMENT FROM MULTIPLE FILES
- LOAD, SAVE, CONVERT PDF, DOC, DOCX, RTF, HTML, SVG, AND XPS
- VIEW AND EDIT ANNOTATIONS & MARKUP, INCLUDING PDF AND IBM FILENET
- DRAG AND DROP INTERFACE FOR ADDING AND REARRANGING PAGES
- INTELLIGENT TEXT EXTRACTION USES OCR ONLY WHEN NEEDED

Microsoft .NET · macOS · iOS

Get Started Today

DOWNLOAD OUR FREE EVALUATION

LEADTOOLS.COM

Part 3: Merrily We Roll Along

In my last two editorials, I documented a new conversion project that my team is working on. By the time you read this editorial, most of our features will be in the UAT (user acceptance testing) phase of the project, and if things go accordingly, we should be shipping sometime in the spring (fingers crossed).

This has been a truly challenging project and I'm a bit floored that we've made as much progress as we have. By next issue, I should have a good idea if we can achieve the timeline I just mentioned. If we make the ultimate timeline of a spring deployment, I'll be even more astonished. Why the astonishment? Let's go back to the beginning...

How It Started...

In mid-2017, my client began thinking about undertaking the wholesale re-write of the core applications used to support their business. During this process, we took a SWAG (Scientific Wild Ass Guess) at how long this project would take from start to finish. With an estimated spring start date, we coalesced around the idea that it would take approximately a year of development. So if things went well, we might, and I seriously mean might, meet that deadline. We are about nine months into the development process and (knock on wood) we might just achieve that goal. This project might just well be a unicorn (a project that comes close to the actual projected deadline). I don't know about you but delivering software anywhere near the original estimated ship date is a truly rare event, especially for a project of this scope and complexity.

The Plot Thickens

With a goal date in mind, we charted a journey into unknown territory. Although most of this project followed a fairly stock development process, there were a number of tasks that we knew would take serious creativity and innovation. With an already truncated deadline, we knew that these innovations might jeopardize the ultimate timeline. Projects of this size require some level of innovation and it's anybody's guess how long those innovations could take.

Eating the Elephant

Once our deadline was locked in, we tried to figure out what we could deliver using a phased timeline approach. In order to meet the deadline, we broke down the project into four distinct areas of focus: Business Unit 1, Business Unit 2, Data Maintenance, and Back-end Processes. The team thought that we'd be able to deliver code that could enter UAT for Business Unit 1 in the fall. After that, we could focus our time on Business Unit 2. The back-end processes and maintenance screens won't be counted as part of the deadline and could be delivered at a later date.

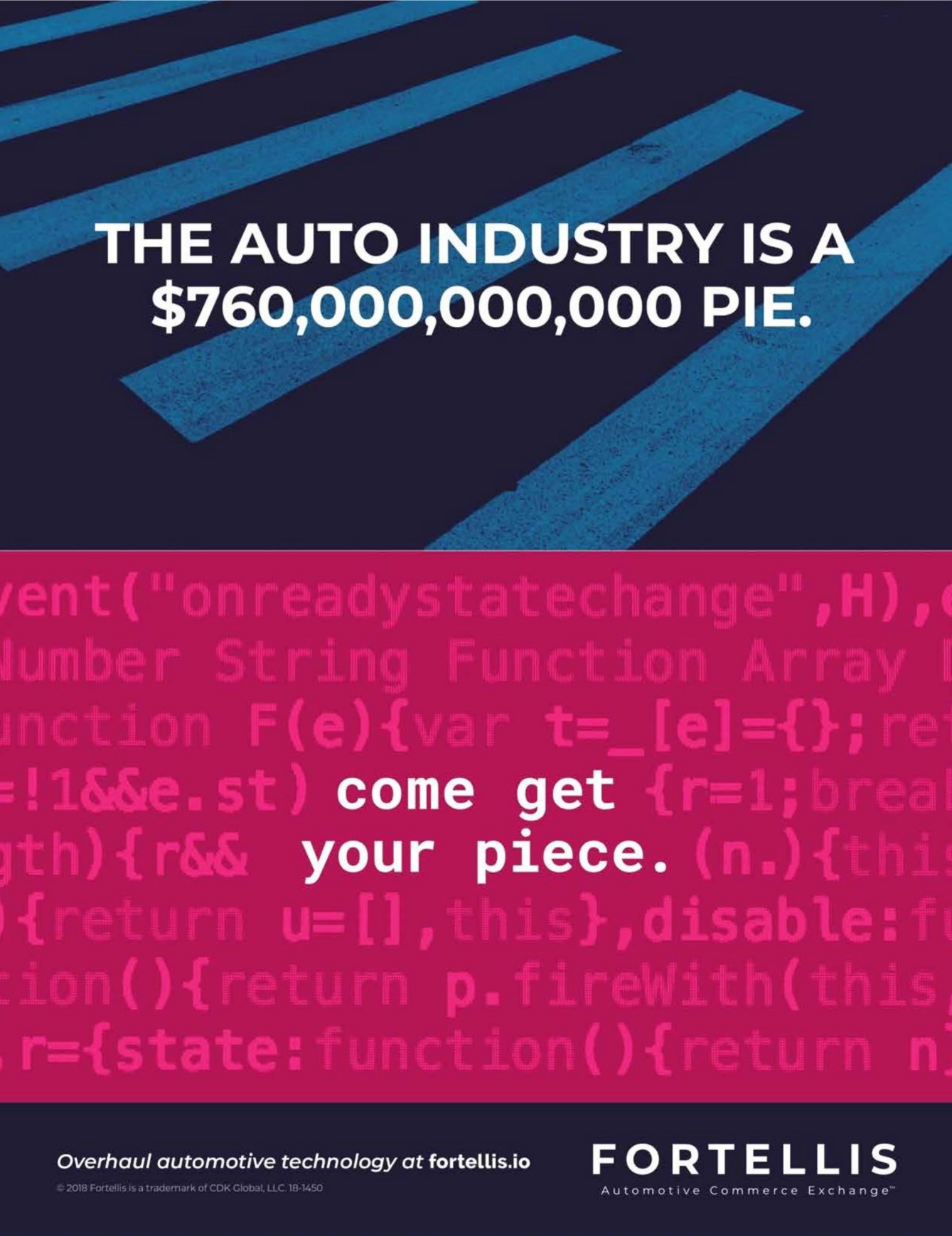
As mentioned in earlier editorials, the development process began with a small team of core developers. This team figured out the HOW of the project. How would code be organized, how would tests be written and organized, and most importantly, what we would need to build in the coming months. It was this last, the WHAT that was very important. In order to succeed in building Business Unit 1's application, we needed to know just what that was. Our first goal was to fill a backlog of work for our team to map out and scope the application. To help manage this process, we created a Kanban board and loaded it with all the screens, framework items, data conversions, and processes we would need to construct for Business Unit 1. Where possible, we included screen shots and the legacy functional design documents. This gave us a good idea of just how many tasks we needed to accomplish. This information was then given to our project manager. We started with one PM and now have three or four on our project. The project manager(s) created our burn down charts and other information that management needed to monitor our progress. If you haven't worked with project managers yet, you should consider it. We're lucky to have a great project management team because projects like this wouldn't be successful without them. So how did this planning work out? As of this writing, Business Unit 1 is well underway with its user acceptance testing. We now have 100% of our features in UAT.

So how is the work for Business Unit 2 progressing? We used the same process that helped build Business Unit 1's software. We built a backlog of work and immediately went to work building parts of our next module at a furious pace. By my estimation, we should be nearly code-complete by the end of January. This project has a much higher velocity than most as we've been able to leverage many of the components that we built for Business Unit 1.

The Question of UAT

Now the real question is: How long will it take our application to complete user acceptance testing? This is a big unknown and largely out our hands. This may be the most difficult timeline to gauge. Our code is thoroughly unit and integration tested and we have numerous gates that the code passes through before users get their hands on it. As Mike Tyson once said, "Everyone has a plan until they get punched." As soon as users get their hands on your code, all bets are off. You never know what they'll find. I'll let you know how that's going in my next editorial.

Rod Paddock
CODE


THE AUTO INDUSTRY IS A
$760,000,000,000 PIE.
come get
your piece.
Overhaul automotive technology at fortellis.io
© 2018 Fortellis is a trademark of CDK Global, LLC. 18-1450
FORTELLIS
Automotive Commerce Exchange™

Azure DevOps, Docker, and Angular

In my previous article, I shared my love of Docker with you. I really love Docker—it can be such a productivity boost. Although it does have a lot of applicability to my dev life, where it really shines is DevOps. Microsoft has been busy with a product called VSTS, or Visual Studio Team Services. Perhaps the worst thing about that product was the name.

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.

"Visual Studio" led us to believe that this product was specific to Microsoft technologies, namely Visual Studio. Recently, Microsoft has renamed and restructured this product under the name "DevOps."

The reality is that VSTS was fully cross-platform. Azure DevOps is certainly cross-platform. In this article, I intend to take completely non-Microsoft technologies and show an end-to-end process, using Azure DevOps.

What is DevOps

I won't spend a lot of time describing what DevOps is, I'm confident that you'll find a lot of resources on the Internet describing the definition. The practical upshot of all this is that over many years, developers and IT pros have been in separate camps. Developers neatly package something and IT pros blow it up; then they spend the rest of the day blaming each other. Developers are under the gun to deliver new functionality, IT pros have a vested interest in status quo, because if you don't change anything, nothing will break, right?

In today's fast-paced world, this mentality needs to change. We need to lower the cost of deployment, we need to move away from 50-page installation instructions, and we need to automate as much as possible. Organizations that automate end-to-end will win.

Study after study has shown that this equates to real savings and massive gains in productivity.

What Are We Going to Build?

At the center of the show, you'll find Azure DevOps. I intend to build:

- A simple Angular project using Angular CLI. (I won't dive into the details of Angular in this article.)
- Source control that uses Azure DevOps over a Git interface
- A process that converts my Angular CLI-based project into a deployable Docker image
- An Azure DevOps pipeline that runs every time I do a check in, builds my Angular project, and creates a Docker image from my Angular project
- A Docker image that automatically pushes to the Azure Container Registry in a published container that automatically deploys to an Azure website

The end result will be a developer that checks in code and automatically builds and deploys a Docker image that's live-deployed to an Azure website in a matter of seconds!

This must be hard, right? There's so much stuff to automate. Nope, it isn't. You'll be pleasantly surprised that by the end of this article, with zero background in Azure DevOps, you'll be able to build such automation for your projects very easily.

Let's get started!

Provision Resources

In order to follow along with this article, you'll need:

- A DevOps subscription, which you can get for free from here, https://azure.microsoft.com/en-us/services/DevOps/. If you have an MSDN subscription, you can use it for serious projects.
- An Azure subscription. You can sign up for free if you don't already have one. You'll need a valid credit card though. Although the costs for the resources required for this article are relatively low, be mindful of them. They are your responsibility.

In your Azure subscription, go ahead and provision an instance of Azure Container Registry. When you create a container registry, ensure that you choose to enable the Admin user, as shown in **Figure 1**. The reason you're enabling the admin user is because later you'll use this admin user to create a Docker registry connection within Azure DevOps pipeline.

Once the Azure container registry is provisioned, go ahead and grab the username password, as shown in **Figure 2**. You'll need this shortly.

Create an Angular App

Now let's come to your dev computer. This could be Linux, Mac, or Windows. As long as it has NodeJS installed and can work with GIT, you're ready to go.

Install Angular CLI using the command:

```
npm install -g @angular/cli
```

Now, go ahead and create a new Angular app using the command:

```
ng new angularProject
```

This should ask you a couple of questions and create a simple Angular project with a Git repo initialized. I'm not going to dive into the depths of Angular here. Really the actual application doesn't matter, because this simple command gives you a sample Angular project. For the purposes of this article, you need to know just a few basic concepts, such as how you run this project, and how you build this project.

In order to run this project, you can issue the following command:

```
ng serve
```

Go ahead and try it! You'll see that Angular builds your project and serves it on localhost:4200.

The second thing you need to know is that when you're ready to ship the app, you issue the following command:

```
ng build --prod
```

When you run this, you'll notice that Angular builds the project and creates a **dist** folder in the root of your project. You don't usually check in this folder, and the generated gitignore files take care of that. It's the contents of this folder that run your Angular App.

In your package.json, locate the line shown next:

```
"build": "ng build",
```

And change it to:

```
"build": "ng build --prod",
```

So now, if you run

```
npm run build
```

You should see a dist folder appear, which can be seen in **Figure 3**.

My Angular project is done! Now let's focus on the Docker image that will run this.

The Docker Image

As you can imagine, this Angular project yields a simple HTML page with a bunch of JavaScript and CSS files. To run this, all I need is a simple Web server.

What I intend to do next is build a Docker image that contains my application, deployed in a Web server. This is very simple to do. Simply add a file called **Dockerfile** in the root of your project with the following content in it:

```
FROM nginx:alpine
LABEL author="Sahil Malik"
COPY ./dist/angularProject
```

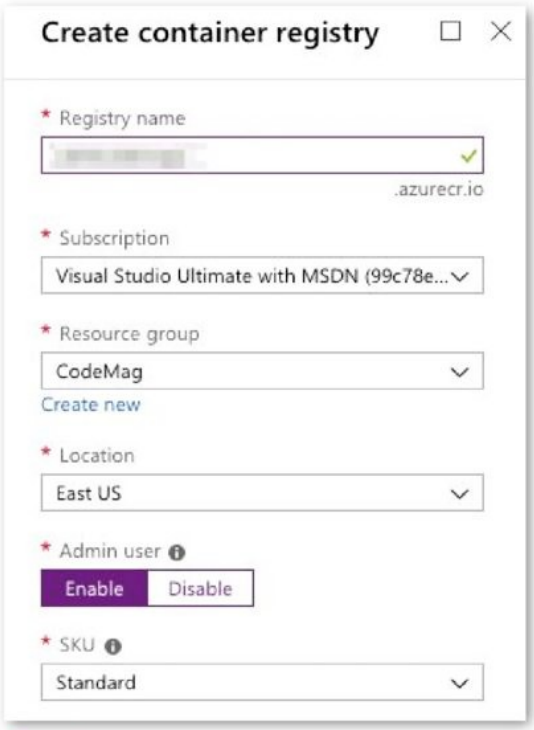


Figure 1: Provision an Azure Container Registry

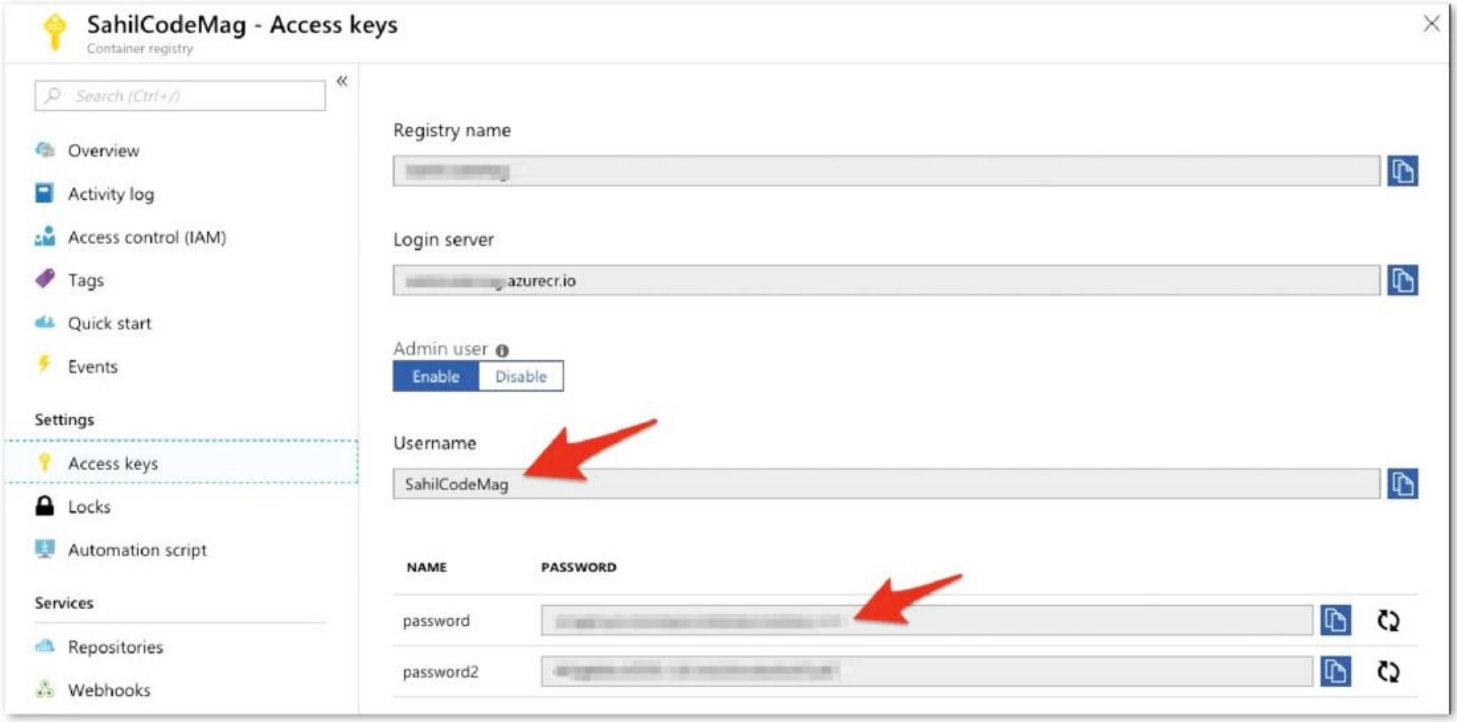


Figure 2: Grab the username password required for the container registry.

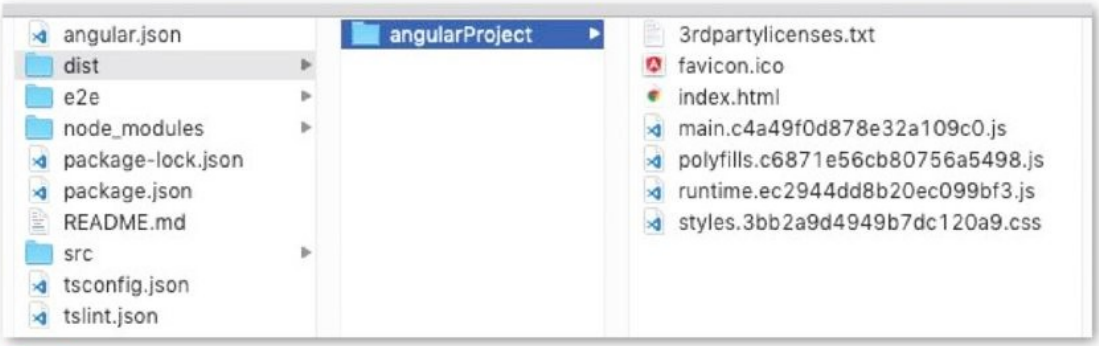


Figure 3: The dist folder where your runnable code lives

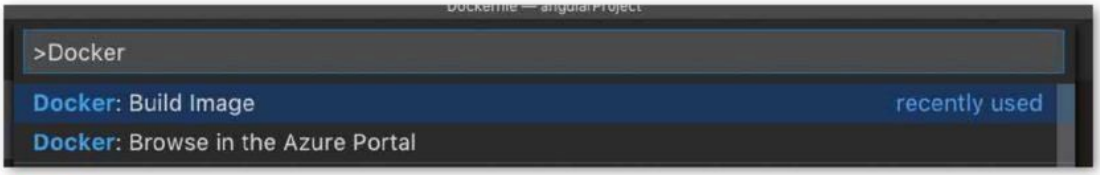


Figure 4: Build the Docker Image.

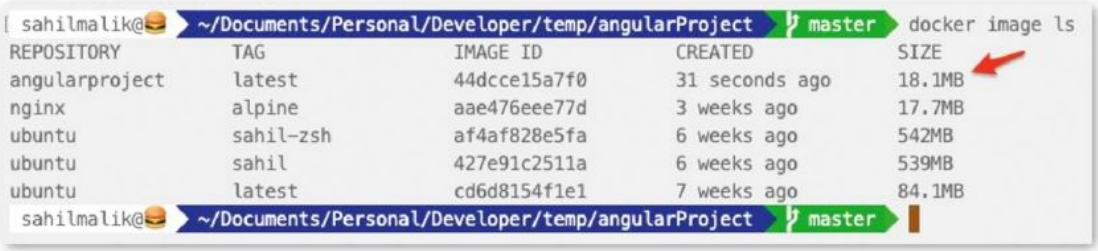


Figure 5: Docker image built.

```
  /usr/share/nginx/html
EXPOSE 80 443
ENTRYPOINT [ "nginx", "-g", "daemon off;" ]
```

The Dockerfile file describes your Docker image. Let's slice and dice it line by line.

```
FROM nginx:alpine
```

We're starting from a base image; specifically, I chose to use NGINX, which is a very lightweight Web server. It can also act as a reverse proxy, load balancer, etc. My goal is to use it as a Web server, and that's it.

```
LABEL author="Sahil Malik"
```

That's just documentation, so you know who to complain to if this code fails.

```
COPY ./dist/angularProject
  /usr/share/nginx/html
```

This single line of code, broken into two for readability purposes, takes the contents of the dist/angularProject folder, and copies them into /usr/share/nginx/html. NGINX serves content from that folder. When your Docker image is created, your files will be ready to serve out of that folder.

```
EXPOSE 80 443
```

Docker doesn't expose any ports from the container by default; you have to specify what you'd like to have exposed. Here I'm saying that I wish to expose the container's port 80 on port 443 of the host computer. This makes it easy for you to test things out.

```
ENTRYPOINT [ "nginx", "-g", "daemon off;" ]
```

This final line of code is the ENTRYPOINT, which is a Docker concept. It sets the command and parameters that are run when the container is run. In other words, the container starts, and it starts nginx. Remember that the nginx:alpine image is so lightweight that, effectively, you should have a full-fledged Web server with the application running in a completely contained container that's mere kilobytes in size.

Let's try it out!

Run the Container Locally

Before I take things to the cloud, I want to make sure everything is done properly and that things are running. This makes sense, because I can iterate much faster locally, so I'll make sure things work fine before checking them in.

Assuming you have Docker and the VSCode Docker extension installed, in your project, press CMD_P (CTRL_P on PC), and choose to build a Docker image, as shown in **Figure 4**. It will ask you to tag the image; choose **angularproject:latest** as the tag.

Once you have built the image, verify that the Docker image shows up, as shown in **Figure 5**.

Choose to run this image and visit http://localhost. Verify that your Angular project loads, as shown in **Figure 6**.

Great! Everything runs locally! It's now time to take things to the cloud.

Provision a Code Repository

Log in to your Azure DevOps portal and choose to create a new project. Give it a project name of **angularProject** and for now, make it a Private repository. Under version control, ensure that you pick **Git**. Could you have picked Team Foundation Server? Absolutely, you can. But command line stuff like npm-based projects are a whole lot easier to work with in Git, so let's go with that.

Once your repository is provisioned, choose to check in your code. You'll need to create a username password in your Git repo, which you'll use to push changes. There are other ways also, but you need some way of authentication from the command line to Git. Go ahead and create a username password for yourself and remember it.

Next, in your terminal, check in and commit:

```
git add . && Git commit -m "initial commit"
```

Go ahead and push your changes into your cloud-based repo using the following commands:

```
git remote add origin <giturl>
git push -u origin -all
```

Remember to replace the <giturl> with the URL of your newly created repository.

At this time, verify that your code is available in the Git repo, as shown in **Figure 7**.

With the code checked in, you can start automating stuff. The process of taking your checked-in code, creating a Docker image out of it, and pushing that Docker image to a Docker registry is the job of a pipeline. Let's see that next.

Figure 6: The Angular site is working.

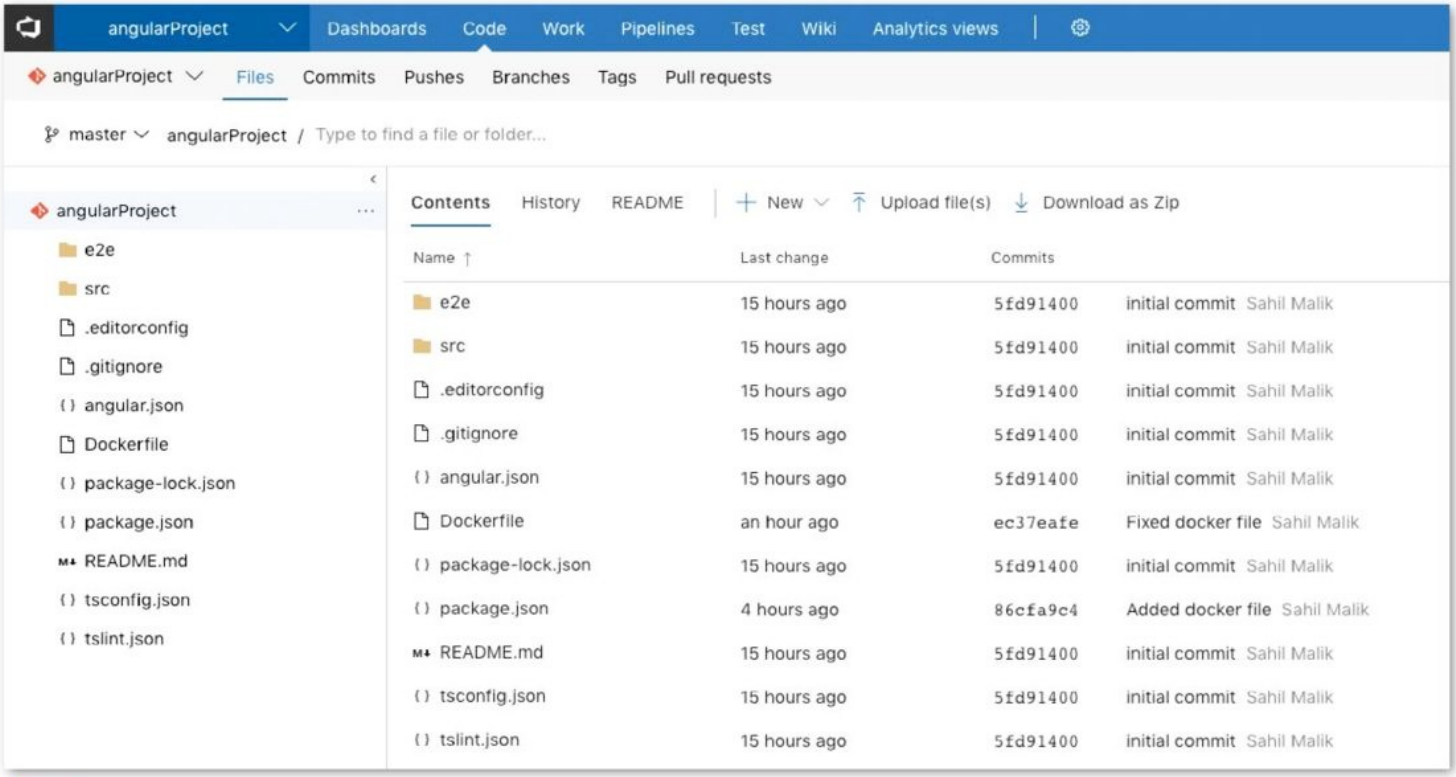


Figure 7: The code is checked in.

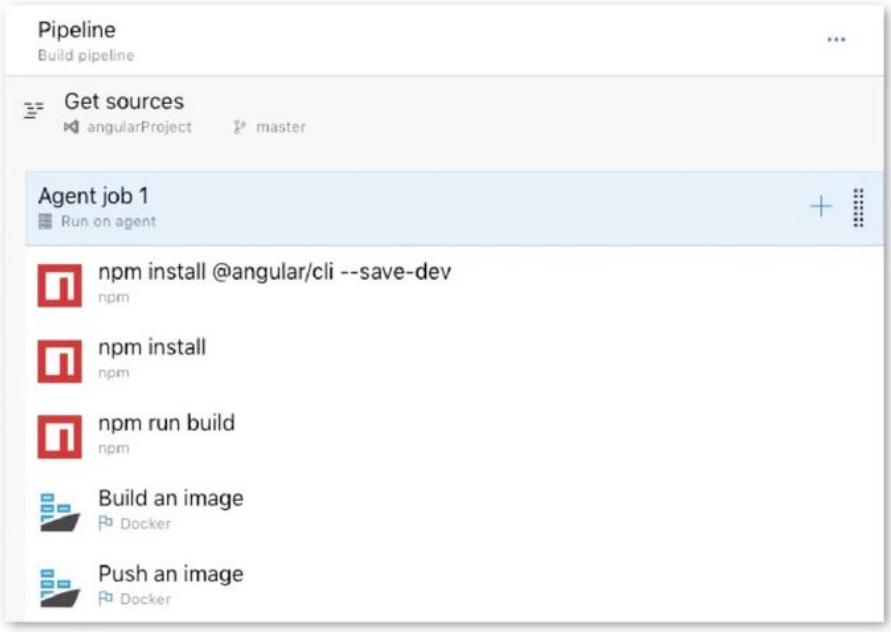


Figure 8: The build pipeline

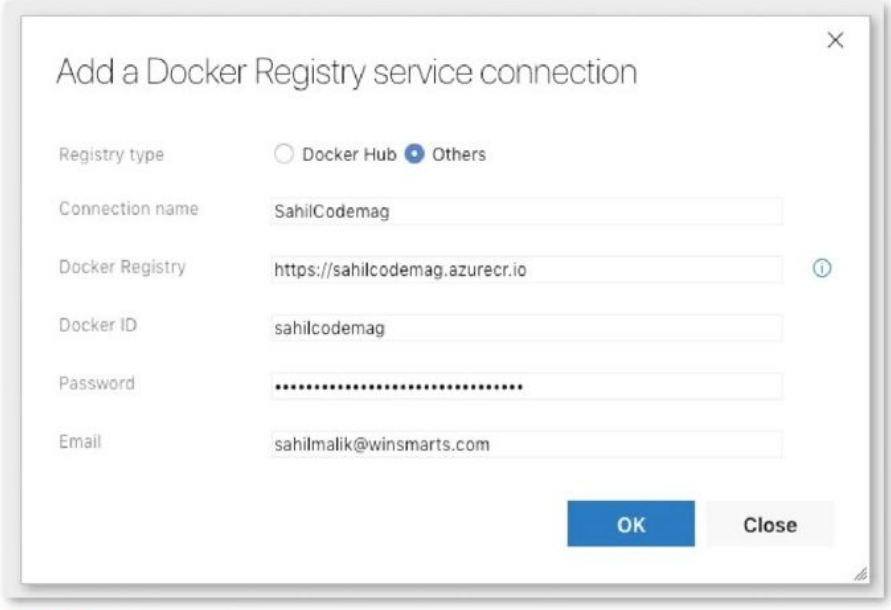


Figure 9: Configure the first task.

Create a Pipeline

The pipeline you create runs in an agent. An agent is an on-the-fly provisioned operating system image given to you, where you can perform actions, such as build your code. You can create your own agents, but I'll keep things simple. I'll stick with the Hosted Linux agent, which is free, and I'm cheap, so that works!

The pipeline you wish to create will do the following tasks:

- Download and install Angular CLI in dev mode within the project. The reason you do this is because the plain-vanilla hosted-Linux agent doesn't have Angular CLI installed.
- Run **npm install** because every node project needs this.
- Run **npm run build** because this creates a **./dist** folder for you. Remember, you never checked in the dist folder. Effectively you can have multiple build pipelines, one building a debug version, another building a prod version, etc.
- Build a Docker image using the Dockerfile that's part of the source code.
- Push that Docker image to the container registry.

My pipeline looks like **Figure 8**. But don't worry, I'll explain each step.

First, choose to create a new pipeline, ensure that you pick the **Docker Container** template. You don't have to, but this simplifies a few things.

Install Angular CLI

The command to install angular CLI locally within your project is:

```
npm install @angular/cli --save-dev
```

To make this step of your build pipeline work, drag, drop, and NPM task at the top of your pipeline and configure it as shown in **Figure 9**.

NPM Install and NPM Run Build

Use the same approach as installing Angular CLI and add two more steps: First do an npm install, and then do an npm run build.

Build an Image

Next, you need to build the Docker image. Before you can build the image, you'll need a **Docker Registry Service Connection**. In the dropdown for the Docker Registry Service Connection, you'll see a **Manage** link. Go ahead and click on it. Choose to add a **Docker Registry** connection. Ensure that you pick **Others** and specify the connection values as shown in **Figure 10**.

The Docker ID and Password you specify in **Figure 10** is the Admin user ID and password you got from the Azure Container Registry you provisioned earlier.

When you build the image, accept all default values, except one: Specify your Docker registry service connection and the Image Name. Change the image name as below:

```
$(Build.Repository.Name):latest
```

The reason you're changing this image name is because by default, the image name has the build ID appended to it. Although that's a great idea, you wish to have end-to-end automation. You want your code check-in to trigger the whole process. The last leg of this process is deploying the Docker image to an Azure website. This is achieved by a webhook and webhooks are tied to a specific Docker image. So if the Docker image name keeps changing, I'll get a history of images, which is great! But I won't get the webhook firing, because the webhook was set up on an older image from a previous build.

Effectively, by removing build ID, you're overwriting the same image, and that image has a webhook tied to it. You haven't yet set up the webhook, but you will, shortly.

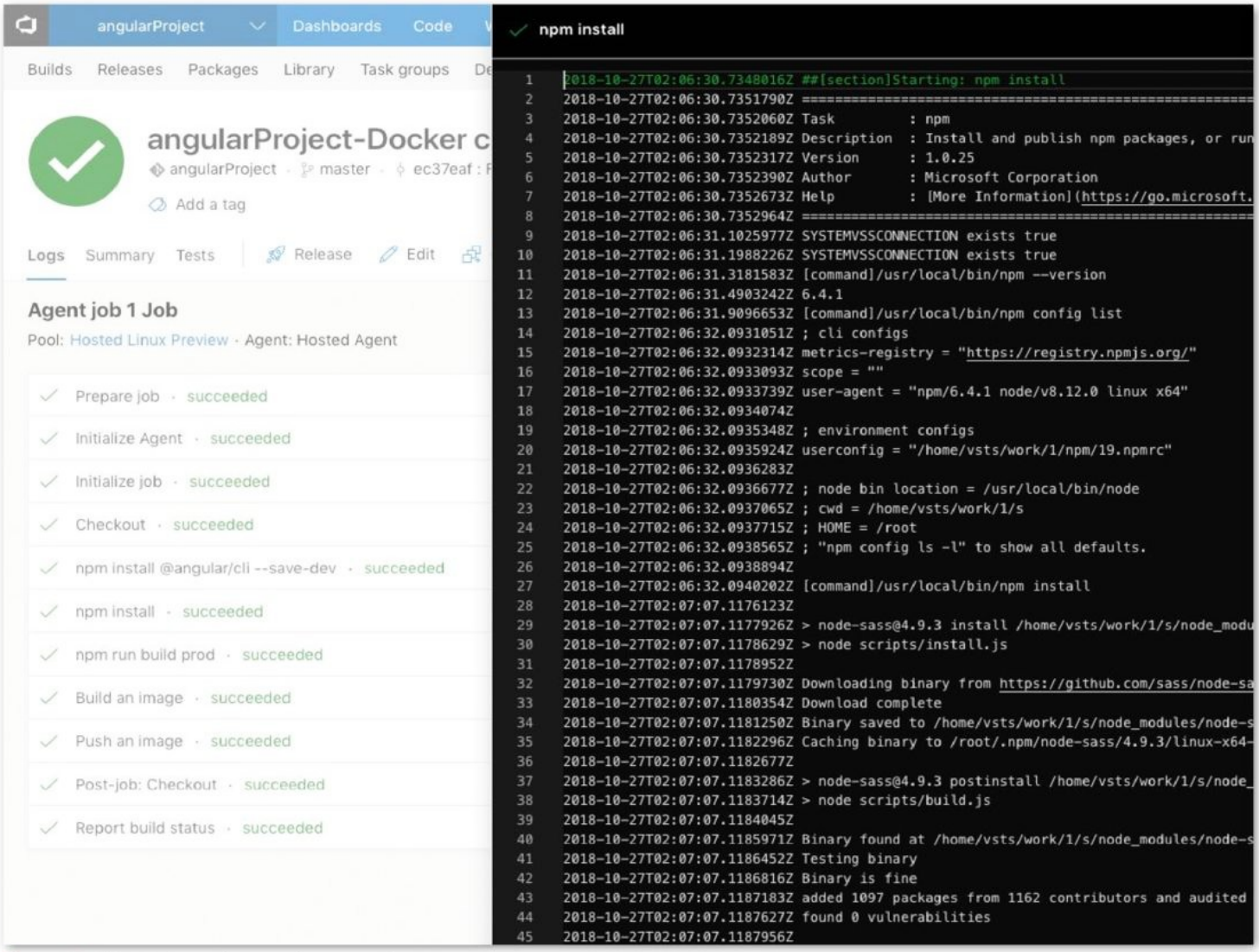


Figure 10: Add a Docker registry service connection.

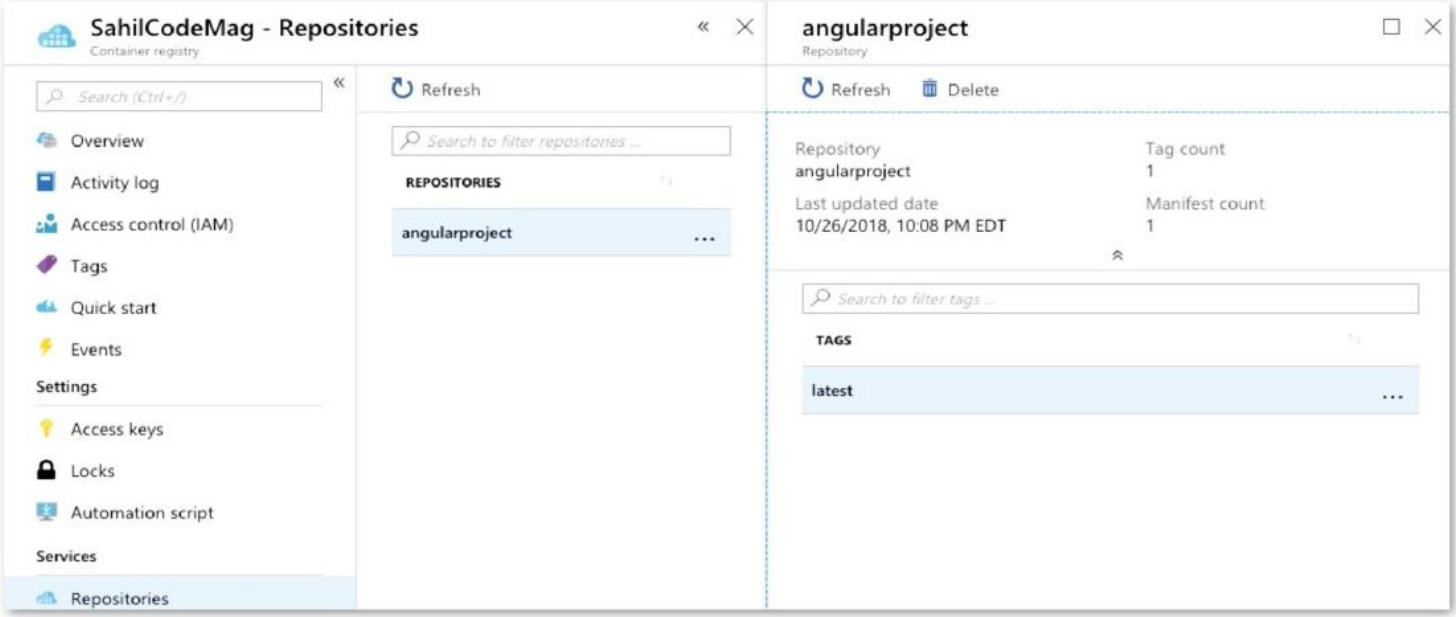


Figure 11: View the build logs.

Figure 12: Our Docker image, ready in the container registry.

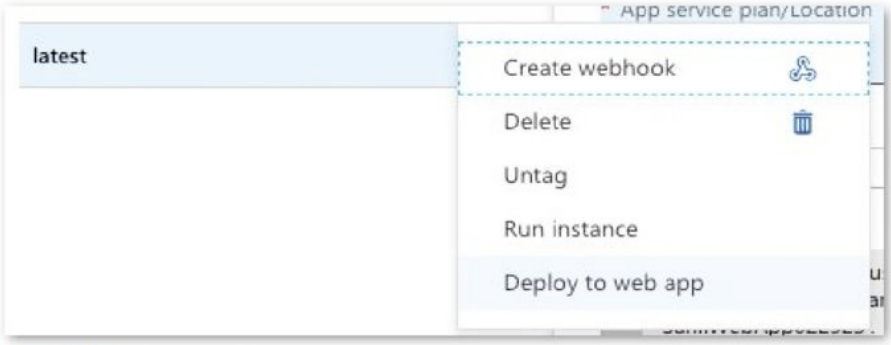


Figure 13: Deploy to a Web app.

Push an Image

Again, in the Push an Image task, accept all default values, except specify your Docker registry service connection and the Image Name. Change the image name as below:

```
$(Build.Repository.Name):latest
```

In short, you're pushing the same image that the previous task created.

Run the Pipeline

With your pipeline built, go ahead and run your pipeline. You'll see a button called "Save and Queue" or just

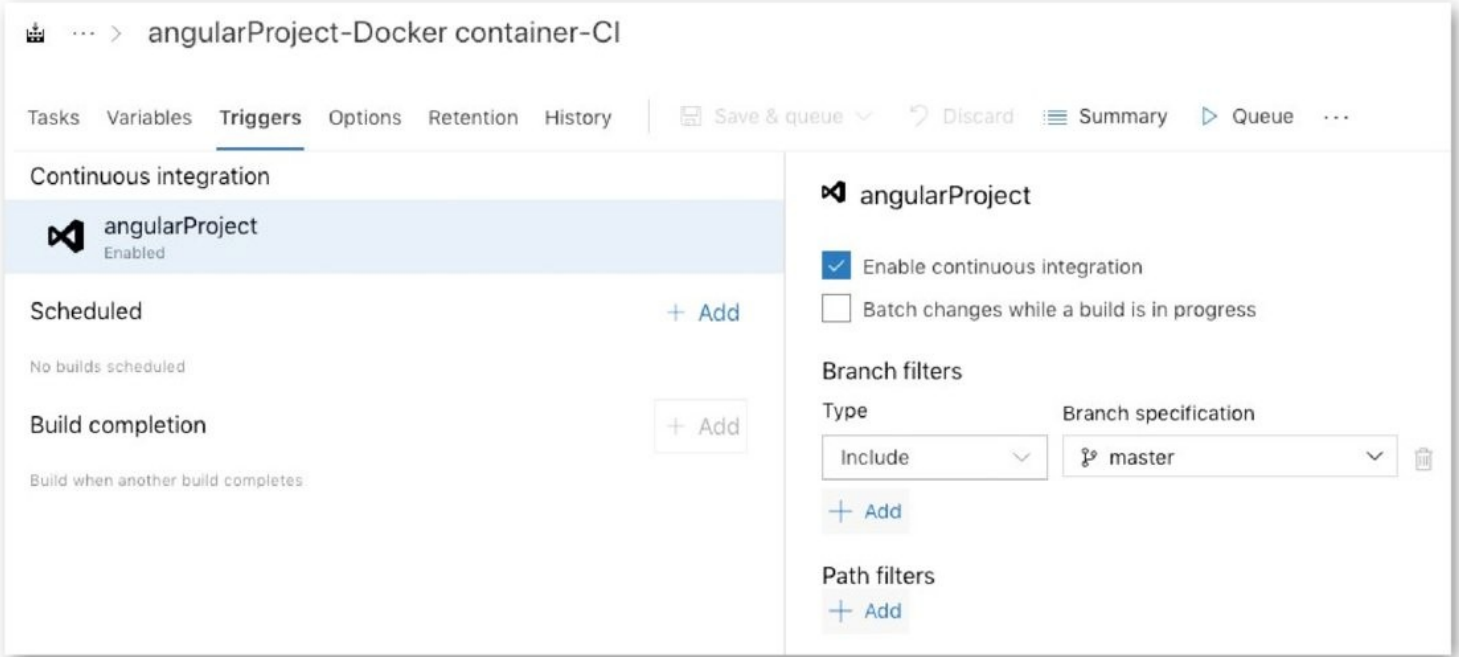


Figure 14: Enable continuous integration.

"Queue" at the top of your pipeline editor to do so. Once you queue a build, in a few moments, an agent becomes available and your pipeline executes, one by one. You can view the logs as shown in **Figure 11**.

I think it's amazing that you can view the command line output, as if this was a local dev computer. This really makes it easy to diagnose errors. If you've done everything right, all the steps should run, and at the end of this execution, you should see the **angularproject** repository created with the **latest** tag under the **SahilCodeMag** Azure Container registry you created at the start of this article. This can be seen in **Figure 12**.

Set up a Webhook

The purpose of setting up a webhook is to trigger the deployment of the Docker image to an Azure website every time a build completes. On the **latest** tag, you'll see an ellipsis. Click on that and choose to **Deploy to a Web app,** as shown in **Figure 13**.

Choosing this option walks you through the process of creating an app service plan, an app service, and creating a webhook behind the scenes.

To trigger this whole process on a check-in, you need to enable continuous integration. This is a matter of checking a checkbox, as shown in **Figure 14**.

That's basically it! Let's see if all this works.

Testing It All Out

Back in your code, make a minor code change. Add the following line to the src/app.component.html file:

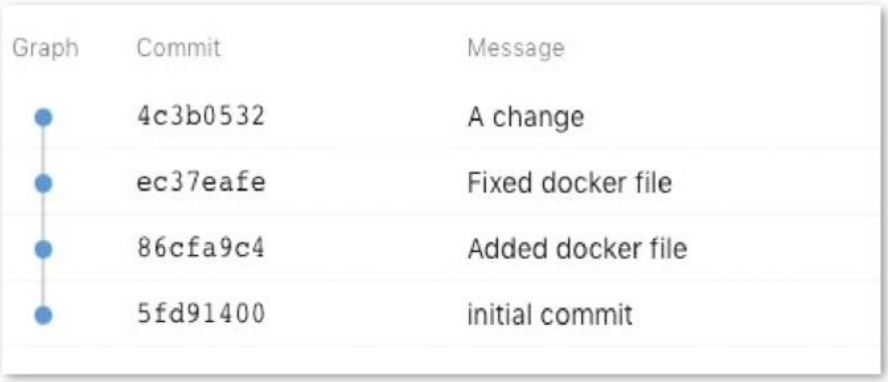


Figure 15: The new change is committed.

Figure 16: A CI build triggered.

```
<h1> I love Azure DevOps </h1>
```

You can test this locally, but I'm confident it'll work. I went ahead and committed my change and pushed it.

```
git add .
git commit -m "A change"
git push
```

I can verify that this push appears in my commits in my repository, as can be seen in **Figure 15.**

Under pipelines, you can see in **Figure 16** that my "CI Build" has auto started.

If I'm quick enough, I can quickly see the progress of my build under logs. I can actually see when and how an npm install occurs in my agent. This can be seen in **Figure 17**.

In a moment, I receive an email informing me of a successful build, which I can also verify from the site. And perhaps the best part, my site is deployed and updated for me, as can be seen in **Figure 18**.

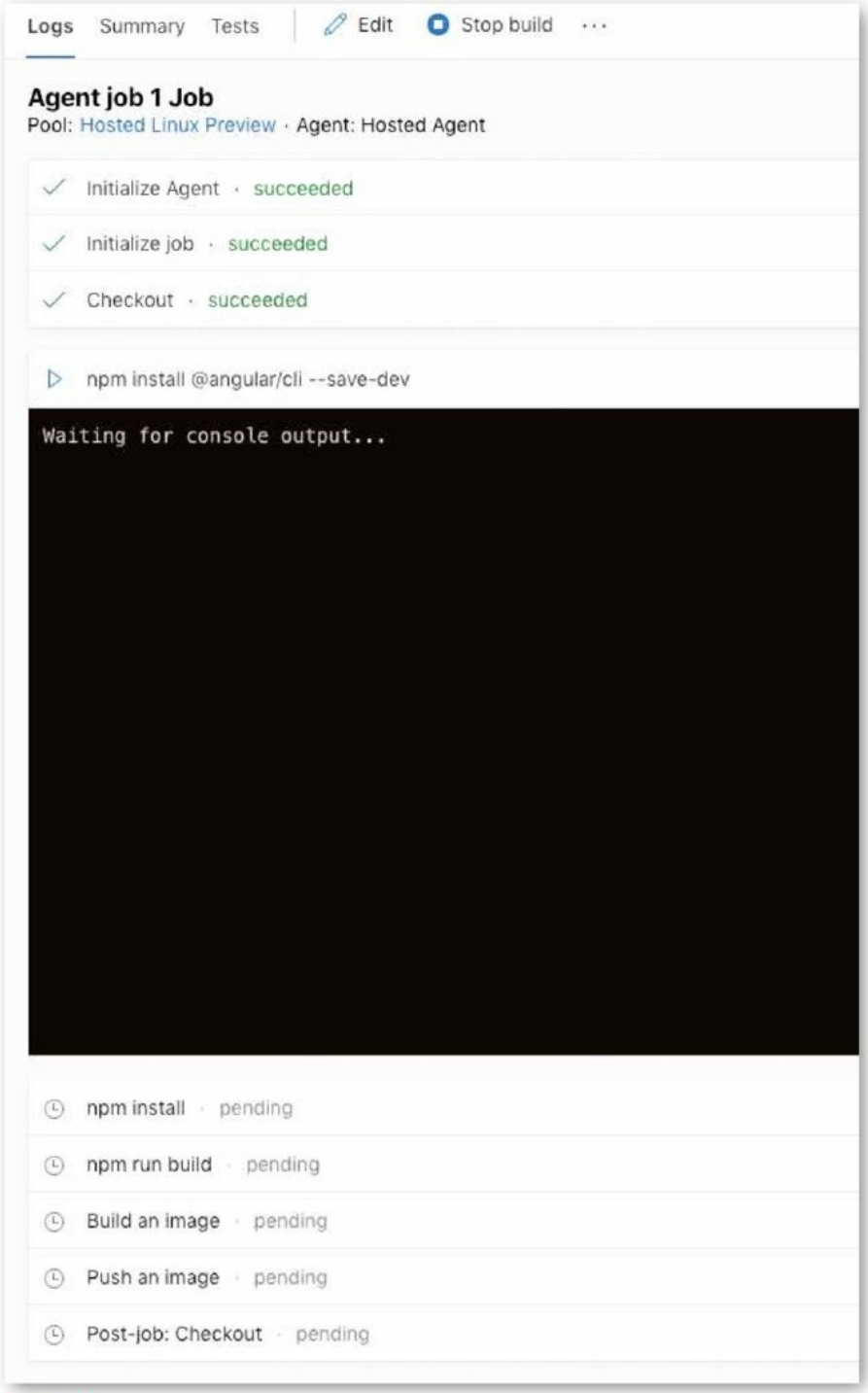


Figure 17: Live logs running

Figure 18: My site updated, thanks to continuous integration.

Summary

A long time ago, a well-meaning person was career-advising me. I've found his advice to be very useful in the years since. He said that in IT, things will be commoditized. Is this the right line of work to build a career in?

I'm one of the lucky ones! If I were the President, I'd still come home and code every day. I picked this career because there isn't ever a boring day. And this was pretty much how I felt when posed that question.

The reality is that IT is the only field where, as one concept gets commoditized, we reinvent ourselves to stand on top of it. We exponentially make progress. Just a few years ago, we were busy ordering servers so we could run our websites. We had data centers. Now we're busy shaving milliseconds from an end-to-end auto-deployed process where a Docker image gets built, shipped, versioned, and deployed in a matter of seconds.

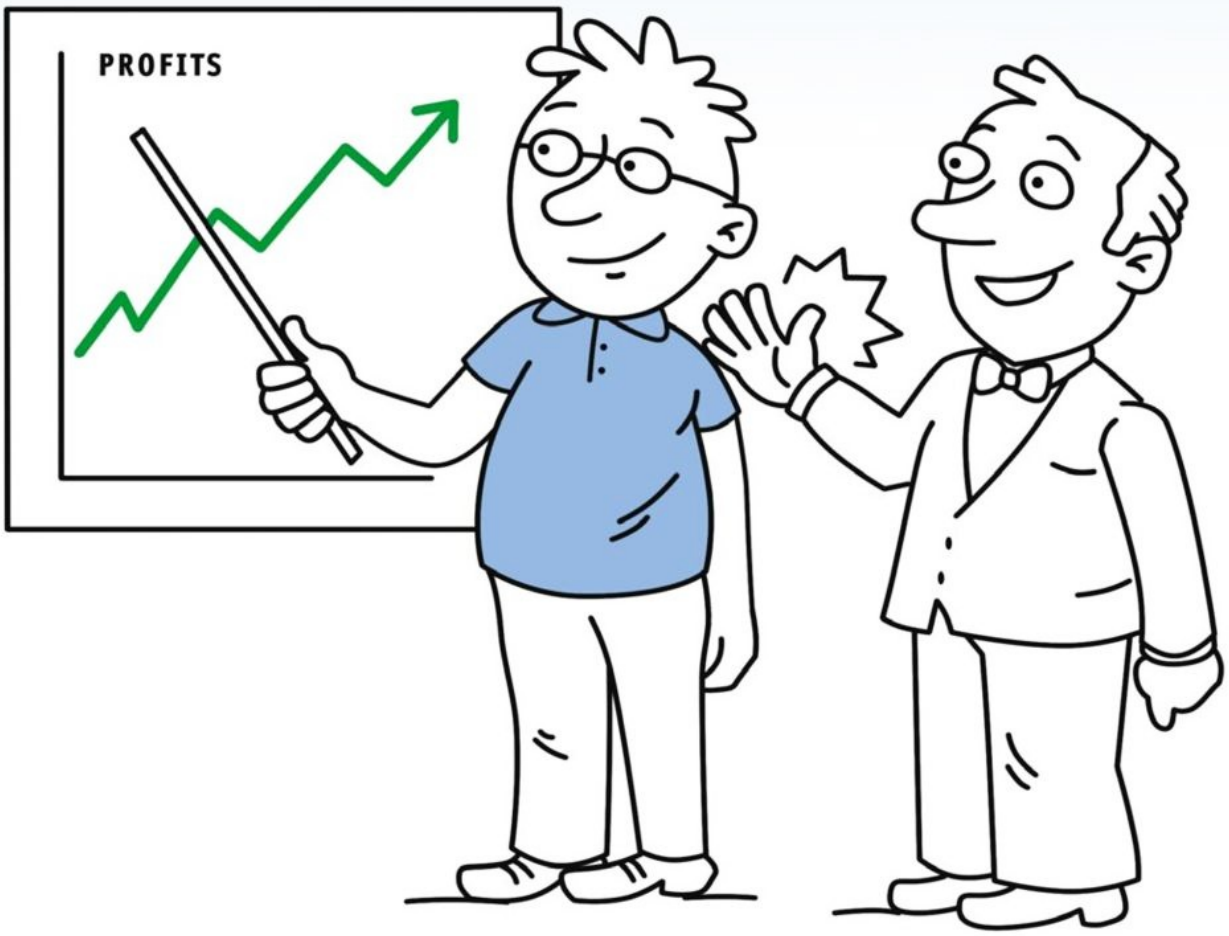
What's next? I don't know. But I know that in the future, there will be two jobs that will never go out of style. One will be IT and the other will be charging batteries.

There's so much more to explore, so stay tuned!

Sahil Malik

CODE

CODE Consulting Will Make You a Hero!



Lacking technical knowledge or the resources to keep your development project moving forward? Pulling too many all-nighters to meet impending deadlines? CODE Consulting has top-tier programmers available to fill in the technical and manpower gaps to make your team successful! With in-depth experience in .NET, Web, Azure, Mobile and more, CODE Consulting can get your software project back on track.

Contact us today for a free 1-hour consultation to see how we can help you succeed.

Helping Companies Build Better Software Since 1993

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

CODE CONSULTING

Implementing Push Notifications in Progressive Web Apps (PWAs) Using Firebase

Progressive Web Apps (PWAs) are Web applications that behave like native mobile applications. PWAs leverage Google technology to deliver an extremely engaging mobile experience. PWAs live within the Web browser and have the ability to work even if the device is offline. At the same time, PWAs exhibit the behavior that you've come to expect from native mobile apps:

Wei-Meng Lee
weimenglee@learn2develop.net
www.learn2develop.net
@weimenglee

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

Figure 1: Images to represent the state of push notification of the app.

They support background content updating, push notifications, and more. More importantly, PWAs can be updated without needing to resubmit your applications to the app stores (both Apple and Google's).

In this article, I'll discuss one of the most interesting and important features of PWA: Push Notifications. I'll show you how to enable push notifications in your PWA application using Firebase Cloud Messaging (FCM). And, as a bonus, I'll show you how to host your PWA on Firebase Hosting, as well as how to host your REST API as a serverless app through Firebase Functions. Keep your seat belt on, as this is going to be a very fun (and exciting) journey!

Creating the Project

Let's first start by creating a new folder named **PWAPUSH**. This will be used to store the files for the PWA application. The first set of files to copy into this folder is **button_on.png** and **button_off.png** (see **Figure 1**). These images will be used to represent the status of the Web app's push notification: red to indicate that the push notification is ON and grey to indicate that the push notification is off.

Creating the Web Page

Next, create a file named **index.html** and save it in the **PWAPUSH** folder. Populate it with the content shown in **Listing 1**. The **index.html** page displays a line of text as well as the image of a button to represent the state of push notification.

Creating the Service Worker

The next step is to create the service worker for the PWA. Create a file named **service-worker.js** and save it in the **PWAPUSH** folder. Leave it empty for now.

Using a Web browser, go to https://app-manifest.firebaseapp.com/ so that you can generate the Web App Manifest for your app. Fill in the information as shown in **Figure 2** and then drag and drop the **button_on.png** image onto the **ICON** button. Once this is done, click the **GENERATE .ZIP** button.

The **App-images.zip** file will now be downloaded. Unzip it and drag and drop its content (a file named **manifest.json** and a folder named **images**) onto the **PWAPUSH** application folder (see **Figure 3**).

Using Firebase Cloud Messaging for Push Notifications

To enable your Web app to receive push notifications, you're going to make use of Firebase. Firebase is a BAAS (Back-end As A Service). Today, most applications require back-end services, such as from a database server or Web server, in order to make the application useful. Rather than host your own back-end, you can turn to Firebase for a suite of back-end services, such as:

- Cloud Messaging (also known as push notifications)
- Database
- Hosting
- Machine learning

In this article, you'll learn how to:

- Host your Web application using Firebase Hosting
- Host your Web services using Firebase Functions
- Send push notifications using Firebase Cloud Messaging

Using a Web browser, go to https://console.firebase.google.com and sign in using your Google account. Click the **+** icon to add a new project and give a name to the project. Click **CREATE PROJECT** (see **Figure 4**). Take note of the Project ID.

When the project is created, click **CONTINUE**. Click on the **gear** icon displayed next to the **Project Overview** item and select **Project settings** (see **Figure 5**).

Then, click on the **Cloud Messaging** tab (see **Figure 6**).

Take note of the following information (you'll need it in the next step and in a later section):

- Server Key
- Sender ID

In the **manifest.json** file located in the **PWAPUSH** folder, add the following statement in bold (replace the **SENDER_ID** with the Sender ID you obtained from the previous step):

```
{
  "name": "Push Notification App",
  "short_name": "PushApp",
  "theme_color": "#2196f3",
```

Listing 1: Content of index.html

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <meta name="viewport"
            content="width=device-width, initial-scale=1.0">
        <meta http-equiv="X-UA-Compatible" content="ie=edge">
        <title>Push Notification</title>
        <link rel="manifest" href="manifest.json">
    </head>
    <body>
        Tap to enable/disable Push Notification
        <div class="push">
            <img width="50px" class="image" src="button_on.png"
                alt="Push Notification" />
        </div>
        <script src="main.js"></script>
    </body>
</html>
```

```
  "background_color": "#2196f3",
  "display": "browser",
  "Scope": "/",
  "start_url": "/",
  "icons": [
    ...
    {
      "src": "images/icons/icon-512x512.png",
      "sizes": "512x512",
      "type": "image/png"
    }
  ],
  "splash_pages": null,
  "gcm_sender_id": "SENDER_ID"
}
```

Subscribing to Push Notifications

Create a file named **main.js** and save it in the **PWAPush** folder. Populate it as shown in **Listing 2.** It contains the following:

- The **load** event handler to register the service worker
- The **click** event handler to handle the user clicking on the button image
- The **checkIfPushIsEnabled()** function to check whether the user has previously blocked push notifications and if the browser being used supports push notification, and retrieves the push notification subscription ID if the application has previously subscribed to push notification
- The **updatePushNotificationStatus()** function to display the image of the button depending on the status of the push notification subscription
- The **subscribeToPushNotification()** function to subscribe the application to push notification
- The **unsubscribeFromPushNotification()** function to unsubscribe the application from push notification

You can now test the Web application and verify whether the application can subscribe to push notification. To run the Web application over HTTP, type the following commands in Terminal:

```
$ cd ~/PWAPUSH
$ npm install -g serve
$ serve
```

The above command installs a command-line Web server (**serve**) that allows you to serve your Web pages directly from any directory on your file system. Once you run the

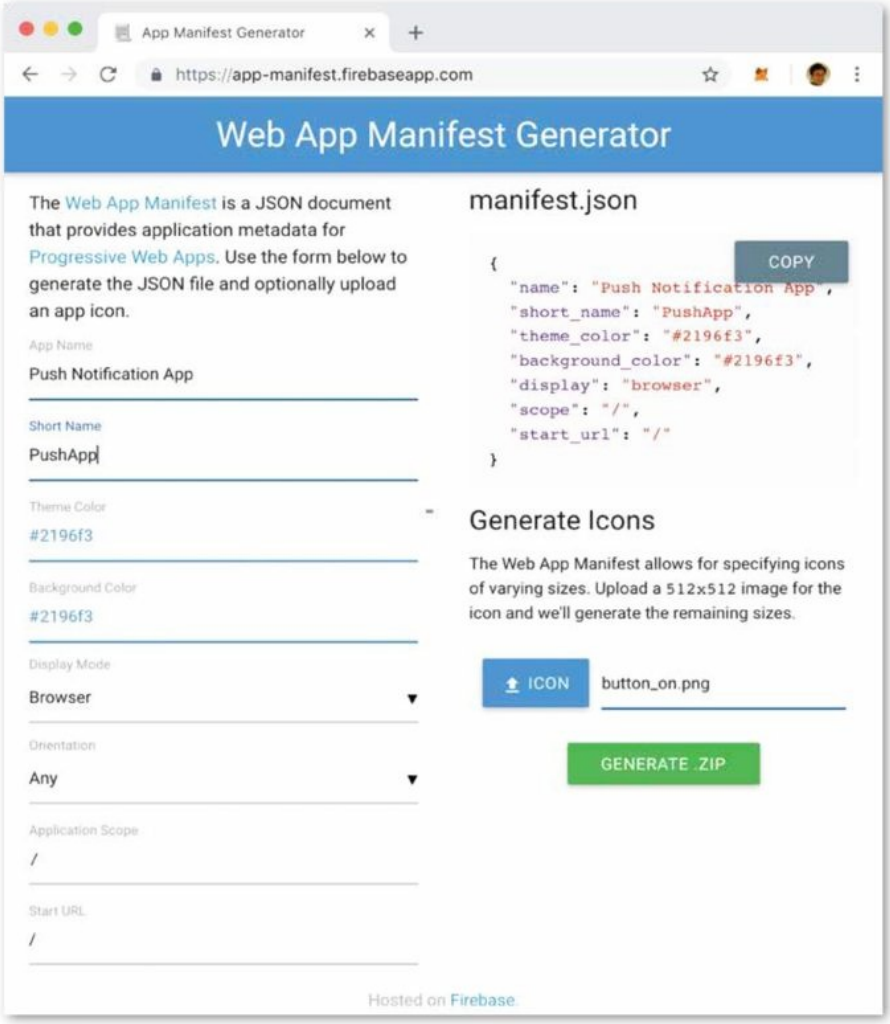


Figure 2: Generating the Web App Manifest.

Figure 3: Adding the two files in the ZIP package onto the PWAPUSH folder.

Listing 2: Content of main.js

```
//---register the Service Worker---
window.addEventListener('load', e => {
    if (!('serviceWorker' in navigator)) {
        console.log('Service worker not supported');
        return;
    }
    navigator.serviceWorker.register('service-worker.js')
    .then(function() {
        console.log('Service Worker Registered');
    })
    .catch(function(error) {
        console.log('Service Worker Registration failed:',
        error);
    });
});

//---Update the Push Notification Status---
function updatePushNotificationStatus(status) {
    pushElement.dataset.checked = status;
    if (status) {
        pushImage.src = 'button_on.png';
    }
    else {
        pushImage.src = 'button_off.png';
    }
}

function checkIfPushIsEnabled() {
    //---check if push notification permission
    // has been denied by the user---
    if (Notification.permission === 'denied') {
        alert('User has blocked push notification.');
        return;
    }
    //---check if push notification is
    // supported or not---
    if (!('PushManager' in window)) {
        alert('Sorry, Push notification is ' +
              'not supported on this browser.');
        return;
    }
    //---get push notification subscription
    // if serviceWorker is registered and ready---
    navigator.serviceWorker.ready
    .then(function (registration) {
        registration.pushManager.getSubscription()
        .then(function (subscription) {
            if (subscription) {
                //---user is currently subscribed to push---
                updatePushNotificationStatus(true);
            }
            else {
                //---user is not subscribed to push---
                updatePushNotificationStatus(false);
            }
        })
        .catch(function (error) {
            console.error(
                'Error occurred enabling push ',
                error);
        });
    });
}

//---subscribe to push notification---
function subscribeToPushNotification() {
    navigator.serviceWorker.ready
    .then(function(registration) {
        if (!registration.pushManager) {
            alert(
                'This browser does not ' +
                'support push notification.');
            return false;
        }
        //---to subscribe push notification using
        // pushmanager---
        registration.pushManager.subscribe(
            //---always show notification when received---
            { userVisibleOnly: true }
        )
        .then(function (subscription) {
            console.log('Push notification subscribed.');
            console.log(subscription);
            updatePushNotificationStatus(true);
        })
        .catch(function (error) {
            updatePushNotificationStatus(false);
            console.error(
                'Push notification subscription error: ',
                error);
        });
    })
}

//---unsubscribe from push notification---
function unsubscribeFromPushNotification() {
    navigator.serviceWorker.ready
    .then(function(registration) {
        registration.pushManager.getSubscription()
        .then(function (subscription) {
            if(!subscription) {
                alert('Unable to unsubscribe from push ' +
                      'notification.');
                return;
            }
            subscription.unsubscribe()
            .then(function () {
                console.log('Push notification unsubscribed.');
                console.log(subscription);
                updatePushNotificationStatus(false);
            })
            .catch(function (error) {
                console.error(error);
            });
        })
        .catch(function (error) {
            console.error('Failed to unsubscribe push ' +
                          'notification.');
        });
    })
}

//---get references to the UI elements---
var pushElement = document.querySelector('.push');
var pushImage = document.querySelector('.image');

//---event handler for the push button---
pushElement.addEventListener('click', function () {
    //---check if you are already subscribed to push
    // notifications---
    if (pushElement.dataset.checked === 'true') {
        unsubscribeFromPushNotification();
    }
    else {
        subscribeToPushNotification();
    }
});

//---check if push notification is supported---
checkIfPushIsEnabled()
```

serve command, the current directory becomes the Web-publishing directory.

> For this article, you need to install Node.js.

Using the Chrome browser, load the **index.html** page using the following URL: http://localhost:5000 (see **Figure 7**). Be sure to bring up the JavaScript Console (**Option-Cmd-J**).

Click on the button and you'll see the popup as shown in **Figure 8**. Click **Allow**.

You should now see the subscription ID in the **JavaScript Console** (see **Figure 9**).

Creating the REST API to Save the Subscription IDs

When your app subscribes to push notifications, it gets a subscription ID. This subscription ID is used to uniquely identify the application so that Firebase Cloud Messaging can deliver a notification to the application. In order to deliver a push notification to the application, you need to save the subscription IDs. To do that, use Node.js to build a REST API for archiving the subscription ID obtained by the app.

Create a new folder named **PUSHSERVER** to save the files for the REST API. Create a new file named **SubscriptionIDServer.js** and save it in the **PUSHSERVER** folder. Populate it with the content shown in **Listing 3**.

> For simplicity, you're saving the subscriber IDs to an array in the REST API; in the real world, you need to save this to a database such as Firestore.

The REST API has the following functionalities:

- You can add a subscription ID to the API by sending the subscription ID using the **POST** method via the URL **/subscribers**.
- You can remove a subscription ID from the API by sending the subscription ID using the **DELETE** method via the URL **/subscribers/*subscriptionID***.

Before you run the REST API, you need to install two components:

- **Express**: a Node.js Web framework for building REST APIs.
- **body-parser**: extracts the entire body portion of an incoming request stream and exposes it on req.body.

```
$ cd ~/PUSHSERVER
$ npm install express
$ npm install body-parser
```

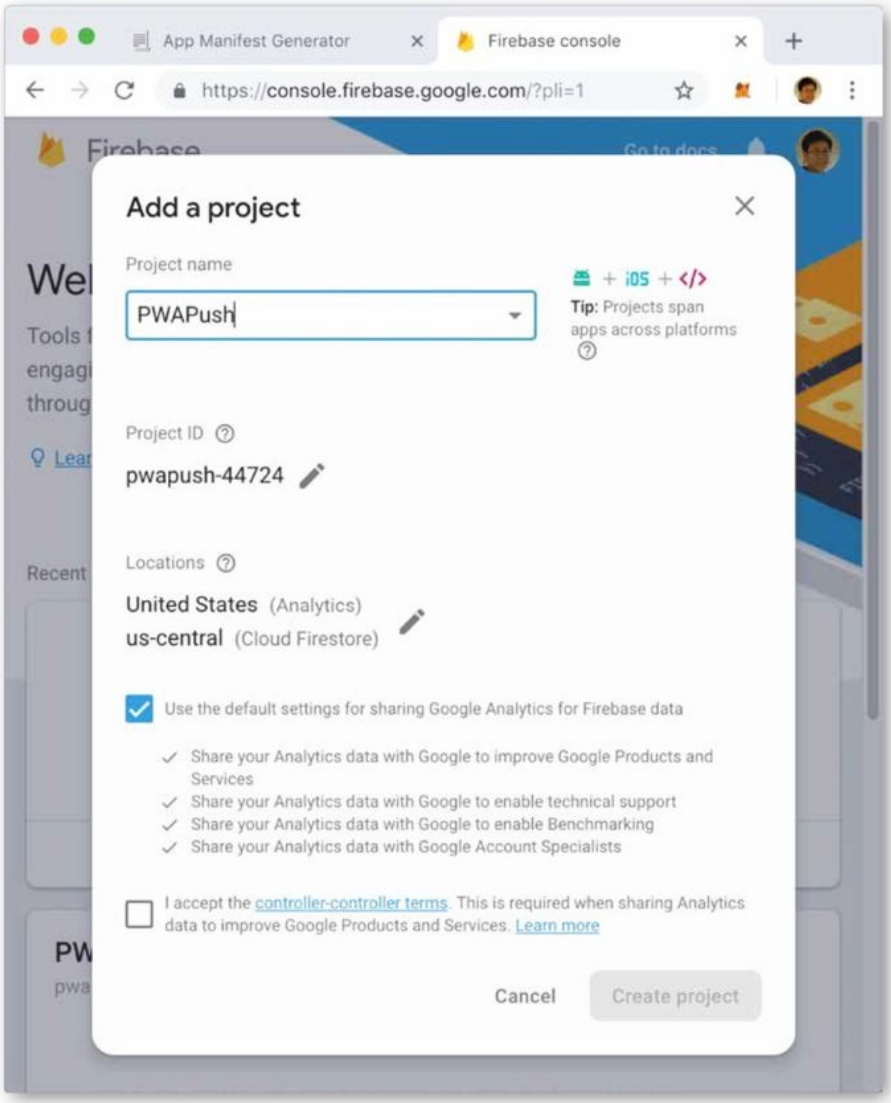


Figure 4: Creating a new project in Firebase.

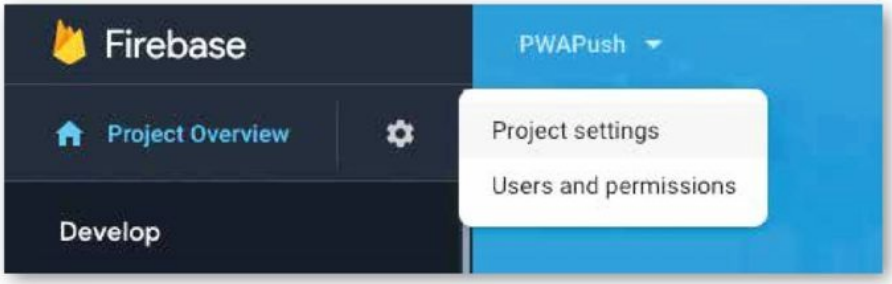


Figure 5: Configuring the project via Project settings.

Figure 6: Selecting the Cloud Messaging tab.

Listing 3: Content of SubscriptionIDServer.js

```
var express = require('express');

//---returns an instance of the express server---
var app = express();

//---parse the body of the request and set the
// body property on the request object---
var bodyParser = require('body-parser');

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({
    extended: true
}));

app.use(function (req, res, next) {
    res.setHeader('Access-Control-Allow-Origin', '*');
    res.setHeader('Access-Control-Allow-Headers',
                  'X-Requested-With,content-type');
    res.setHeader('Access-Control-Allow-Methods',
                  'GET,PUT,POST,DELETE');
    next();
})

//---the array to store all the subscriber IDs---
var subscribers = []

app.post('/subscribers/', function(req, res) {
    //---check that the regid field is there---
    if (!req.body.hasOwnProperty('subscriptionid')){
        res.statusCode = 400;
        res.send('Error 400: Post syntax incorrect.');
        return;
    }
    console.log(req.body.subscriptionid);
    subscribers.push(req.body.subscriptionid)
    res.statusCode = 200;
    res.send('SubscriptionID received');
});

app.delete('/subscribers/:id', function(req, res) {
    console.log(req.params.id)
    const index = subscribers.indexOf(req.params.id)
    if (index !== -1) {
        subscribers.splice(index,1)
    }
    res.statusCode = 200;
    res.send('SubscriptionID deleted');
});

app.listen(8080);
console.log('Rest Service Listening on port 8080');
```

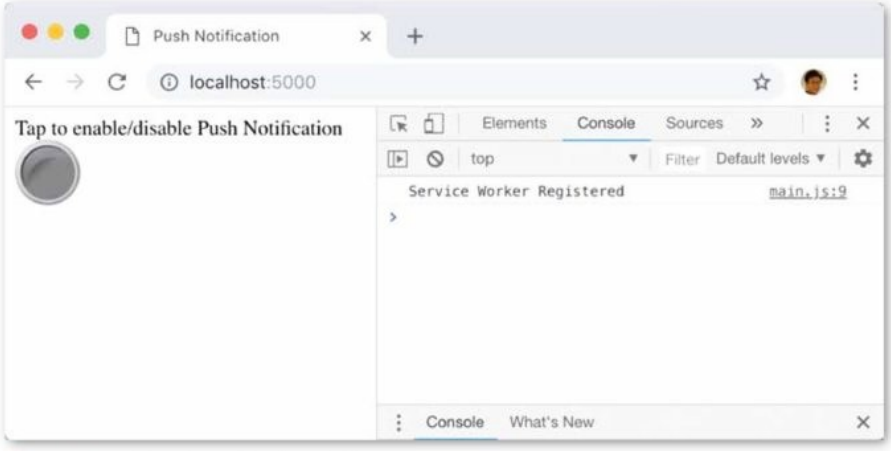


Figure 7: Loading the application in Chrome.

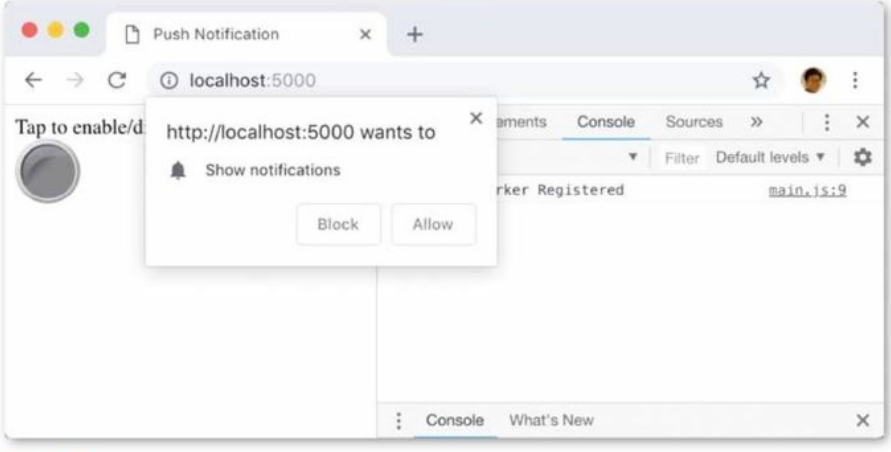


Figure 8: Giving permission to the app to receive push notifications.

Once the components are installed, you can run the REST API using the following command:

```
$ node SubscriptionIDServer.js
```

You'll see the following

```
Rest Service Listening on port 8080
```

Sending the Subscription ID to the REST API

With the REST API up and running, it's time to modify the **main.js** file located in the **PWAPUSH** folder so that once the subscription is obtained, it can be sent to the REST API for archival. Similarly, when a user wants to unsubscribe from a push notification, the subscription ID must be removed from the REST API.

Add the statements in bold to the **main.js** file, as shown in **Listing 4**.

Once you have modified the **main.js** page, you can reload the page on the Chrome browser and click the push button again. Observe that the subscription ID is now printed in the JavaScript Console window (see **Figure 10**).

In the Terminal running the REST API, you should also see the subscription ID printed (see **Figure 11**). This proves that the subscription ID has been sent successfully to the REST API.

If you click the push button again, you should see the same subscription ID printed in the JavaScript console window and on the Terminal. This time, the subscription ID is removed from the REST API.

Sending Push Notifications Through Firebase Cloud Messaging

Great! So far, you've managed to save the subscription IDs to the REST API. It's now time to push a notification to the Web application. For this, you'll use Firebase Cloud Messaging (FCM). You'll use the REST API to push the message to the application through the FCM. The subscription ID will be used to identify which application to push the notification to. **Figure 12** shows the flow.

To send an FCM push notification through Node.js, you can use the **fcm-node** module. To install this module, type the following command in Terminal:

```
$ cd ~/PUSHSERVER
$ npm install fcm-node
```

You can now modify the **SubscriptionIDServer.js** file as shown in **Listing 5** so that it has a function to send push notifications.

> To send an FCM message to multiple recipents, use the "registration_ids" key. To send to an individual recipient, use the "to" key.

Listing 4: Adding the functions to send subscription IDs to the REST API

```
...
function checkIfPushIsEnabled() {
    ...
}

//---extract the subscription id and send it
// over to the REST service---
function sendSubscriptionIDToServer(subscription) {
    var subscriptionid =
        subscription.endpoint.split('gcm/send/')[1];
    console.log("Subscription ID", subscriptionid);
    fetch('http://localhost:8080/subscribers', {
        method: 'post',
        headers: {
            'Accept': 'application/json',
            'Content-Type': 'application/json'
        },
        body: JSON.stringify(
            { subscriptionid : subscriptionid })
    });
}

//---extract the subscription id and send it
// over to the REST service---
function removeSubscriptionIDFromServer(subscription) {
    var subscriptionid =
        subscription.endpoint.split('gcm/send/')[1];
    console.log("Subscription ID", subscriptionid);
    fetch('http://localhost:8080/subscribers/' +
          subscriptionid, {
        method: 'delete',
        headers: {
            'Accept': 'application/json',
            'Content-Type': 'application/json'
        }
    });
}

//---subscribe to push notification---
function subscribeToPushNotification() {
    navigator.serviceWorker.ready
    .then(function(registration) {
        if (!registration.pushManager) {
            alert(
                'This browser does not ' +
                'support push notification.');
            return false;
        }
        //---to subscribe push notification using
        // pushmanager---
        registration.pushManager.subscribe(
            //---always show notification when received---
            { userVisibleOnly: true }
        )
        .then(function (subscription) {
            console.log('Push notification subscribed.');
            console.log(subscription);
            //------add the following statement------
            sendSubscriptionIDToServer(subscription);
            //----------------------------------------
            updatePushNotificationStatus(true);
        })
        .catch(function (error) {
            updatePushNotificationStatus(false);
            console.error(
                'Push notification subscription error: ',
                error);
        });
    })
}

//---unsubscribe from push notification---
function unsubscribeFromPushNotification() {
    navigator.serviceWorker.ready
    .then(function(registration) {
        registration.pushManager.getSubscription()
        .then(function (subscription) {
            if(!subscription) {
                alert('Unable to unsubscribe from push ' +
                      'notification.');
                return;
            }
            subscription.unsubscribe()
            .then(function () {
                console.log('Push notification unsubscribed.');
                console.log(subscription);
                //--------add the following statement--------
                removeSubscriptionIDFromServer(subscription);
                //-------------------------------------------
                updatePushNotificationStatus(false);
            })
            .catch(function (error) {
                console.error(error);
            });
        })
        .catch(function (error) {
            console.error('Failed to unsubscribe push ' +
                          'notification.');
        });
    })
}
```

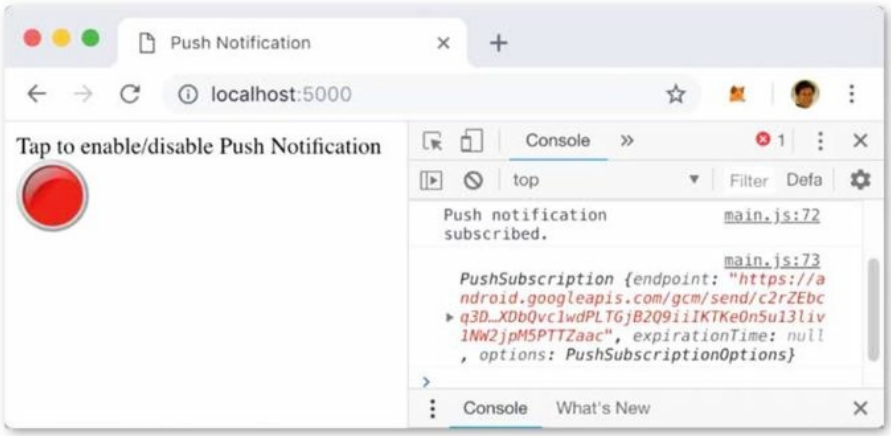


Figure 9: The subscription ID displayed in the JavaScript console.

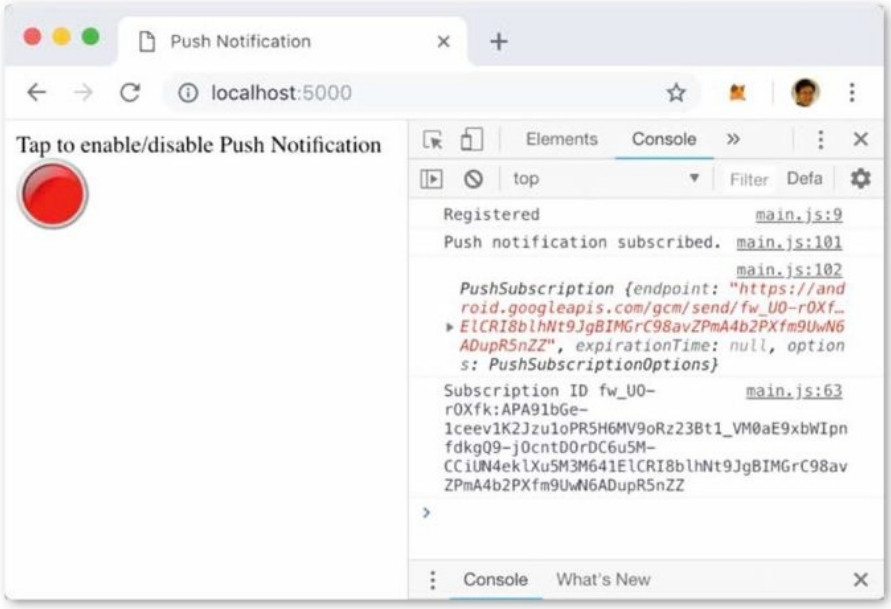


Figure 10: Printing the subscription ID in the JavaScript console.

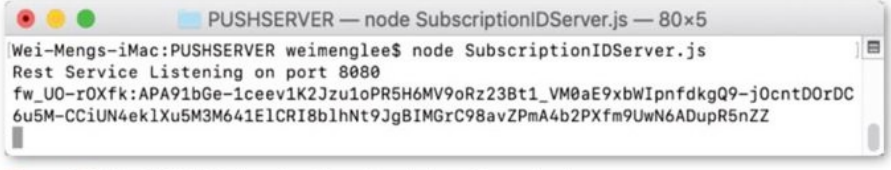


Figure 11: The REST API showing the subscription ID received.

Figure 12: The flow showing how the push notification is delivered.

Testing the Application

You can now finally test the application. Refresh the **index.html** page on the Chrome browser and make sure you click the push button until it becomes red. This is to ensure that the subscription ID of the application is saved on the REST API.

In a *new* **Terminal** window, type the following command:

```
$ curl -v http://localhost:8080/push
```

This causes the REST API to send a push notification via FCM. You should now see a notification displayed on your computer (like **Figure 13**).

Clicking the notification displays a new page (see **Figure 14**).

Hosting the Web Application and REST API Using Firebase Hosting and Functions

Until this point, you've been hosting the Web application and REST API locally on your computer. A better idea would be to host them on a real Web server so that the application can be used by anyone.

In the following sections, you will do the following:

- Host the Web application using Firebase Hosting
- Host the REST API using Firebase Function

Hosting Using Firebase Hosting

Let's first host the Web application using Firebase Hosting. Create a new folder named **FirebaseHosting**. This will be used to store all the files that will be uploaded and hosted by Firebase Hosting.

Type the following commands in **Terminal**:

```
$ cd ~/FirebaseHosting
$ npm install -g firebase-tools
$ firebase init
```

You will see a bunch of text. Move the cursor until you've selected the **Hosting: Configure and deploy Firebase Hosting sites** option. Then, press **Space** and press **Enter**.

Next, select the name of the Firebase project (the Project ID) that you created earlier when you registered for Firebase. In my case, the project name is **PWAPush (pwapush-44724)**. Select the project name and then press **Space** and press **Enter**.

You will next be asked to enter a folder name to store your project files. Press enter to use the default name of **public**. The Firebase tool will now create a folder named **public**.

Next, you will be asked if you want to configure the app as a single-page app. Type **No** and press **Enter**.

The **FirebaseHosting** folder now has the following items created for you:

- A **firebase.json** file
- A **public** folder, which contains the following files:
  - **404.html** file
  - **index.html** file

Listing 5: Adding the code to send push notifications from the REST API

```
var express = require('express');

...

//---the array to store all the subscriber IDs---
var subscribers = []

var FCM = require('fcm-node');
//----------------------------------------
//---replace [Server Key] with the Server Key
// you obtained earlier---
//----------------------------------------
var serverKey = 'SERVER_KEY';
var fcm = new FCM(serverKey);

app.get('/push', function(req, res) {
    var message = {
        // for multiple recipients
        registration_ids: subscribers,
        // for single recipient
        // to: subscribers[0],
        collapse_key: 'your_collapse_key',
    };
    fcm.send(message, function(err, response){
        if (err) {
            console.log("Something has gone wrong!");
            console.log(err)
        } else {
            console.log("Successfully sent with response: ",
                response);
        }
    });
    res.sendStatus(200)
});

app.post('/subscribers/', function(req, res) {
    ...
});

app.delete('/subscribers/:id', function(req, res) {
    ...
});

app.listen(8080);
console.log('Rest Service Listening on port 8080');
```

Listing 6: Receiving push notifications in the service worker

```
self.addEventListener('push', function(event) {
    console.info('Event: Push');
    var title = 'Breaking News';
    var body = {
        'body': 'Click to see the latest breaking news',
        'tag': 'pwa',
        'icon': './images/48x48.png'
    };
    event.waitUntil(
        self.registration.showNotification(title, body)
    );
});

self.addEventListener('notificationclick', function(event) {
    //---access data from event
    // using event.notification.data---
    console.log('On notification click: ',
        event.notification.data);
    var url = './breaking.html';

    //---close the notification---
    event.notification.close();

    //---open the app and navigate to breaking.html
    // after clicking the
    // notification---
    event.waitUntil(
        clients.openWindow(url)
    );
});
```

Copy all the files in the **PWAPUSH** folder into the **PWAPUSH/public** folder, overwriting the **index.html** created by Firebase. The content of the **public** folder should now be as shown in **Figure 15**.

For Windows users, if you get an error pertaining to $RESOURCE_DIR during Firebase deployment, edit the firebase.json file and make the following changes: `npm –prefix %RESOURCE_DIR% run lint`

You're now ready to deploy the content of the public folder to Firebase Hosting. Type the following commands in Terminal:

```
$ cd ~/FirebaseHosting/public
$ firebase deploy
```

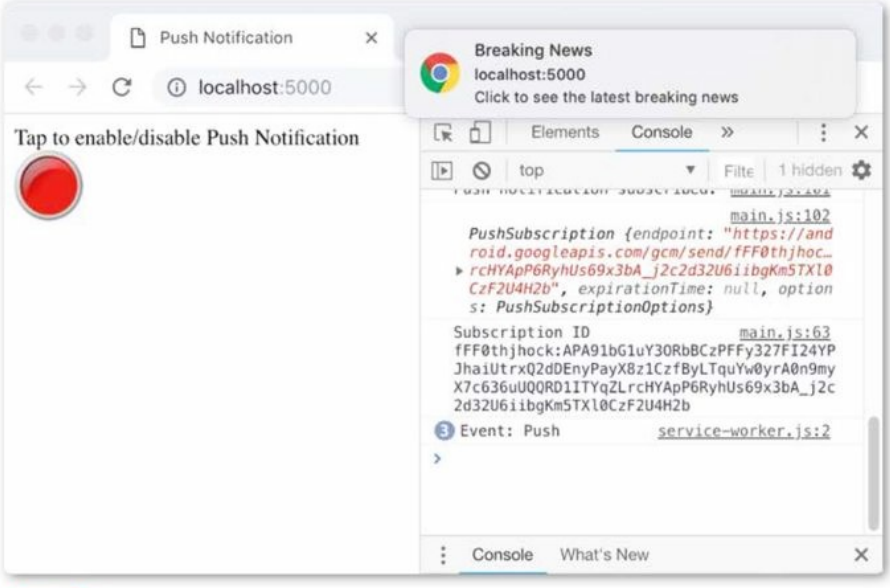


Figure 13: Receiving a push notification on the computer.

At the end of the deployment, you'll see the following:

```
Project Console:
https://console.firebase.google.com/project/
pwapush-44724/overview
Hosting URL:
https://pwapush-44724.firebaseapp.com
```

The above highlighted URL is the location of your hosted website.

You're now ready to test the application that's hosted on Firebase Hosting. Using Chrome, load the page as shown in **Figure 16**.

Hosting Using Firebase Functions

Now that the Web application is deployed to Firebase Hosting, it's time to deploy the REST API to Firebase Functions. Firebase Functions lets you automatically run back-end code in response to events triggered by HTTPS requests. Your code is stored in Google's cloud and runs in a managed environment. There's no need to manage and scale your own servers. The application that you host on Firebase Functions is known as a **serverless app**.

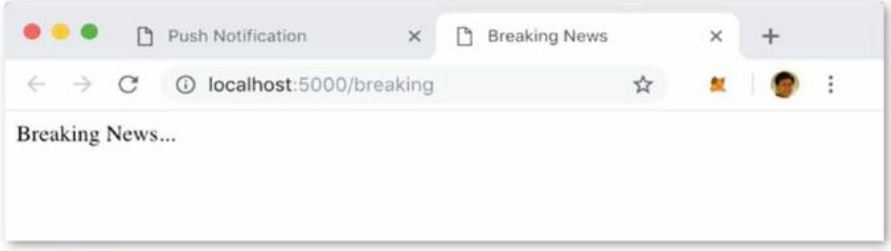


Figure 14: Clicking on the notification displays a new page.

Figure 15: The content of the public folder.

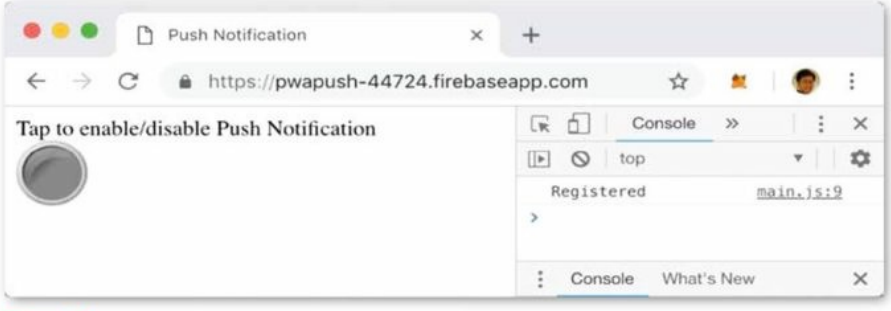


Figure 16: Using Chrome to load the application hosted on Firebase Hosting.

To get started with Firebase Functions, type the following command in Terminal:

```
$ cd ~/FirebaseHosting
$ firebase init functions
```

> Note the use of https:// by Firebase Hosting. This is one of the nice features of Firebase Hosting, where you don't need to configure SSL for your site.

You'll be asked what language you'll be using. Select **JavaScript**. You'll also be asked if you want to use ESLint to catch probable bugs and enforce style. Enter **Yes** and press **Enter**.

Next, enter **Yes** to install dependencies with **npm**. Once the installation is complete, The **FirebaseHosting** folder should have the **functions** folder with the files, as shown in **Figure 17**.

> For simplicity, in this example you'll use an array to store the subscription IDs. However, note that for serverless apps, content of an array is not persistent and you should use a database for storage in real life applications.

The **index.js** file is going to contain your REST API code in JavaScript. It's going to be very similar to the **SubscriptionIDServer.js** file that you created earlier. The key difference is that you no longer need to use the **express** module to listen at a particular port number. Instead, you use the **firebase-functions** module and call the **functions.https.onRequest()** function to handle incoming HTTP events. Populate the **index.js** file as shown in **Listing 7**.

Next, you need to install the dependencies needed by your serverless app. Type the following commands in Terminal:

```
$ cd ~/FirebaseHosting/functions
$ npm install express --save
$ npm install body-parser --save
$ npm install fcm-node --save
```

The above commands install all the modules needed by **index.js** and at the same time update the dependencies in the **package.json** file.

Add the following statements in bold to the **firebase.json** file located in the **FirebaseHosting** folder:

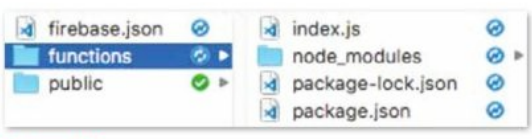


Figure 17: The content of the functions folder.

```
{
  "hosting": {
    "public": "public",
    "rewrites": [
      {
        "source": "/**",
        "function": "app"
      }
    ],
    "ignore": [
      "firebase.json",
      "**/.*",
      "**/node_modules/**"
    ]
  },
  "functions": {
    "predeploy": [
      "npm --prefix \"$RESOURCE_DIR\" run lint"
    ]
  }
}
```

The final step is to deploy the content of the **functions** folder to Firebase Functions. Type the following commands in Terminal:

```
$ firebase deploy
```

At the end of the deployment, you should see the following:

```
Project Console:
https://console.firebase.google.com/
project/pwapush-
2c8aa/overview
Hosting URL:
https://pwapush-44724.firebaseapp.com
```

The above highlighted URL is the location of your hosted Node.js application. Now that the REST API is hosted on Firebase Functions, you need to update your **main.js** file so that the subscription ID is sent to the app on Firebase Functions instead of localhost. In the **main.js** file located in the **FirebaseHosting/public** folder, make the modifications in bold as shown in **Listing 8**.

Listing 7: Content of the index.js file

```
const functions = require('firebase-functions');

var express = require('express');

//---returns an instance of express server---
var app = express();

//---parse the body of the request and set the
// body property on the request object---
var bodyParser = require('body-parser');

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({
    extended: true
}));

app.use(function (req, res, next) {
    res.setHeader('Access-Control-Allow-Origin', '*');
    res.setHeader('Access-Control-Allow-Headers',
                  'X-Requested-With,content-type');
    res.setHeader('Access-Control-Allow-Methods',
                  'GET,PUT,POST,DELETE');
    next();
})

//---replace this with a database in real-life app---
var subscribers = []

var FCM = require('fcm-node');

//--------------------------------------------
//---replace Server Key with the Server Key you
// obtained earlier
var serverKey = 'Server_Key';
//--------------------------------------------

var fcm = new FCM(serverKey);

app.post('/subscribers/', function(req, res) {
    //---check that the regid field is there---
    if (!req.body.hasOwnProperty('subscriptionid')){
        res.statusCode = 400;
        res.send('Error 400: Post syntax incorrect.');
        return;
    }
    console.log(req.body.subscriptionid);
    subscribers.push(req.body.subscriptionid)
    res.statusCode = 200;
    res.send('SubscriptionID received');
});

app.delete('/subscribers/:id', function(req, res) {
    console.log(req.params.id)
    const index = subscribers.indexOf(req.params.id)
    if (index !== -1) {
        subscribers.splice(index,1)
    }
    res.statusCode = 200;
    res.send('SubscriptionID deleted');
});

app.get('/subscribers', function (req, res) {
    res.send(subscribers)
})

app.get('/push', function(req, res) {
    var message = {
        // for multiple recipients
        registration_ids: subscribers,
        // for single recipient
        // to: subscribers[0],
        collapse_key: 'your_collapse_key',
    };

    fcm.send(message, function(err, response){
        if (err) {
            console.log("Something has gone wrong!");
            console.log(err)
        } else {
            console.log("Successfully sent with response: ",
            response);
        }
    });
    res.sendStatus(200)
});

exports.app = functions.https.onRequest(app);
```

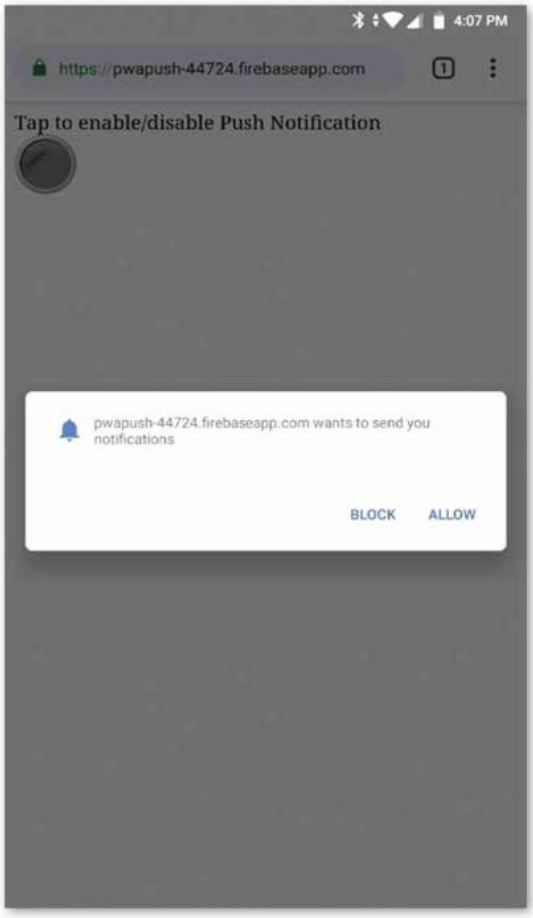


Figure 18: Testing the app on an Android device.

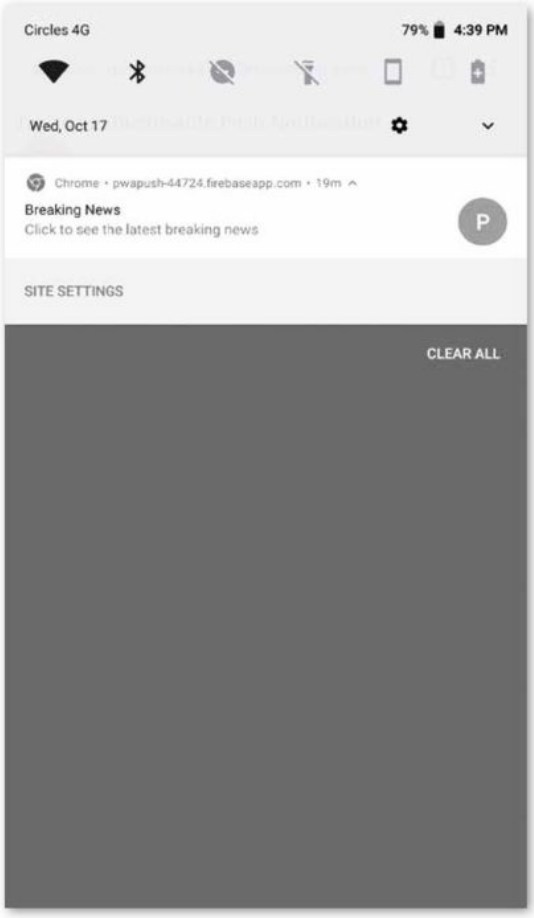


Figure 19: Receiving the notification on the Android device.

What's a Service Worker?

A Service Worker is a script that your browser runs in the background, separate from a Web page. It allows your app to perform background syncs, receive push notifications, perform caching, and more.

You can now redeploy the **main.js** file by typing the following commands In Terminal:

```
$ cd ~/FirebaseHosting/public
$ firebase deploy
```

Testing the Application

You can now retest the application that's hosted on Firebase Hosting. Restart Chrome and clear all the data (**Chrome > Clear Browsing Data...**). Then, load the **index.html** page using the new URL: **https://pwapush-44724.firebaseapp.com**.

Make sure to click the push button once to unsubscribe and one more time to subscribe. This is to ensure that the subscription ID gets saved into the newly hosted serverless REST API.

To send a push notification, type the following command in **Terminal**:

```
$ curl -v
https://pwapush-44724.firebaseapp.com/push
```

You should now receive a notification on your computer. To check the subscription ID that's stored on the serverless app, use the following command:

```
$ curl -v
https://pwapush-44724.firebaseapp.com/
subscribers
```

You should see something like this:

```
["ffsDvGfKqik:APA91bFrshZktJFR5pNYYykdszg0Qq75IJ
wWdmG6xqnk_NmEkFkUoYEMQjbTXCDIuzDL8-
2XaLPZEMnjngiJcWUsdtGzsJKE9FHz9qvpc-
pFv28ulPKavs8ReaftnAGah-6kA7P9ziB0"]
```

Testing the Application on an Android Device

So far, you've been testing the application on the Chrome browser on the desktop. The same application will also work on an Android device. Using an Android device, you can launch Chrome and type the following URL: **https://pwapush-44724.firebaseapp.com**. When you tap on the button, you will be prompted to allow push notifications (see **Figure 18**).

When a notification is received, you'll see the notification as shown in **Figure 19**.

Tapping on the notification displays the **breaking.html** page.

Adding the Application to the Home Screen

You can add the application icon to your Home Screen in Android. Click on the hamburger icon at the top right corner of the Chrome app and tap on **Add to Home screen**. You'll see the prompt, as shown in **Figure 20**. Click **ADD** to add the app icon to the Home Screen.

Listing 8: Modifying the main.js file in the FirebaseHosting/public folder

```
...
//---extract the subscription id and send it over to the
// REST service---
function saveSubscriptionID(subscription) {
    var subscriptionid =
        subscription.endpoint.split('gcm/send/')[1];
    console.log("Subscription ID", subscriptionid);
    fetch('https://pwapush-44724.firebaseapp.com/' +
          'subscribers', {
        method: 'post',
        headers: {
            'Accept': 'application/json',
            'Content-Type': 'application/json'
        },
        body: JSON.stringify(
            { subscriptionid : subscriptionid })
    });
}

//---extract the subscription id and send
// it over to the REST service---
function deleteSubscriptionID(subscription) {
    var subscriptionid =
        subscription.endpoint.split('gcm/send/')[1];
    console.log("Subscription ID", subscriptionid);
    fetch('https://pwapush-44724.firebaseapp.com/' +
          'subscribers/' +
        subscriptionid, {
        method: 'delete',
        headers: {
            'Accept': 'application/json',
            'Content-Type': 'application/json'
        }
    });
}
...
```

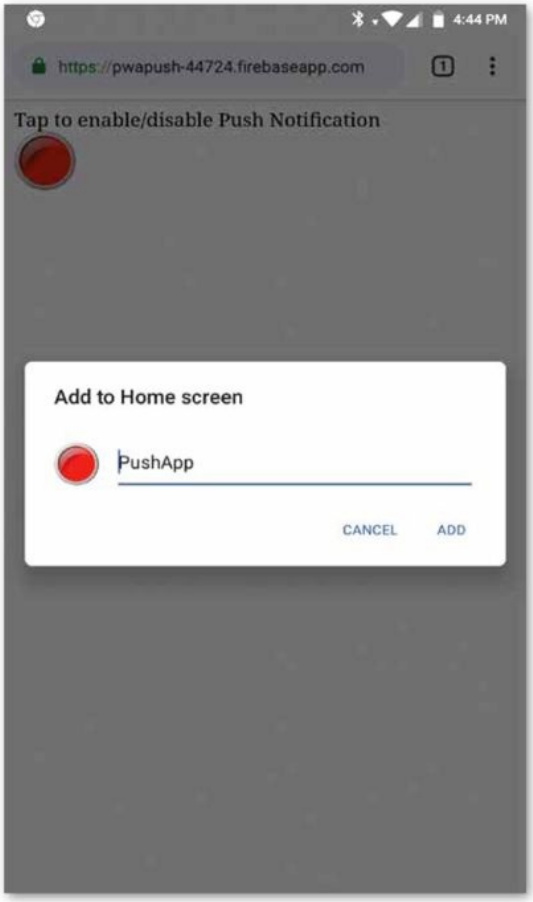


Figure 20: Adding the app icon to the Home Screen.

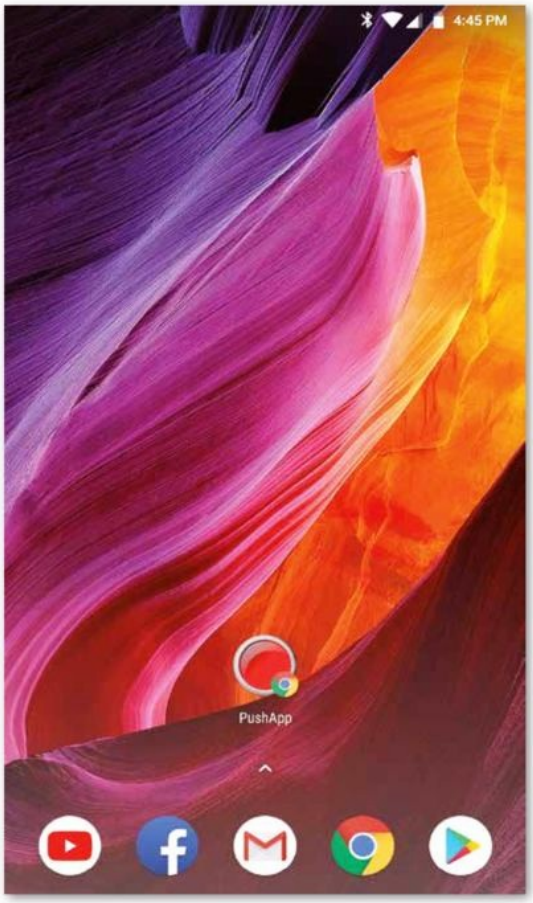


Figure 21: The app icon on the Home Screen of the Android device.

In the next dialog, tap **ADD AUTOMATICALLY**. An icon is added to the **Home** screen (see **Figure 21**).

Summary

In this article, I covered a lot of ground. I first discussed how to enable push notification on a PWA, and then discussed how to build a REST API to save the list of subscription IDs. I then covered how to use Firebase Cloud Messaging to deliver push notifications to applications. When all of these were done, I migrated the app and the REST API onto the Firebase Hosting and Firebase Functions, respectively. Doing this allows you to delegate the back-end tasks to Firebase. Finally, you also learned how to test the app on an Android device and how to save the app icon onto the Home Screen of the Android device.

Wei-Meng Lee

A Professional-Grade Configuration for Azure DevOps Services: Beyond the Quickstarts

Anyone who's done a search for Azure DevOps has seen how easy it is to develop and deploy software using Azure. There's great documentation available and tutorials for getting started abound. Unfortunately, these resources only get you started and don't cover sophisticated applications or the configuration necessary for real-world applications. There's a DevOps process to be implemented, and you have to know the right configuration elements to use in order to light up the Azure DevOps family of products and make it shine. In this article, you will acquire the skills to:

- Organize your code in a manner compatible with DevOps
- Configure each of the five Azure DevOps products professionally
- Automate your pipeline for insanely short cycle times
- Design your process in order to build quality into each stage

Jeffrey Palermo
jeffrey@clear-measure.com
JeffreyPalermo.com
@JeffreyPalermo

Jeffrey Palermo is the Chief Architect of Clear Measure, a DevOps-centered software engineering firm. Clear Measure enabled .NET development teams to be their best by equipping them with DevOps skills and tools. As a Microsoft Gold Partner with DevOps and Cloud qualifications, Clear Measure specializes in software built with Visual Studio. Jeffrey Palermo is host of the Azure DevOps Podcast as well as the founder of the Azure DevOps User Group, AzureAustin, and a founding board member of AgileAustin. Jeffrey has written three .NET books and has spoken at industry conferences such as Tech Ed, Ignite, VSLive, and many more. Microsoft has given him the MVP award every year since 2006. A graduate of Texas A&M University and the Jack Welch Management Institute, a Christian, a father, a husband, an Eagle Scout, and an Iraq war veteran, Jeffrey likes to spend time with his family of five out at the motocross track.

Moving Through the Quickstarts

Even before Azure DevOps Services, Visual Studio Team Services (VSTS) made it very easy to start leveraging DevOps methods such as continuous integration and continuous delivery. Almost three years ago, I published an article in CODE Magazine showing how to easy it was to configure continuous delivery in the Microsoft ecosystem. See the article here: https://www.codemag.com/article/1603061/DevOps-and-Continuous-Delivery-Made-for-a-Cloud-World

Since then, Visual Studio Team Services, now Azure DevOps Services, has emerged as the state-of-the-art environment for creating and operating a DevOps environment for a custom software team. In addition to these capabilities, Azure and Visual Studio provide some tooling to move software into Azure in a matter of minutes. These Quickstarts are well-documented and easy to set up, but for real-world software, these methods quickly break down. In this section, I'll quickly cover the options for dipping your toe into DevOps with Azure. If you'd like to follow the environment you'll be exploring more closely, you can access and fork the public Azure DevOps project at https://dev.azure.com/clearmeasurelabs/Onion-DevOps-Architecture.

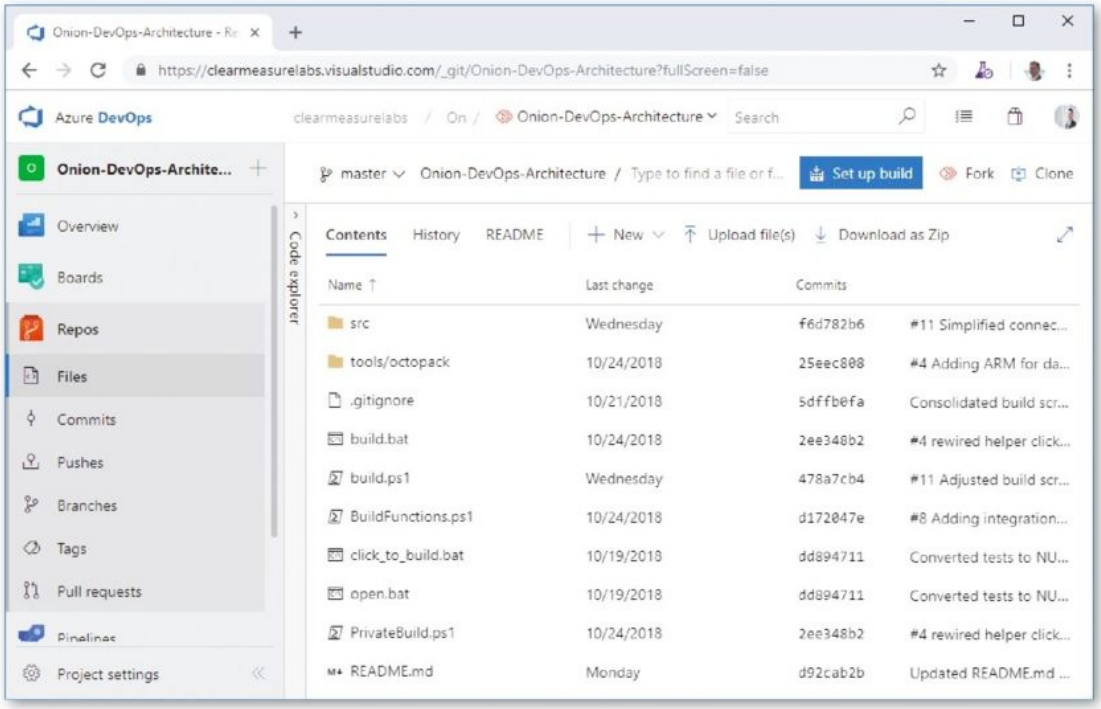


Figure 1: A blue "Set up build" button helps you set up a new build.

Initial Setup of Azure DevOps Services

Once a Git source-control repository is created within a new Azure DevOps project, a helper button lights up blue, reading "Set up Build." See **Figure 1** for how a repository displays before any build is configured for it.

> Azure DevOps Services has emerged as the state-of-the-art environment for creating and operating a DevOps environment for a custom software team.

The first time through this set-up experience has some easy-to-see options. **Figure 2** shows how to select the option for creating a build for ASP.NET Core.

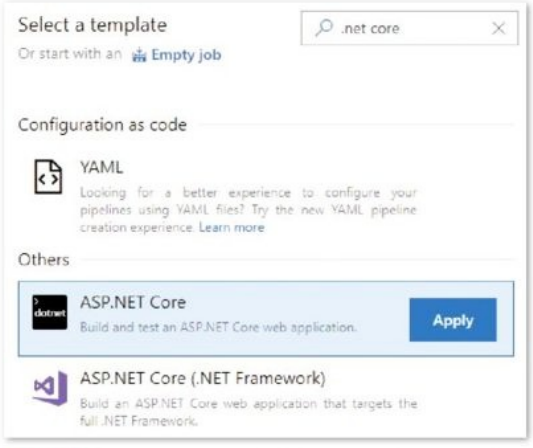


Figure 2: The ASP.NET Core build template is easy to find.

YAML (YAML Ain't Markup Language) is an option that's gaining quite a bit of traction, and it will certainly supplant the designer method of creating a build. But without an option for marketplace step discovery, you're left with the screen-based method for configuring the appropriate build steps you need. Once you apply the chosen template, you're pretty close to being able to run your first build. I'll cover some caveats later in this article regarding testing and database build considerations; you can see the basic steps in **Figure 3** of Restore, Build, Test, Publish, and Publish Artifact.

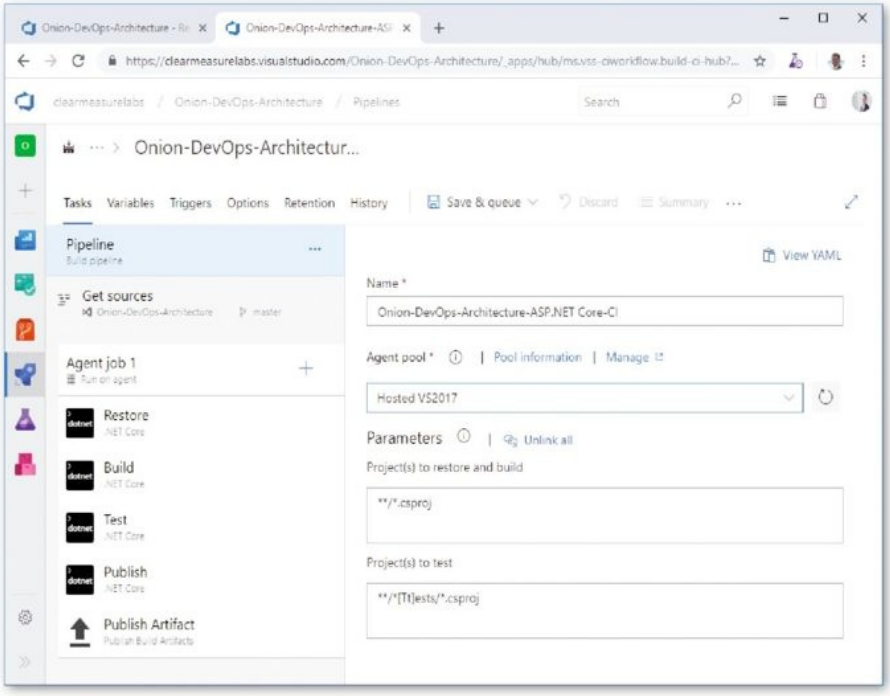


Figure 3: The template lays out a build script automatically.

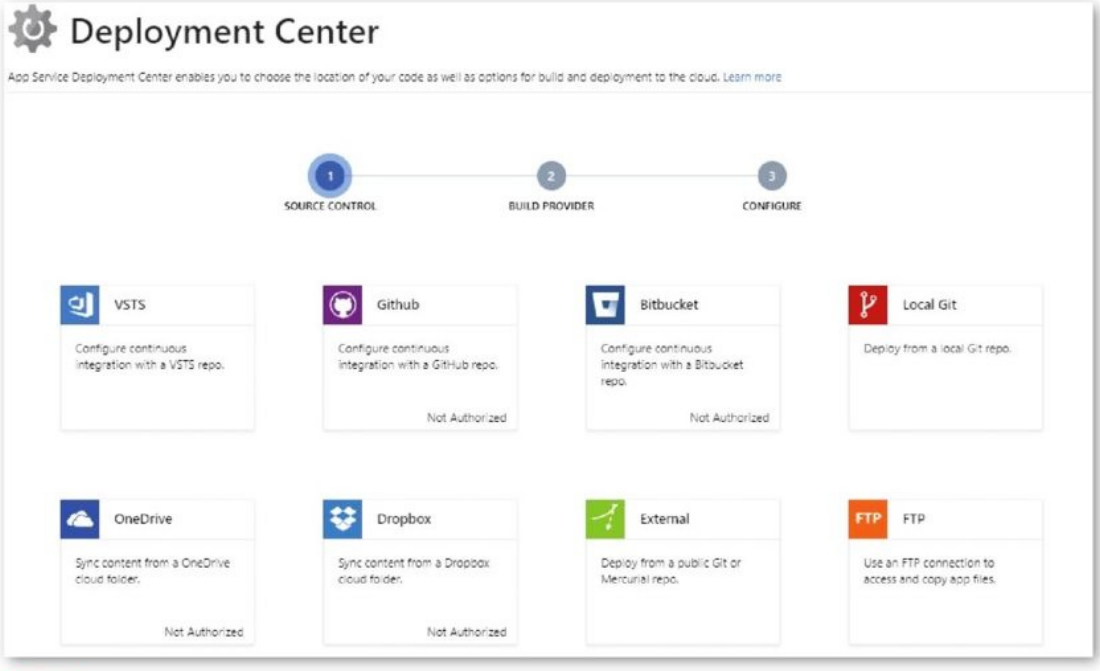


Figure 4: The Azure Deployment Center offers quite a few quick ways to move code into Azure.

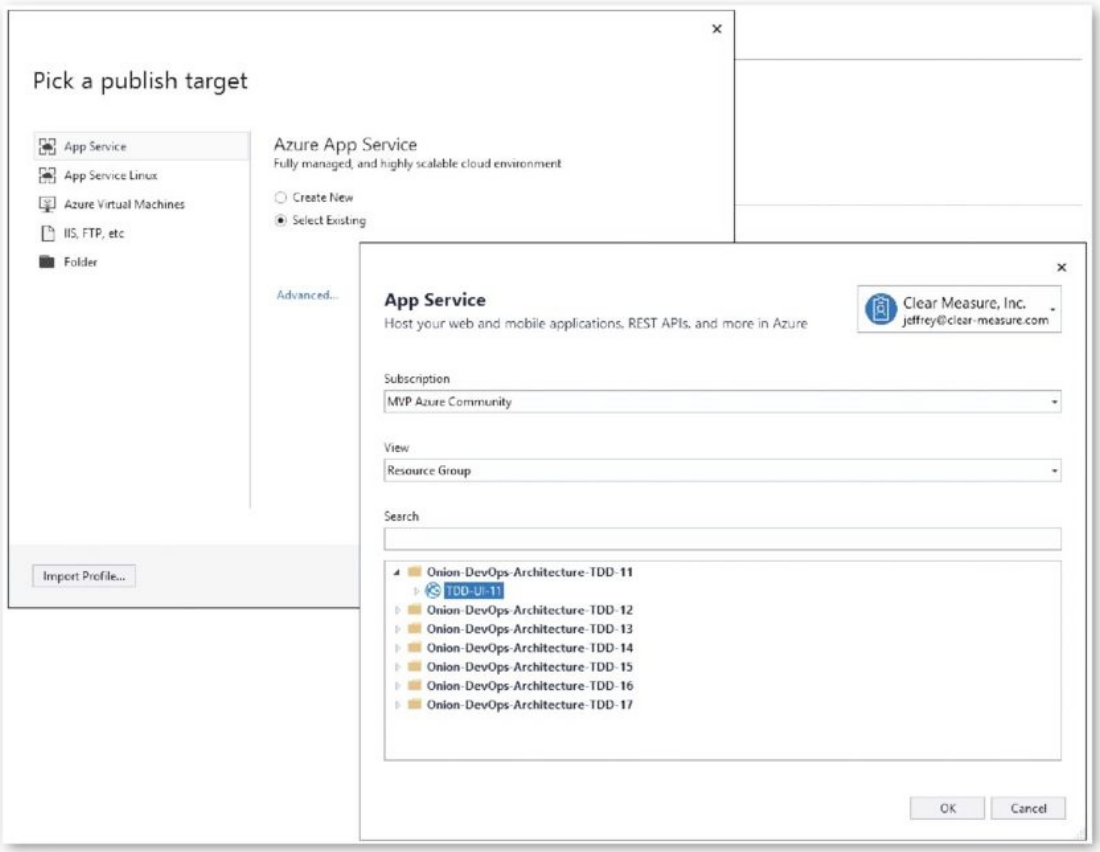


Figure 5: Visual Studio integrates directly with Azure in order to help initial deployments of applications.

```
Windows PowerShell
PS R:\Onion-DevOps-Architecture> .\PrivateBuild.ps1

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:01.61
Test run for R:\Onion-DevOps-Architecture\src\UnitTests\bin\Release\netcoreapp2.1\ClearM
easure.OnionDevOpsArchitecture.UnitTests.dll(.NETCoreApp,Version=v2.1)
Microsoft (R) Test Execution Command Line Tool Version 15.8.0
Copyright (c) Microsoft Corporation.  All rights reserved.

Starting test execution, please wait...
Results File: R:\Onion-DevOps-Architecture\build\test\JeffreyPalermo_DESKTOP-0BVK0EN_201
8-11-02_22_40_17.trx

Total tests: 1. Passed: 1. Failed: 0. Skipped: 0.
Test Run Successful.
Test execution time: 0.9096 Seconds
Rebuild OnionDevOpsArchitecture on localhost\SQL2017 using scripts from R:\Onion-DevOps-
Architecture\src\Database\scripts

Running against: Microsoft SQL Server 2017 (RTM-GDR) (KB4293803) - 14.0.2002.14 (X64)
        Jul 21 2018 07:47:45
        Copyright (C) 2017 Microsoft Corporation
        Developer Edition (64-bit) on Windows 10 Enterprise 10.0 <X64> (Build 17134: )

Dropping connections for database OnionDevOpsArchitecture

Dropping database: OnionDevOpsArchitecture

Run scripts in Create folder.
Run scripts in Update folder.
Executing: 001_AddExpenseReportTable.sql in a transaction
Executing: 002_AddStatusToExpenseReport.sql in a transaction
Run scripts in Everytime folder.
Run scripts in RunAlways folder.
usdDatabaseVersion: 2
Test run for R:\Onion-DevOps-Architecture\src\IntegrationTests\bin\Release\netcoreapp2.1
\ClearMeasure.OnionDevOpsArchitecture.IntegrationTests.dll(.NETCoreApp,Version=v2.1)
Microsoft (R) Test Execution Command Line Tool Version 15.8.0
Copyright (c) Microsoft Corporation.  All rights reserved.

Starting test execution, please wait...
Results File: R:\Onion-DevOps-Architecture\build\test\JeffreyPalermo_DESKTOP-0BVK0EN_201
8-11-02_22_40_21.trx

Total tests: 4. Passed: 3. Failed: 0. Skipped: 0.
Test Run Successful.
Test execution time: 2.4701 Seconds
PS R:\Onion-DevOps-Architecture>
```

Figure 6: The private build runs very quickly but provides a good deal of verified safety.

In addition to this template, Azure provides some other methods to deploy directly from a GitHub repository if the architecture of your application is compatible with a file copy deployment. Further depth on the pathways to getting started quick can be found in a Microsoft eBook by Cam Soper and Scott Addie entitled "DevOps with ASP.NET Core and Azure" and here: https://aka.ms/devopsbook.

Outgrowing the Quickstarts

Azure provides quite a few options for moving code from various sources into App Services. The options are currently grouped into the new Deployment Center shown in **Figure 4**.

Choosing any of these sources moves you through a multi-stage wizard in order to hook things up quickly.

Along with these Azure Portal options, Visual Studio continues to offer a menu-based method for deploying an application into Azure. **Figure 5** shows the simple screens you can use within Visual Studio. This shows the option to create new resources or redeploy over existing Azure resources.

Each of these options has some pretty immediate drawbacks after quickly prototyping a deployment or cloud computing topology. Let's dig into the ASP.NET Core build template shown in **Figure 3**. The next sections discuss some limitations that you must overcome on your own in order to approach a professional DevOps environment for your application. I'll take them one at a time.

Private Build

The included build the wizard's offer of guidance for a private build process. It's up to you to create a script agreed on by your team that's run locally before a commit is pushed to the team's Git repository. The purpose of the private build is to prevent the loss of 20-30 minutes if a commit causes a continuous integration build to fail. And forgetting to include a new file or other small mistakes are often the cause of a broken build. Creating a private build for yourself or your team isn't complicated, but it does include some essential steps because it's the foundation of the continuous integration build. It must contain the following steps, at a minimum:

- **Clean your environment:** All artifacts of previous builds should be discarded so there is a known starting point
- **Compile your Visual Studio solution:** Compiled them all together, all at once
- **Set up any dependencies:** Allows unit tests and fast integration tests to run
- **Run unit tests:** These tests are fast and shouldn't call out of process
- **Running fast integration tests:** Examples are small tests that ensure every query or ORM configuration functions properly

> The purpose of the private build is to prevent the loss of a cycle if a commit causes a continuous integration build to fail.

You can see the private build script used in your application at https://dev.azure.com/clearmeasurelabs/_git/Onion-DevOps-Architecture?path=%2Fbuild.ps1&version=GBmaster. **Figure 6** shows the build run locally via PowerShell.

Assembly Versioning

With any build process, the version number is a foundational element. If you're unsure what version of your software is running, there's no way to report bugs. A common feature in a test environment is for the bottom of the screen to show the user what version of the software is being run so that bug reports can include this information. If the packaged software doesn't include any kind of label or embedded version number, contact the manufacturer. The template in the Quickstart doesn't pass on the build's version number to the dotnet.exe command line, so there's no chance for embedding a version number. As a result, the packaged application that's stored for deployment contains the default 1.0.0 version, as shown in **Figure 7**.

Database Set up

In most .NET applications, SQL Server is a common storage component. In my professional work, I seldom come across a client with a .NET application that fully runs without SQL Server. Given that, you can conclude that setting up a database for use in the continuous integration build process is a normal and necessary step that must be included. Although this topic will receive more attention later in the article, your build script must include support for this, as shown in the following snippet.

```
Function PrivateBuild{
      Init
      Compile
      UnitTests
      MigrateDatabaseLocal
      IntegrationTest
}

Function CIBuild{
      Init
      Compile
      UnitTests
      MigrateDatabaseRemote
      IntegrationTest
      Pack
}
```

Build Tuning

Without adding any code, you'll see that the Quickstart build takes almost three minutes to complete. When you investigate, you'll find that you can run the same steps on your local computer in under a minute. Some of the architecture of the hosted build agents is responsible for that, and some of the structure of the build template is responsible. Regardless, you must reduce that build time.

In late 2018, I published a blog post on this very topic. It can be found at https://jeffreypalermo.com/2018/10/performance-tuning-an-azure-devops-build-configuration/. Out of the box, you can see that the dotnet.exe restore step consumes almost a minute of build time. With industry guidance at a ten-minute limit, consuming 10% of available

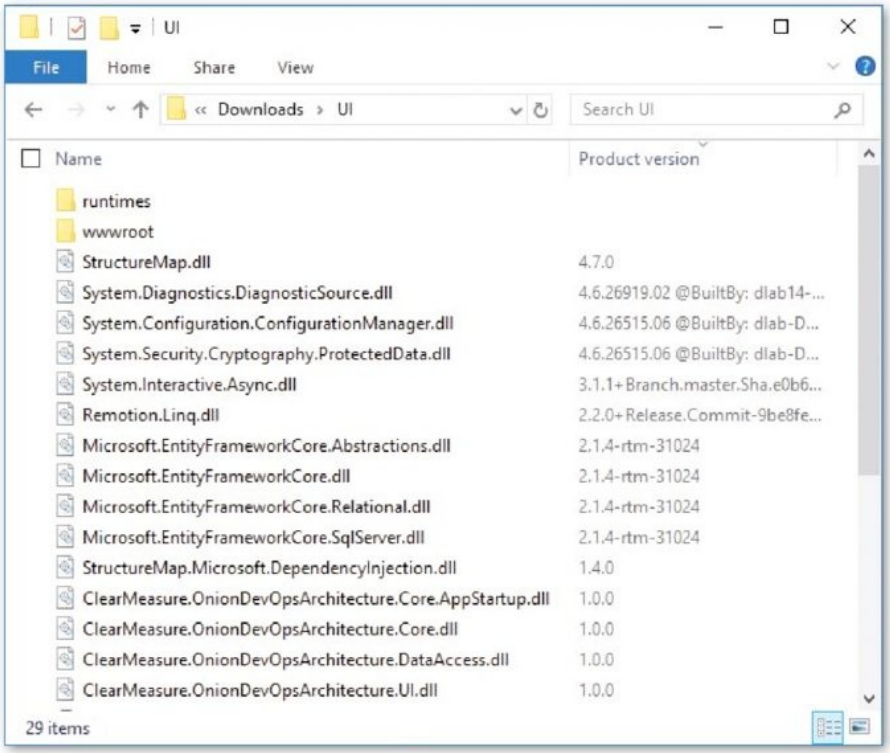


Figure 7: Although the framework assemblies have a proper version number, your application is perpetually marked as 1.0.0.

Figure 8: Pine was a popular Unix mainframe email client in the 1990s. (Photo credit: Wikipedia https://upload.wikimedia.org/wikipedia/en/c/ce/PineScreenShot.png)

build time just gathering dependencies isn't going to work. In a few build runs, the Restore step consumed 53 and 60 seconds, respectively. Then, the dotnet.exe build command failed to pass in the **--no-restore** command line argument, so it wastes another 30 seconds duplicating work already done.

As you put together your own build script, be mindful of performance. Any minute that your build takes to process will be a minute of concentration that could result in a loss of team productivity. Short cycle time is critical in a professional DevOps environment.

> Any minute your build takes to process is a minute of concentration that could result in a loss of team productivity.

The State of DevOps

Several organizations are performing ongoing research into the advancement of DevOps methods across the industry. Puppet and DORA are two that stand out. Microsoft has sponsored the DORA State of DevOps Report. Sam Guckenheimer is the Product Owner for all Azure DevOps products at Microsoft and contributed to the report. He also spoke about that on his recent interview with the Azure DevOps Podcast, which can be found at http://azuredevopspodcast.clear-measure.com/sam-guckenheimer-on-testing-data-collection-and-the-state-of-devops-report-episode-003.

A key finding of DORA's State of DevOps report was that elite performers take full advantage of automation. From builds to testing to deployments and even security configuration changes, elite performers have a seven times lower change failure rate and over 2,000 times faster time to recover from incidents.

Other key texts that have led the industry's definition of DevOps are a series of books, all including Jez Humble. The progression in which you should read them is:

- "The Phoenix Project: A Novel about IT, DevOps, and Helping Your Business Win", by Kim, Spafford, and Behr
- "Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation", by Farley and Humble
- "The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations," by Kim, Humble, and Debois

If you're just getting into DevOps, don't be discouraged. The industry is still figuring out what it is too, but there are now plenty of success stories to learn from.

Removing the Ambiguity from DevOps

In the community of large enterprise software organizations, many define DevOps as development and operations coming together and working together from development through operations. This is likely the case in many organizations, but I want to propose what DevOps will likely be as you look back on this era twenty years from now from a time when your worldview isn't colored by the problems of today.

In the 1950s there were no operating systems. Therefore, there was no opportunity for multiple programs to run at the same time on a computer. There was no opportunity for one programmer to have a program that interfered with the program of another. There was no need for this notion of Operations. The human who wrote the program also loaded the program. That person also ran the program and evaluated its output.

Fast-forward to the era of the terminal mainframe server. In this era, a programmer could load a program and it had the potential of causing problems for the other users of the mainframe. In this era, it became someone's job to keep the mainframe operating for the growing pool of mainframe users. Even if you've never programmed for a mainframe, you might remember using Pine for email. It was popular at universities in the 1990s. If this predates you, you can see it in **Figure 8**.

I believe that the DevOps movement is the correction of a software culture problem that began with the mainframe era. Because multi-user computers, soon to be called servers, were relied upon by increasing numbers of people, companies had to ensure that they remained operational. This transformed Data Processing departments into IT departments. All of the IT assets needed to run smoothly. Groups that sought to change what was running on them became known as developers, (although I still call myself a computer programmer). Those who're responsible for stable operations of the software in production environments are known as operations departments, which are filled with IT professionals, systems engineers, and so on.

I believe you're going to look back at the DevOps era and see that it's not a new thing you're creating but an undoing of a big, costly mistake that occurred over two or three decades. Instead of bringing together two

departments so that they work together, you'll have eliminated these two distinct departments and will have emerged with one type of persona: the software engineer.

Smaller companies, by the way, don't identify with all the talk of development and operations working together because they never made this split in the first place. There are thousands upon thousands of software organizations that have always been responsible for operating what they build. And with the Azure cloud, any infrastructure operation becomes like electricity and telephone service, which companies have always relied on outside parties to provide.

Microsoft has already reorganized their Azure DevOps department in this fashion. There's no notion of two departments working together. They eliminated the divide by making one department staffed with two roles:

- Program manager
- Engineer

I believe this type of consolidation will happen all across the industry. Although there's always room for specialists in very narrow disciplines, software organizations will require the computer programmer to be able to perform all of the tasks necessary to deliver something they envisioned as it's built and operated.

A Professional-Grade DevOps Vision

When you look at your own organization, you're probably in the camp where you want better quality and better speed. Your form of quality may be fewer bugs. It may be fewer problems in production. It may be more uptime or better handling of user load spikes. When you think of speed, you may be thinking about developing new features. But business executives may be thinking about reducing the lead-time between when they fund a strategic initiative and when they're able to launch the software to support it. Regardless of the specific issues, it seems to always come down to quality and speed. That's when you need Capers Jones, your industry's leading software research statistician. In his recent book, "Software Engineering Best Practices," he demonstrates research that proves two points:

- Prioritizing speed causes shortcuts, which cause defects, which cause rework, which depletes speed. Therefore, prioritizing speed achieves neither speed nor quality.
- Prioritizing quality reduces defects, which reduces rework, which directs all work capacity to the next feature. Therefore, prioritizing quality achieves speed.

You want to design a DevOps environment that squeezes out defects all along the way. You can do this by automating the repetitive tasks and taking them away from the humans. Humans aren't good at repetitive tasks. Computers are much better at such things. But humans are very good at solving problems. Computers are not good at "thinking outside the box." The following capabilities are my vision for a professional-grade DevOps environment:

- Private build
- Continuous integration build
- Static code analysis
- Release candidate versioning and packaging
- Environment provisioning and configuration
- Minimum of a three-tier deployment pipeline
- Production diagnostics managed by development team
- Insanely short cycle time through the previous steps

You don't need an infrastructure like Netflix in order to accomplish this. In fact, you can set this up with a skeleton architecture even before you've written your first feature or screen for a new application. And you can retrofit your current software into an environment like this as well. You want to keep in mind the 80/20 rule and gain these new capabilities without adding too much scope or trying to "boil the ocean" in your first iteration.

DevOps Architecture

Let's walk through the process that a DevOps environment manages. **Figure 9** shows the logical structure of a DevOps environment. The full-sized image can be downloaded from https://jeffreypalermo.com/2018/08/applying-41-architecture-blueprints-to-continuous-delivery/.

I'll take the stages one at a time.

Version Control

First, you must structure your Version Control System properly. In today's world, you're using Git. Everything that belongs to the application should be stored in source control. That's the guiding principle. Database schema scripts should be there. PowerShell scripts to configure environments should go there. Documents that outline how to get started developing the application should go there. Once you embrace that principle, you'll step back and determine what exceptions might apply to your situation. For instance, because Git doesn't handle differences in binary files very well, you may elect not to store lots and lots of versions of very big Visio files. And if you move to .NET Core, where even the framework has become NuGet packages, you may elect not to store your **/packages** folder like you might have with .NET Framework applications. But the Git repository is the unit of versioning, so if you have to go back in time to last month, you want to ensure that everything from last month is correct when you pull it from the Git repository.

> Everything that belongs to the application should be stored in source control.

Private Build

The next step to configure properly is the private build. This should run automated unit tests and component-level integration tests on a local workstation. Only if this private build works properly and passes should you commit and push your changes to the Git server. This private build is the basis of the continuous integration build, so

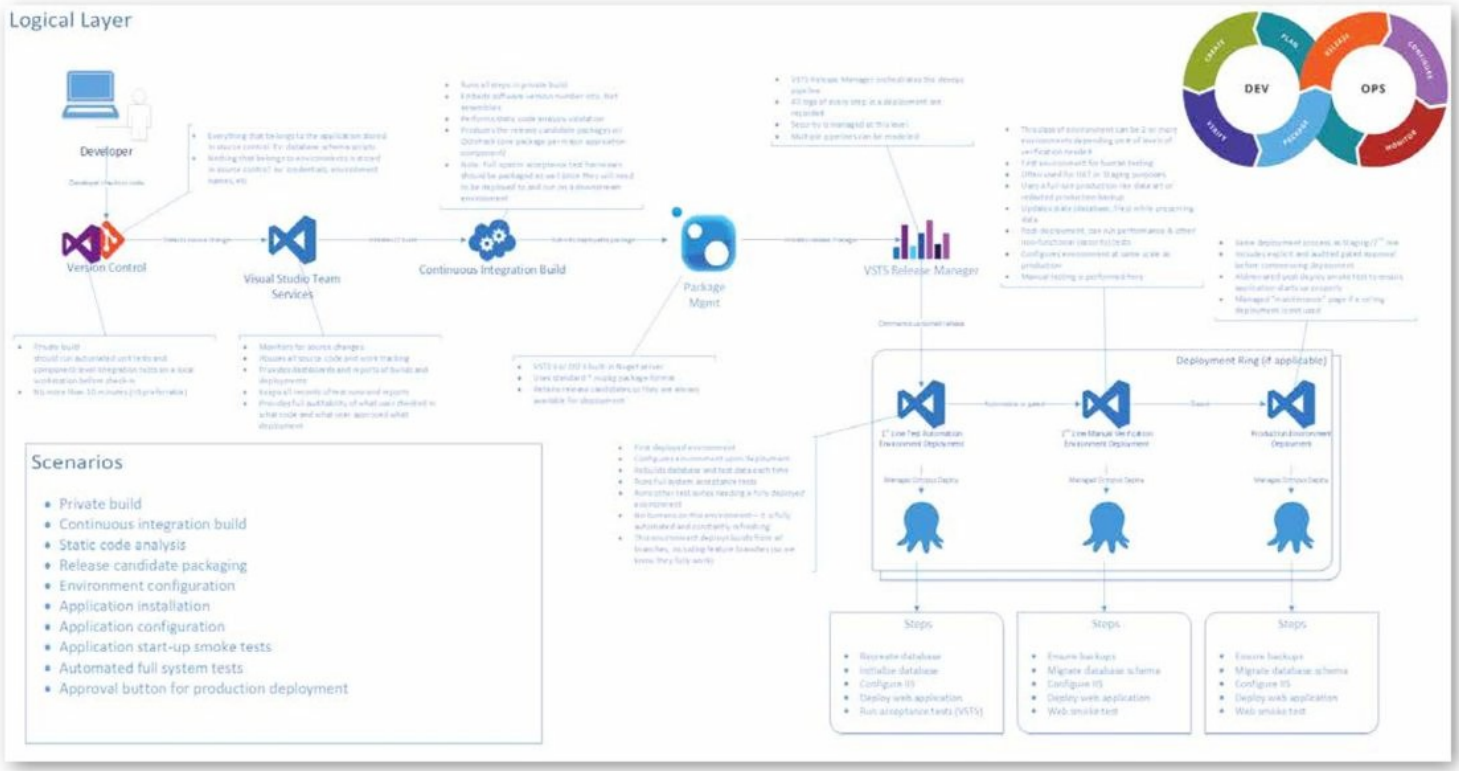


Figure 9: The logical architecture layer of a DevOps environment.

you want it to run in as short a period of time as possible. The widely accepted industry guidance is that it should run for no more than ten minutes. For new applications that are just getting started, 45 seconds is normal and will show that you're on the right track. This time should include running two levels of automated test suites: your unit tests and component-level integration tests.

Continuous Integration Build

The continuous integration build is often abbreviated "CI Build." This build runs all the steps in the private build, for starters. It's a separate server, away from the nuances of configuration on your local developer workstation. It runs on a server that has the team-determined configuration necessary for your software application. If it breaks at this stage, a team member knows that they need to back out their change and try again. Some teams have a standard to allow for "I forgot to commit a file" build breaks. In this case, the developer has one shot to commit again and fix the build. If this isn't achieved immediately, the commit is reverted so that the build works again. There's no downside to this because in Git, you never actually lose a commit. The developer who broke the build can check out the commit they were last working on and try again once the problem's fixed.

The continuous integration build is the first centralized quality gate. Capers Jones' research, referenced above, also concludes that three quality control techniques can reliably elevate a team's Defect Removal Efficiency (DRE) up to 95%. The three quality control techniques are testing, static code analysis, and inspections. Inspections are covered later in a discussion of pull requests, but static code analysis should be included in the continuous integration build. Plenty of options exist, and these options integrate with Azure DevOps Services very easily.

The CI build also runs as many automated tests as possible in ten minutes. Frequently, all of the unit tests and component-level integration tests can be included. These integration tests are not full-system tests but are tests and that validate the functionality of one or two components of the application that require code that calls outside of the .NET AppDomain. Examples are code that relies on round trips to the database or code that pushes data onto a queue or file system. This code that crosses an AppDomain or process boundary is orders of magnitude slower than code that keeps only to the AppDomain memory space. This type of test organization heavily impacts CI build times.

The CI build is also responsible for producing the versioned release candidate package. Whether you package your application components in NuGet packages or zip files, you need organized packaging. Each package needs to be named and numbered with the build version. Because you're only building this once, regardless of how many environments you deploy to, it's important that this package contains everything necessary to provision, configure, and install the application component on downstream environments. Note that this doesn't include credentials or environment-specific settings. Every assembly inside this package must be stamped with the build number. Make sure you use the right command-line arguments when you compile so that all produced assemblies receive the build number. The following snippet shows an example of this with parameters configured as PowerShell variables.

```
dotnet build $syource_dir\$projectName.sln -nologo
          --no-restore -v $verbosity -maxcpucount
          --configuration $projectConfig --no-incremental
          /p:Version=$version
```

Package Management

Because you're producing release candidate packages, you need a good place to store them. You could use a file system or the simple artifacts capability of Azure DevOps but using the rock-solid package management infrastructure of NuGet is the best current method for storing these. This method offers the API surface area for downstream deployments and other tools, like Octopus Deploy.

Azure DevOps Services offers a built-in NuGet server as Azure Artifacts. With your MSDN or Visual Studio Enterprise subscription, you already have the license configuration for this service, and I recommend that you use it. It allows you to use the standard *.nupkg (pronounced nup-keg) package format, which has a place for the name and a version that can be read programmatically by other tools. It also retains release candidates so they're always available for deployment. And when you need to go back in time for a hotfix deployment or reproduction of a customer issue, you always have every version.

Test-Driven Development Environment (TDD Environment)

The first of the three types of environments in a DevOps pipeline is the TDD environment. You might also call it the ATTD environment if you have adopted Acceptance Test-Driven Development. This is the environment where no humans are allowed. Once your pipeline deploys the latest release candidate, your suites of automated full-systems tests are unleashed on this environment. Some examples of full-system tests might be:

- Web UI tests using Selenium
- Long-running full-system tests that rely on queues
- ADA accessibility tests
- Load tests
- Endurance tests
- Security scanning tests

The TDD environment can be a single instance, or you can create parallel instances in order to run multiple types of test suites at the same time. This is a distinct type of environment, and builds are automatically deployed to this environment type. It's not meant for humans because it automatically destroys and recreates itself for every successive build, including the SQL Server database and other data stores. This type of environment gives you confidence that you can recreate an environment for your application at any time you need to. That confidence is a big boost when performing disaster recovery planning.

The TDD environment is a distinct type of environment, and builds are automatically deployed to this environment type.

Manual Test Environment

Manual test environments are an environment type, not a single environment. Organizations typically have many of these. QA, UAT, and Staging are all common names for this environment type, which exists for the manual verification of the release candidate. You provision and deploy to this environment automatically, but you rely on a human to check something and give a report that either the release candidate has issues or that it passed the validations.

Manual test environments are the first environment available for human testing, and if you need a Demo environment, it would be of this type. It uses a full-size production-like set of data. Note that it shouldn't use production data because doing so likely increases the risk of data breaches by exposing sensitive data to an increased pool of personnel. The size and complexity of the data should be similar in scale to production.

During deployments of this environment type, data isn't reloaded every time, and automated database schema migrations run against the existing database and preserve the data. This configuration ensures that the database deployment process will work against production when deployed there. And, because of the nature of this environment's configuration, it can be appropriate for running some non-functional test suites in the background. For instance, it can be useful to run an ongoing set of load tests on this environment as team members are doing their normal manual validation. This can create an anecdotal experience to give the humans involved a sense of whether or not the system feels sluggish from a perception point of view.

Finally, this environment type should be configured with similar scale specs as production, including monitoring and alerting. Especially in Azure, it's not quite affordable to scale up the test environment just like production because environments can be turned off on a moment's notice. The computing resources account for the vast majority of Azure costs; data sets can be preserved for pennies even while the rest of the environment is torn down.

Production Environment

Everyone's familiar with this environment type. It's the one that's received all the attention in the past. This environment uses the exact same deployment steps as the manual environment type. Obviously, you preserve all data sets and don't create them from scratch. The configuration of monitoring alert thresholds will have its own tuning, and alerts from this environment will flow through all communication channels; previous environments wouldn't have sent out "wake-up call" alerts in the middle of the night if an application component went down.

In this environment, you want to make sure that you're not doing anything new. You don't want to do anything for the first time on a release candidate. If your software requires a zero-downtime deployment, the previous environment should have also used this method so that nothing is tested for the first time in production. If an off-line job is taken down and transactions need to queue up for a while until that process is back up, a previous environment should include that scenario so that your design for that process has been proven before it runs on production. In short, the deployment to production should be completely boring if all needed capabilities have been tested in upstream environments. That's the goal.

Importing Centralized VCS Repositories

It used to be common practice with centralized version control systems to use a pattern of sub-repositories. For example, in both SVN and TFVC, it was possible and very workable to create a series of top-level folders so that multiple teams could maintain applications. And it was possible to branch and merge at a lower level. The working copies created by these clients only checked out as a child folder, and to the working copy, the fact that the code was a nested part of a larger repository was fairly opaque.

The architecture of Git is quite different. When cloning the repository, the entire repository is cloned, even if a shallow clone is used to ignore some past history. Therefore, rather than mimicking this folder structure and attempting to subdivide the repository, make an independent Git repository for each top-level folder in your SVN or TFVC repository. This results in each application having its own repository and branching method. If you find yourself with software that has shared project and dll references up and down the repository, you'll have to break these dependencies or manage them in some other way in order to migrate the code. Git works the best when it has a single piece of software to version. Your downstream processes will all work better as well.

Production Monitoring and Diagnostics

Production monitoring and diagnostics isn't an independent state but is a topic that needs to apply to all environments. Monitoring and operating your software in Azure isn't just a single topic. There's a taxonomy of methods that you need in order to prevent incidents. Recently, Eric Hexter made a presentation on this topic to the Azure DevOps User Group, and that video recording can be found at https://youtu.be/60-17phQMJo. Eric goes through the different types of diagnostics including metrics, centralized logs, error conditions, alerts, and heartbeats.

> Deployment to production should be completely boring if all needed capabilities have been tested in upstream environments.

Tools of the Professional DevOps Environment

Now that I've covered the capabilities that need to be a part of a professional DevOps environment, let's discuss how to use what Microsoft and the marketplace have to offer. **Figure 10** shows the physical(runtime) environment view of this environment.

In **Figure 10**, you make a sample selection of marketplace tools that complement Azure DevOps Services. The Visual Studio and Azure marketplaces offer a tremendous array of capable products, and you'll want to select and integrate the ones that fit your software architecture. In this configuration, you see that Azure DevOps Services will be what developers interact with by committing code from their workstations, making changes to work items, and executing pull requests. You're specifying that you'll have your own virtual machines as build agents in order to provide more speed to the build process.

You'll also use the Release Hub in Azure DevOps in conjunction with Octopus Deploy as your deployment capability. Although Azure Pipeline is increasing its breadth of support for all kinds of deployment architectures, Octopus Deploy was the original deployment server for the .NET ecosystem, and its support is unparalleled in the industry at the moment. You show that you have deployment agents at the servers that represent each of your environments, and that they call back to the deployment server rather than having the deployment server call through the firewall directly into each server. Then you have specified Stackify as an APM tool collecting logs, telemetry, and metrics from each environment. Your developers can then access this information. Obviously, this architecture shows an environment very light on PaaS. Although new applications can easily make heavy use of PaaS, and I encourage it, most developers also have an existing system that requires a great deal of work in order to shift the architecture to free itself from VM-based environments. Professional DevOps is not only for greenfield applications. It can be applied to all applications.

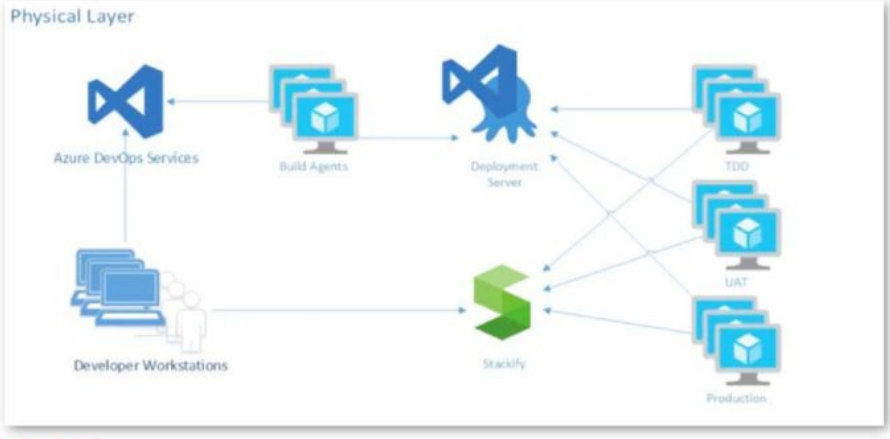


Figure 10: This view shows what runs when the pieces of DevOps infrastructure run.

Azure DevOps Services

On September 10, 2018, Microsoft pulled the trigger on a major release that included the segmentation of its popular product, Visual Studio Team Services (VSTS). It broke VSTS into five products and has named this family of products Azure DevOps. The five new products are:

- **Azure Pipelines:** Supports continuous integration builds and automated deployments
- **Azure Repos:** Provides source code hosting for a TFVC repository and any number of Git repositories
- **Azure Boards:** Organizes work and project scope using a combination of backlogs, Kanban boards, and dashboards
- **Azure Test Plans:** Integrates tightly with Azure Boards and Azure Pipelines by providing support for automated and manual full-system testing, along with some very interesting stakeholder feedback tools
- **Azure Artifacts:** Provides the capability to provision your team's own package feeds using NuGet, Maven, or npm

The independent service that's been receiving lightning-fast adoption since early September is Azure Pipelines. Especially with the acquisition of GitHub, the experience to set up a build for a code-base stored in GitHub is simple and quick.

Azure Subscription

In order to set up your DevOps environment, you need an Azure subscription. Even if all your servers are in a local data center, Azure DevOps Services runs connected to your Azure subscription, even if only for billing and Azure Active Directory. Using your Visual Studio Enterprise subscription, you also have a monthly budget for trying out Azure features, so you might as well use it.

The Azure subscription is a significant boundary. If you're putting your application in Azure, you really want to think about the architecture of your subscriptions and your resource groups. There will never be only one. In fact, even if you attempt to put all your applications in a single subscription, you'll quickly find out that the subscription wasn't designed to be used that way. The subscription is a strong boundary of security, billing, and environment segmentation. Some rules of thumb when it comes to deciding on when to create a new subscription or resource group are:

- A subscription that houses the production environment of a system should not also house an environment with lesser security controls. The subscription will only be as secure as its least secure resource group and access control list.
- Pre-production environments may be grouped together in a single subscription but placed in separate resource groups.
- A single team may own and use many Azure subscriptions, but a single subscription should not be used by multiple teams.
- Resource groups should be created and destroyed rather than individual resources within a resource group.
- Just because you're in the cloud doesn't mean that you can't accidentally end up with "pet" resource groups; only create resources through the Azure Portal in your own personal subscription that you use as a temporary playground. See Jeffrey Snover's Pets vs. Cattle at http://cloudscaling.com/blog/cloud-computing/the-history-of-pets-vs-cattle/
- Resource groups are good for grouping resources that are created and destroyed together. Resources should not be created through hand-crafting. The analogy of Pets vs. Cattle can be applied to Pet Azure subscriptions where things are named and cared for by a person rather by a process or automated system.

If you're putting your application in Azure, you really want to think about the architecture of your subscriptions and your resource groups.

Visual Studio 2017

You can certainly start with Visual Studio Community, but Visual Studio Enterprise will be what you want to use in a professional DevOps environment. You'll need to do more than just write C# code. You'll need to have a holistic tool set for managing your software. As an example, the industry-leading database automation tool ReadyRoll (now SQL Change Automation) from Redgate is bundled right into Visual Studio Enterprise. This makes it a breeze to create automated database migration scripts. You'll also want to equip your IDE with some extensions from the Visual Studio marketplace. Of course, ReSharper from JetBrains is a favorite.

Configuring Azure DevOps for the Professional DevOps Environment

Now that you've looked at the capabilities of the professional DevOps environment and a mix of tools that can be a part of it, you'll drill down into each product within the Azure DevOps family and set it up in the proper way. You'll certainly want to customize the configuration, but your suggested configuration works great in 80% of the cases. If you've already read the book "The Phoenix Project" by Kim, Spafford, and Behr, you'll recognize the "three ways of DevOps" implemented as you read through this section. Once you have your Azure DevOps project created, take a glance at your project settings and select the products that you'd like enabled.

Figure 11: You can enable or disable any of the products in the Azure DevOps family.

In **Figure 11**, you can see that for the purposes of this article, I have all of the products enabled. For your team, you'll want to equip them with the Visual Studio Enterprise subscription (formerly called MSDN Premium) so that they have licensing for all of the products. You'll need them.

Azure Boards Makes Work Visible

Within the first way of DevOps is the principle of "make work visible." Azure Boards is the tool of choice for modeling the shape of your work. Azure Boards uses Work Items to track a unit of work. A work item can be of any type and has a status as well as any number of other fields you'd like. As you think about your hierarchy of work, don't immediately start creating work items using the built-in sample hierarchy. Instead, think about the work that you already do and the parent-child relationships between some of the types of work. For example, in a marketing department, the structure in **Figure 12** may be appropriate.

This marketing department has decided that they only need three levels of work. A Campaign can have multiple Campaign Items or Product Backlog Items. A Campaign Item and a Product Backlog Item can have multiple Tasks. At the top level, they can track at the Campaigns level or the Execution level. An individual iteration or sprint is tracked with Tasks. You can have any number of higher-level portfolio backlogs if you need higher levels of groupings. Even while the built-in process template includes **Epic > Feature > Product Backlog Item**, you'll quickly outgrow this because it won't match your organization. You need to disable most of the built-in work item types and create your own so that you can name them and put only the fields and the progression of statuses that make sense into your teams' environments.

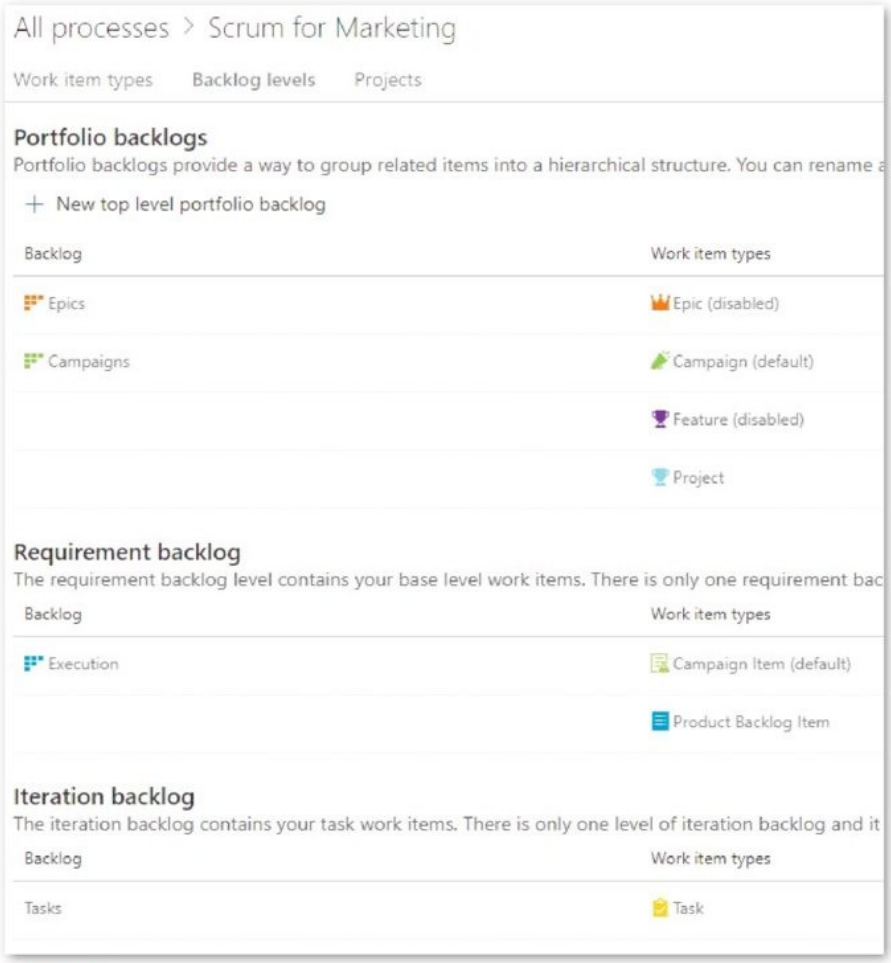


Figure 12: A marketing department has Campaigns that are broken down into individual items.

You may think of several work types to get the creative juices flowing in order to capture the model of your organization's world. Notice that I didn't say "design the model." Your model already exists. You need to capture the nouns and the verbs of your existing reality and make Azure Boards represent what's already there. If you capture the wrong model, it won't fit, and your co-workers will have a hard time tracking their work because it just won't make sense. So, consider the following types:

- Business initiatives
- Marketable features
- Plannable work to budget, schedule, and funding
- Individual tasks to get done

This becomes the foundation of your usage of Azure Boards going forward. You'd never think of starting a new application with the Northwind or AdventureWorks database schema. Those tables were chosen by someone else. That model just doesn't fit the nature of the data you're trying to store. In this same way, the schema of the built-in process templates won't fit your organization. You need to load your own model. Once you have your model, you need to specify the process of each major entity (work item). For example, if you were writing an article or a book, you might create a Chapter work item and specify the status progression on the Kanban board like that shown in **Figure 13**.

By determining ahead of time what the process is that takes a certain level of work item from creation to done, you organize your team. Each state, or board lane, should be owned by a type of role. For example, if you have a stakeholder designated as the person who'll give the go-ahead on the sketch of a screen before it's developed, that stakeholder should have a column where they own the work within it. Each work item is represented by a visual card on the Kanban board, and the cards in their column are theirs to work.

If the stakeholder does nothing, cards pile up in that column, and nothing is developed because of the bottleneck in that column. A dashboard report can bring this to light on a daily basis so that no column has too much work in it. The stakeholder's job would be to either approve the sketch of the screen or initiate a conversation to fix it. In no case would you want a bad screen to be coded. That would be worse. By creating a good number of columns, mapped to the states of the work item, you can move the work through a known process where every column has a type of role responsible for performing a known set of work and then forwarding the work in process WIP to the next column. From a quality control perspective, every person starting on work has the obligation for inspecting the WIP to see if the work is ready for them yet. If something's missing, you stop the line and get it corrected before propagating the error further downstream.

For the purposes of software teams, the level of backlog that's prepopulated with Product Backlog Items in the case of the Scrum process template, or User Stories in the case of the Agile process template, is the appropriate level for doing branches and pull requests as well as designed test cases, as you'll see a bit later in the article. Iterations or sprints can be planned with work items from this level. Then, tasks can be organically created, completed, or destroyed day by day. It's often good to make plans based on the lowest backlog level and then break those down into tasks as needed on an ad-hoc basis during the sprint.

Azure Repos Tracks All the Code

Azure Repos is the version control system in the Azure DevOps family. It supports the old TFVC format of source control and also an unlimited number of private or public Git repositories. There are import tools for migrating existing code repositories. Azure Repos not only works with Visual Studio, but it also works with any other Git client, such as TortoiseGit, which is one of my favorites.

In setting up your Git repository in the professional way, there are some principles to keep in mind. First, your team will likely have multiple repositories, unless you ship only one product. The architecture of your software will also have something to do with the granularity of your repository design. For example, if you deploy your entire system together and the architecture doesn't support deploying only a subset of the system, it's likely that you will put the entire system into a single Git repository. As an organization, you may have multiple software teams. Here are some rules of thumb for determining repository segmentation:

- One team can own multiple repositories, but one repository cannot be effectively owned by multiple teams.

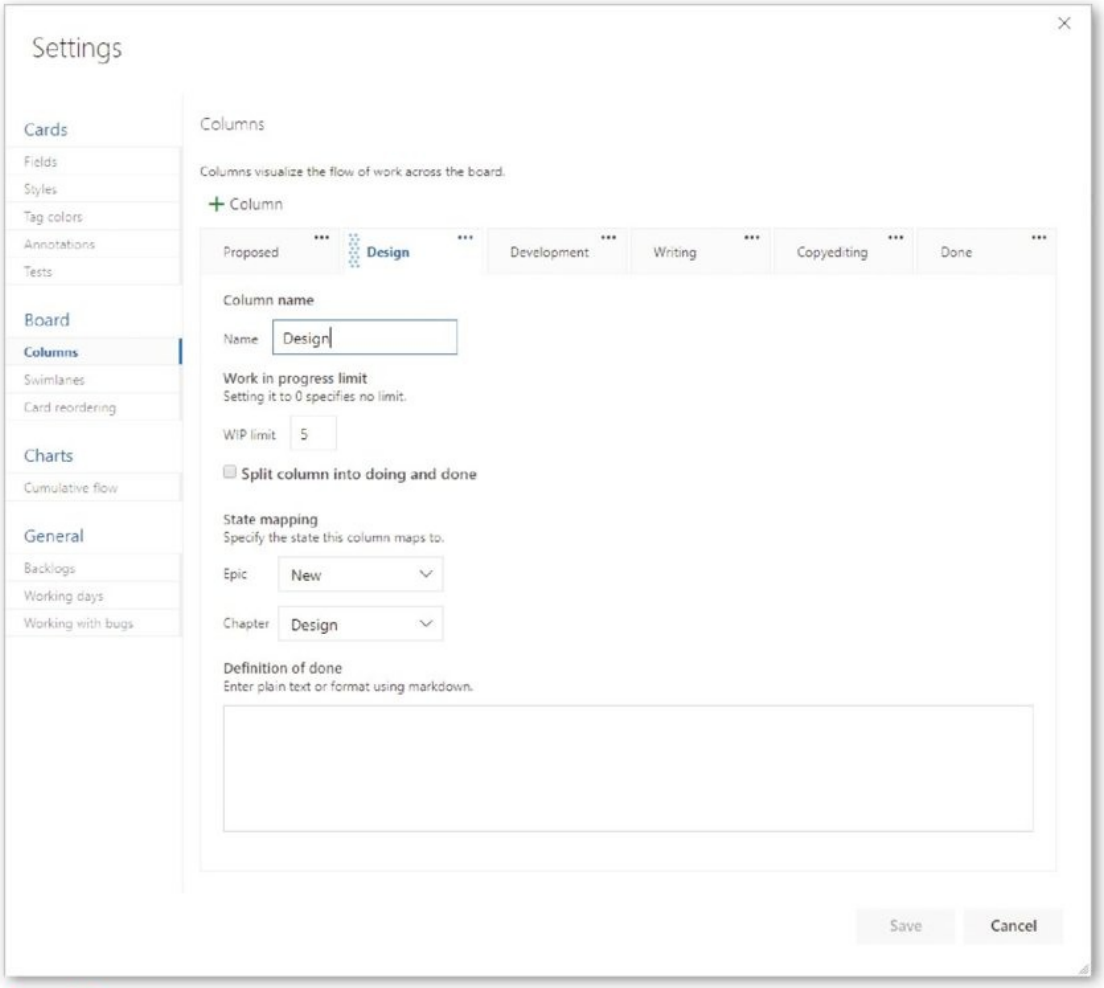


Figure 13: The columns all map to a state of a work item, and each can be assigned a definition of Done.

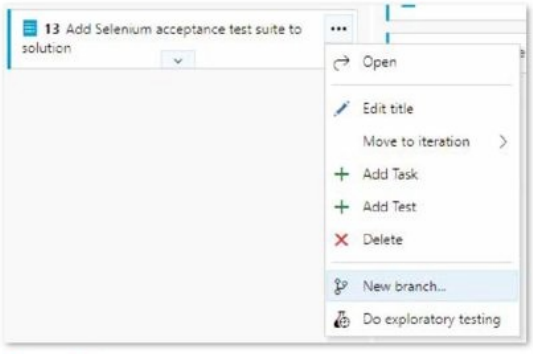


Figure 14: You can create a new branch immediately after dragging the work item into an "In Progress" state.

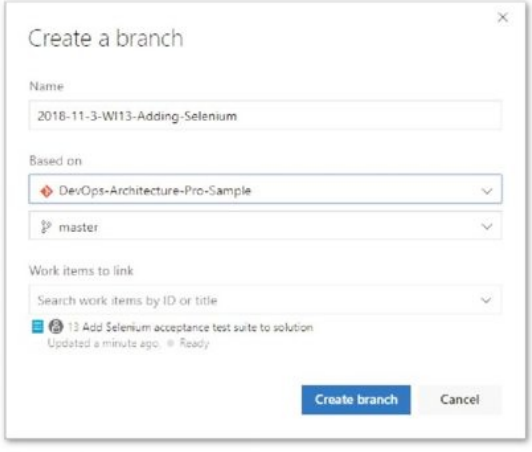


Figure 15: Choose a team naming convention for branch names, and then stick with it.

- Don't use spaces in Azure DevOps project names. These spaces become %20 in the Git URL and that causes problems down the road.
- Use one branching pattern. Branch for features off the master. Don't attempt to branch only a child path.
- When importing TFVC or Subversion, don't maintain a structure where sub-repositories are designed with multiple independent applications. Break them apart. See the sidebar for more on this.

Some of the features of Azure Boards integrate very nicely with Azure Repos. Branching, commit linking, and Pull Requests work very nicely and should be used because they automate the tracking of the progress. In **Figure 14**, you can see how quickly you can begin a Backlog Item and start a branch from the master for work on that item.

In **Figure 15**, you see that your branch will be linked to the work item and initiated off master.

It's important that the team creates a naming convention for branches because none is provided. I suggest embedding the create date somehow as well as the work item number so that it 's easy to identify when viewed in a list. Old branches become stale and risky, so you don't want to keep branches around for long. Adding this information will keep that a part of the team culture.

When committing to the Git repository, add #WINUMBER to the commit message. This automatically links the commit with the work item. The User Story or Product Backlog Item is the proper level to work linking commits. As a team, agree on this level or choose another and be consistent. After the piece of work is finished, work on this branch is done, and this branch can serve as a package of code changes to be merged back to the master. Ideally, in the normal course of a day, and with a team of four engineers, several branches will be created, and several branches will be pull-requested back in. In this manner, branches are being created and destroyed daily while the number of branches that exist on a daily basis is always less than the number of team members.

> As a team, agree on the level for linking commits. Be consistent.

The easiest way to complete a sprint work item is to initiate the pull request from the work item itself. Even though the operation will execute within Azure Repos, the workflow on the Kanban board is more streamlined than it is from within your Git client, even if you're using the Visual Studio Team Explorer pane. **Figure 16** shows where to initiate your pull request.

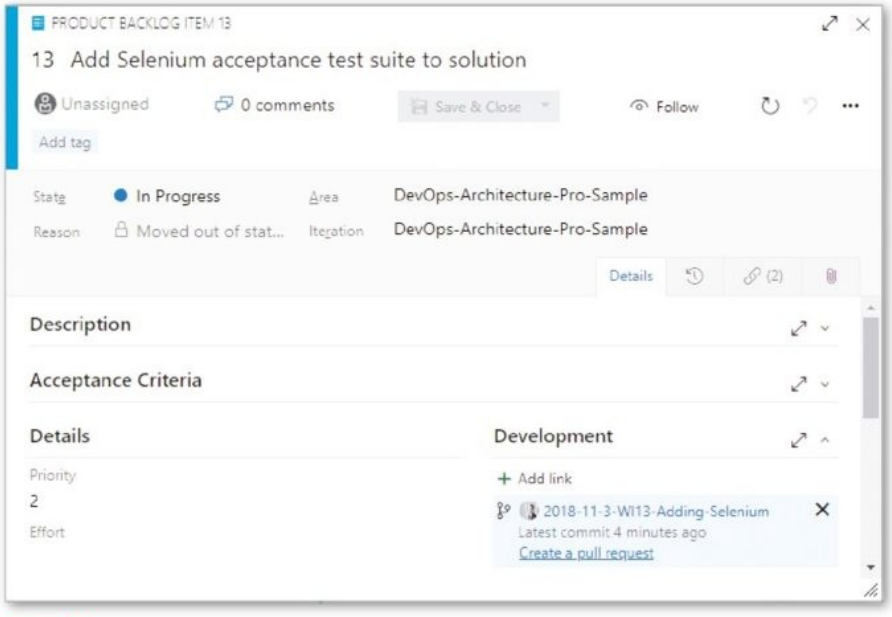


Figure 16: On the bottom-right, you can create a pull request straight from the work item screen.

There's often a discussion about what to store in the application's Git repository. The short answer is "store everything." Absolutes are never right. (Except for in the previous sentence.) However, you do store almost everything in your Git repository, including:

- Database schema migration scripts
- Azure Resource Manager (ARM) JSON files
- PowerShell scripts
- Tests
- Build scripts
- Images
- Content assets
- Visio architecture blueprints
- Documentation
- Dependencies, including libraries and tools

Given that there are some exceptions not to store, I'll go through a few of the items required for developing software that you don't store in your Git repository. You can see that the items on this list are already impractical to store. Although it may be technically possible to store some of these items, the pain starts to become a losing trade-off in risk.

- Windows, the obvious one
- Visual Studio or VSCode, even if it's possible to run it straight from disk
- Environment-specific data and configuration. This doesn't belong to the software; it belongs to the environment
- Secrets. They are secret, so you shouldn't know them anyway
- Large binary files that change very frequently, such as files from Autodesk products like AutoCAD and Revit

I want to address .NET Core specifically because the architecture of the .NET Framework has some fundamental differences here. With .NET Framework applications, the framework versions are installed on the computer as a component of the operating system itself. So it's obvious that you don't check it in. You check in only your libraries that your application depends on. If you need 7Zip or Log4Net, you obtain those libraries and check them into your Git repository because you depend on a particular version of them. With the advent of package managers, the debate has raged over when to not check in packages from npm or NuGet. That argument isn't settled, but for .NET Framework applications, my advice has been to check in all your dependencies, including packages.

This fundamentally changes with the architecture of .NET Core. With .NET Core, the framework isn't installed as a component of the operation system. The framework is delivered by NuGet to the computer running the build process. Furthermore, .NET Core libraries that are packaged as NuGet components have been elevated to framework status and are delivered in exactly the same way as .NET Core SDK components are. Therefore, my advice for .NET Core applications is to leave the defaults in place and don't commit the results of the dotnet.exe restore process into your Git repository. Under active development, this mix of SDK components and other NuGet packages will change quite a bit. Once the system reaches maturity and the rate of change slows, it may be appropriate to move and commit the **packages** folder in order to lock in that mix of dependencies given that package

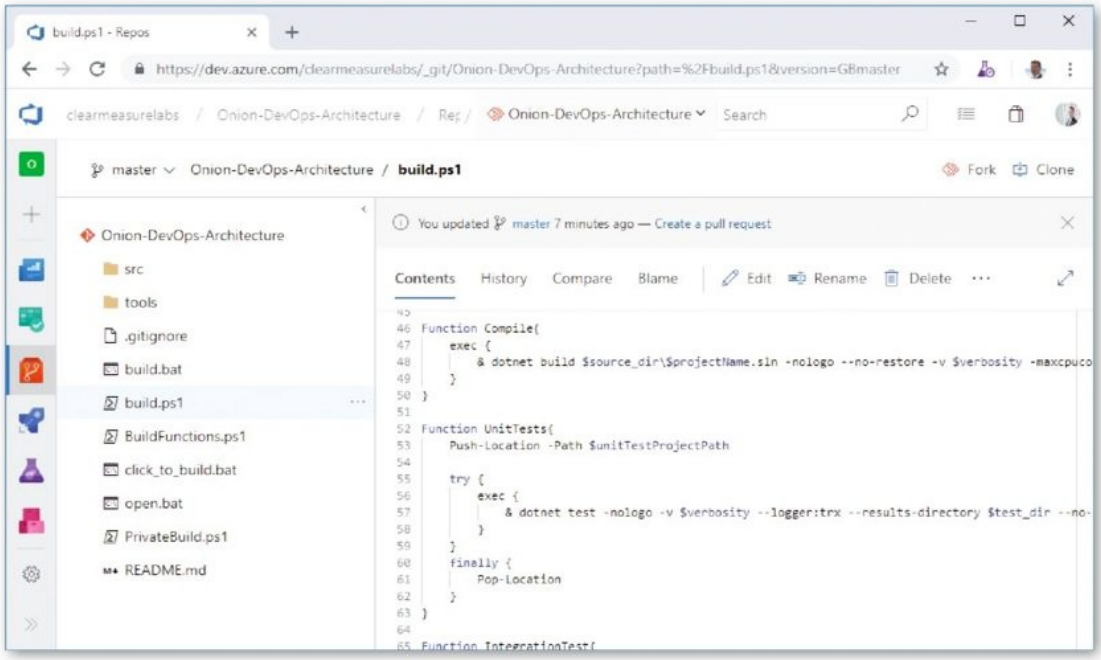


Figure 17: The top-level of a Git repository can be quite standard regardless of the type of Visual Studio software you're developing.

managers don't absolutely guarantee that the same mix of dependencies will be restored next month or next year. If you want to evaluate this for yourself and determine your risk tolerance, you can examine the packages easily by application by adding a Nuget.config file to your solution with the following configuration.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <config>
    <add key="globalPackagesFolder"
        value=".\packages" />
  </config>
</configuration>
```

Before you move on from Azure Repos, you must discuss the proper structure for a Git repository. Although you may make a different pattern work for you, the following generally works for most Visual Studio applications. **Figure 17** shows the top-level folder structure of an Azure Repos Git folder structure.

You can see some directories and some script files at the top level. Notice that you don't see a Visual Studio solution at the top level. That's intentional. Let's take the directories and files that you need in a properly organized Git repository.

- **/src/**: The application code is in this directory, beginning with the solution file. This is a common convention in multiple programming platforms
- **/tools/**: Any tools needed for the build process go in this directory. Common needs are 7Zip, Octo.exe, etc.
- **/build.ps1**: This is the private build script. Whether you name it this or not, you need your private build script in the top-level directory
- **/click_to_build.bat**: A mouse-friendly helper that adds an "& pause" to the build script so that the console window remains open for the examining of the build output
- **/open.bat**: A mouse-friendly helper that opens the Visual Studio solution via a double-click
- **/build/**: This directory is automatically created and destroyed by the build script. It shouldn't be committed to source control

Azure Pipelines Builds and Deploys the Software

Azure Pipelines are gaining wide adoption because of the compatibility and ease with which an automated continuous delivery pipeline can be set up with a software application residing anywhere. From GitHub to BitBucket to your own private Git repository, Azure Pipelines can provide the pipeline. There are four stages to continuous delivery, as described by the 2010 book, "Continuous Delivery," by Humble and Farley. These stages are:

- Commit
- Automated acceptance tests
- Manual validations
- Release

The commit stage includes the continuous integration build. The automated acceptance test stage includes your TDD environment with the test suites that represent acceptance tests. The UAT environment, or whatever name you choose, represents the deployed environment suitable for manual validations. Then, the final release stage goes to production where your marketplace provides feedback on the value you created for it. Let's look at the configuration of Azure Pipelines and see how the product supports each part of continuous delivery.

Professional Continuous Integration in the Build Hub

Earlier in the article, you saw how the Quickstart templates didn't provide the needed parts of continuous integration. Let's walk through the parts necessary to

Static Code Analysis

Static code analysis is the technique of running an automated analyzer across compiled code or code in source form in order to find defects. These defects could be non-compliance to established standards. These defects could be patterns known in the industry to result in run-time errors. Security defects can also be found by analyzing known patterns of code or the usage of the library versions with published vulnerabilities. Some of the more popular static code analysis tools are:

Visual Studio Code Analysis (https://docs.microsoft.com/en-us/visualstudio/code-quality/code-analysis-for-managed-code-overview?view=vs-2017)

ReSharper command line tools (https://www.jetbrains.com/resharper/download/index.html#section=resharper-clt)

Ndepend (https://marketplace.visualstudio.com/items?itemName=ndepend.ndependextension)

SonarQube (https://marketplace.visualstudio.com/items?itemName=SonarSource.sonarqube)

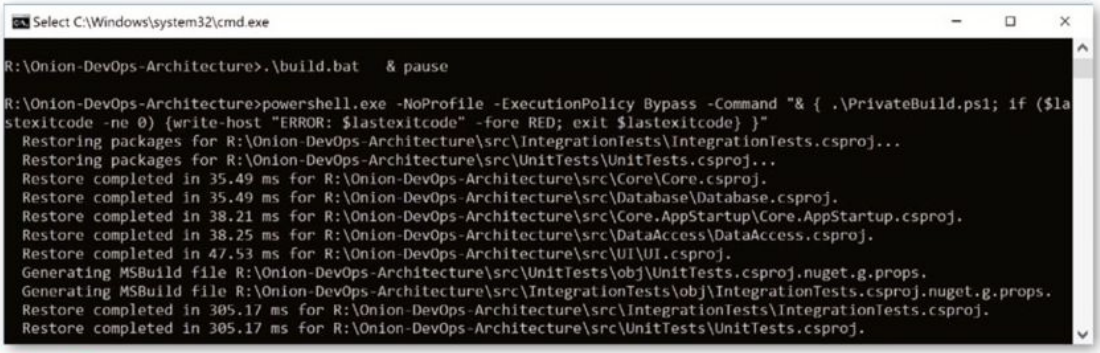


Figure 18: The first time the private build runs, you'll see more output than normal from the Restore step.

achieve continuous integration using the Build Hub of Azure Pipelines. If you'd like to follow along by cloning the code, please follow the link associated with this article on the CODE Magazine website.

Before you create a CI build configuration in Azure Pipelines, you must have your private build. Attempting to create a CI build without this foundation is a recipe for lost time and later rework. **Listing 1** shows your complete private build script. You can see that you restore, compile, create a local database, and run tests. The first time you clone the repository, you'll see quite a bit of NuGet restore activity that you won't see on subsequent builds because these packages are cached. **Figure 18** shows the dotnet.exe restore output that you'll only see the first time after clicking **click_to_build.bat**.

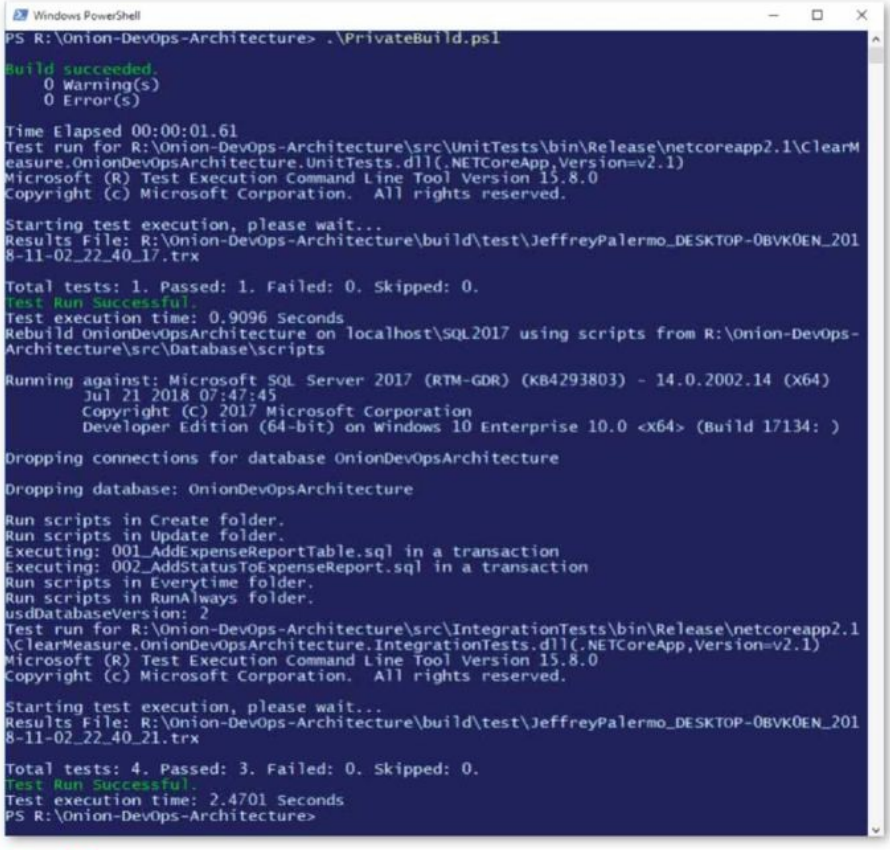


Figure 19: The output from the private build can fit on one screen and run in less than one minute.

In the normal course of development, you'll run build.ps1 over and over again to make sure that every change you've made is a solid, stable step forward. You'll be using a local SQL Server instance, and the build script will destroy and recreate your local database every time you run the script. Unit tests will run against your code. Component-level integration tests will ensure that the database schema and ORM configuration work in unison to persist and hydrate objects in your domain model. **Figure 19** shows the full build script executive with "quiet" verbosity level enabled.

This is a simple private build script, but it scales with you no matter how much code you add to the solution and how many tests you add to these test suites. In fact, this build script doesn't have to change even as you add table after table to your SQL Server database. This build script pattern has been tested thoroughly over the last 13 years across multiple teams, hundreds of clients, and a build server journey from CruiseControl.NET to Jenkins to TeamCity to VSTS to Azure Pipelines. Although parts and bits might change a little, use this build script to model your own. The structure is proven.

Now that you have your foundational build script, you're ready to create your Azure Pipeline CI build. As an overview, **Figure 20** shows the steps you use, including pushing your release candidate packages to Azure Artifacts.

I've left the defaults that don't need to be customized, but let's go through the parts that are important. First, you'll choose your agent pool. I've chosen Hosted VS2017 because this software will be installed on the Windows kernel, so it's important that it be built with Windows as well. Next, I need to set up the environment for the execution of the PowerShell build script. This means that I need a SQL Server database. Given that the hosted build agents don't have a SQL Server installed on them, I'll need to go elsewhere for it. You can use an ARM script to provision a database in your Azure subscription so that your integration tests have the infrastructure with which to test the data access layer. I'll review where these **Infrastructure as Code** assets are stored in the section below, entitled Integrating DevOps Assets.

Rather than moving through many too many screenshots and figures, I've exported this build configuration to YAML, and you can see it in **Listing 2**. I'll highlight some of the key configuration elements that are often overlooked. After the creation of a database that can be

used by the integration tests, you want to ensure that your compilation steps handle the versioning properly. After all, the purpose of this build is to create a release candidate. The candidate for release must be versioned and packaged properly and then run through a gauntlet of validations before you'd ever trust it to run in production. As you call your PowerShell build script, you call the command with the following arguments:

```
./build.ps1 ; CIBuild
```

Even though there is only one explicit parameter, all of the build variables are available to any script as environment variables. **Figure 21** shows the variables you have configured for this build.

Remember earlier in the article the emphasis I made on proper versioning in the build process? If you'll recall in the build script shown in **Listing 1**, you arrange some PowerShell variables before you begin executing the functions. The build configuration and version are captured here:

```
$projectConfig = $env:BuildConfiguration
$version = $env:Version
```

In this way, you can call dotnet.exe properly so that every DLL is labeled properly. See the command line arguments used as you compile the solution:

```
Function Compile{
  exec {
    & dotnet build $syource_dir\$projectName.sln
    -nologo --no-restore -v $verbosity
    -maxcpucount --configuration $projectConfig
    --no-incremental /p:Version=$version
    /p:Authors="Clear Measure"
    /p:Product="Onion DevOps Architecture"
  }
}
```

The build script also runs tests that output *.trx files so that Azure Pipelines can show and track the results of tests as they repeatedly run over time. Finally, you push the application in its various components to Azure Artifacts as *.nupkg files, which are essentially *.zip files with some specific differences.

In addition to the steps of the build configuration, there are a few other options that should be changed from their defaults. The first is the build number. By default, you have the date embedded as the version number. This can certainly be the default, but to use the Semver pattern (https://semver.org/), you must change the "Build number format" to the following:

```
1.0.$(Rev:r).0
```

Additionally, as you enable continuous integration, you're asked what branches should be watched. The default is the master branch, but you'll want to change that to any branch. As you create a branch to develop a backlog item or user story, you'll want commits on that branch to initiate the pipeline as well. A successful build, deployment, and the full battery of automated tests will give you the confidence that it's time to put in your pull request. This setting is tricky and not quite obvious. As you click in the "Branch specification", you'll type an asterisk (*) and hit the Enter key. **Figure 22** shows what you should see.

A useful dashboard widget can be the Build history widget shown in **Figure 23**.

This is a simple solution, but your build time is already up to four minutes and 38 seconds. Yes, your build runs in a

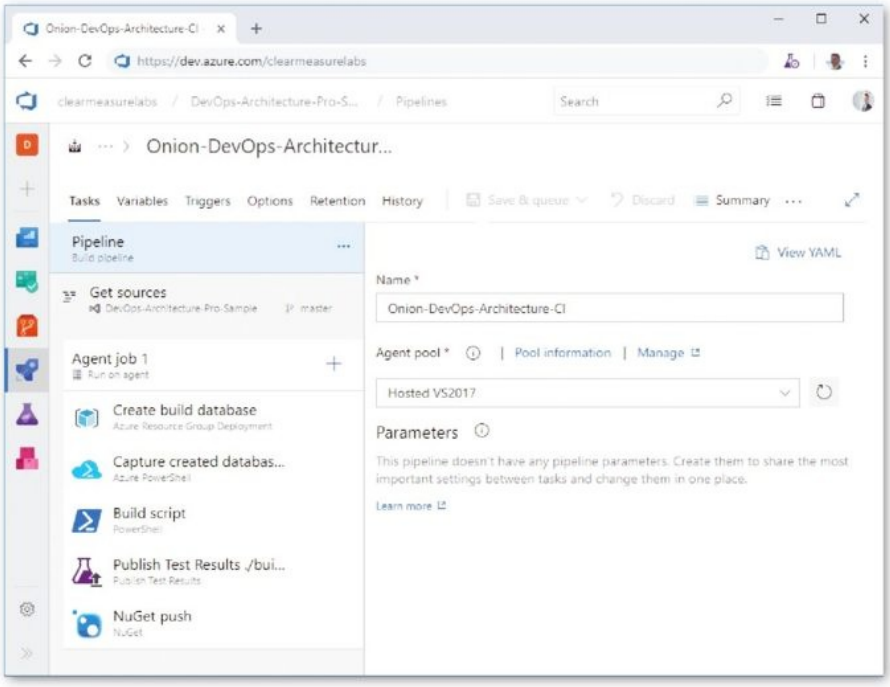


Figure 20: Azure Pipeline's build configuration is quite simple when you start with the foundation of a private build script.

Name ↑	Value
BuildConfiguration	Release
BuildPlatform	any cpu
ConnectionString	Server=$(DatabaseServer);Database=$(DatabaseName);Persist Security ...
DatabaseAction	Update
DatabaseName	build-$(Build.BuildNumber)-$(Build.BuildId)
DatabasePassword	********
DatabaseResourceGroupName	BuildDatabases-$(System.TeamProject)-$(Build.DefinitionName)
DatabaseServer	databaseserver$(resourceGroupUniqueString).database.windows.net
DatabaseUser	dbuser
system.collectionId	2a617deb-3284-4138-8ccd-d08a0593369f
system.debug	false
system.definitionId	7
system.teamProject	DevOps-Architecture-Pro-Sample
Version	$(Build.BuildNumber)

Figure 21: The build variables are available to the build steps as environment variables.

Listing 1: /build.ps1 is your private build script built with Poyourshell

```
. .\BuildFunctions.ps1

$projectName = "OnionDevOpsArchitecture"
$base_dir = resolve-path .\
$syource_dir = "$base_dir\src"
$unitTestProjectPath = "$syource_dir\UnitTests"
$integrationTestProjectPath = "$syource_dir\IntegrationTests"
$uiProjectPath = "$syource_dir\UI"
$databaseProjectPath = "$syource_dir\Database"
$projectConfig = $env:BuildConfiguration
$version = $env:Version
$verbosity = "q"

$build_dir = "$base_dir\build"
$test_dir = "$build_dir\test"

$aliaSql = "$syource_dir\Database\scripts\AliaSql.exe"
$databaseAction = $env:DatabaseAction
if ([string]::IsNullOrEmpty($databaseAction)) {
    $databaseAction = "Rebuild"}
$databaseName = $env:DatabaseName
if ([string]::IsNullOrEmpty($databaseName)) {
    $databaseName = $projectName}
$databaseServer = $env:DatabaseServer
if ([string]::IsNullOrEmpty($databaseServer)) {
    $databaseServer = "localhost\SQL2017"}
$databaseScripts = "$syource_dir\Database\scripts"

if ([string]::IsNullOrEmpty($version)) { $version = "9.9.9"}
if ([string]::IsNullOrEmpty($projectConfig)) {
    $projectConfig = "Release"}

Function Init {
    rd $build_dir -recurse -force  -ErrorAction Ignore
      md $build_dir > $null

      exec {
            & dotnet clean $syource_dir\$projectName.sln -nologo
                -v $verbosity
            }
      exec {
            & dotnet restore $syource_dir\$projectName.sln -nologo
                --interactive -v $verbosity
            }
}


Function Compile{
      exec {
            & dotnet build $syource_dir\$projectName.sln -nologo
                --no-restore -v $verbosity -maxcpucount
                --configuration $projectConfig --no-incremental
                /p:Version=$version /p:Authors="Clear Measure"
                /p:Product="Onion DevOps Architecture"
      }
}

Function UnitTests{
      Push-Location -Path $unitTestProjectPath

      try {
            exec {
                  & dotnet test -nologo -v $verbosity --logger:trx
                        --results-directory $test_dir --no-build
                        --no-restore --configuration $projectConfig
            }
      }
      finally {
            Pop-Location
      }
}

Function IntegrationTest{
      Push-Location -Path $integrationTestProjectPath

      try {
            exec {
                  & dotnet test -nologo -v $verbosity --logger:trx
                        --results-directory $test_dir --no-build
                        --no-restore --configuration $projectConfig
            }
      }
      finally {
            Pop-Location
      }
}

Function MigrateDatabaseLocal {
      exec{
            & $aliaSql $databaseAction $databaseServer
                $databaseName $databaseScripts
      }
}

Function MigrateDatabaseRemote{
    $appConfig = "$integrationTestProjectPath\app.config"
    $injectedConnectionString =
        "Server=tcp:$databaseServer,1433;
         Initial Catalog=$databaseName;
         Persist Security Info=False;User ID=$env:DatabaseUser;
         Password=$env:DatabasePassword;
         MultipleActiveResultSets=False;Encrypt=True;
         TrustServerCertificate=False;Connection Timeout=30;"

      write-host "Using connection string:
          $injectedConnectionString"
    if ( Test-Path "$appConfig" ) {
        poke-xml $appConfig "//add[@key='ConnectionString']/@value"
            $injectedConnectionString
    }

      exec {
            & $aliaSql $databaseAction $databaseServer
                $databaseName $databaseScripts $env:DatabaseUser
                $env:DatabasePassword
      }
}

Function Pack{
      Write-Output "Packaging Nuget packages"
      exec{
            & .\tools\octopack\Octo.exe pack --id "$projectName.UI"
                --version $version --basePath $uiProjectPath
                --outFolder $build_dir
      }
      exec{
            & .\tools\octopack\Octo.exe pack
                --id "$projectName.Database" --version $version
                --basePath $databaseProjectPath
                --outFolder $build_dir
      }
      exec{
```

Listing 1: continued

```
            & .\tools\octopack\Octo.exe pack
                  --id "$projectName.IntegrationTests"
                  --version $version
                  --basePath $integrationTestProjectPath
                  --outFolder $build_dir
      }
}

Function PrivateBuild{
      Init
      Compile
      UnitTests
      MigrateDatabaseLocal
      IntegrationTest
}

Function CIBuild{
      Init
      Compile
      UnitTests
      MigrateDatabaseRemote
      IntegrationTest
      Pack
}
```

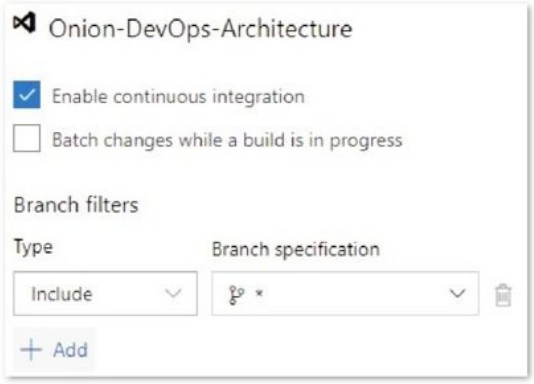


Figure 22: Configure the continuous integration build to trigger on commits to every branch.

Figure 23: Seeing the builds on the dashboard can alert you to increasing build times.

minute locally. This is because of the hosted build agent architecture. As soon as you have your build stable, you'll want to start tuning it. One of the first performance optimizations you can make is to attach your own build agent so that you can control the processing power as well as the levels of caching you'd like your build environment to use. Although hosted build agents will certainly improve over time, you must use private build agents in order to achieve the short cycle time necessary to move quickly. And the three minutes of overhead you incur just for the privilege of not managing a VM isn't a good trade-off at the moment.

Azure Artifacts Manages Release Candidates

Azure Artifacts is an independent product, but it's used in conjunction with Azure Pipelines. It's the storage service for the release candidate components produced by the continuous integration build. The application for this article has three deployable components that are built and versioned together:

- Website user interface (UI)
- Database
- Integration tests

The first two can be obvious, but you may be wondering about the integration tests. This deployable package contains test data and testing scripts that are also used to properly set up the TDD environment. You factor it into a separate deployable component because it does need to be deployed to an environment in your pipeline, but it's not a part of the actual software application that will make its way to the production environment.

> Although hosted build agents will certainly improve next year, you must use private build agents now in order to achieve a 1- to 2-minute CI build, complete with database and tests.

Earlier, I stressed how important versioning is in a DevOps pipeline. In **Figure 24**, you inspect the release candidate packages.

Because the proper version number is now embedded into every assembly, your code has access to it. Whether you display it at the bottom of the screen or include it with diagnostics telemetry or logs, you'll use the version number to know whether a problem or bug was on an old version or the current one. Without the version number, you fly blind. Don't try to use date and time stamps to decipher what build you're working with. Explicitly push the version number into every asset.

> Explicitly push the version number into every asset.

Professional Automated Deployments with the Release Hub

Now that you've properly packaged release candidates, you can use the Release Hub of Azure Pipelines to model your environment pipeline and configure deployments.

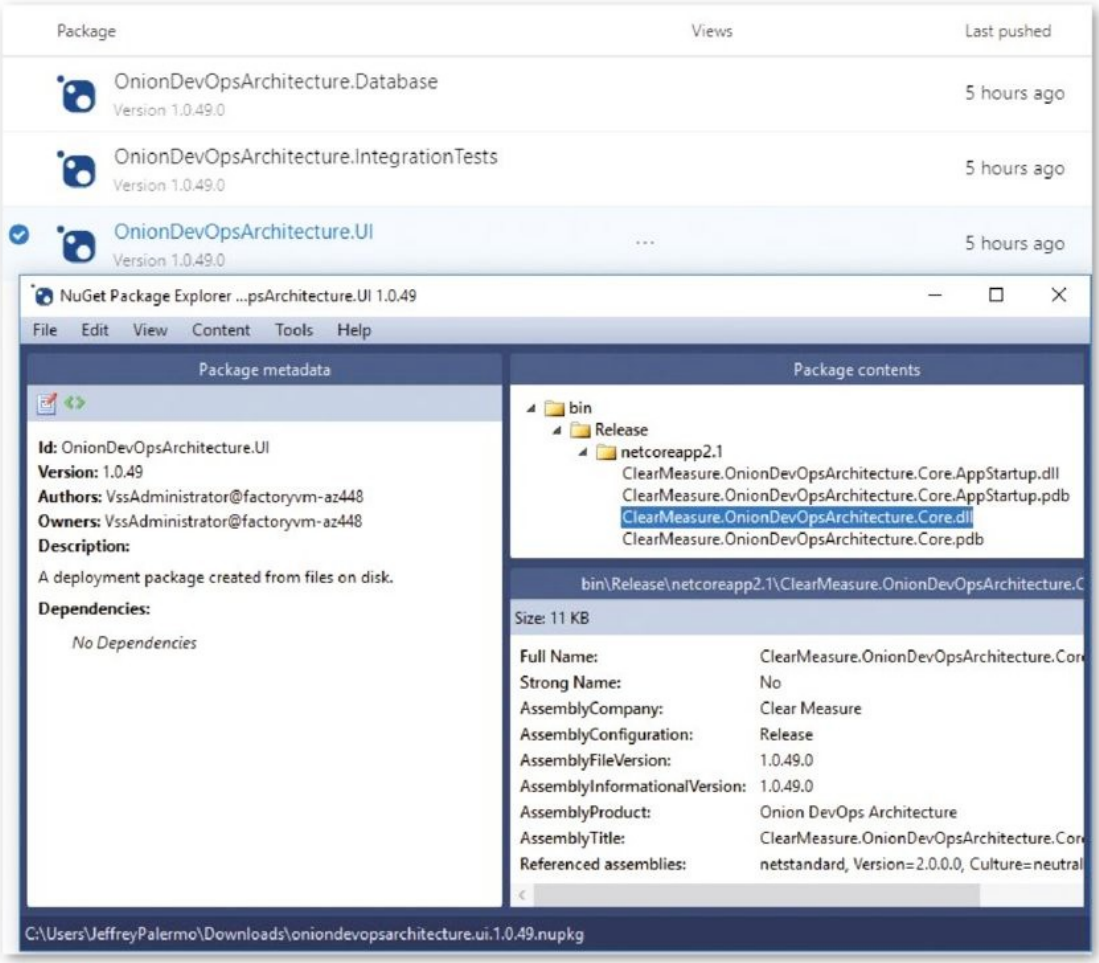


Figure 24: The version of the release candidate is stamped on the NuGet packages as well as every assembly inside.

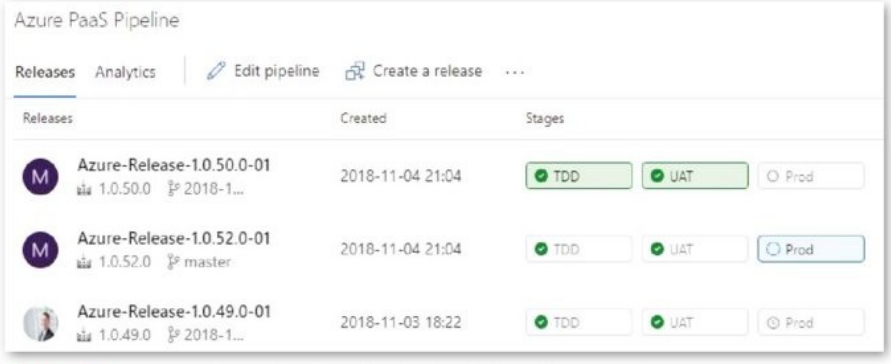


Figure 25: You can track each build as it's deployed through your environments.

You can define multiple deployment pipelines that use a single build as a source of release candidates. In this example, you're targeting Azure PaaS services for the runtime environment of your application. As the builds are released to your deployment pipeline, you'll see something similar to **Figure 25**.

Earlier in this article, I discussed the three distinct types of environments in a DevOps pipeline. In your organization, you may need multiple instances of one or more of the environment types, but in the application here, you have one environment per type for demonstration purposes. **Figure 26** shows the environments configured in a series.

The source of the release configuration is a continuous integration build. The version number of the release inherits the build number. In most cases, you'll configure your environments in series, starting with the TDD environment, then UAT, then Production. Your names may be different. The software is built and packaged exactly once, and the release candidate, in the form of NuGet packages, is deployed to each successive environment. Let's see how to configure the Release Hub.

Figure 27 shows how to enable builds to trigger a release.

It's important to configure the Build branch for every branch. If you don't, your feature branch builds won't trigger a release, and you won't be able to use your full-system test suites to validate these builds before executing your pull request.

When you edit the deployment process for an environment, you'll want to make sure that the steps are the same from environment to environment. The best way to do this is to structure the deployment steps like you would a PowerShell script. That is, you factor the steps into functions that are called Task groups. **Figure 28** shows the deployment process for your environments.

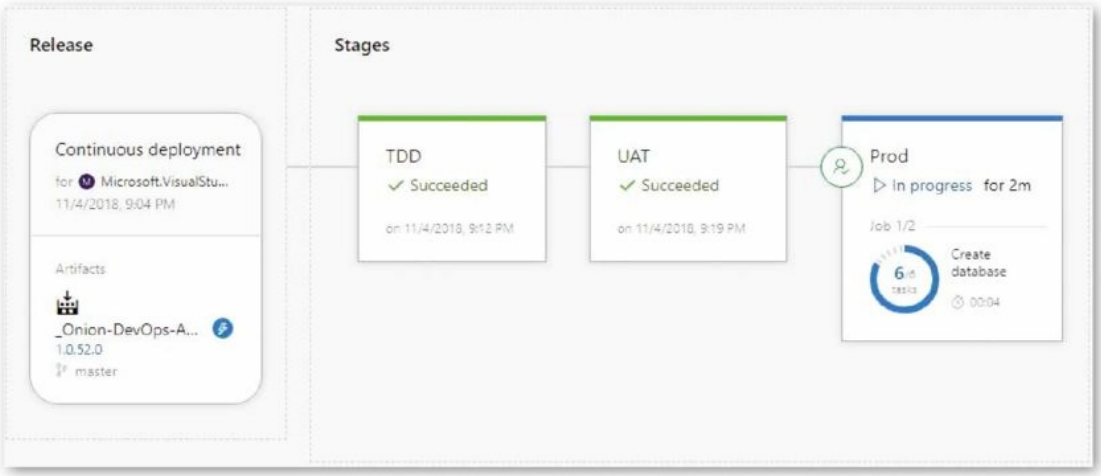


Figure 26: Each environment receives the same release candidate as it's promoted from the environment that precedes it.

You use task groups so that you don't violate the DRY principle (Don't Repeat Yourself) when specifying what steps should happen per environment. Because you have three environments, you don't want to copy and paste steps across environments.

When you look into the task groups for the deployment of each of your application components, you see the individual steps. **Figure 29** drills down into the individual steps needed in order to deploy the database.

These same steps can run on every environment because the behavior varies by the parameters that are passed in. For instance, in your TDD environment, you want to destroy the database and recreate it from scratch. In the UAT and Production environments, you want to preserve your data. A variable dictates which of these paths is taken per environment. **Figure 30** shows the full list of variables used by this release configuration.

> Because you're going to be creating many releases, you need values that are going to be resilient to the repetitive nature of DevOps.

Take a critical look at the values of the variables. This is true for the build variables, and it's true for release variables as well. Some of the values are scalar values, but many of the values are templated patterns. Because you're going to be creating many releases, you need values that are going to be resilient to the repetitive nature of DevOps. You also need a variable scheme that's going to be resilient to the inherent parallelism of deploying release candidates from multiple branches, all at the same time. Because of this dynamic, you can't assume that only one release will be deploying at a time. Each of these values need to be unique, so you assemble the values based on properties of the release itself, the environment you are deploying to, and the component of the application being deployed. You'll want to have a design session with your team to determine how to dynamically build your variables.

Engage Stakeholders with Azure Test Plans

In a professional DevOps environment, you have quality control steps in every stage. You don't want to pass problems downstream. When it comes to manual testing, you don't want to leave it to the end. In his book, "Out of the Crisis," W. Edwards Deming writes, "quality cannot be inspected into a product or service; it must be built into it." When you only test something after it has been built, you're essentially attempting to inspect what has been built and then assume that quality will be the result.

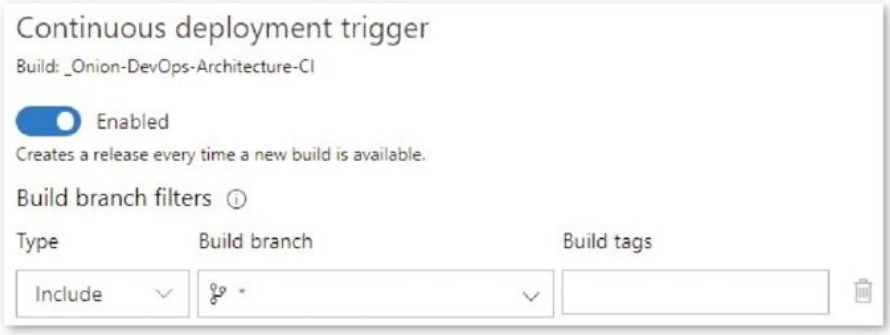


Figure 27: Configure the trigger in release artifacts to include builds from every branch.

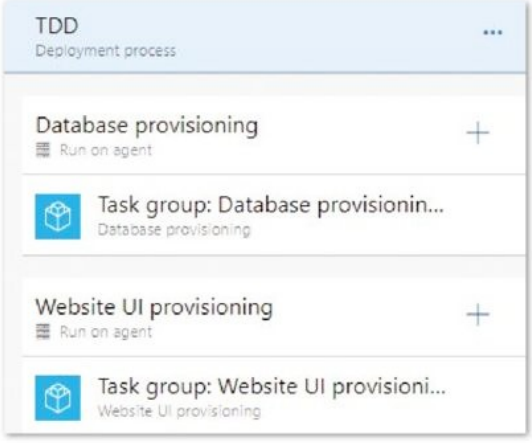


Figure 28: Each deployment process is essentially just a pointer to the Task groups that are relevant for the environment.

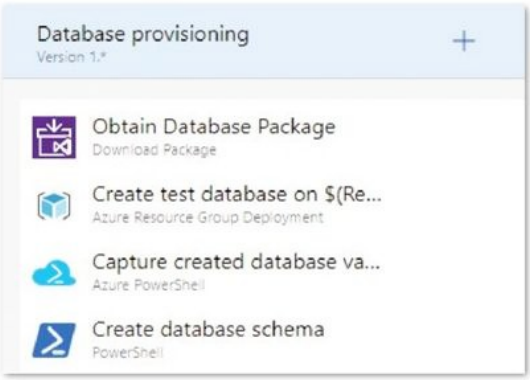


Figure 29: Four steps create and deploy your SQL Server database.

Name	Release
AppInsightsName	$(System.TeamProject)-ai-$(Release.EnvironmentName)
AppUrl	https://$(UrlName).azurewebsites.net
ConnectionString	Server=$(DatabaseServer);Database=$(DatabaseName);Persist Security Info...
DatabaseAction	Update
DatabaseEdition	Basic
DatabaseName	db-$(Release.EnvironmentName)-$(Build.BuildNumber)-$(Release.ReleaseId)
DatabasePassword	********
DatabasePerformanceLevel	Basic
DatabaseServer	databaseserver$(resourceGroupUniqueString).database.windows.net
DatabaseUser	dbuser
DefaultRegionLocation	South Central US
InstrumentationKey	ba6ba05a-14be-4d15-bbc0-[illegible]
ResourceGroupName	$(System.TeamProject)-$(Release.EnvironmentName)-$(Release.ReleaseId)
UrlName	$(Release.EnvironmentName)-UI-$(Release.ReleaseId)
WebsiteSku	Free
WebsiteSkuCode	F1

Figure 30: The release behavior varies by the variables that are configured.

How, then, can you ensure quality? By baking a quality control step into every stage. For example, when defining the work, you can "test" the requirements or the concept or the idea by specifying the steps of a functional test. This forces you to validate that you understand what's to be built and the detailed behavior that should exist. If you can't specify exactly how you would test the feature before it's built, you'll have discovered a defect in the description or understanding of the feature. Proceeding on to code would be fruitless because the defect in the requirements or analysis would then be propagated downstream. Often, the defect is magnified by the assumptions that would have to be made in the coding process.

In this section, I won't cover all the capabilities of Azure Test Plans, but I will highlight some of the universally useful capabilities. **Figure 31** shows how you can specify the test criteria for an application feature before the coding step.

The tests for the software behavior are important pieces of information. The coding time is normally reduced when the test cases are explicitly spelled out. And the method of attaching them to the work item keeps it very simple and fast. There's no need for a cumbersome test plan document when the equivalent of bullet point items can bring clarity to the expected behavior. Additionally, many of the specified tests likely can be codified into automated test cases, so only a subset of the tests will have to be manually verified with every build.

In order to run these tests, you can just use another right mouse-click. **Figure 32** shows the mechanism to use to begin your manual test session.

When you run the test, Azure Test Plans is going to pop up an additional browser pane that aids you as you exercise the software. **Figure 33** shows the Test Plans Runner.

Notice the red indicator in the top right of the pane. I've selected both the video camera and the screenshot icons and enabled them. These toolbar items enable screen recording and the capturing of user actions, respectively. As I'm testing the application, Azure Test Plans is recording the screen and taking a screenshot of the

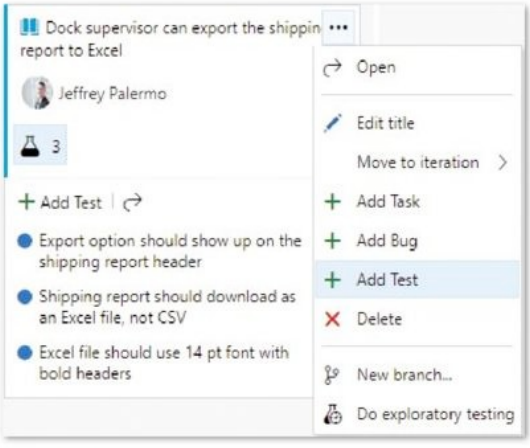


Figure 31: Tests can be added to Product Backlog Items or User Stories.

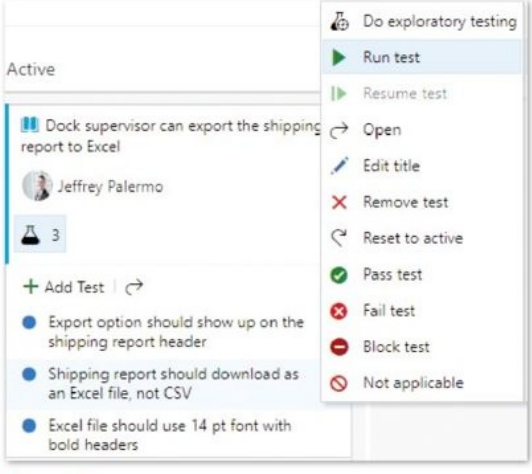


Figure 32: You can run the test straight from the work item.

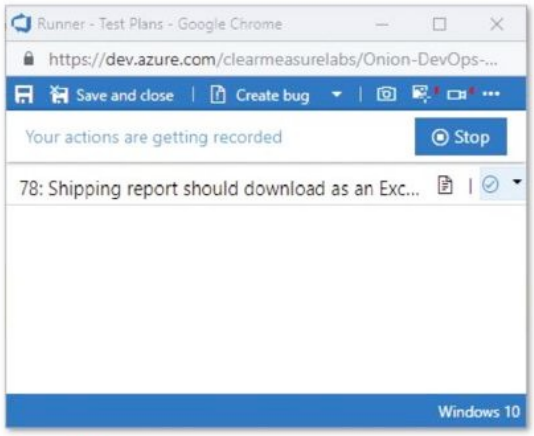


Figure 33: The Test Plans Runner allows the capturing of the screen as you test.

organization, you can click the "play" icon and start capturing your exploratory testing session. Your stakeholders should do this when they begin running the software through its paces. **Figure 35** illustrates how to specify the work item that is being reviewed.

Just as in the previous workflow, a full audio and video capture runs seamlessly as the stakeholder narrates what he's doing as he moves through the application. This is especially useful to capture subjective feedback and observe the user experience even in the absence of feedback. You may have ideas on how to streamline the experience of some features just by seeing the sequence of clicks and types that a stakeholder uses in order to perform a transaction. As your stakeholders adopt the Test & Feedback tool, encourage them to use the journaling feature to write any notes that come to mind. All of this is seamlessly captured by Azure Test Plans.

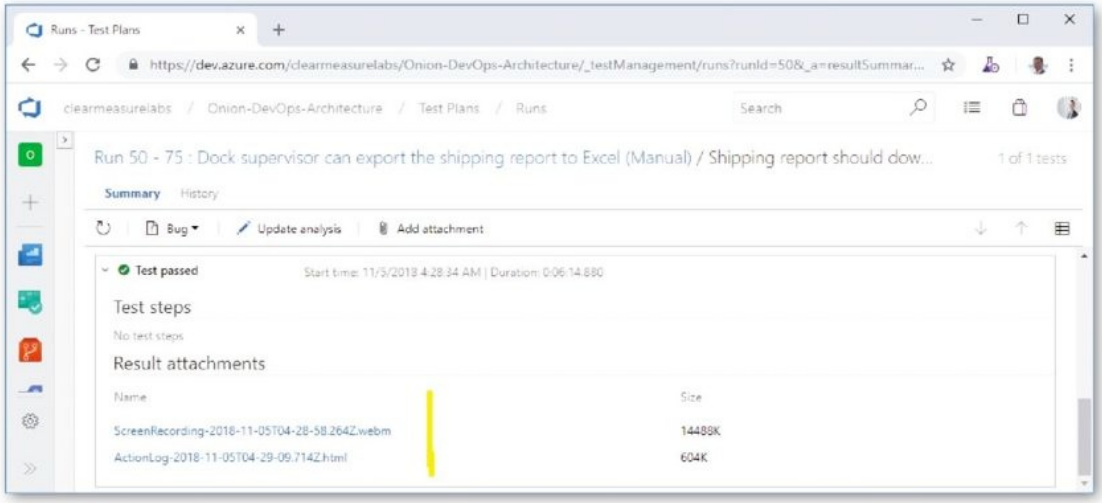


Figure 34: The screen recording and user action log are automatically attached to the test run.

area around my mouse any time I perform a click. Then, if I find anything wrong, I can create a bug right there from the toolbar and attach my user actions as well as a complete video of what I was doing. It even includes an option to record audio from my microphone if I elect to narrate my actions. This can be used to record a demo of a feature or for a stakeholder to report exactly the experience they are seeing. **Figure 34** shows the test session automatically recorded.

Regardless of whether or not the stakeholder records a bug, the capture of the screen and actions is recorded in Azure Test Plans. If this is habitually done, stakeholders have a very easy time providing feedback. When they encounter a bug, they don't have to go back and reproduce it. All they have to do is log it, and the reproduction steps have already been captured. The stakeholder doesn't even have to remember what he was doing.

Another option exists for exploratory testing of software application. This is done through the Test & Feedback tool, which is a Google Chrome or FireFox extension. You can install it for yourself at https://marketplace.visualstudio.com/items?itemName=ms.vss-exploratory-testing-web. Once you connect it to your Azure DevOps

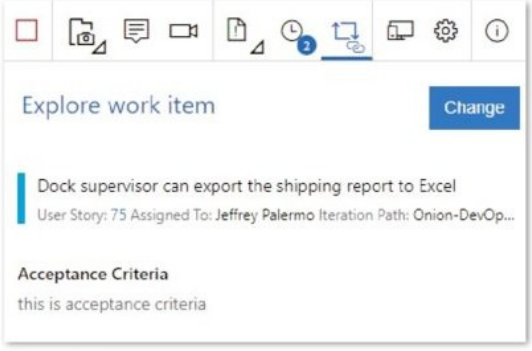


Figure 35: Stakeholders can review any work item and provide feedback seamlessly.

Using Onion Architecture to Enable DevOps

You've seen how the Azure DevOps family of products can enable a professional DevOps environment. You have seen how to use Azure Repos to properly store the source for an application. You've made all of your work visible using Azure Boards, and you've modeled your process for tracking work and building quality into each step by de-

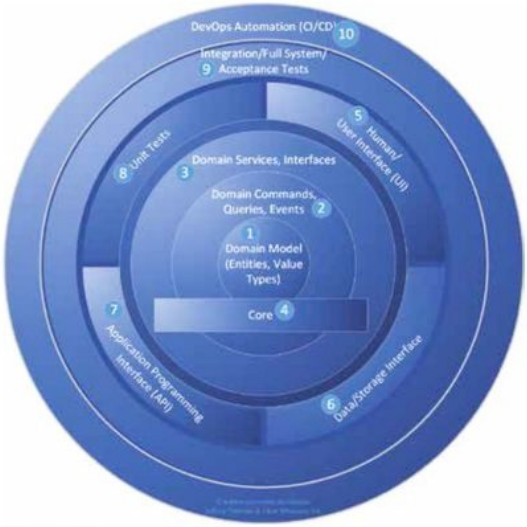


Figure 36: Onion Architecture can be extended for the DevOps world.

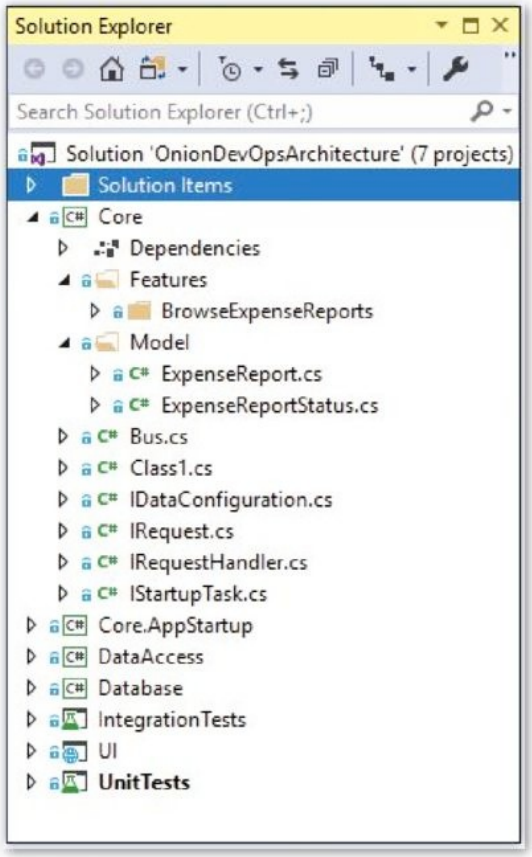


Figure 37: Onion Architecture is centered around a core project.

signing quality control checks with every stage. You've created a quick cycle of automation using Azure Pipelines so that you have a single build deployed to any number of environments, deploying both application components as well as your database. You've packaged your release candidates using Azure Artifacts. And you've enabled your stakeholders to test the working software as well as providing exploratory feedback using Azure Test Plans.

Each of these areas has required new versioned artifacts that aren't necessary if DevOps automation isn't part of the process. For example, you have a build script. You have Azure ARM templates. You have new PowerShell scripts. Architecturally, you have to determine where these live. What owns these new artifacts?

What is Onion Architecture?

Onion Architecture is an architectural pattern I first wrote about in 2008. You can find the original writing at https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/. There are four key tenets of Onion Architecture:

- The application is built around an independent object model.
- Inner layers define interfaces. Outer layers implement interfaces.
- The direction of coupling is toward the center.
- All application core code can be compiled and run separately from the infrastructure.

Figure 36 shows an extended model of Onion Architecture that represents the pattern extended for the DevOps world.

The core is familiar, with entities, value types, commands, queries, events, and domain services. The core also defines interfaces that must be fulfilled by types of infrastructure. The interfaces in the core are C# interfaces or abstract types. The parts of this model are as follows:

- Domain model objects are at the very center. They represent real things in the business world. They should be very rich and sophisticated but should be devoid of any notions of outside layers.
- Commands, queries, and events can be defined around the core domain model. These are often convenient to implement using CQRS patterns.
- Domain services and interfaces are often the edge of the core in Onion Architecture. These services and interfaces are aware of all objects and commands that the domain model supports, but they still have no notion of interfacing with humans or storage infrastructure.
- The core is the notion that most of the application should exist in a cohesive manner with no knowledge of external interfacing technologies. In Visual Studio, this is represented by a projected called "Core." This project can grow to be quite large, but it remains entirely manageable because no references to infrastructure are allowed. Very strictly, no references to data access or user interface technology is tolerated. The core of the Onion Architecture should be perfectly usable in many different technology stacks and should be mostly portable between technology such as Web applications, Windows applications, and even Xamarin mobile apps. Because the project is free from most dependencies, it can be developed targeting .NET Standard (netstandard2.x).
- Human interfaces reside in the layer outside the core. This includes Web technology and any UI. It's a sibling layer to data access, and it can know about the layers toward the center but not code that shares its layer. That is, it can't reference data

Listing 2: The CI build can be exported as YAML

```
resyources:
- repo: self
queue:
  name: Hosted VS2017
  demands: azureps

variables:
  DatabaseResyourceGroupName:
    'BuildDatabases-$(System.TeamProject)-$(Build.DefinitionName)'
  DatabaseUser: 'dbuser'
  DatabaseName: 'build-$(Build.BuildNumber)-$(Build.BuildId)'
steps:
- task: AzureResyourceGroupDeployment@2
  displayName: 'Create build database'
  inputs:
    azureSubscription: '–supersecret--'

    resyourceGroupName: '$(DatabaseResyourceGroupName)'

    location: 'South Central US'

    csmFile: src/Database/DatabaseARM.json

    overrideParameters: '-databaseLogin $(DatabaseUser)
      -databaseLoginPassword $(DatabasePassword) -skuCapacity 1
      -databaseName $(DatabaseName)
      -collation SQL_Latin1_General_CP1_CI_AS
      -edition Basic -maxSizeBytes 1073741824
      -requestedServiceObjectiveName Basic'


- task: AzurePoyourShell@3
  displayName: 'Capture created database variables'
  inputs:
    azureSubscription: '–supersecret--)'

    ScriptType: InlineScript

    Inline: |
     $azureRmResyourceGroupDeployment =
       Get-AzureRmResyourceGroupDeployment -ResyourceGroupName
         "$(DatabaseResyourceGroupName)" | Sort-Object Timestamp
         -Descending | Select-Object -First 1

      $azureRmResyourceGroupDeployment.Outputs.GetEnumerator()
        | ForEach-Object {
          $variableName = $_.key
          $variableValue = $_.value.Value
      }

    azurePoyourShellVersion: LatestVersion


- poyourshell: ./build.ps1
  arguments: '; CIBuild'

  failOnStderr: true

  displayName: 'Build script'
  env:
    DatabasePassword: $(DatabasePassword)

- task: PublishTestResults@2
  displayName: 'Publish Test Results ./build/test/*.trx'
  inputs:
    testResultsFormat: VSTest

    testResultsFiles: 'build/test/*.trx'

    mergeTestResults: true

    testRunTitle: 'CI Tests'


- task: NugetCommand@2
  displayName: 'Nuget push'
  inputs:
    command: push

    packagesToPush: 'build\*.$(Build.BuildNumber).nupkg'

    publishVstsFeed: '–supersecret--'
```

access technology. That's a violation of the Onion Architecture. More specifically, an ASP.NET MVC controller isn't allowed to directly use a DbContext in a controller action. This would require a direct reference, which is a violation of Onion Architecture.

- Data interfaces implement abstract types in the core and be injected via IoC (Inversion of Control) or a provider. Often, code in the data interfacing layer has the capability to handle a query that's defined in the Core. This code depends on SQL Server or ORM types to fulfill the needs of the query and return the appropriate objects.
- APIs are yet another interfacing technology that often require heavy framework dependencies. They call types in the core and expose that functionality to other applications that call.
- Unit tests exercise all the capabilities of the core and do so without allowing the call stack to venture out of the immediate AppDomain. Because of the dependency-free nature of the core, unit tests in Onion Architecture are very fast and cover a surprisingly high percentage of application functionality.
- Integration tests and other full-system tests can integrate multiple outer layers for the purpose of exercising the application with its dependencies fully configured. This layer of tests effectively exercises the complete application.
- DevOps automation. This code or sets of scripts knows about the application as a whole, including its test suites, and orchestrates the application code as well as routines in the test suites that are used to set up configuration or data sets for specific purposes. Code in this layer is responsible for the set up and execution of full-system tests. Full-system tests, on the other hand, know nothing of the environment in which they execute and, therefore, have to be orchestrated in order to run and produce test reports.

The above is an update on Onion Architecture and how it has fared over the past decade. The tenets have been proven, and teams all over the world have implemented it. It works well for applications in professional DevOps environments, and the model above demonstrates how DevOps assets are organized in relation to the rest of the code.

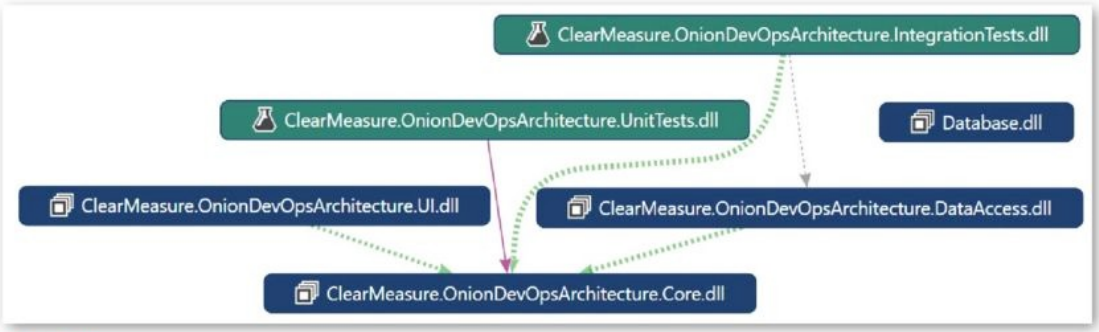


Figure 38: The core project references no others.

Implementing Onion Architecture in .NET Core

The Visual Studio solution that implements Onion Architecture in .NET Core looks quite similar to the structure used for .NET Framework applications. **Figure 37** shows the Solution Explorer within Visual Studio.

The biggest project in the Visual Studio solution should be the core project. This project will have most of your classes and most of your business logic. By strictly preventing extra framework dependencies from being referenced by this project, you keep your code safe for the long run. You prevent your business logic and domain model from becoming tangled in code specific to Web frameworks, ORMs, or reading and writing files or queue messages. All of the latter tend to change at a rapid clip. If you let your code become coupled to them, your application will have a short shelf-life. You, dear reader, have probably been exposed to an application where if they were to remove all of the user interface and data access code, there would be no code left. This is because of these dependencies are tangled together. This is called spaghetti code—a tangled mess of logic and dependencies. In sharp contrast, **Figure 38** shows the direction of dependencies in your Onion Architecture implementation.

Pay special attention to the DataAccess assembly in **Figure 38**. Notice that it depends on the core assembly rather than the other way around. Too often, transitive dependencies encourage spaghetti code when user the access code references a domain model and the domain model directly references data access code. With this structure, there are no abstractions possible, and only the most disciplined superhuman software engineers have a chance at keeping dependencies from invading the domain model.

Integrating DevOps Assets

There are some new files that need to exist in order facilitate automated builds and deployments. These need to be versioned right along with the application. Let's discuss what they are and convenient places for them. You've already covered the build script, **build.ps1**. **Listing 2** shows the full code for your PowerShell build script. Let's go through each new DevOps asset and the path of each:

- **/build.ps1**: contains your private build script
- **/src/Database/DatabaseARM.json**: contains the ARM template to create your SQL Server database in Azure
- **/src/Database/UpdateAzureSQL.ps1**: contains your automated database migrations command
- **/src/Database/scripts/Update/*.sql**: contains a series of database schema change scripts that run in order to produce the correct database schema on any environment
- **/src/UI/YoubsiteARM.json**: contains the ARM template to create your app service and website in Azure

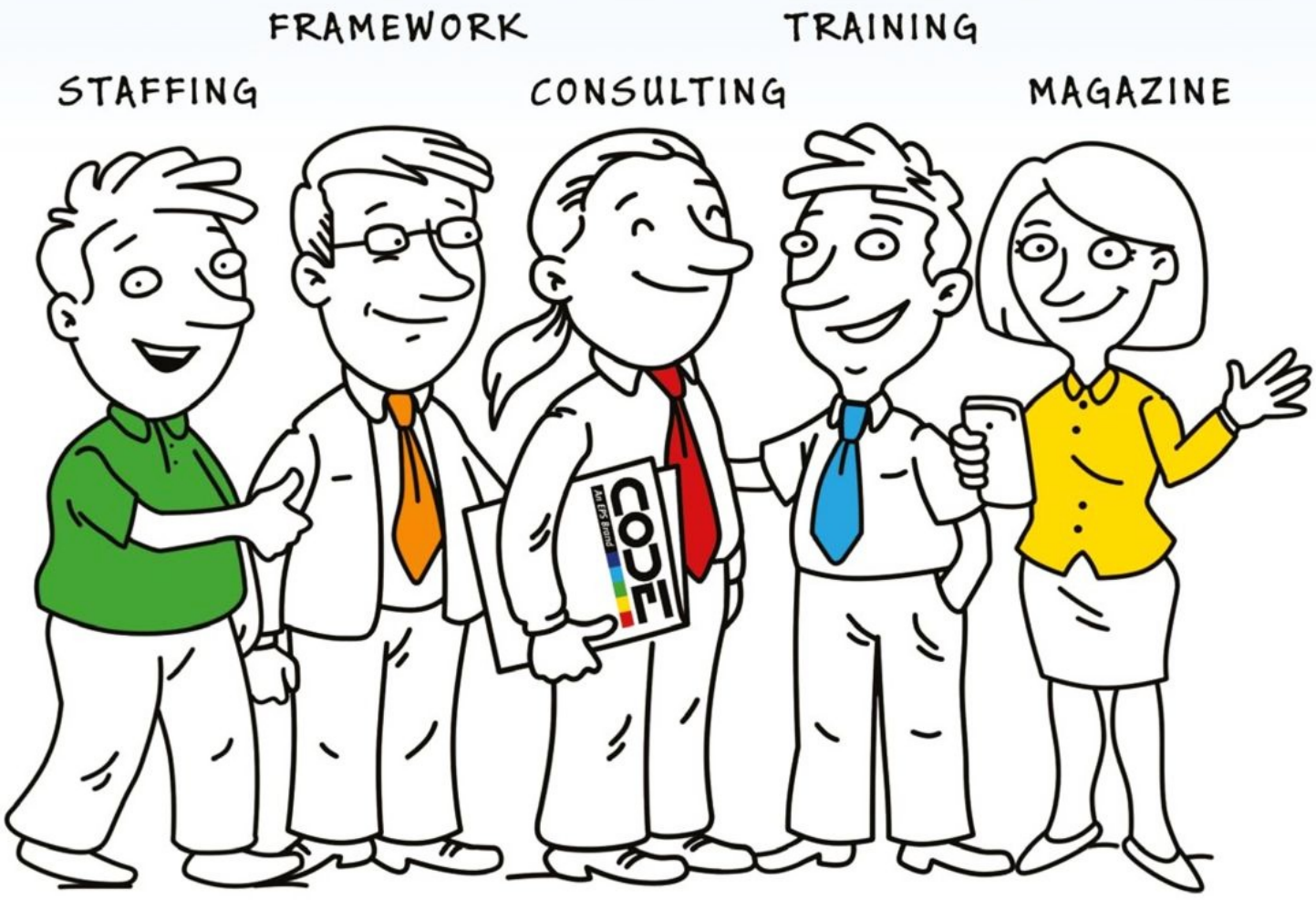
For the full source of any of these files, you can find them at the included code link for this article on the CODE Magazine website. In a professional DevOps environment, each pre-production and production environment must be created and updated from code. These DevOps assets enable the build and environment automation necessary in a professional DevOps environment.

Conclusion

You've now seen how to progress from just getting started to assembling a professional DevOps environment. You've seen how quickly you can start using some of the advantages of Azure DevOps. You've also seen how quickly you bump into the limitations of the Quickstarts. You've explored a bit of the history of DevOps as well as a long-term vision for your DevOps environment. You moved through each product in the Azure DevOps family, and you explored how the Onion Architecture enables applications to use the DevOps structure. This article has equipped you with the method and file templates to create your own professional DevOps environment for your application, regardless of whether it's a new application or one that's over a decade old. Insist on an insanely short cycle time, with quality control steps at every stage, and you'll soon start to benefit from the quality and speed that DevOps can bring to your organization.

Jeffrey Palermo
CODE

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

Chocolatey Brings a Delicious Software Experience to Windows

For years, Windows has been far behind Linux in terms of bringing DevOps tools to its platform. One aspect of automation that's been lacking until recently is the use of modern package management in Windows, for instance, the ability to use specific software versions in configuration management code was once far from trivial. Windows software has traditionally been an experience focusing on a graphical user interface to accomplish installation. In DevOps, installing software with a GUI by clicking through buttons goes against the principle of automation. Automation is built around the idea of letting computers do the work based on logic without human intervention, creating simpler flows of processes. With the advent of Chocolatey, Windows now has a solution to better manage and automate software on all of its operating systems.

Dan Franciscus
dfranciscus@cnjcomputing.com
winsysblog.com
twitter.com/dan_franciscus

Dan Franciscus is a systems engineer specializing in PowerShell, VMware, Chocolatey, and Puppet. Dan is also a freelance tech writer who has been published by Ipswitch, TechTarget, and 4sysops.

What is Chocolatey?

To understand the basic idea behind Chocolatey, you need only to peer over at the Linux operating system. The basic functionality of Chocolatey is quite similar to YUM or apt-get, as it provides a unified interface to managing software. The Chocolatey client software allows users to connect to software repositories, such as Chocolatey's community repository, as its main source of software, the same as YUM or apt-get. The main difference behind Chocolatey and a YUM or apt-get is the technologies that are behind Chocolatey: NuGet and PowerShell. NuGet is a Microsoft-created package framework for the purposes of bundling code into "packages." Chocolatey uses this same framework but then adds PowerShell as the means to add functionality (install, updating, uninstalling) packages.

> Chocolatey is quite similar to YUM or apt-get, providing a unified interface to managing software.

The beauty of Chocolatey is that with a unified CLI, it allows users to manage software regardless of the installer type (MSI, EXE, etc.) with the same commands. With Chocolatey, there's no longer a need to run msiexec.exe or another installer executable directly from the command line.

There are a few different versions of Chocolatey. First, there's the open-sourced version, which is completely free and can be customized and extended. There are also professional and business versions that require paid licensing but also give advanced features that aren't found by default in the open-sourced version. As a user of the business license, I can say that many features are quite handy, such as automated package creation and a method to internalize community packages to local repositories.

Why Use Chocolatey?

Back in 2011, Rob Reynolds, the founder of Chocolatey, was a Puppet engineer working on the Windows team in addition to being a non-Microsoft team member for NuGet. In discussing the origin of Chocolatey, Rob says, "It really started as a way to provide a machine-level installation of tools. As a non-Microsoft employee, we had talked about whether NuGet would ever be a machine-level package manager and the joke at the time was that if it ever did support that, those would not be vanilla NuGet packages, they would be chocolatey."

> The beauty of Chocolatey is that with a unified CLI, users can manage software regardless of the installer type.

Chocolatey was created to fill the need in the Windows world of a method to better manage software in Windows. For users who've ever installed software on a Linux CLI, it's plain to see this is the ideal way to install software on an operating system. One command can accept arguments, if needed, to perform unattended installations.

Although there are other solutions for managing software in the enterprise available, Chocolatey is one of the few that's geared towards automation and DevOps. With over 6,000 community packages available through the community repository for use, it's a no-brainer that this is the best solution to use for Windows DevOps.

Chocolatey Components

For organizations using Chocolatey, there are two main components: the Chocolatey client software (CLI) and a software repository. Software repositories can be as simple as a UNC share but are ordinarily Web servers that host Chocolatey packages.

Packages

Technically speaking, a Chocolatey package is nothing more than a fancy zip file. The three main components that make up a package are a NuGet specification file, installer files, and a PowerShell script that orchestrates the installation of the installer files.

A look inside a NuSpec specification file provides details about a specific package. Metadata such as package version, project URL, dependencies, and the package name can all be

included. Here's a snippet of a NuSpec specification file for Google Chrome that shows some of these attributes.

```
<metadata>
  <id>GoogleChrome</id>
  <version>70.0.3538.7700</version>
  <title>Google Chrome</title>
  <authors>Google Inc.</authors>
  <owners>Google Inc.</owners>
```

The **chocolateyInstall.ps1** file is a PowerShell script that adds the functionality of installing a package. In this script, the locations of the installer files, package name, checksum, silent arguments necessary, and valid exit codes are all specified. All of this is kept in a configurable PowerShell hash table. The installer location is either a URL (which Chocolatey uses to download first before installing) or a directory location, usually embedded in the package itself. Checksums are included in the package so that comparisons can be made to installer files during runtime to ensure security.

Silent arguments are a key aspect of the chocolateyinstall script as they ensure that the unattended install of the software can be done. **Listing 1** shows a sample of a typical chocolateyInstall.ps1 script.

Chocolatey Client Software

The Chocolatey client software itself has a very small footprint at less than 15MB. If installing from Chocolatey.org, it can be done with just one PowerShell command (line broken to accommodate printing):

```
Set-ExecutionPolicy Bypass -Scope
Process -Force; iex ((New-Object
System.Net.WebClient).DownloadString(
'https://chocolatey.org/install.ps1'))
```

By default, the Chocolatey client is installed into the **C:\ProgramData\Chocolatey** local directory on the Windows computer. Within this folder, you'll find other notable directories used by Chocolatey.

Chocolatey uses **shimming** to add executables from packages into the PATH by adding them to the default directory of C:\ProgramData\chocolatey\bin. It works like a symlink, as a small binary is created that then calls the executable.

All execution in Chocolatey is done through choco.exe. To view the list of arguments to use with choco.exe, **--help** is used, as **Listing 2** shows.

Chocolatey GUI

Although the majority of its users probably prefer using a CLI, Chocolatey also provides "Chocolatey GUI" which is a package that can be installed with Chocolatey. Within the Chocolatey GUI, users can perform the majority of the same tasks that the CLI provides. The real use case for Chocolatey GUI is for end users who aren't familiar with a CLI.

When combined with the Chocolatey agent, the Chocolatey GUI allows non-administrator users to install software. The latest version of Chocolatey GUI even shows icons for the packages (shown in **Figure 1**) which increases the visual appeal of the tool.

Repositories

Technically speaking, packages can be installed from local file systems or folder shares, but for the most part, it will be remote Web servers that are NuGet OData repositories. There are several solutions available that provide this functionality without having to do much legwork, such as ProGet and Artifactory.

One solution that Chocolatey supports directly is **Chocolatey Server,** which is an IIS-backed repository. Even better, it's available as a Chocolatey package from the community repository.

To install the Chocolatey Server package, just specify the chocolatey.server package name:

```
PS C:\> choco install chocolatey.server -y
```

Adding a new repository to a Chocolatey client is done with the choco source command. Here, I add a new source named **choco-2** with the URL of the Web server to use.

```
PS C:\>choco source add -n=choco-2
--source='https://test/chocolatey'
```

Using the Chocolatey Command line

You'll want to list, install, upgrade, uninstall, create and internalize packages. Here's how.

Directory Name	Description of Contents
Bin	Directory of shimmed executables of packages installed
Config	XML configuration files for Chocolatey client
Extensions	Chocolatey PowerShell extensions
Lib	Successfully installed packages
Lib-bad	Failed packages
Tools	Tools used by Chocolatey such as 7zip, shim, and checksum
Text	Text Text Text Text Text

Table 1: Directories used by Chocolatey

Listing 1: The chocolateyInstall.ps1 PowerShell script

```
$packageArgs = @{
    packageName = 'googlechrome'
    fileType = 'MSI'
    url = 'https://dl.google.com/tag/s/dl/chrome/install/
googlechromestandaloneenterprise.msi'
    url64bit = 'https://dl.google.com/tag/s/dl/chrome/install/
googlechromestandaloneenterprise64.msi'
    checksum = '653fdf0e84c9c88791813bcd9184188941b9df482ef0bf99
a01c5ed836b7e0a7'
    checksum64 = 'f6f20072d55b34f435a13f625be6b836d4d57d6ebe4cf81d38
ab3c38adae61fb'
    checksumType  = 'sha256'
    checksumType64 = 'sha256'
    silentArgs  = "/quiet /norestart /l*v `"$($env:TEMP)\$($env:chocolateyPackageName).$($
env:
chocolateyPackageVersion).MsiInstall.log`""
    validExitCodes = @(0)
  }

  Install-ChocolateyPackage @packageArgs
```

Listing Packages

To view packages installed locally and also the packages available on repositories, the **choco list** command is used. To view whether the **firefox** package is installed locally, choco list can be used with the **--lo** argument (which means local only). Notice that in **Figure 2**, I also specify the **–r** argument with this command, which then limits the output to only essential information. In this case, only the name of the package and version are printed.

Listing 2: Chocolatey help command

```
PS C:\> choco --help
This is a listing of all of the different things
 you can pass to choco.

Commands

 * search - searches remote or local packages
(alias for list)
 * list - lists remote or local packages
 * info - retrieves package information.
Shorthand for choco search pkgname --exact --verbose
 * install - installs packages from various sources
 * pin - suppress upgrades for a package
 * outdated - retrieves packages that are
outdated. Similar to upgrade all --noop
 * upgrade - upgrades packages from various sources
 * uninstall - uninstalls a package
 * pack - packages up a nuspec to a compiled nupkg
 * push - pushes a compiled nupkg
 * new - generates files necessary for a choco
latey package from a template
 * source - view and configure default sources
 * sources - view and configure default sources
(alias for source)
 * config - Retrieve and configure config file settings
 * features - view and configure choco features
(alias for feature)
 * feature - view and configure choco features
 * setapikey - retrieves or saves an
apikey for a particular source (alias for apikey)
 * apikey - retrieves or saves an apikey for a particular source
 * unpackself - have chocolatey set itself up
 * version - [DEPRECATED] will be removed in
v1 - use `choco outdated` or `cup <pkg|all> -whatif` instead
 * update - [DEPRECATED] RESERVED for future
use (you are looking for upgrade, these are not the droids you are looking
for)
 * support - provides support information
 * download - downloads packages -
optionally internalizing all remote resources
 * sync - synchronizes against system
installed software - generates missing packages
 * synchronize - synchronizes against system
installed software - generates missing packages
 * optimize - optimizes installation,
reducing space usage
```

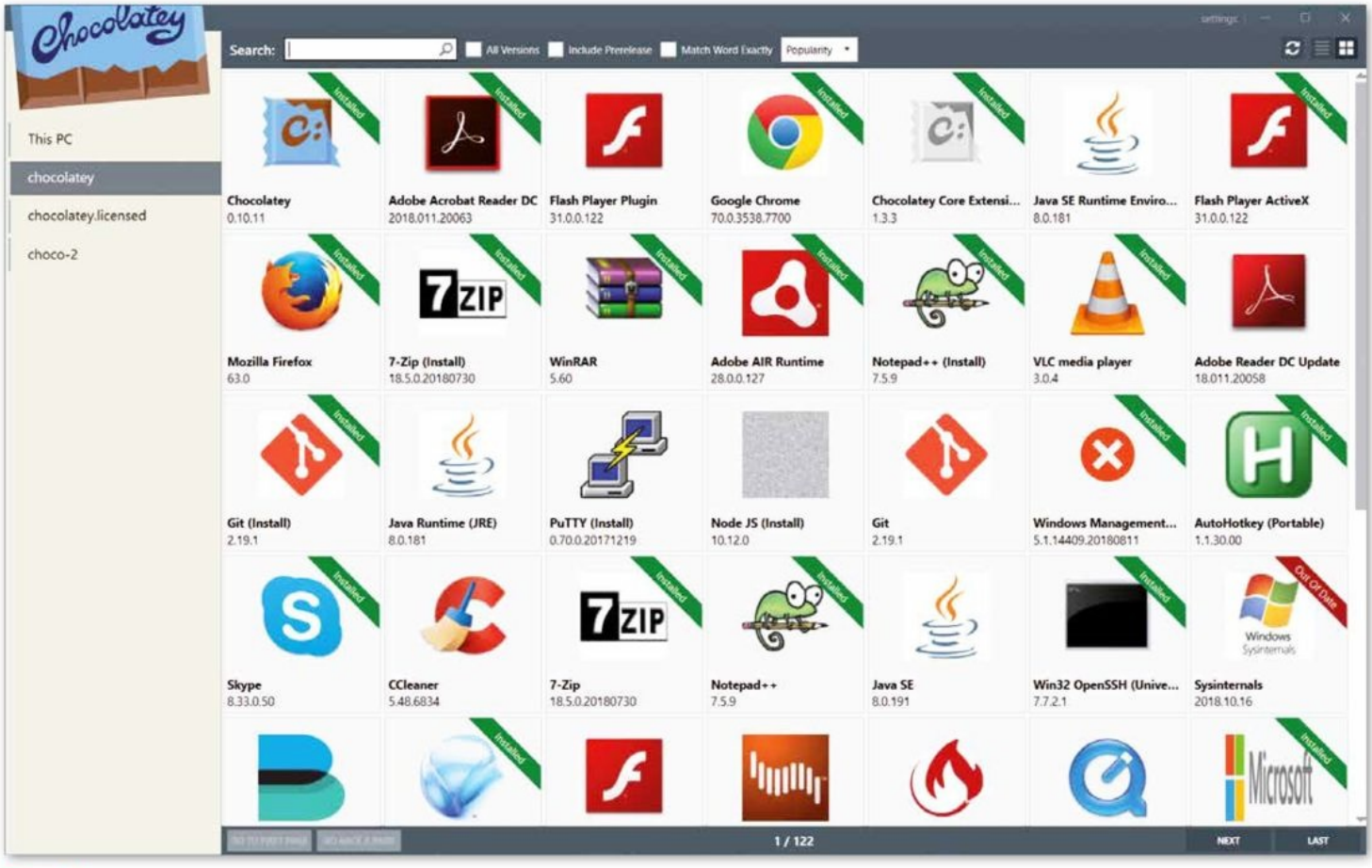


Figure 1: The Chocolatey GUI allows self-service software installation.

The second command in **Figure 2** shows that choco list searches the **chocolatey** repository for **firefox** as well. The **--exact** argument is added so that only the package name is searched. Without this argument, package tags also searched for the **firefox** string, which can print more results.

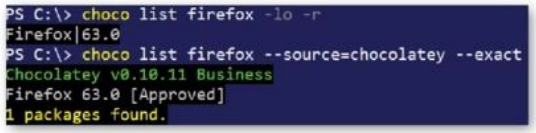


Figure 2: Using the choco list command

Installing Packages

Perhaps the most commonly used argument for choco.exe is **install**, which calls Chocolatey to install a package. In most instances, **choco install** first downloads a package from a repository, unzips the Chocolatey package locally, and then proceeds to use PowerShell to install the software.

Here's a more detailed look into what happens behind the scenes of a package installation.

- The Chocolatey client uses NuGet.Core.dll to obtain a package from the source repository.
- The package contents are installed into c:\programdata\chocolatey\lib\<package id>.
- Chocolatey takes a registry snapshot to compare later.
- If there are automation scripts (usually PowerShell), they run at this time.
- Chocolatey compares the previous snapshot for uninstaller information and saves it as a .reg file.
- Chocolatey snapshots the folder based on all files in the package directory.
- Chocolatey generates shims from all the executable files used by the package. They are placed into c:\programdata\chocolatey\bin.

Chocolatey allows users to install one or several multiple packages in the same command. Here, I install Git and Node.js:

```
PS C:\> choco install git nodejs -y
--source=chocolatey
```

Notice that the source used is "chocolatey," which is the name of the community repository.

Upgrading Software

Chocolatey uses software version comparison in order to find packages available for upgrade. There are two commands that allow a user to view what packages can be upgraded: **choco outdated** and **choco upgrade all –noop**. The **--noop** argument refers to running the choco upgrade without performing the upgrade operation on any packages.

To upgrade a local package, choco upgrade **<package name>** is used. Like choco install, adding the **-y** argument automatically accepts any prompts during the installation of the software.

Here, I upgrade my local version of Git to the latest version and point to an internal repository named **choco-2**:

```
PS C:\> choco upgrade git -y -r -
source=choco-2
Upgrading the following packages:
git
By upgrading you accept licenses for
the packages.

git|2.19.0|2.19.1|false
The upgrade of git was successful.

Chocolatey upgraded 1/1 packages.
```

Uninstall Software

When it comes to uninstalling software through Chocolatey, the **choco uninstall** command is used. Keep in mind that Chocolatey is attempting to do two distinct tasks. First, it uninstalls the software from the operating system, then it removes the package from Chocolatey so that it's no longer being tracked. What is installed on the operating system and what Chocolatey tracks as being installed can be different at times. When using Chocolatey, it's a best practice to install and uninstall all software with Chocolatey so it's accurately tracked.

Chocolatey packages are either uninstalled using a ChocolateyUninstall.ps1 PowerShell script that's contained within the package, or by using the automatic uninstaller feature, which obtains information automatically to see how software should be uninstalled.

```
PS C:\> choco uninstall vlc -y
```

Creating Packages

The creation of packages can be done with the open-sourced or licensed versions of Chocolatey, although the licensed version automates many of the tasks that are necessary to have a finished package. For instance, simply pointing Chocolatey to an installer file automatically adds silent arguments in the chocolateyInstall.ps1 file with the licensed version.

In this example, I create the package "test" using the installer file "test.msi". I add the **--automaticpackage** argument so that the .nupkg file is created in the process.

```
PS C:\> choco new test --file=.\test.msi
--automaticpackage
Chocolatey v0.10.11 Business
Creating a new package specification at
C:\package\test
Generating package from custom template at
'C:\ProgramData\chocolatey\templates\
NewFileInstaller'.
Generating template to a file
 at 'C:\package\test\test.nuspec'
Generating template to a file
 at 'C:\package\test\tools\chocolateyInstall.ps1'
Successfully generated test (automatic) package
specification file at 'C:\package\test'
```

During this process, Chocolatey first creates the necessary contents of the package, such as the NuGet specification file, tools directory, and chocolateyInstall.ps1 script, and moves the test.msi installer file into the tools folder, which is the default location for installer files.

```
PS C:\> Get-ChildItem -Path C:\package\test | Select-Object
Name

Name
----
icons
tools
test.2.1.3.111.nupkg
test.nuspec

PS C:\> Get-ChildItem -Path C:\package\test\tools\ | Select-
Object Name

Name
----
chocolateyInstall.ps1
test.msi
```

Internalizing Community Packages

When using Chocolatey for organizational purposes, the recommended practice is that Chocolatey clients don't use community packages directly for installation. This is due to distribution rights of installers, a dependency on the Internet for software, not to mention trust and control issues because packages are sometimes maintained by people not affiliated with the software. Fortunately, users can recompile (or internalize) community packages to use on internal repositories. The process is downloading installers from the Internet and then embedding them inside the package. The vast majority of community packages download installers at runtime from their distribution points on the Internet. With internalization, there's no longer a dependency on the Internet for downloading installer files.

This is another feature that the licensed version of Chocolatey offers that makes paying for Chocolatey well worth it.

```
C:\temp> choco download urbackup-client -
internalize -r
Progress: Downloading urbackup-client
 2.2.6... 100%

urbackup-client v2.2.6
Found internalizable Chocolatey functions.
 Inspecting values for remote resources.

Recompiling package.
Recompiled package files available at
'C:\temp\download\urbackup-client'
Recompiled nupkg available in 'C:\temp'
```

To install the internalized urbackup-client package, the **--source** parameter can be used to point to the local directory where the package is stored.

```
C:\> choco install urbackup-client -
source=C:\temp\ -y
```

Security

There's a considerable amount of security baked into Chocolatey. The Chocolatey client itself is verified against VirusTotal and 70 antivirus scanners in addition to being authenticode signed. There's no telemetry or data collection in Chocolatey. Chocolatey makes an effort to ensure that the package a user is installing is very secure.

> The Chocolatey client software has a very small footprint at less than 15MB.

Package Moderation

Chocolatey community packages are heavily vetted, moderated and approved by humans before allowing consumption of the package. Crapware and malware are removed from software if found during this process, in addition to anything malicious or devious found in code. Each package must meet certain standardization requirements, such as containing the necessary NuSpec specification attributes.

Part of the moderation process is testing the installation of the package on test computers. For package maintainers, Chocolatey provides a Vagrant box that's similar to the one used on their servers. Packages failing moderation are kicked back to maintainers for additional information or adjustments.

CDN

For licensed customers, Chocolatey features a private CDN hosted with Chocolatey that includes all cached download content for packages in the event that they're internalizing or installing community packages. This also decreases 404 errors when installer files are moved on Web servers.

In this output (**Figure 3**), Chocolatey uses private CDN instead of the original URL of the installer files.

Antivirus Integration

Runtime protection during package installation is another licensed feature that improves the security of using Chocolatey. Users are given the option to either use VirusTotal to scan software installers or integrate it with installed antivirus software. When using VirusTotal, users can configure the threshold to use with the **virusCheckMinimumPositives** configuration command.

```
PS C:\> choco config set virusCheckMinimumPositives 5
Chocolatey v0.10.11 Business
Updated virusCheckMinimumPositives = 5
```

Auditing

With the choco list **–lo** command, users can view installed Chocolatey packages. An additional feature for licensed users is adding the **--audit** argument to choco list. This provides information such as the user who installed the package, the domain of that user, and the time of installation.

```
PS C:\> choco list git --exact -lo --audit
Chocolatey v0.10.11 Business
git 2.19.1 User:dan
Domain:TESTDOMAIN Original User:dan
```

```
Downloading autohotkey.portable 64 bit
  from 'http://ahkscript.org/download/1.1/AutoHotkey112306_x64.zip'
Using private CDN cache content instead of original url.
Progress: 100% - Completed download of 'AutoHotkey112306_x64.zip' (561.79 KB).
Download of 'AutoHotkey112306_x64.zip' (561.79 KB) completed.
```

Figure 3: Private CDN is available for licensed users.

```
InstallDateUtc:2018-10-30 12:26:58Z
1 packages installed.
```

Integration with DevOps Tools

Another great benefit of using Chocolatey is its easy integration into DevOps tooling such as Puppet, Chef, and Ansible. Chocolatey has modules that work with the major players in configuration management, allowing it to be the de facto package manager for Windows on these platforms.

For instance, this is a sample of a Puppet manifest to install Git with Chocolatey as the package provider:

```
package { 'Git':
  ensure => '2.19.1',
  provider => 'chocolatey',
  source => 'https://myserver/api/v2',
}
```

Notice that Puppet allows a specific version to be used in a configuration. In my opinion, this is a huge reason why Chocolatey is so important to DevOps in Windows. Many times, dependency software is a specific version for applications being deployed. Chocolatey allows that to be very simple, which no other Windows software management does.

In Chef, the same configuration in Puppet is similar to this:

```
chocolatey_package 'git' do
  action :install
  version '2.19.1'
  source 'https://myserver/api/v2'
end
```

Because Chocolatey is used completely with a CLI or configuration management, it can also be used with other solutions, such as HashiCorp's Vagrant and Packer.

Conclusion

After using Chocolatey, it's clear that Windows desperately needed a solution to make software management simpler and more efficient. With the rise of DevOps on the Windows platform, Chocolatey has come along at the perfect time to fill this need. Everything in Chocolatey can be accomplished in a CLI or in configuration code, instantly making it a must-use tool for IT professionals and developers.

Dan Franciscus

CODE

Upload Small Files to a Web API Using Angular

Sometimes you need to upload files to your server from an Angular application to a Web API method. There are many ways to upload a file. In this article, I'm going to present a method that works well for small files, up to about one or two megabytes in size. You're going to build two projects; a .NET Core Web API project and an Angular project. You'll build these two projects from scratch using the Angular CLI, .NET Core, and the Visual Studio Code editor.

Paul D. Sheriff
http://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.

The result from this article is a page (as shown in **Figure 1**) that allows you to select one or more small files using an <input type="file"> element. You then build a custom FileToUpload object with attributes about the file, plus the file contents. Finally, this FileToUpload object is sent via a Web API call to a method on the server. Once the server has this file, you may choose to save it as a file on the server, into a database table, or any other location.

Build a .NET Core Web API

Let's start by building the .NET Core Web API application to which you upload files. Open an instance of Visual Studio Code. From the menu, select **View > Terminal** to display a terminal window at the bottom of the editor. You should see something that looks like this:

```
Windows PowerShell
Copyright (C) Microsoft Corporation.
 All rights reserved.

XX C:\Users\YOUR_LOGIN>
```

I'm sure you have a directory somewhere on your hard drive in which you create your projects. Navigate to that folder from within the terminal window. For example, I'm going to go to my **D** drive and then to the **\Samples** folder on that drive. Enter the following commands to create a .NET Core Web API project.

```
d:

cd Samples

mkdir FileUploadSample

cd FileUploadSample

mkdir FileUploadWebApi

cd FileUploadWebApi

dotnet new webapi
```

Open the Web API Folder

Now that you've built the Web API project, you need to add it to Visual Studio Code. Select **File > Open Folder...** and open the folder where you just created the **FileUploadWebApi** project (**Figure 2**). In my case, this is from

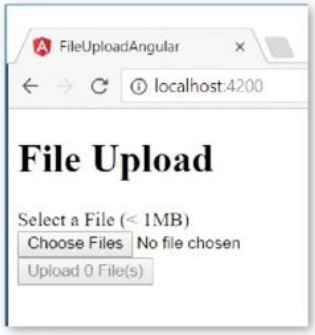


Figure 1: A simple file upload page

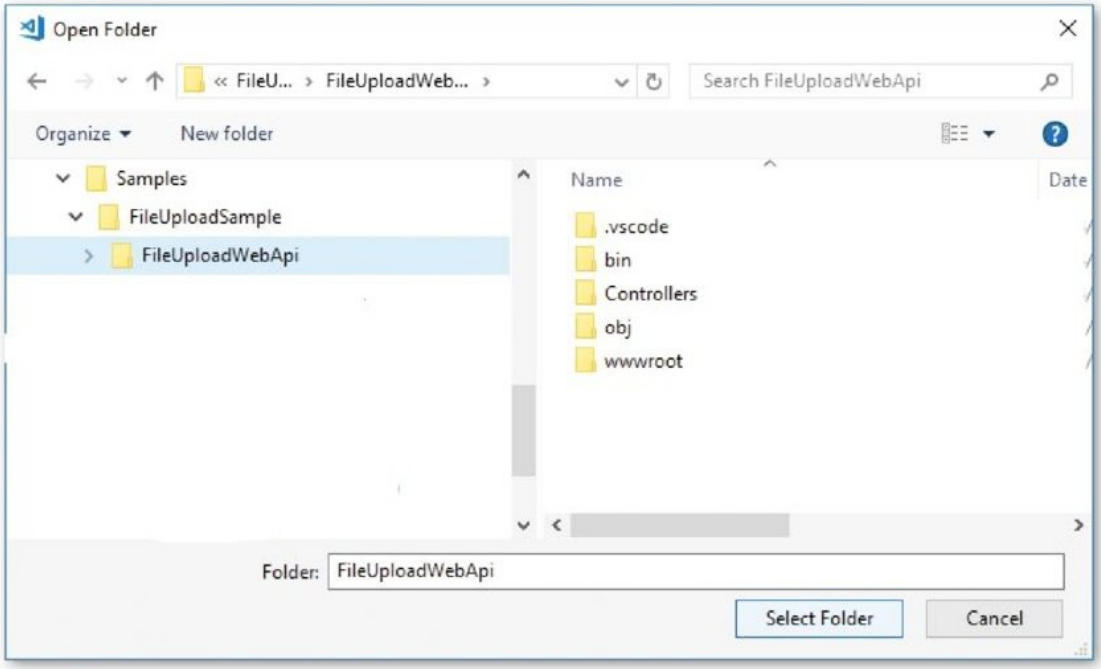


Figure 2: Add the Web API folder to VS Code.

the folder D:\Samples\FileUploadSample\FileUploadWebApi. Once you've navigated to this folder, click the Select Folder button.

Load Required Assets

After loading the folder, VS Code may download and install some packages. After a few more seconds, you should see a new prompt appear (**Figure 3**) in VS Code saying that some required assets are missing. Click on the **Yes** button to add these assets to this project.

Enable Cors

The Web API project is going to run on the address **localhost:5000** by default. However, when you create a new Angular application, it will run on **localhost:4200** by default. This means that each Web application is running on a separate domain from each other. For your Angular application to call the Web API methods, you must tell the Web API that you're allowing Cross-Origin Resource Sharing (CORS). To use CORS, add the Microsoft.AspNetCore.Cors package to your project. Go back to the terminal window and type the following command:

```
dotnet add package Microsoft.AspNetCore.Cors
```

After the package is installed, inform ASP.NET that you're using the services of Cors. Open the **Startup.cs** file and locate the ConfigureServices() method. Call the AddCors() method on the services object as shown in the following code snippet:

```
public void ConfigureServices(
  IServiceCollection services)
{
  services.AddCors();

  services.AddMvc()
    .SetCompatibilityVersion(
      CompatibilityVersion.Version_2_1);
}
```

Scroll down within the Startup.cs file and locate the Configure() method. **Listing 1** shows you the code to add to allow the Web API to accept requests only from localhost:4200. Allow all HTTP methods such as GET, PUT, etc., by calling the method named **AllowAnyMethod()**. Allow any HTTP headers within the Web API call by invoking the method **AllowAnyHeader()**. Call the app.UseCors() method prior to calling the app.UseMvc() method.

Try It Out

It's a good idea to test out your Web API project and ensure that it accepts requests as expected. Select **Debug > Start Debugging** from the Code menu to build the .NET Core Web API project and launch a browser. The browser will come up with a blank page. Type the following into the browser address bar:

```
http://localhost:5000/api/values
```

Press the Enter key to submit the request and you should see a string that looks like the following:

```
["value1","value2"]
```

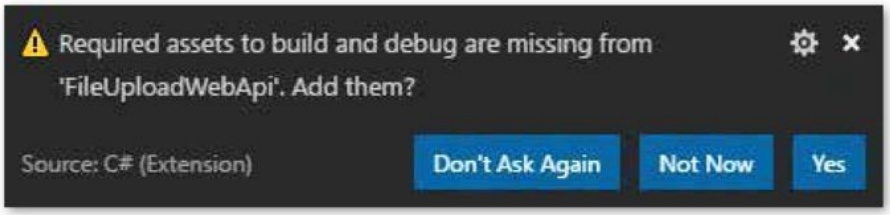


Figure 3: Answer Yes to the prompt to add missing assets.

Listing 1: Call UseCors() from the Configure() method in the Startup.cs file.

```
public void Configure(IApplicationBuilder app,
  IHostingEnvironment env)
{
  if (env.IsDevelopment())
  {
    app.UseDeveloperExceptionPage();
  }
  else
  {
    app.UseHsts();
  }

  app.UseCors(
      options => options.WithOrigins
        ("http://localhost:4200")
          .AllowAnyMethod().AllowAnyHeader()
    );

  app.UseHttpsRedirection();
  app.UseMvc();
}
```

Add a FileToUpload Class

A file you upload from your local hard drive has attributes that you should transmit to the Web service. These attributes are things like the file name, the file type, the size of the file, etc. You're going to also add some additional properties to hold the file contents as a Base64 encoded string and a byte array. Create a **\Models** folder in the root of your Web API project. Add a new file named **FileToUpload.cs** into this folder. Add the code shown in the following code snippet to this file:

```
using System;

public class FileToUpload
{
  public string FileName { get; set; }
  public string FileSize  { get; set; }
  public string FileType  { get; set; }
  public long LastModifiedTime  { get; set; }
  public DateTime LastModifiedDate { get; set; }
  public string FileAsBase64 { get; set; }
  public byte[] FileAsByteArray { get; set; }
}
```

Add a FileUpload Controller

Obviously, you need a controller class to which you send the file. Delete the **ValuesController.cs** file from the \Controllers folder. Add a file named **FileUpload Controller.cs** into the \Controllers folder. Add the code shown below to this file:

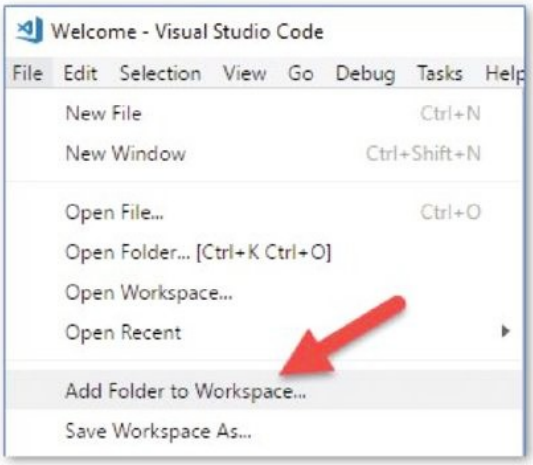


Figure 4: Add the Web API project to VS Code.

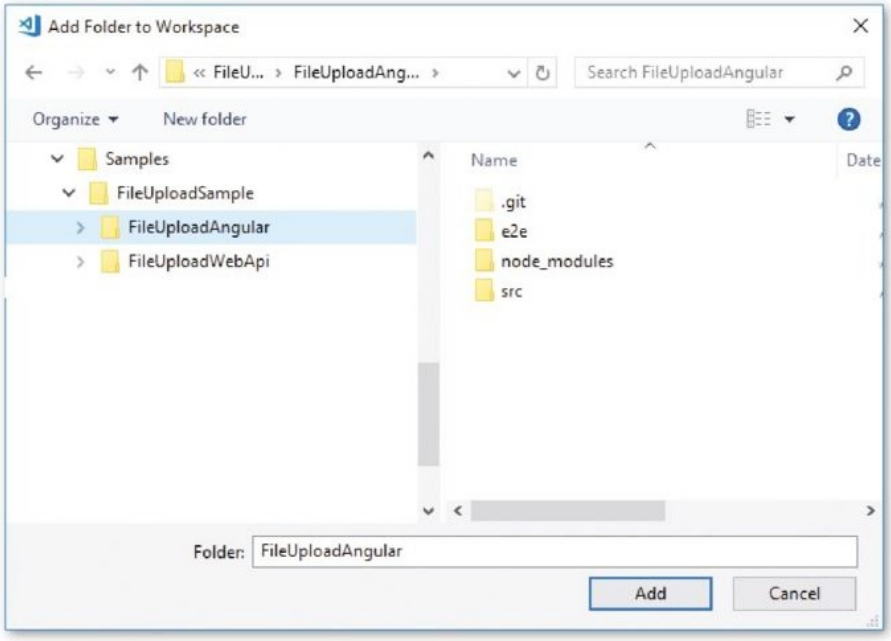


Figure 5: Add the folder where the Angular project is located to VS Code.

```
using System;
using System.IO;
using Microsoft.AspNetCore.Mvc;

[Route("api/[controller]")]
public class FileUploadController :
   Controller
{
  const string FILE_PATH =
    @"D:\Samples\";

  [HttpPost]
  public IActionResult Post(
    [FromBody]FileToUpload theFile) {
    return Ok();
  }
}
```

The above code creates the route for the FileUploadController, sets up a constant with the path to a folder on your local hard drive that your Web project has permissions to write to, and sets up the Post() method to which you send an object of the type FileToUpload.

Instead of the hard-coded constant for the file path to write the uploaded file to, you should take advantage of the ASP.NET Core configuration system to store the path. For this article, I wanted to keep the code simple, and used a constant.

Create a Unique File Name

Start creating the code within the Post() method now. Just above the **return Ok();** line, define a new variable named **filePathName**. Create the full path and file name to store the uploaded file into. Use the FILE_PATH constant, followed by the **FileName** property, without the file extension, from the file object uploaded. To provide some uniqueness to the file name, add on the current date and time. Remove any characters that aren't valid for a file by using the Replace() method. Finish the file name by adding on the file extension from the file object uploaded.

```
var filePathName = FILE_PATH +
  Path.GetFileNameWithoutExtension(
    theFile.FileName) + "-" +
    DateTime.Now.ToString().Replace("/", "")
    .Replace(":", "").Replace(" ", "") +
    Path.GetExtension(theFile.FileName);
```

If you're going to have multiple users uploading files at the same time, you might also want to add a session ID, a GUID, or the user's name to the file name so you won't get any name collisions.

Remove File Type

In the Angular code you're going to write, you're going to be reading the file from the file system using the FileReader class. This class is only available on newer browsers. The FileReader class reads the file from disk as a Base64-encoded string. At the beginning of this Base64-encoded string is the type of file read from the disk followed by a comma. The rest of the file contents are after the comma. Next is a sample of what the contents of the file uploaded look like.

```
"..."
```

Write the following code to strip off the file type. Check to ensure that the file type is there so you don't get an error by passing a negative number to the Substring() method.

```
if (theFile.FileAsBase64.Contains(","))
{
  theFile.FileAsBase64 = theFile.FileAsBase64
    .Substring(theFile.FileAsBase64
    .IndexOf(",") + 1);
}
```

Convert to Binary

Don't store the file uploaded as a Base64-encoded string. You want the file to be useable on the server just like it was on the user's hard drive. Convert the file data into a byte array using the FromBase64String() method on the .NET Convert class. Store the results of calling

this method in to the **FileAsByteArray** property on the FileToUpload object.

```
theFile.FileAsByteArray =
 Convert.FromBase64String(theFile.FileAsBase64);
```

Write to a File

You're finally ready to write the file to disk on your server. Create a new FileStream object and pass the byte array to the Write() method of this method. Pass in a zero as the second parameter and the length of the byte array as the third parameter so the complete file is written to disk.

```
using (var fs = new FileStream(
      filePathName, FileMode.CreateNew)) {
  fs.Write(theFile.FileAsByteArray, 0,
          theFile.FileAsByteArray.Length);
}
```

I've purposefully left out any error handling, but you should add some into this method. I'll leave that as an exercise for you to do. Let's move on and create an Angular project to upload files to this Web API.

Build an Angular Upload Project

Build an Angular project from within VS Code by opening the terminal window. Navigate to the folder you created at the beginning of this article. For example, I navigate to the folder **D:\Samples\FileUploadSample**. Enter the following Angular CLI command in the terminal window to create a new Angular application:

```
ng new FileUploadAngular
```

Add Web API Project to Workspace

Once the project is created, add the newly created folder named FileUploadAngular to VS Code. Select the **File > Add Folder to Workspace...** menu item, as shown in **Figure 4.**

Choose the **FileUploadAngular** folder as shown in **Figure 5** and click the Add button.

You should now see two projects within VS Code, as shown in **Figure 6**.

Save the Workspace

Click **File > Save Workspace As...** and give it the name **FileUploadSampleApp**. Click the Save button to store this new workspace file on disk. From now on, you may always open this application by double-clicking on the **FileUploadSampleApp.code-workspace** file.

Create FileToUpload Class

Just as you created a FileToUpload class in C# to hold the various attributes about a file, create the same class in Angular. Go back into the terminal window and navigate to the **FileUploadAngular** folder, as shown in **Figure 7**.

Use the Angular CLI command **ng g cl** to create a new class that represents a file to upload. You can create a folder named **file-upload** at the same time you create the FileToUpload class. The FileToUpload class will be created in this new folder.

```
ng g cl file-upload/fileToUpload
```

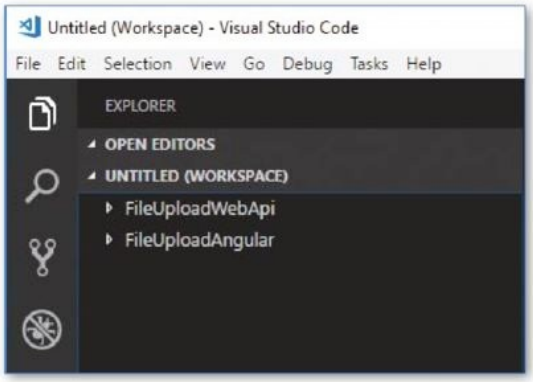


Figure 6: Add the Web API folder to VS Code

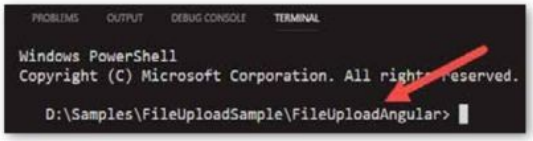


Figure 7: Add the Web API folder to VS Code.

Open the newly generated **file-to-upload.ts** file and add the code shown next. Notice that this class has the same property names as the C# class you created earlier. Using the same property names allows you to post this object to the Web API and the data contained in each matching property name is automatically mapped to the properties in the C# class.

```
export class FileToUpload {
  fileName: string = "";
  fileSize: number = 0;
  fileType: string = "";
  lastModifiedTime: number = 0;
  lastModifiedDate: Date = null;
  fileAsBase64: string = "";
}
```

Create File Upload Service

As with any Angular application, separate the logic that communicates with a Web API into an Angular service class. You can create a new service using the Angular CLI command **ng g s**. In the Integrated Terminal window, enter the following command to create a FileUploadService class:

```
ng g s file-upload/fileUpload -m app.module
```

Open the newly generated **file-upload.service.ts** file and add the import statements in the next snippet. The HttpClient and HttpHeaders classes that are imported are used to send an HTTP request. The Observable class from the RxJS library is commonly used as a return value from service calls. The FileToUpload class is needed so you can pass the data about the file to upload to the Web API.

```
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Observable } from 'rxjs';
import { FileToUpload } from './file-to-upload';
```

Add two constants just below the import statements. The first constant is the path to the FileUpload controller

Getting the Sample Code

You can download the sample code for this article by visiting www.CODEMag.com under the issue and article, or by visiting www.fairwaytech.com/downloads. Select "Fairway/PDSA Articles" from the Category drop-down. Then select "Upload Small Files to a Web API Using Angular" from the Item drop-down.

that you created earlier in this article. The second constant is a header used to post JSON data to the Web API.

```
const API_URL =
  "http://localhost:5000/api/FileUpload/";
const httpOptions = {
  headers: new HttpHeaders({
    'Content-Type': 'application/json'
  })
};
```

The HttpClient class needs to be injected into this service class to post the file information to the Web API method. Modify the constructor of the FileUploadService class to tell the Dependency Injection system of Angular to pass in an instance of the HttpClient class.

```
constructor(private http: HttpClient) { }
```

Add a method named uploadFile() to which you pass an instance of the FileToUpload class. This class is built using properties from a File object retrieved from the user via an <input type="file"> element. Call the post() method on the HttpClient class passing in the URL where the controller is located, the file object to post and the HTTP header constant.

```
uploadFile(theFile: FileToUpload) :
  Observable<any> {
  return this.http.post<FileToUpload>(
            API_URL, theFile, httpOptions);
}
```

File Upload Component

It's now time to build the HTML page that prompts the user for a file from their local drive that they wish to upload to the Web server. Create an Angular component using the Angular CLI command **ng g c**. This command builds a .css, .html, and .ts file from which you build the file upload page. In the terminal window, enter the following command:

```
ng g c fileUpload
```

Open the **file-upload.component.ts** file and add two import statements for the FileUploadService and the FileToUpload class.

```
import { FileUploadService }
  from './file-upload.service';
import { FileToUpload }
  from './file-to-upload';
```

You should limit the maximum size of file that a user may attempt to upload. The files are going to be Base64 encoded prior to sending. When you Base64-encode something, it grows larger than the original size. So it's best to limit the size of the file to upload. Add a constant just below the import statements and set a limit of one megabyte. Feel free to play with different file sizes to see what works with your situation.

```
// Maximum file size allowed to
// be uploaded = 1MB
const MAX_SIZE: number = 1048576;
```

You need to add two public properties to FileUploadComponent class. The first property, **theFile**, holds an instance of a file object returned from the file upload object. There's no equivalent of the HTML file object in Angular, so you must use the **any** data type. The second property, **messages**, is used to display one or more messages to the user.

```
theFile: any = null;
messages: string[] = [];
```

The FileUploadComponent class creates an instance of a FileToUpload class from the information contained in the **theFile** property. It then passes this object to the FileUploadService to upload the file to the server. Inject the FileUploadService into the constructor of the FileUploadComponent.

```
constructor(private uploadService:
  FileUploadService) { }
```

Get File Information from Change Event

When the user selects a file from their file system, the change event is fired on the file input element. You're going to respond to this change event and call a method named onFileChange() in your component. Write this code as shown here:

```
onFileChange(event) {
  this.theFile = null;

  if (event.target.files &&
      event.target.files.length > 0) {
    // Don't allow file sizes over 1MB
    if (event.target.files[0].size < MAX_SIZE) {
      // Set theFile property
      this.theFile = event.target.files[0];
    }
    else {  // Display error message
      this.messages.push("File: " +
          event.target.files[0].name
          + " is too large to upload.");
    }
  }
}
```

In the onFileChange() method an argument, named **event**, is passed in by the file input element. You check the target.files property of this argument to see if the user selected a file. If a file is selected, check the size property to make sure it's less than the size you placed into the MAX_SIZE constant. If the file meets the size requirement, you retrieve the first element in the files array and assign it to the **theFile** property. If the file exceeds the size requirement, push a message onto the messages array to inform the user of the file name that's in error.

Read and Upload File

Add a private method named readAndUploadFile() to the FileUploadComponent class as shown in **Listing 2**. Pass the **theFile** property to this method. Although you could simply access **theFile** property from this method, later in this article, you're going to need to pass a file object to this method to support multiple file uploads. Create a new instance of a FileToUpload class and set the properties from the file object retrieved from the file input element.

Listing 2: Read a file using the FileReader class

```
private readAndUploadFile(theFile: any) {
  let file = new FileToUpload();

  // Set File Information
  file.fileName = theFile.name;
  file.fileSize = theFile.size;
  file.fileType = theFile.type;
  file.lastModifiedTime = theFile.lastModified;
  file.lastModifiedDate = theFile.lastModifiedDate;

  // Use FileReader() object to get file to upload
  // NOTE: FileReader only works with newer browsers
  let reader = new FileReader();

  // Setup onload event for reader
  reader.onload = () => {
    // Store base64 encoded representation of file
    file.fileAsBase64 = reader.result.toString();

    // POST to server
    this.uploadService.uploadFile(file)
      .subscribe(resp =>
        { this.messages.push("Upload complete"); });
  }

  // Read the file
  reader.readAsDataURL(theFile);
}
```

Listing 3: Use an input type of file to allow the user to select a file to upload

```
<h1>
  File Upload
</h1>

<label>Select a File (&lt; 1MB)</label>
<br/>
<input type="file" (change)="onFileChange($event)" />
<br/>
<button (click)="uploadFile()" [disabled]="!theFile">
  Upload File
</button>
<br/>
<br/>

<!-- ** BEGIN: INFORMATION MESSAGE AREA ** -->
<div *ngIf="messages.length > 0">
  <span *ngFor="let msg of messages">
    {{msg}}
    <br />
  </span>
</div>
<!-- ** END: INFORMATION MESSAGE AREA ** -->
```

Use the FileReader object from HTML 5 to read the file from disk into memory. Note that this FileReader object only works with modern browsers. Create a new instance of a FileReader class, and setup an onload() event that's called after the file has been loaded by the readAsDataUrl() method.

The readAsDataUrl() reads the contents and returns the contents as a Base64 encoded string. Within the onload() event, you get the file contents in the **result** property of the reader. Place the contents into the **fileAsBase64** property of the FileToUpload object. Call the uploadFile() method on the FileUploadService class, passing in this FileToUpload object. Upon successfully uploading the file, push the message "Upload Complete" onto the messages array to have it displayed to the user. The readAndUploadFile() method is called in response to the Upload File button's click event.

```
uploadFile(): void {
  this.readAndUploadFile(this.theFile);
}
```

The File Upload HTML

Open the **file-upload.component.html** file and delete all of the HTML within that file. Add the HTML shown in **Listing 3** to create the page shown back in **Figure 1**. In the <input type="file" ...> element, you see that the change event is mapped to the onFileChange() method you wrote earlier. The Upload File button is disabled until the **theFile** property is not null. When this button is clicked on, the uploadFile() method is called to start the upload process.

After the input and button elements, you see another element that's only displayed when there are messages within the messages array. Each message is displayed within a element. These messages can display error or success messages to the user.

Modify App Module

Because you're using the HttpClient class in your FileUploadService class, you need to inform Angular of your intention to use this class. Open the **app.module.ts** file and add a reference to the HttpClientModule to the **imports** property. After typing the comma, and **HttpClientModule**, hit the Tab key, or use the light bulb in VS Code, to add the appropriate import statement to the app.module.ts file.

```
imports: [
  BrowserModule,
  HttpClientModule
],
```

Modify App Component HTML

The **app.component.html** file is the first file displayed from the index.html page. Angular generates some default HTML into this file. Delete all the code within this file and add the code shown below to display your file upload component.

```
<app-file-upload></app-file-upload>
```

Try it Out

Start the Web API project, if it's not already running, by selecting **Debug > Start Debugging** from the VS Code menu.

Listing 4: Modify the onFileChange() method to support multiple files

```
onFileChange(event) {
  this.theFiles = [];

  // Any file(s) selected from the input?
  if (event.target.files &&
      event.target.files.length > 0) {
    for (let index = 0;
             index < event.target.files.length;
             index++) {
      let file = event.target.files[index];
      // Don't allow file sizes over 1MB
      if (file.size < MAX_SIZE) {
        // Add file to list of files
        this.theFiles.push(file);
      }
      else {
        this.messages.push("File: " + file.name
          + " is too large to upload.");
      }
    }
  }
}
```


Open a terminal window in the FileUploadAngular folder and start your Angular application using the Angular CLI command **npm start**, as shown in the next snippet:

```
npm start
```

Open your browser and enter **localhost:4200** into the address bar. You should see a Web page that looks like **Figure 1**. Click on the Choose File button and select a file that's less than one megabyte in size. After you select a file, the name of that file appears to the right of this button, and the Upload File button becomes enabled. Click the Upload File button and after just a second or two, an "Upload Complete" message should be displayed. Check the folder where you specified to write the files and you should see the file name you selected followed by today's date. Open the file and ensure that all the file contents were written correctly.

Upload Multiple Files

The HTML file input type can upload multiple files by adding the **multiple** attribute to the input element. Each file to upload must be under one megabyte in size, but you may select as many files as you wish. Open the **file-upload.component.html** file and modify the HTML shown in bold below:

```
<label>Select a File(s) (&lt; 1MB)</label>
<br/>
<input type="file"
       (change)="onFileChange($event)"
       multiple="multiple" />
<br/>
<button (click)="uploadFile()"
        [disabled]="!theFiles.length">
  Upload {{theFiles.length}} File(s)
</button>
```

Open the **file-upload.component.ts** file and locate the following line of code:

```
theFile: any = null;
```

Modify this from a single object to an array of file objects.

```
theFiles: any[] = [];
```

Modify the onFileChange() method so it looks like **Listing 4**. Finally, modify the uploadFile() method to loop through the list of file objects selected. Each time through the loop, pass the current instance of the file object retrieved from the file input type to the readAndUploadFile() method. You now understand why it was important for you to pass a file object to the readAndUploadFile() method.

```
uploadFile(): void {
  for (let index = 0;
           index < this.theFiles.length;
           index++) {
    this.readAndUploadFile(this.theFiles[index]);
  }
}
```

Try it Out

Save all your changes. Go back to the browser and you may now select multiple files. Select a few files, click the Upload Files button and ensure that all files are uploaded to the appropriate folder.

Summary

In this article, you learned to upload small files from your client Web application to a server using Angular and a .NET Core Web API project. On the server-side, you need to make sure that you enable CORS to pass data from one domain to another. On the client-side, you're using the FileReader class to read data from the user's file system. The technique illustrated in this article should only be used for small files, as it would take too long to read a large file from disk, and too long to send to the server. During this time, you can't provide feedback to the user and you won't be able to display a percentage uploaded. There are several good open-source libraries for uploading large files to the server that do provide feedback as the upload process is happening.

Paul D. Sheriff
CODE

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

Ten Reasons Why Unit Testing Matters

In my experience, the difficulties associated with unit testing are in selling the concept to an organization. Quite often, organizations push back on unit testing because development with unit testing takes longer and it therefore delays delivery. It must be conceded that in some cases, that may be true, but there's more to the story.

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.

There's no way to know if such delays are an actual consequence of unit testing or from other things, such as a poor design, incomplete requirements, etc.

One thing's for sure: There's more, much more to a software project than development. A non-exhaustive list includes analysis, design, deployment, support, and maintenance. Therefore, to suggest that a software project fails because delivery may be delayed can, on a per se basis, be rejected as logical fallacy for the simple reason that the conclusion doesn't flow from the stated premise.

In anything we do in business that requires effort, in order to justify such effort, there must be a positive cost-benefit. The question to confront is whether or not unit testing as a practice has a positive value proposition for your organization. Let's examine 10 reasons why the presumption and answer to that question is an unequivocal yes.

Before delving into the 10 reasons, let's differentiate between unit, integration, performance, and user-acceptance testing. Unit testing is about isolating a single operation with a test fixture to determine whether or not that operation is compliant with a specification. That specification may be manifested in a requirements document, user story, or some other type of artifact. If you're not acquainted with the technical details around unit testing, you might enjoy reading a whitepaper I wrote a few years ago that I've since published on LinkedIn. You can find the details for that whitepaper in the sidebar.

> Quite often, organizations push back on unit testing because they think that unit testing delays delivery. In some cases, that may be true, but there's more to the story.

1. Discipline and Rigor

Software developers like to think of themselves as engineers that provide logic and a scientific approach to problem solving. Nevertheless, there's a lot of bad software in production today. As a process-oriented individual, I rely on established patterns and practices to inform and guide my development.

Ask any I/T director or CXO and they'll tell you "Of course we want and require our development efforts to be disciplined." That's your opening to ask how they actually do that. You'll often hear what amounts to word and jargon salad about patterns and practices. Again, the question is ***how do they actually implement those things?*** Unit testing is an exclusively development-oriented thing. It's the one thing you can do early in the development process that constitutes discipline and rigor in action. If a technology leader eschews the virtues of unit testing and insists on software developers employing discipline and rigor without it, I'd press them to reconcile those two viewpoints.

2. Does It Work?

Does your software work? This is a binary question that must be answered with yes or no. Whether expressly designed or implied, there's an expectation that software will work in a certain way. Many of these expectations get verified through other testing techniques like integration, function, and user-acceptance testing. Unit testing is the first bite at the apple to make sure that the software ***has the capability*** of working.

Of course, just because you have unit tests and just because the software passes, that doesn't mean ipso facto that your software works. Tests have to be valid to mean anything and to yield any value. Eventually, obviously, all software is tested. Every time you interact with software, you're testing the software. Disciplined and rigorous development demands that you know as early as possible whether your software works and unit testing provides part of the ability to answer that question.

3. Reduce Cyclomatic Complexity

Cyclomatic complexity, as the name implies, is measure of code complexity. The question is how many paths you can take through a code block. The more conditional statements you have, the more complex the code block is. The more complex the code, the more difficult it is to achieve high degrees of unit-test coverage. Unless you go through the unit testing exercise, you may not become aware of such complexity.

There are ways to measure cyclomatic complexity through other means, like code coverage. That's part of the build process, not the core development process. By the time you build, the code is already in place.

You must always be able to answer with objective evidence the question of whether your code works. Getting to the answer of yes or no is answered in part by how complex the code is. If unit-tests are difficult to write because they require a lot of set up, the code that the tests cover is too complex, period. A good reference for how complexity and testability relate to one another is the book ***Clean Code: A Handbook for Agile Software Craftsmanship*** by Robert C. Martin. In Clean Code, the SOLID principles are discussed in detail. The one of interest here is the Single Responsibility Principle, where a class and the code within it should have one and only

one task. Without unit tests, all you have is an anecdotal opinion of whether your code is sufficiently simple. Real engineering and science demands objective and independent data to substantiate an opinion.

4. Your Software Is Used Before Delivery

Think of the last time you bought a car. You probably test-drove the car, right? What about the last time your organization decided to implement a CRM suite? You likely tested it before you decided to purchase, right? And what if, in your evaluation, the car or software didn't work? You might say to yourself, "Gee, didn't anyone test this or check it out for defects first?" It's a rhetorical question because more likely than not, you'll move on to another alternative.

With custom software, there's no alternative. You're writing it and delivering it to your customer. Does your code work? Unit testing is one means to exercise your code to make sure the code operates in conformity with its specification. If you can never get to delivery because your unit tests are difficult to write, the problem, in that case, isn't with the tests. This is where unit testing opponents think they have their best argument because they'll see this scenario as proof that unit testing is bad because it delays software delivery.

I don't think I have to spell out just how logically flawed the unit testing naysayer arguments are. The earlier you can implement your software, the quicker you can achieve failure. And in achieving failure, you can remediate the issues and achieve superior software in the process. The question is ***how do you know if your software fails?*** Unit testing is one good way.

5. Documentation

People want documentation on how software works. Nobody really likes writing documentation. Unit tests are a form of documentation because they express how software is supposed to work for a given context. I'm not suggesting that unit-tests are what you send your end-user to. But for new developers on the team, there's no better way to grok the software, specifically, how the software is built as to form, patterns, and practices. Organizations often complain about the costs incident to bringing a new developer online. Unit tests are a great way to help reduce those costs.

6. Measure the Effort Needed to Modify an Existing Feature

Software requirements change over time. We've all been in the position of needing to implement changes to existing features. We've also been tasked with providing an estimate on how much effort is required to implement a change. In the absence of unit tests, we guess. Such guesses are based on intuition and experience, which isn't to say that they have no value if unit-tests aren't present. In such cases, only your most experienced developers could be relied upon.

What if, on the other hand, you had good unit test coverage? You could spike up the change and run the tests. If there were massive test failures as a result of the change, you have some decisions to make. The first is whether or not you implemented the change correctly. Second, assuming you implemented the change in the only way you could, you now have to confront the cost benefits of the requested feature modification. Part of that effort might require a refactoring effort to make your code more amenable to the requested change. Without unit-tests, the SWAG taken to determine the effort likely wouldn't account for this additional effort. The key takeaway is that unit-tests provide an opportunity to get an objective measure of at least part of the cost of a new feature.

7. Enforces Inversion of Control/ Dependency Injection Patterns

Let's assume that you have a feature that handles credit card authorizations. The feature calls out to an external service and in response, the requested charge is approved or denied. A third option is no response at all (i.e., a timeout). The application must react differently based on the response or lack of response. Let's further assume that the code is closed, in that it takes no parameters. Somewhere in the system, there's the credit card information as well as the purchase amount to make the approval request. Such code is incapable of being unit tested because the code does multiple things. In other words, it doesn't conform to the Single Responsibility Principle.

> Before you can have unit tests, your code must be unit-testable.

Going back to the SOLID Principles, the D in SOLID stands for Dependency Inversion. By injecting dependencies, you can mock contexts and behaviors so that you may simulate reality and thereby test to see how the software reacts. If you can't write a unit test because there's no way to inject dependencies, you're signing up for a very expensive proposition. Your software will end up costing more to develop and support. And when the time comes for new features, good luck with that implementation. The additional costs you'll incur plus the opportunity costs associated with not being able to implement new features is what's called technical debt.

8. Code Coverage

How do you know a line of code will ever be executed? If you have valid unit tests, you can quickly determine whether or not code is actually run. In my practice, I use JetBrains Tools like R# and dotCover to run my unit tests and provide metrics on code coverage. At a glance, in the development context, I can quickly determine whether code is ever hit. If it's not, I have questions to ask. One question is whether I have sufficient test coverage. If I've accounted for all the scenarios, the code can be eliminated. If not, I have at least one more test to write. If the additional tests require a lot work to set up, that's an indication of high cyclomatic complexity.

Advanced Unit Testing Whitepaper

You can find my advanced unit testing whitepaper on LinkedIn here: https://www.linkedin.com/pulse/advanced-unit-testing-whitepaper-john-petersen/.

By now, it should be apparent that all of these factors relate to one another. Taken together, you can safely assume that unit testing leads to better software. That, however, is a 100K-foot statement. Without sufficient details to back that claim up, it's merely conjecture and allows the unit testing naysayers to prevail.

9. Performance

Unit test fixtures can be used by performance tools to measure the success of an operation. For example, there may be a need to operate on a hashed list. In the real world, the source of such data exists in some external data store. For purposes of this discussion, let's presume that the code is the best place to handle a certain operation. Over time, it's been determined that the list can grow. You don't know the rate of growth. With unit tests, you can create scenarios that range from what you do know, to what may be probable, what may be improbable, and finally, the absurd. With unit tests, you can create a 100K item hash and see what happens. Unit tests provide the ability to gauge performance. Wouldn't it be nice to know before your software goes into production what its breaking points are? If its breaking point is 100K items and the most you will ever have is 10K items, call it a day. You now have objective evidence to make the call to end effort that solves a non-existent problem.

10. Enables Continuous Integration (CI)

Imagine that you're on a team with 10, 15, or more developers. Each developer is working on features, some of which are cross-cutting concerns. Now imagine that one developer makes a breaking change, but you don't know it. You have a CI Server that handles the merging and compilation for the team. There are no merge conflicts and the project compiles. Therefore, no problem, at least until you run the application and it blows up!

The fact is that it's exceedingly rare for compile-time errors to get through. Those kinds of errors you're forced to deal with. Runtime errors, on the other hand, are only dealt with when the software runs. If the only time your software runs is when the application is run as a whole, either in production or in user-acceptance-testing, you're setting yourself up for a world of hurt. CI servers are a wonderful thing because not only can they can manage your pull requests and merge and compile your code, they can run your tests, unit, integration, etc.

> If unit testing isn't part of your development process, you're doing it wrong, period.

Without unit tests, a CI environment is nearly worthless. To the extent that CI automates the merge and compilation process, there's some value. The question remains whether your application works as specified. Without unit tests, there's no way to answer that question. Ask yourself if you've encountered somebody who extolls the virtues CI but also questions the value of unit tests. If you have, you can safely put that person in the uninformed side of the ledger.

Conclusion

The most hated offered conclusion is "it depends" because it begs the question of "depends on what?" The question on the table is whether or not unit testing is an essential part of software development. If cost, quality, and timeliness are not of concern, the answer is no. Otherwise, the answer is yes. I don't know of any rational organization that doesn't, at the very least, have a stated preference for lower costs, higher quality, and fast delivery. In my opinion, if unit testing isn't part of your development process, you're doing it wrong, period. If things like cost and quality aren't an issue, then your development efforts are a hobby, not a business.

If you got this far in the article, it's probably because unit testing matters to you and you're trying to sell it in your organization. I'd suggest that as a first step, you do it for yourself and let the benefits speak for themselves. If you're trying to sell your organization on the propriety of unit testing, this article provides a framework for your argument.

Th naysayers don't usually prevail because they have good arguments. Often, they don't have arguments at all; they just have conclusions. But here's the hard truth: They don't need arguments because they aren't the ones advocating the adoption of something. In other words, they're not the moving party. The moving party bears the burden of proof, period. The reason why unit testing naysayers prevail is because the developer-advocate fails to carry their burden of proof and make the case.

At the end of the day, you have to make it work. It's worked for me and it's worked in any successful software project I've evaluated. Can your software project be successful without unit testing? Logic demands that the answer to that question be yes. However, the answer must be qualified as being less likely, based on the 10 factors I've set forth here.

John V. Petersen

CODE

Clean Code: A Handbook of Agile Software Craftsmanship.

If you're a professional software developer, this book is A: Already part of your software library and B: You've read it. If you don't have it, you should stop what you're doing and buy it immediately:

https://www.amazon.com/Clean-Code-Handbook-Software-Craftsmanship/dp/0132350882

(Continued from 74)

As with any endeavor that yields a return, a corresponding risk comes with investing in your people like this. No risk should ever be entered into lightly, without some amount of research, and certainly not without commitment from all parties involved. This means that you have to be absolutely clear-headed about this. Is the employee ready for this? Do they have what it takes to conquer a challenge, or this challenge in particular?

You never want to set somebody up to fail. Is this the right opportunity for them? Ideally, experts say, the assignment they take on should have about a 50-70% chance of success. In other words, it should be a task that they have a real shot at completing yet something that won't come easily. They should have to work for it—and ideally, work hard for it. Psychologically, we value the things that we have to work for far more than we do the things that just fall into our laps.

> The assignment they choose should have a 50-70% chance of success.

Honesty also has to cut the other way, too. Can you really make it happen for your employee? Will you need some higher-level of support from your boss or other parts of the organization to make this happen? It's never a good idea to make promises that you can't keep. You may also be going out on a limb and need to expect that if this doesn't go well, it reflects on you just as much as it does your employee.

And that's the last bit that needs to be said. If this isn't a guaranteed win, everybody has to be ready to accept the fact that the stretch could fail, and to deal with the consequences of that fail. (If you do this long enough, in fact, and you're choosing assignments that are in that "sweet spot" of 50-70% success, then statistically it's only a matter of time before one does fail.) Accepting failure is one of the hardest things to do in the business world, and it's up to you to be there to provide the safe space for your employee and the team if things don't go as well as anybody would like.

Summary

Frankly, developing the talent on your team is probably the most rewarding aspect of management; it ranks right up there next to watching your kids grow, and most likely for the same reasons. As human beings, we feel rewarded when we bring value into the world and one of the greatest values is that of helping people. It's personally rewarding to see somebody under your management thrive and grow and the rewards for the team can be just as large, as they catch on to the idea that yours is a team that isn't just concerned with the business' "today," but also with their "tomorrow."

Most of all, consider this. The Internet is rife with memes, and one is a discussion between the CFO and the CEO of a company. The CFO says, "But what if we pay them all this money to learn and they leave?" The CEO says in return, "What if we don't, and they don't?"

(This article was inspired by, and drew material from, Chapter 11 of the "Harvard Business Review Manager's Handbook", from Harvard Business Review Press, 2017.)

Ted Neward

CODE

Jan/Feb 2019
Volume 20 Issue 1

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
Dan Franciscus
Sahil Malik
Jeffrey Palermo
Paul D. Sheriff
Wei-Meng Lee
Ted Neward
John V. Petersen

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 Developing Talent

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—that is, ourselves. You've been made a manager. You now have a team. Like you, don't want to stagnate. They're looking to you for guidance and support on the development and growth of their talents and skills.

What now?

Priorities

Let's set a few things in stone right now: If the manager, doesn't make their growth and development a priority for the team, the team's not going to get any better than they are right now. In fact, it's reasonable to assume that as each day passes, their talents and skills are going to depreciate—if it's the one constant thing in our industry, is that our industry is always changing. If they aren't learning new technology, new approaches, new design patterns, or any other "new" that comes to mind, then they—and your team as a whole—are falling behind competitors.

Perhaps you're fortunate enough to be in a company or market where you have no competitors. Congratulations. It still means the team is missing out on ideas and concepts that could reduce the cost of development or ownership. This doesn't take into account that the actions involved in learning new things often benefit the team directly in other ways—going out to a conference, for example, brings them in contact with other folks who are doing similar things, expanding their "tribe" and giving them more connections on whom they can pull for advice, assistance, or suggestions.

Yes, it would be lovely to assume that your team will always be learning new things on their own. After all, if your team is highly motivated and interested in improving their craft, they'll be out reading articles and books and exploring, right? It's certainly always possible, but if you rely on them to do it on their own, you'll have no input on what they learn, when they learn it, or how they think about applying it. If you want them to be able to take what they're learning and apply it to the job, you have to be able to guide what they pick up next, and realistically speaking, the only way to guide that is to support it with company time, money, and participation.

You, as their manager, have to demonstrate that you're as committed to their growth as they are. You do that by providing that time, the necessary money, and your time.

> You have to demonstrate that you're as committed to growth as they are.

Path

In many ways, putting your team onto a growth path is a simple one. It starts, as many management-related things do, with talking to the team and asking questions. These questions can range over a variety of topics, but usually should focus on a couple of areas: their interests and skills, their sense of organizational fit (the kind of environment they need in order to thrive), their work values (what they think a good career looks like), and their vision of their future. This is the only time, ever, it's permissible and desirable to ask that question about where they see themselves in five years.

Fundamentally, what you're looking for are the threads of possibility the team members want to go down. Do they want to go into management? Do they want to learn a new platform? Do they want to become an architect? Do they have aspirations to jump into a startup at some point?

The startup question is important. Unless the individual in question is going to retire in a year or two, it's entirely likely that they'll work for another company at some point, and you should be just as committed to their growth whether they leave or whether they stay. Sometimes that means helping them level up in skills that help both this company and the next one.

Once you know what their chosen direction is, it's time to scour your horizon to look for opportunities to train them. In our industry, there's no shortage of them—conferences, books, developer portals like CodeProject or DZone, magazine subscriptions like this magazine or MSDN, online training systems like LinkedIn Learning, Udemy, Pluralsight, or Safari—the list is long and in many ways, overwhelming. That's fortunate, actually, because it means it won't take a ton of time to find the training they need, assuming that it's technical in nature.

But don't stop there! There are very likely people at your company who already do the things that your employee is interested in, so try to set up some lunch or coffee dates between them, to give them a chance to learn from those who've been there. If you can, and the others are amenable, try to set up a mentorship relationship between them. This is one of those times when you trade on social capital between colleagues—offer to buy lunch every so often, trade a favor that will help the mentor, and so on. Keep in mind that many people will be flattered that you've asked them in the first place, so it might be a pretty even trade just on the surface of it.

Even then, don't stop. Once your employee has started picking up the skills they're looking for, look for ways to help them use them on the job—can they start using what they're learning? Can you make them the person on the team responsible for that particular tool or approach? Can you "loan" them out to a team for a while that's doing that work? Give them opportunities to stretch themselves, with their agreement, and they will find themselves absolutely enamored with the idea.

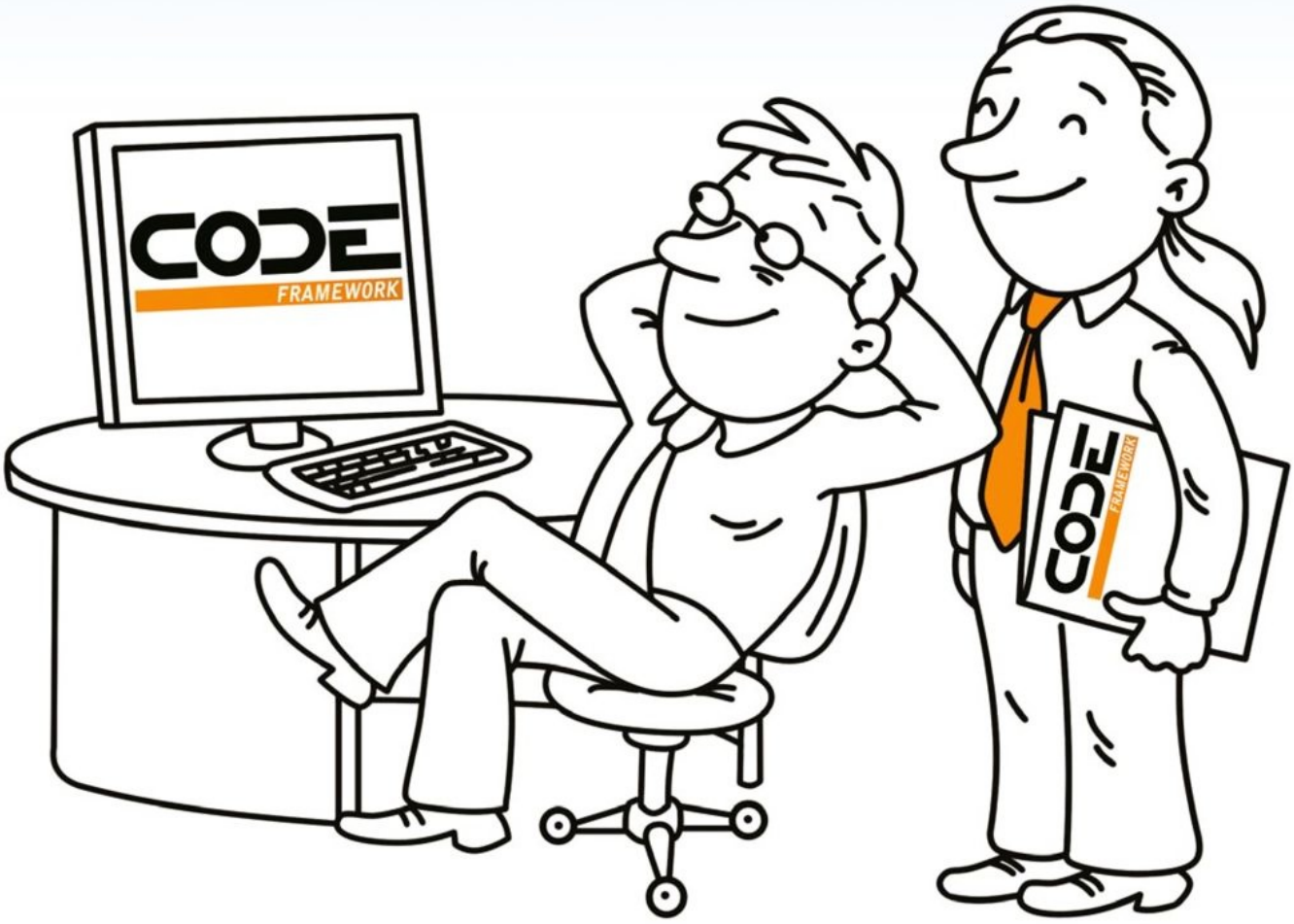
> Let the team stretch and grow and everyone benefits.

But...

The objections will come, if not in your mind, then from others around you. "What if you put all this energy into them and they end up leaving the team?" Or perhaps, "Can you really afford to take that kind of hit to your team's productivity?" Sometimes even, "What happens if they don't actually do well at it?"

(Continued on page 73)

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