Issue 35 | Mar-Apr 2018

DNCMagazine

www.dotnetcurry.com

Designing
Data Objects in C#

Kubernetes!

Software Craftsman
Mindset

Microsoft's DevOps Story

SQL Graph

Angular +
ASP.NET Core

ASP.NET Core
E2E Testing

Creating Apps on different **.NET Framework Versions**

A lap around
Visual Studio App Center

the team.

Editor In Chief : Suprotim Agarwal

Art Director : Minal Agarwal

Technical Reviewers : Yacoub Massad, Suprotim Agarwal, Subodh Sohoni, Roopesh Nair, Ravi Shanker, Keerti Kotaru, Damir Arh, Aseem Bansal, Abhijit Zanak.

Next Edition : May 2018

contributors.

Yacoub Massad · Shreya Verma · Sandeep Chadda · Rahul Sahasrabuddhe · Ravi Kiran · Gerald Versluis · Daniel Jimenez Garcia · Andrei Dragotoniu · Damir Arh

Suprotim Agarwal

letter from the editor.

There are a few key factors I can clearly say have shaped me as an educator and as an IT professional. One of them is attending Conferences and Summits.

I recently got the invitation to attend the MVP Global Summit at the Microsoft headquarters in Redmond (USA). The MVP Summit is an exclusive event reserved for active MVPs and Regional Directors (RDs), and is all about celebrating community leaders, networking, exchanging Microsoft product feedback and ideas worth exploring!

With all of us digitally connected, and feedback just a tweet away, is there any need to take time off work, travel for 30+ hours (I travelled from India) and attend these professional events anymore?

Yes! Perhaps now more than ever before.

There is a great deal of difference in "knowing" someone online, and a true face-to-face communication - be it at breakfast, during a session, during a product group party or by fixing private meetings.

Not only do these professional events reignite enthusiasm, there is always something powerful about meeting the people who you need to meet. (pics on Pg. 66). When I attend a professional conference, I always have a renewed sense of excitement about the work I do, I get new ideas generated from unexpected conversations, rekindle old friendships and cherish the wonderful new friends I have made.

The experience from these conferences also influences the quality of your work.

...and then above all, there's the fun factor.

There's a lot to get out of these professional events, and often these events bring out the biggest opportunities to learn and grow. The key mantra is to keep your eyes open and your ears perked throughout the events.

If you have never attended conferences before, go take the leap!

THANK YOU

FOR THE 35th EDITION


@shreyavermakale


@dani_djg


@rahul1000buddhe


@yacoubmassad


@sandeepchads


@jfversluis


@sravi_kiran


@subodhsohoni


@andrei74_uk


@damirarh


@suprotimagarwal


@saffronstroke


@abhijit_zanak


@keertikotaru

WRITE FOR US

mailto: suprotimagarwal@dotnetcurry.com

contents.

Sandeep Chadda

Microsoft's Devops story

This is a story about the Cloud & Enterprise engineering group at Microsoft that encompasses products & services such as PowerBI, Azure, Visual Studio, Visual Studio Team Services, Windows Server, and System Center. It is a story about transformation of Microsoft's culture, organization structure, technology, and processes to allow enterprises to deliver faster, continuously learn, and experiment.

This is also a story of 84,000+ Microsoft engineers (as of Feb 2018) who have moved to Visual Studio Team Services (VSTS) to create unparalleled value for their customers using DevOps.

DevOps – Frequently Asked Questions

Few frequently asked questions around Devops that this article attempts to answer:

Image Courtesy: Flikr

What is DevOps?

- Where do we start with Devops?
- Is Agile based product management enough for DevOps
- If we continuously deploy, would that constitute DevOps
- Can we achieve DevOps in 1-2 years?

Organization changes

- Can Devops be a top down approach or a bottoms-up?
- Do we need to modify our team structure to achieve Devops?
- Can the teams continue to operate as-is without any major changes or do we need to relook the org structure?

Engineering investments

- How do we ensure that our teams can write code and deploy it faster? How would testing look in this new world of faster delivery?
- How do we work on features that span multiple sprints?
- How do you expose features to few early adopters to get early feedback?
- If there are many teams working on a product then how do they make sure that we are always shippable?
- How does integration testing work in a DevOps enabled engineering system?
- How long does it take you to deploy a change of one line of code or configuration?

How do we measure success?

- Our intent was to experiment, learn, and fail fast but how do we measure failure or success.
- If we want to ensure we deploy often, then how do we measure the service metrics.

PS: Some of the anecdotes used in this article are from the author's own experiences at Microsoft.

Microsoft's Mindset

I start with Microsoft's mindset.

Though this is not the first thing that we tackled at Microsoft, it is important to understand the mindset of

the Microsoft employees to better appreciate the transformation in the organization.

As Satya writes in his book "Hit refresh", the new Microsoft is built on the scaffolding of growth mindset & collaboration.

THERE IS A TRANSFORMATION FROM "I KNOW EVERYTHING" TO "I WANT TO LEARN EVERYTHING".

All employees in Microsoft are encouraged every day to experiment, learn, and continuously improve. It is a transformation from celebrating success to celebrating failure and learning from those mistakes and course correct to add value to customers. This has fostered a start-up culture within Microsoft which lives and breathes on maximizing user value.

Now you really can't experiment much and react fast if the people are not thinking that ways, if the processes are not tuned for this, and the toolset is not an enabler.

This process, toolset, and mindset is what I refer as DevOps.

Agile Transformation | Kanban

Across Microsoft and particularly within the Cloud + Enterprise engineering group, we have been focusing on shipping software more frequently. Therefore, it was critical for us to gradually transform from a traditional waterfall model with lots of phases and a multi-year release cycle, to a rapid 3-week sprint cycle.

So, let's first understand why Agile?
We were earlier planning our product release many years in advance and contemplating what customers asks would be, when the product releases. Time and again we found ourselves trying to answer genuine feature requests from our customers and giving them an ETA (Estimated time of arrival) several years ahead of time. This was not bad, since the whole software industry was working this way. However, the question is **whether there is a scope of improvement or is there wastage in the system?**

We frequently found ourselves in uncomfortable territories where a genuine customer request would only be fulfilled after few years since our release cycle were years apart. It was also an interesting challenge to ensure that we were able to adhere to release deadlines as well.

We soon realized that the biggest issue with the waterfall model was that we were delaying value that we delivered to our customers. Our first prerogative was to reduce the cycle time from ideation to delivering customer value, and thus we adopted Agile.

We trained everyone on Agile and these trainings continue to happen for those joining the organization afresh. The key focus has been on the flow of value to our customers.

Agile did not happen to us overnight. When we first moved to Agile, we experimented with two weeks and four-week sprints. We also experimented with 5 sprints of three weeks each followed by a stabilization sprint. This was an interesting experiment since it had some interesting outcomes.

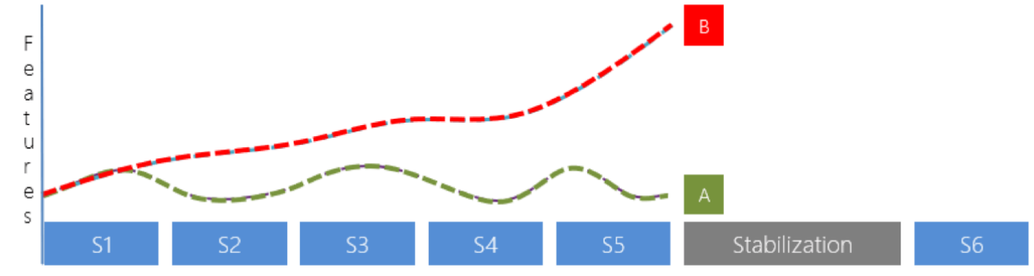


Figure 1: Experiment with stabilization phase was not successful

Because of the stabilization phase, one of the teams, "TEAM A" continued to develop features and kept the debt in check. Another team, "TEAM B" went overboard with feature work and looked to reduce the debt in the stabilization phase. You would assume that "Team B" worked to get their technical debt down in the stabilization phase while "Team A" continued the path of feature development, but it did not turn out that ways.

Even "Team A" had to reduce "Team B's" debt.

Now imagine what was Team A's motivation to do the right thing. We were almost penalizing "Team A".
In this experiment we learnt quickly and got rid of the explicit "stabilization phase". Now, the teams are expected to keep the debt in check on an ongoing basis with each sprint. We have a staunch foundation of measuring team health to ensure teams do not pile on debt.

I will talk about this later in the "Build, Measure, Deploy" section of this article.
Team structure and planning

We follow Scrum, but we customized it to our needs to fit the scales that we needed. For example, a team is often referred to as feature crew that is responsible for owning a product feature. Just to give an example, we have a feature crew for "Kanban boards" and another that manages "pull requests" and all git related workflows.

A feature crew (FCs) typically comprises of a product owner and the engineering team. All FCs in Microsoft are driven by the dual principle of team autonomy and organizational alignment.

Team Autonomy: It is the freedom that each team in Microsoft enjoys defining its own goals to maximize the customer impact

Organization Alignment: It is the alignment of team's goals with the organization's objectives. Think of it as the glue between the operations and strategy that acts as the guiding light of each Microsoft employee.

To give an example.

A feature crew owns its own backlog. It decides what will be shipped in the next few sprints. It decides the backlog based on the conversations that each team is having with its customers and partners. This is team autonomy.

While at the same time, all feature crews within a product are aligned to speak the same taxonomy in the organization. e.g. when I am writing this article, all 45 feature crews for Visual Studio Team Services are on Sprint 131. We all know what we will ship in sprint 132 and 133 and this is the information we share with our customers in our release notes and plans. This makes us efficient in terms of processes when managing conversations with our customers or other teams.

Reference: Sprint 130 release notes
https://docs.microsoft.com/en-us/vsts/release-notes/2018/feb-14-vsts

All feature crews are multidisciplinary and typically comprise of 8-12 engineers, 1 Engineering manager, and 1 program manager. Mostly a feature crew stays together for 1 to 1.5 years and many stay together much longer. There is a support structure with each feature crew that may comprise of user experience designers, researchers etc. that works alongside the feature crew.

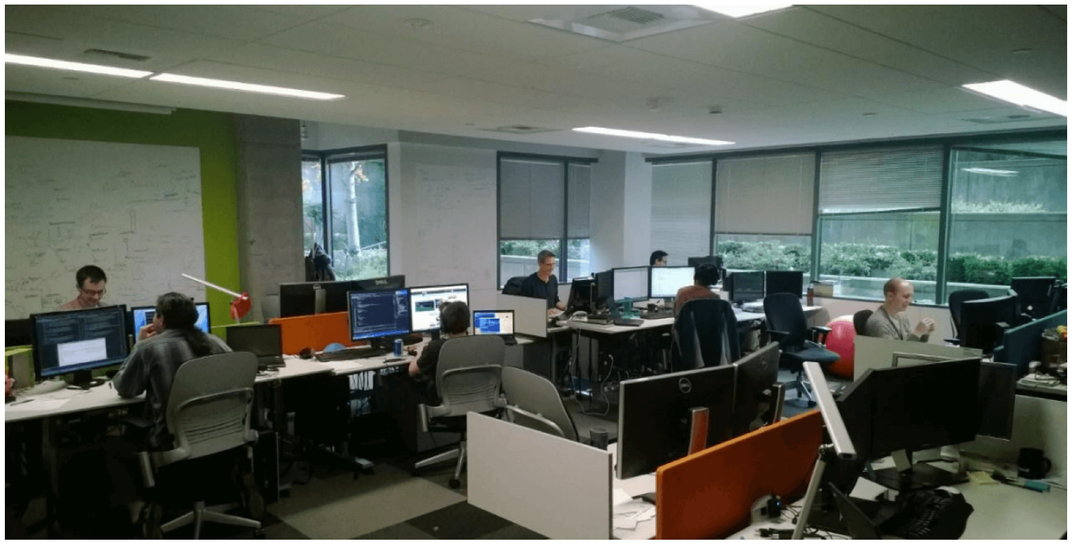


Figure 2: A typical feature crew seated together in a team room - Microsoft, Redmond office

While feature crews are formed with a specific objective, there are moments when another feature crew may be asked to balance work for a short duration.

A feature crew always sits together in a team room. Think of team room as a dedicated room where all members of the feature crew sit together and can talk freely. Each team room has a focus room for quick huddles of 3-4 members. We encourage conversations in the team room but focus rooms help having a focused conversation if the whole team does not need to be involved or disturbed. Each team room has a television on display that projects the team's Kanban board or important metrics that team wants to track. The Kanban board comes handy during stand-ups when we simply stand-up in our rooms to discuss the day and work.

The product manager is the thread between customers and the feature crew. She is responsible for ensuring the product backlog is up to date. We have a sprintly ritual of sending the sprint email to all feature crews and leadership. Each sprint mail typically contains 3 sections:

1. What we did in this sprint
2. What we plan to do in the next sprint
3. Sprint videos celebrating the value we delivered in the sprint

Here is mine from few sprints back:

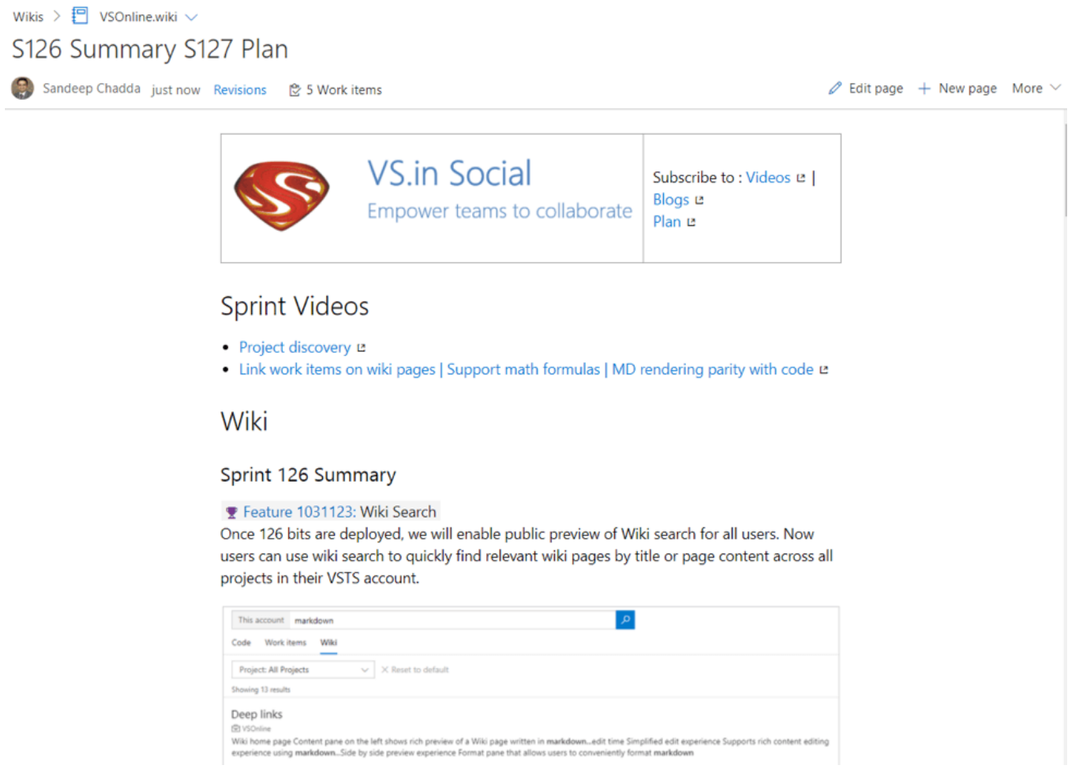


Figure 3: Sample sprint email including customer value delivered to our customers

The planning exercise is very lightweight. The feature crew is driven by an 18-month vision, which lays the foundation or objective of the feature crew's stint. This 18-month epic may be split into chunks of features or spikes that are the fundamental drivers for a team's backlog. The 18-month epic is typically split into 6-month planning cycles.

After every three sprints, the feature crew leads get together for a "feature chat," to discuss what is coming up next. These feature chats are also shared with leadership and serves to help keep feature crew priorities in alignment with overall product objectives. You would often hear feature crews saying that certain feature asks are not in the next 3 sprint plan or 6-month plan since we plan this way.

The feature crew then delivers features every sprint. With each sprint we communicate to our customers regarding the work completed and this also serves as our feedback channel.

User voice - When the feature crew completes a feature from the user voice you will typically see comments updated by the product manager

You can view the Wiki full text search user voice here:

https://visualstudio.uservoice.com/forums/330519-visual-studio-team-services/suggestions/19952845-wiki-fulltextsearch

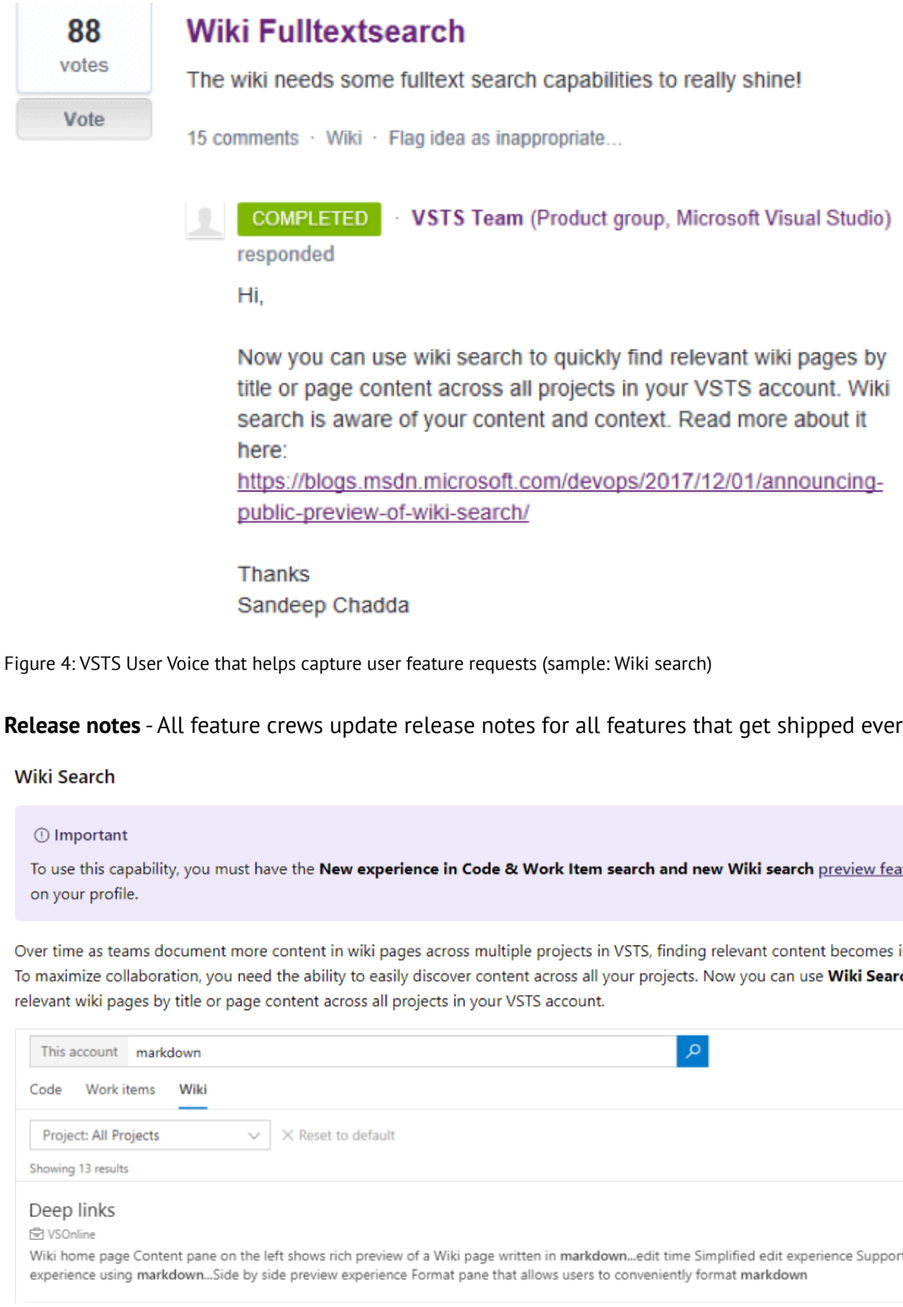


Figure 4: VSTS User Voice that helps capture user feature requests (sample: Wiki search)

Release notes - All feature crews update release notes for all features that get shipped every sprint

Figure 5: Release notes published publicly (sample: wiki search)

Blogs - All major announcements are blogged as well. Here's an example:

https://blogs.msdn.microsoft.com/devops/2017/12/01/announcing-public-preview-of-wiki-search/

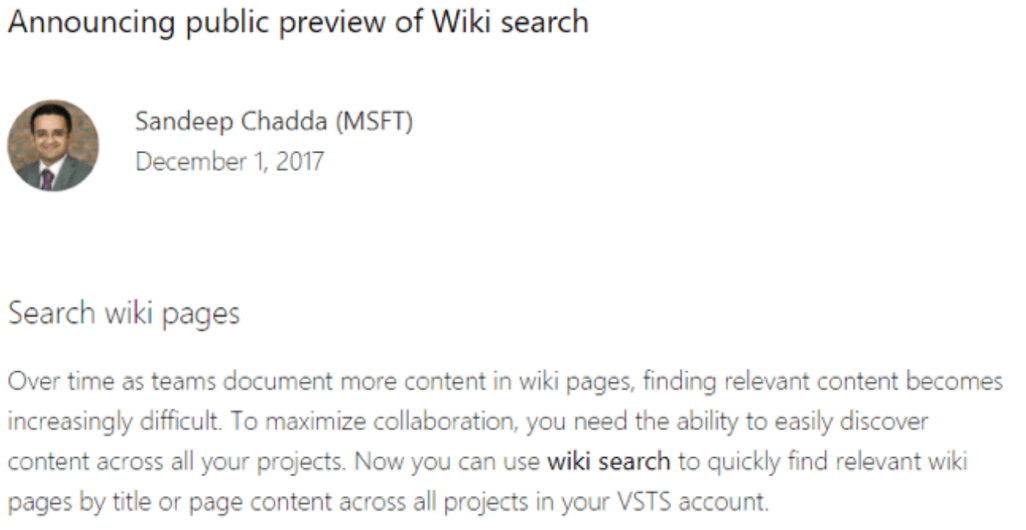


Figure 6: Blog post snippet (sample: wiki search)

This may look like a lot of work, but it is not since all this information flows right into my work items on the Kanban board in VSTS. I can prioritize work based on the user voice of the features and my release notes from the work item flow into the public forum, so I can get feedback from my customers and reach out to them by just updating my work item.

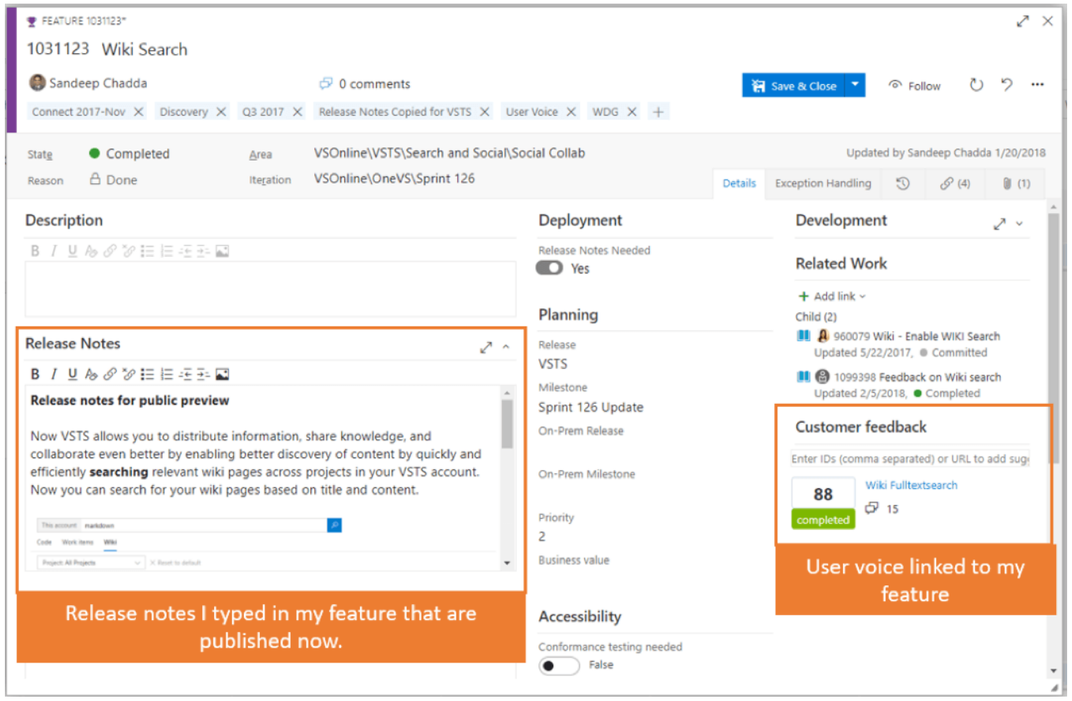


Figure 7: Wiki search work item shows integration with user voice and release notes

Engineering discipline and branching strategy

Earlier, we had an elaborate branch structure and developers could only promote code that met a stringent definition and it included gated check-ins that included the following steps:

1. "Get latest" from the trunk
2. Build the system with the new changesets
3. Run the build policies
4. Merge the changes

The complex branch structure resulted in several issues:
1. we had long periods of code sitting in branches unmerged.
2. created significant merge debt
3. massive amount of merge conflicts by the time code was ready to be merged
4. long reconciliation process

There was a lot of wastage in the system. Eventually we flattened the branch structure and had the guidance for the temporary branches. We also minimized the time between a check-in and the changeset becoming available to every other developer. I talk more about this in the "continuous everything" section.

The next step was to move to Git. This move to a distributed source control from a centralized version control was a big step in our journey. Typically, the code flow would happen as follows:

1. A work item is created for a bug or a user story
2. Developer creates a topic branch from within the work item
3. Commits are made to the branch and then the branch is cleaned up when the changes are merged into the master.

This way all the code lives in master branch when committed, and the pull-request workflow combines both code review and the policy gates.

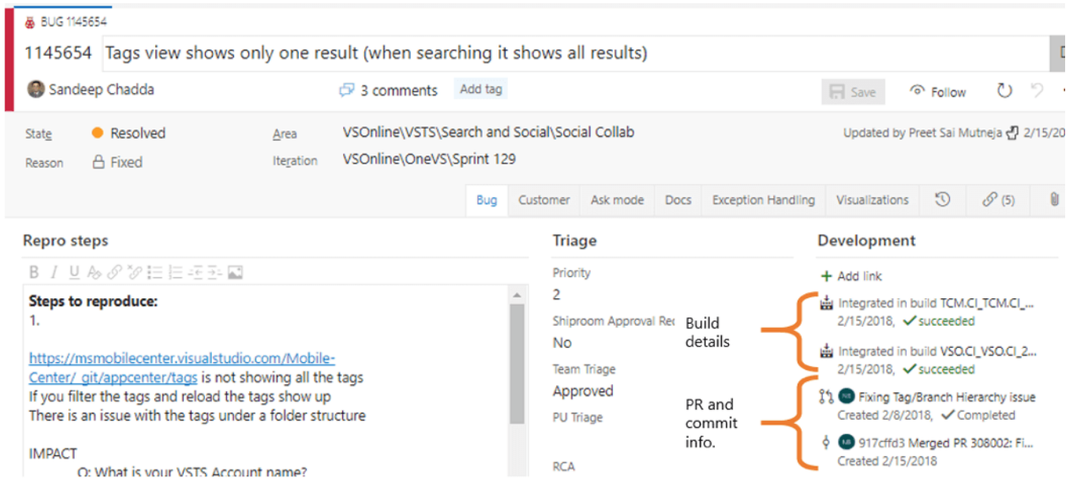


Figure 8: Work item showing commit, PR, build, and release status from the work item itself.

This ensures 100% traceability of work while ensuring that merging remains easy, continuous, and in small batches. This not only ensures that the master branch remains pristine but also that the code is fresh in everyone's mind. This helps in resolving bugs quickly.

Control the exposure of the feature

One problem that we faced going down this route of short lived branches and sprintly cadence of deployment was - "How do I manage master when I am working on a big feature that spans multiple sprints"?

We had two options:

1. Continue to work on the feature on another feature branch and merge it to master when done. But that would take us back in time when we had those merge conflicts, an unstable master, and long reconciliation time.

2. Therefore, we started to use the *feature flags*. A feature flag is a mechanism to control production exposure of any feature to any user or group of users. As a team working on features that would span multiple sprints, we register a feature flag with the feature flag service and set the default to OFF. This basically means, the visibility of this feature is turned OFF for everyone. At the same time, the developer keeps merging changes to the master branch even though the feature is incomplete behind the feature flag. This allows us to save the cost of integration tests or merge conflicts when eventually the feature goes live. When we are ready to let the feature go live, we turn the flag ON in production or the environments of our choice.

You would often hear us say, the feature is currently being dogfooded and it will be enabled for other customers soon. Basically, what this means is that we have turned ON the feature flag for the feature on our dogfood environment and we are getting feedback. The feature is nearly production ready. We will react to the feedback and turn ON the feature flag for the rest of the world soon. Once the FF is turned ON for everyone, we clean it up.

The power of feature flag is beyond comprehension. It not only ensures that our master branch stays pristine, but also empowers us to modify the feature without any visible production impact. It can also be an effective roll back strategy without deploying a single iota of code in production.

By allowing progressive exposure control, feature flags also provide one form of testing in production. We will typically expose new capabilities initially to ourselves, then to our early adopters, and then to increasingly larger circles of customers. Monitoring the performance and usage allows us to ensure that there is no issue at scale in the new service components.

You can also try one of the extension on our marketplace to manage feature flags:
https://marketplace.visualstudio.com/items?itemName=launchdarkly.launchdarkly-extension

Read more about feature flags
https://www.visualstudio.com/learn/progressive-experimentation-feature-flags/

Organization structure

Our organizational chart was by discipline i.e. PMs report to PMs. Developers report to Developers. Testers report to Testers.

As we got into smaller cycles and quicker iterations, there were barriers to having a separate discipline for developers and testers. We started seeing wastage in the system during the handshake between testers and developers.

Some of issues that we faced were:

- If testers are out, developers didn't know how to test
- Some testers knew how to fix the issue, but they could not since they were "testers" and not "developers"
- If the developer is out, the "tester" had not much to do

All this created wastage in the system and slowed the delivery of customer value.

As a company, we made a strategic change to transform into combined engineering. We took the developer discipline and tester discipline and put them together as **one engineering**.

Now engineering is responsible for

- how we are building features
- and we are building with quality

Now, we hire engineers who write code, test code and deploy code. This has helped us to reduce or should I say remove the unnecessary handshake that was causing severe delays in the system, especially when the need of the hour was to go fast and deliver faster.

PS: This strategy worked great for Microsoft however when you plan to implement similar strategies at your end, you should see whether you are able to identify wastage in your system and see what can fit best to your organization needs.

The old model left us with a lot of functional tests that took extremely long to run and were flaky. When the developers were made accountable for testing code, we simultaneously made investments in making the test infrastructure better. We also made investments in making tests faster and reliable. We soon created zero tolerance for flaky tests and the testing became closer to the code. This is often referred to SHIFT LEFT in Microsoft parlance.

This again ties back to the initial requirement to keep the master branch clean & shippable and we were able to do so effectively by shifting our tests left. Now we run around nearly 71k+ L0 tests with every PR that merges code onto the master branch. It takes approximately 6 min 30 seconds to run all these tests.

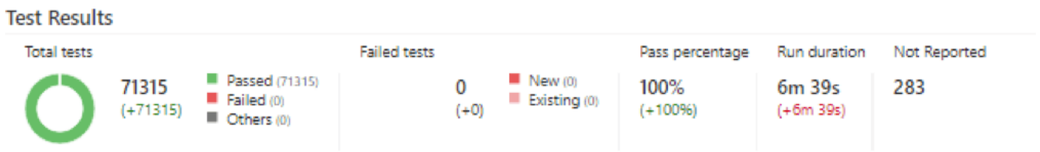


Figure 9: There are more than 71k tests that run with each PR. Screenshot from VSTS.

The number was a little over 60,000 few months back so the rate of increase is significant. This level of automation helps us keep the master branch always shippable. We constantly measure the health of the master branch as you can see in the image below.

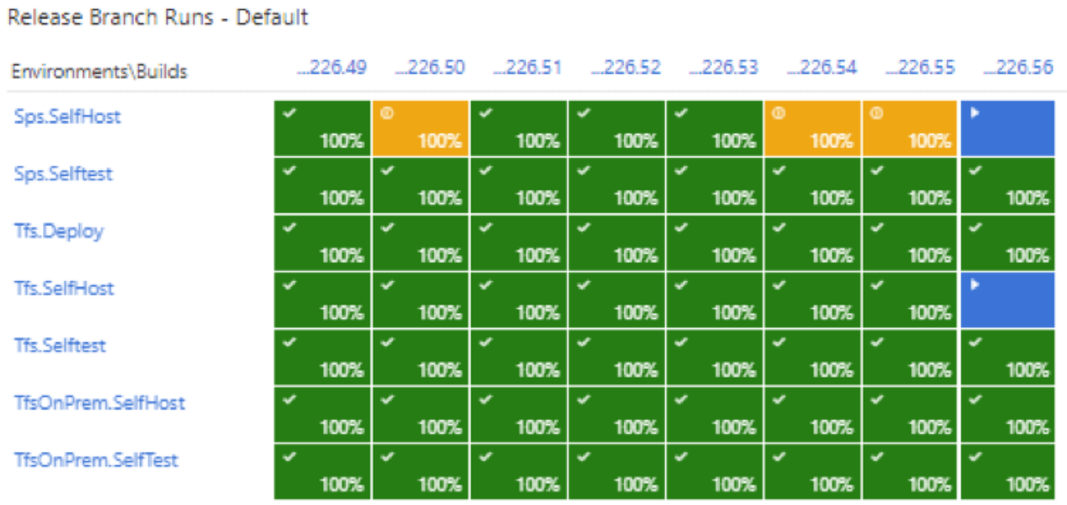


Figure 10: We always track the health of our release branch and master branch. This widget is hosted on a VSTS dashboard.

Build measure learn

If you are in Microsoft, then you would be inundated with acronyms. BML (Build measure learn) is one of them.

BML is both a process and a mindset.

When I joined Microsoft, I was told not to be afraid of taking risks and experimenting as long as I can build, measure, and learn. Also, I was told that making mistakes is fine if we expect, respect, and investigate them.

And that is the philosophy of BML as well. Whatever we build, we measure it and learn from it.

Build - The experiments we run are based on hypothesis. Once the hypothesis is established and accepted, the experiment is executed keeping in mind that we do not spend many engineering cycles in implementing the experiment since we would like to fail fast.

Measure - So what do we measure? Pretty much everything. When we create features or experiments, we measure as much as we can to validate availability, performance, usage, and troubleshooting. On any given day we capture almost 60GB of data and we use this data to enhance our features, to troubleshoot issues, or measure the health of our service.

Learn - Learning from what we measure is the most integral aspect of the whole journey.

For example, when I was creating VSTS Wiki (an easy way to write engineering and project related content in VSTS), I had to set an attachment size limit for adding content to Wiki. This limit would determine the size of the images and docs you can attach to a wiki page. I did not know what size of attachments users would prefer in a wiki page, therefore, I experimented. I went with a meager 5MB attachment size with the

hypothesis that users would only add images and small docs to a wiki page. My hypothesis was invalidated in the first month of Wiki launch and we started seeing a pattern where 90% of the attachments were less than 18MB therefore we found the magical number for our attachment size. Now VSTSWiki supports 18MB of attachment size and most of my users are happy with it.

You can read about all these investments in the DID YOU KNOW section of my blog here: https://blogs.msdn.microsoft.com/devops/2018/01/12/link-wiki-pages-and-work-items-write-math-formulas-in-wiki-keyboard-shortcuts-and-more/

We also continuously monitor health. It could be the service health, team health, engineering health, or feature health.

TEAM HEALTH

Just to give some perspective, here is the engineering health of my team.

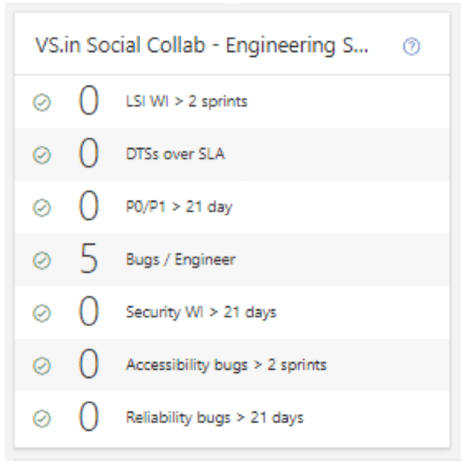


Figure 11: Team health widget in VSTS Dashboards

As you can see, we are at 5 bugs per engineer which is a max threshold we maintain for all feature crews. It basically means that I will have to ask my team to stop all feature work and get our bug count to as low as possible. By the time this article was published, we are down to 2 bugs per engineer.

USER HEALTH

We have integrated all user feedback on public forums such as https://developercommunity.visualstudio.com/spaces/21/index.html etc. into VSTS. The below pie chart indicates all VSTS Wiki feedback from my users. You can see that my user conversations are healthy since we have only 11 feedback tickets out of which all are either Fixed and pending a release or under investigation.

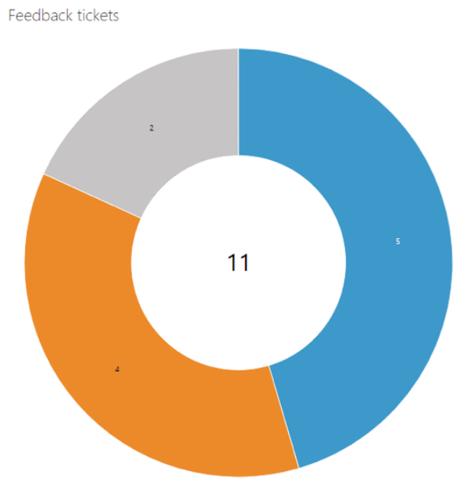


Figure 12: VSTS Dashboard widget indicating user feedback health

USER VALUE FLOW RATE

The below chart shows the rate of delivery of user value by my team. It shows that I am delivering bugs and stories at the rate of 55 days per work item. This is not as good as I would have liked and I need to improve it.

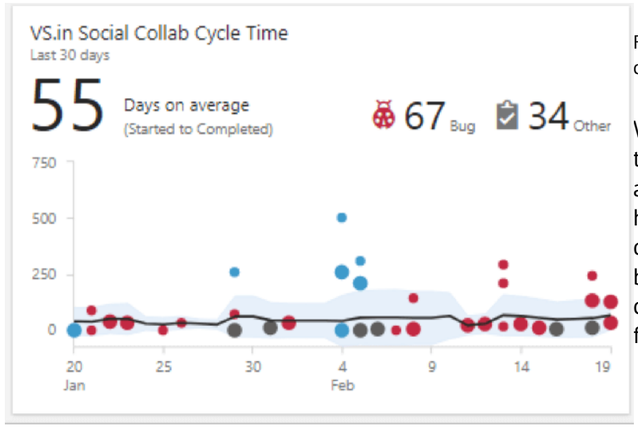


Figure 13: VSTS Dashboard widget indicating team cycle time

While writing this article I investigated that this is an issue with stale bugs that are not closed and are in resolved state. I have added another widget on the VSTS dashboard to identify stale items on my board. This will help to ensure that my cycle time is not as high as 55 going forward.

Continuous everything

I already talked about continuously testing our code with every merge to master. You can call it continuous testing which is powered by continuously integration. We also continuously deploy our code on the Monday after each sprint.

We always deploy our code in the canary instance or the dogfood environment first. Our dogfood environment is as pristine as production since that is the environment on which VSTS is built. If we have a buggy code in VSTS, then we can't use VSTS to fix VSTS therefore the stakes are high even for our dogfood environment.

Environment	Actions	Deployment status	Triggered
Ring 0 - tfs-wcus-0	•••	SUCCEEDED	4 days ago
Ring 0 - tfs-pfcusc	•••	SUCCEEDED	4 days ago
Ring 0 - tfs-ppe-su1	•••	SUCCEEDED	4 days ago
Ring 0 - tfs-eus2-su2	•••	SUCCEEDED	4 days ago
Ring 1 - tfs-sbr-1	•••	SUCCEEDED	3 days ago
Ring 2 - su3	•••	SUCCEEDED	3 days ago
Ring 2 - tfs-wus2-2	•••	SUCCEEDED	3 days ago
Ring 3 - tfs-weu-2	•••	SUCCEEDED	3 days ago
Ring 3 - su6	•••	SUCCEEDED	3 days ago
Ring 4 - su5	•••	SUCCEEDED	8 hours ago
Ring 4 - tfs-cca-1	•••	SUCCEEDED	2 days ago
Ring 4 - tfs-sin-1	•••	SUCCEEDED	2 days ago
Ring 4 - tfs-ea-1	•••	SUCCEEDED	2 days ago
Ring 4 - su7	•••	SUCCEEDED	2 days ago
Ring 4 - tfs-cus-1	•••	SUCCEEDED	2 days ago

Figure 14: We deploy to Ring0 (our canary instance) first and you can see that there is a deliberate delay of a day between Ring0 and Ring1 which is our next set of users.

We deliberately leave few hours before continuing the deployment on other environments so that we give feature crews enough time to react to any issues. Typically, the deployment across all our VSTS users completes in 5-6 days while we ensure that there is 99.9% service availability.

The goal of continuous everything is to shorten cycle time. Continuous Integration drives the ongoing merging and testing of code, which leads to finding defects early. Other benefits include less time wasted on fighting merge issues and rapid feedback for development teams. Continuous Delivery of software solutions to production and testing environments helps us to quickly fix bugs and respond to ever-changing business requirements.

There is much more to talk regarding security, collecting data, our KPIs, live site culture, limiting impact of failure using circuit breakers etc. I would love to gain feedback from this write-up to see what else would you be interested in and I will write about it. Consider this article as my MVP of DevOps.

Take away

If you had the patience to read so far, then first I want to applaud you. Next, I want to leave you with some key takeaway:

This is Microsoft's journey based on the gaps we found in our system and the priorities were set based on what we *felt* was hurting us the most. If you or your customers are trying to onboard to DevOps, then it is important to understand your needs and pain points. The problems or pain points should drive the solutions.

1. DevOps is a journey of continuous improvement and we are still on that journey. We have come a long way in this journey to be where we are however we are still finding flaws and wastages and we improve them … every day

2. DevOps is about your people, processes, and tools which is soaked in the culture and mindset of the organization and driven by business goals. We need all these ingredients to come up with a great DevOps success story.

Who wrote this article?

An entire feature crew that was aligned that it was important for others to learn from the mistakes we made at Microsoft. Happy DevOps journey!!

Sandeep Chadda

Author

Sandeep is a practicing program manager with Microsoft | Visual Studio Team Services. When he is not running or spending time with his family, he would like to believe that he is playing a small part in changing the world by creating products that make the world more productive. You can connect with him on twitter: @sandeepchads. You can view what he is working on right now by subscribing to his blog: aka.ms/devopswiki

Thanks to Roopesh Nair, Aseem Bansal, Ravi Shanker, Subodh Sohoni and Suprotim Agarwal for reviewing this article.

Andrei Dragotoniu

Having a Craftsman Mindset in Software Development

Software development is a highly social activity.

A developer rarely works on a project on their own, unless they build something small. Even then, chances are it will be worked on by someone else in the next weeks, months or years.

In many cases, the software built runs for a long time, years and sometimes even decades. As such, it is important to consider several aspects, before sitting down and writing any code.

Software Craftsmanship - Abstract

The purpose of this article is to cover software development from the point of someone who does more than just sit in a chair and does exactly what they are told.

It highlights that software development is a highly complex process with lots of smaller steps that are followed in order to build something of quality. Requirement gathering, architecture, testing, writing maintainable code, all come together to create a product that can be supported over its entire life cycle.

The article discusses each of these points and goes into details showing why they are important from an end result point of view.

The article also highlights how our own experience and attitude can have a big effect on the end result.

Context

First, we need to understand the context of what we are building.

There are a number of questions we need to answer before we write any code.

Are we building a brand-new application?

Is it a rewrite?

Is it part of a complex system?

Does it work on its own?

Are there any dependencies?

Do we understand what we need to build?

Do we understand how that piece of software will be used?

Do we understand who will use it?

Do we understand the role of this piece of software?

What about the maintenance of this piece of code?

As you can see, there are a lot of questions to answer straight off the bat. We did not even get to coding anything yet. All these questions give us the context, the conditions and restrictions of what we are building. They basically decide the area inside which we operate.

Why are these questions important? Why do we care if this is a new app or a rewrite?

Well, if this application is a rewrite then we need to understand why we are rewriting it.

Rewriting something is more than likely an expensive exercise and as such, there is usually a good reason for it. Maybe it's built using some old tech which doesn't fit with a much larger system change, maybe we need to add new functionality, maybe we need to make it easier to be used by its users. Maybe we need to make it easier to maintain.

Yes, maintenance is very important.

We might have written the most amazing piece of software the world has ever seen, but if no one understands how it's written, if other developers look at it and run away scared, then chances are that every little fix or change will not only take a long time but will more than likely cause issues.

There is a lot we can do to cover some of these questions and making sure there is a good answer to each one of them is usually a good starting point. As developers, we need to consider that we won't always be there to maintain that piece of software. As such, we can take care of the basics: clean code, good naming policy, SOLID principles, tests to prove it works as it's supposed to work.
If these things are taken care of, then we already are in a good place.

Software Architecture

Designing architecture of an application is something we pay attention to, from the beginning. It covers the way the system works and once in place, it's difficult, if not downright impossible, to change.

This is where we start, we get an idea of how everything might work, how things interact, how sub systems talk to each other. It is usually a good idea to do a few proof of concept (POC) little applications to see if everything really works the way we think they should work.

This is also where we might decide that actually, we don't know enough about certain aspects and we either seek help or we learn how to do it.

Knowing when to do each of these things is important. No one knows everything and we should be quite happy to admit what we don't know.

When we know what we don't know, that's when we learn. That's when we know what we need to learn. As a developer, we have to keep learning, there is so much out there that it can be quite daunting. When that happens, it's usually a good idea to take a step back and go back to basics.

Some people decide to specialize in certain aspects, which is good because they develop very good knowledge on something. Others like to be more generalists so they know some things about a lot of things.

None of these approaches is wrong.

The more you know, the better. More are the choices you will have when making any decisions and the easier it gets to explain why you made those decisions in the first place.

Remember how we said that software development is a very social activity? It is, because you need to interact with others, you need to explain your choices, to articulate why certain things should be done a certain way. It's all about communication, it's all about being open to suggestions and being able to spot a good idea, even if it wasn't yours.

I guess this pushes us into a very obvious direction. Understanding the social aspect in software development and using it to the advantage of whatever you're building. If you remove yourself from the equation and see the bigger picture, then it becomes obvious that the goal is to deliver something good. How you reach certain decisions won't be that important any more. It's the results and outcome of those decisions that matter.

You could of course say that the architecture is not really that important, it will be taken care of by the framework you're using. It could be something like MVC for example.

This isn't what I am talking about.

A framework will give you a starting point, it will help you by providing a few ways for you to build upon. It's usually very focused on a very specific type of activity. This is only a small part of what you are building, so it's good to understand that from the beginning.

However, a framework won't cover dependencies and interactions with other applications. You might even use more than one framework to build something. Maybe you add Entity Framework into the mix, maybe you add others as well. Now you have a number of frameworks in place and you need to get them to play nicely together.

The architecture of a system is decoupled from such things. It helps if you visualize things somehow. Even drawing on a whiteboard will do wonders because it will help you see the gaps.

Attitude

Be the solution, not the problem.

Software development is a social activity and you will interact with others a lot. It is very easy to let your ego take charge and fight everything. It's very easy to say you're the one making decisions and don't have to explain everything to everyone, but that usually leads to chaos. This kind of attitude won't be good for the project.

People will lose trust, then they start doing other things, if you don't bother keeping them informed (why should they?). Before you know it, the whole thing is going down the drain and eventually fails, because everyone does things their own way without thinking about the good of the project.

This is where the craftsman mindset shines the most. The project comes first. What this means is that you need to consider the bigger picture and code accordingly.

Quick fixes vs proper fixes

Sometimes people apply a quick fix just to get something working quickly. This is usually done to sort out something small that has a negative effect, usually when the system is in production.

What happens on bigger projects is that these quick fixes become norm, add up and given enough time will make a system pretty much unmaintainable.

As a craftsman, we need to understand these things. We're not just a machine, taking in pizza and coke and churning out code all day and night.

This is a highly logical and creative activity where experience matters, where you can spot the direction where something is going.

Yes, sometimes you might need to apply a quick fix to stop losing money, if a system is losing millions an hour because of an issue, then by all means, fix it as quickly as you can. Once you've done that, make sure a proper fix is in place, don't let these quick fixes add up and kill the system eventually.

What this boils down to is taking the time to understand the issue and put a fix in place which guarantees the problem won't happen again. This is part of a much bigger discussion of course, because we typically have deadlines and only so much time is available. However, being able to articulate and explain the problem, the solution and the dangers of ignoring the problem, is paramount. We're not there just to code, we're there to share our experience and prevent such things from happening.

In a way, attitude is a funny one.

That's because this works better if you are surrounded by people with a similar way of looking at and understanding things. But even if you are not in such a place, then maybe you should show why having this type of attitude works and helps build better software and maybe you can lead the way to change.

The last thing you want is to compromise and accept mediocrity because the product will end up being exactly that, mediocre!

Understanding the requirements and filling in the gaps

Typically, the requirements we get are never complete, they can't be.

Let's take a simple example. One of your requirements is to build a login page with a username and a password. How hard can this be? A page with two fields and a button.

Well, let's see.

The username, is it an email address, a phone number, or something else?

What happens when you press the Login button, but haven't filled in the username?

What if the password is missing as well?

Where do you show the error messages?

What color is used to display the error messages?

Should there be a link to a registration page?

What about a forgotten password link?

What happens if you forgot your username?

How do you get in touch with anyone to recover it?

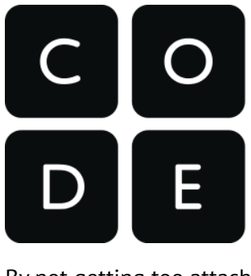
How do you make sure you don't fall victim to a SQL injection attack?

What happens if a user clicks the Login button 50 times very quickly?

The list of questions gets bigger and bigger. This is only a login page, imagine the number of questions when you build something more complex than that.

This is what we typically do, we look at a piece of work and start asking questions to make sure we build the right thing. We think of anything that could go wrong and we draw on the experiences of the past. Once we put a system live, once we go through a security audit, that helps clarify what kind of questions we should ask when building anything. Experience matters!

Don't get attached to code



This is a funny one. It's very easy to get attached to our code and even be defensive about the choices we make. Other people can't possibly understand the beauty of what we are doing, right?

Well, here's the thing. In the real world, no one is going to care. There is one simple truth out there, the code needs to perform a function, that's it. It either does it or it does not.

By not getting too attached, we allow ourselves to see the flaws and short comings.

It allows us to make changes without feeling sorry for doing it. It leaves the ego out the door and we can then focus on what matters, which is the business value. This is very important in my eyes, the code doesn't exist in a parallel world, it simply needs to perform a function and deliver value, maybe save someone's time, maybe it does more than ever, quicker than ever. There is always a reason why that code exists and if the reason goes away, we can then simply remove it, without any second thoughts.

> Focus on the business value first!

Clean code

We touched on programming being a social activity already. There are many things which matter and being able to support and maintain a codebase, is right at the top. Whether we are part of a big team or even a one-man team, we still need to think what will happen when the code we write goes into production.

Will we be able to look at it in a few months or years and still make sense of it?

Will someone else be able to pick it up and maintain it, fix bugs, maybe even add new features to it?

We all dread legacy code, but if we stop to think a bit about it, we will soon discover that every bit of code we write becomes legacy the moment it goes into production. The shiny new framework we think will solve all our problems, give it 6 months and it will be old news and something else will be the flavor of the month.

With all this in mind, it makes sense to focus more on how we write code.

So what exactly is clean code? Well, to make sense of some code we need to be able to read it like a story and understand it. This mean that our variables need to have proper names.

For example, instead of

```
_uRep = whatever
```

we could use something like:

```
_userRepository = whatever
```

If we look at a line of code and it's not possible to grasp very quickly what it does, then it's time to change it. Let's think not only of this moment in time, when everything is clear in our head, let's think what happens in a few months when all that context is gone and we start from scratch.

So, good variable names and good method names can help with understanding the code and make it easier to maintain it when the time arrives.

Another good approach is to avoid smart and complicated one liners. We all know what they are - those beautiful constructs which do a lot into a single line. Break it up, make it look dumb simple, easy to read and understand.

Hopefully by now, a clear pattern emerges; keeping it simple has never been more important.

Start applying some of the SOLID principles. We don't need to blindly apply everything, just the things that make sense. For example, the single responsibility principle is a good one to start with. This means we write classes which have one purpose and methods which do one thing.

This allows us to write testable code and this is very important for future maintenance.

I believe it is time to stop justifying why testing the code is important. This is the norm, not the exception. Most of the time we justify not doing things this way because we have deadlines and not enough time to actually write tests. This pushes us into the next section.

If you have a craftsman mindset, then prove it by doing it

We, software developers, need to stop being defensive about what we are doing. We shouldn't justify why we will spend time to add automated tests around our functionality, we just do it. We can of course, explain what this achieves, but not as a justification, we test things because this is helping the product, because we

are craftsmen and because we know what we are doing. When someone asks us to estimate a piece of work, we will do it, with testing in mind.

> Testing is part of our work, not a nice to have.

No one tells a plumber how to do his work, they quote you for a job, factoring in everything their experiences teach them and so do we. So, let's stop justifying what we know needs to be done and let's just do it. The product will be better off because of it.

This takes us back to the cutting corners side of our industry. As I said, let's be part of the solution not the problem. We know that cutting corners doesn't help, we know that continually introducing technical debt will eventually introduce a huge cost, so it might be better to do whatever we can to minimize it.
Here's a quick test for the current project you work on. Have a look and see how many technologies are used, how many frameworks, how many libraries.

Some nasty surprises might come out of this.

How many of those are there because someone wanted to use a particular thing?

How many libraries are there because someone needed one thing to be done, in one place and instead of writing a small method they introduced a whole new library to do the job?

This can all be solved if we switch the focus from ourselves, to the project itself. Only add something new if there is a clear, long lasting benefit, if it can be honestly justified through something more than just personal desire.

Testing

This is the big elephant in the room. People outside this profession will see that and immediately think, how much longer this will add to the development process. Did you just say 30%? No, we haven't got time for that, we'll test at the end!

How many times have we heard this? Of course, we already know the benefits of writing testable code and then adding tests to actually prove that it works as expected.

Imagine having these unit tests which run very quickly and thus having the certainty that the system behaves in a predictable way. Imagine being able to refactor without the fear that you might break something. Imagine getting a bug report, writing a failing test which highlights the bug, then fixing the code to make the test pass. Imagine having one more test which guarantees that the particular bug you just fixed won't show its ugly head again.

These are all great benefits already but wait, there is more!

Imagine not having to keep testing the whole system every time something changes, imagine the maintenance phase being much shorter and less expensive because you don't need so many manual tests anymore. We need to be ready to articulate why this way of working is better, even among us, even to other developers who may not code the same way. This helps bring everyone up to a higher level, work in a modern way, enjoy the work process for a change instead of running around doing countless extra hours, whenever a launch is looming.

Hacking your own code or security is not something you think about at the end

We are now moving into that area that no one likes to talk about, which is security. It seems to be something we always think about at the end, for some reason.

How do we build secure code? This is not so trivial, but it requires developers to think for themselves, because most of the time this area will not be covered by requirements. So, we are the ones who need to make sure that the beautiful login page we just wrote is actually secure.

We are the ones who see a URL like this: www.mysite/user-details/1 and think to ourselves, "hmm I wonder what happens if I replace 1 with 2 in the URL and hit enter"? We are the ones making sure that sensitive data is not exposed. We are the ones making sure that we build an API that can be accessed only by the intended audience and no one else.

This is a huge subject which deserves an article of its own. For now, because we have the attitude of a crafter, we can look at our login page and start thinking, how would we hack it? Maybe we go and read a bit about ethical hacking, get some ideas, see how the other side of the world functions and try to hack our own code. It won't be perfect, it won't be 100%, but it's a start.

Focus on something for a change

We all want to work on the latest and greatest, but that's not always possible.

Whether it is the latest framework, the latest version of something, maybe a new database system, maybe a new API type, whatever it is, we want it! That's not necessarily a bad thing, it's good to stay up to date with latest developments. It is however, also good to realize there is a balance that needs to be maintained. By jumping very quickly from one thing to another we risk not learning anything deep enough and we can end up with surface only knowledge. That's definitely not a good thing!

There are a lot of products written in languages and versions which have not been current for several years. Someone still needs to maintain and maybe even upgrade them. Businesses will rarely accept a complete rewrite of an application which does the job just fine. That's to be expected and we need to understand that. Yes, there is always something we can improve even in older products and if we can provide enough of a reason for them then certain changes can and will happen.

It's all down to us explaining the value of a change to the clients. It won't be done in a technical language and it will have to be understood by non-technical people. It's all down to communication skill and business value.

Conclusion

As you can see, we touched lightly on a lot of subjects. None of these is truly ground-breaking or amazingly new. It's easy to forget certain aspects though. I hope this article shows you how big a software project is, how many moving parts there are and how important they all are. The most important aspect is our knowledge. We are the ones with experience, who can see the defects a mile away, who can predict if something will work or not, based on hard data and our own tests.

We need to keep learning, keep educating ourselves and keep educating others. If we are in a team of ten developers and we help them improve 10% each, that's a total of 100% for everyone.

> Our own value is not the only one we bring to the table, what we can do for others matters sometimes even more.

We need to have more faith in our skills and take charge, make sure the right things are done, make sure there still is a balance and we can help build a product that can sustain the test of time.

Andrei Dragotoniu

Author

Andrei Dragotoniu is a software developer from Southampton, UK. He currently works for DST Bluedoor as a Lead Backend Developer, working on a financial platform, getting involved in code standards, code reviews, helping junior developers. He is interested in architectural designs, building efficient APIs and writing testable code. In his spare time, he blogs about technical subjects at www.eidand.com.

Thanks to Yacoub Massad for reviewing this article.

Yacoub Massad

Designing Data Objects in C#

In this article, I will give some recommendations on how to design data objects in C#. I will talk about immutability and about making invalid states unrepresentable. I will also present `DataObjectHelper`, a Visual Studio extension I created to help create data objects, followed by a demonstratation on how we can use F# to create data objects and use them from C# projects.

Introduction

In a previous article, Data and Encapsulation in complex C# applications (bit.ly/dnc-encapsultation), I talked about behavior-only encapsulation among other things.

When doing behavior-only encapsulation, we separate data and behavior into separate units. Data units contain only data that is accessible directly, and behavior units contain only behavior. Such behavior units are similar to functions and procedural programming, respectively.

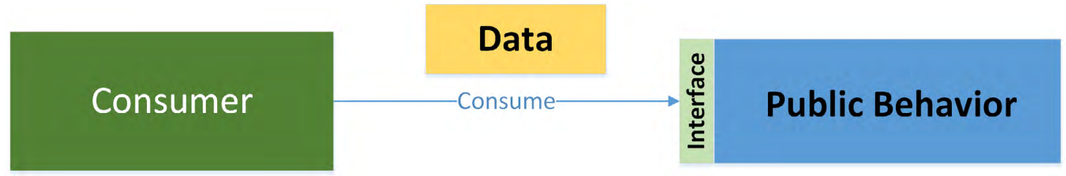


Figure 1: Behavior-only encapsulation

In this article, I am going to give some recommendations on how to design these data units. I also call them data objects because in a language like C#, we create data units using classes; and as far as the language is concerned, instances of classes are "objects".

Make your data objects immutable

This means that once a data object is created, its contents will never change.

In C#, we can do this by making all properties `read-only` and providing a constructor to allow the initialization of these properties like this:

```
public class TextDocument
{
  public TextDocument(string identifier, string content)
  {
    Identifier = identifier;
    Content = content;
  }

  public string Identifier { get; }
  public string Content { get; }
}
```

Benefits of immutability

By making data objects immutable, we get many benefits. I will only mention a couple of them here.

1. Data objects become thread-safe. This is true because all a thread can do to a data object is read from it. Having multiple threads reading something cannot cause a concurrency issue.

2. Code becomes easier to understand. For example, consider that you have some code that calls a function named `DoSomething` that has the following signature:

```
public static ProcessingResult DoSomething(TextDocument document)
```

To understand the effect of this function, we need to answer the following questions:

Q1. How is the return value `ProcessingResult` generated? In particular:

a. How is it affected by the input arguments? (the *document* argument in this case)
b. How is it affected by global state of the application (or even state external to the application such as a database)?

Q2. What effect on global state (or external state) does this function have?

Q3. What changes does this function do to the document argument?

By making data objects immutable, we can eliminate question number 3 and thus make understanding this method, easier.

To understand why this is important, consider the example where the `DoSomething` method passes the *document* argument to some other method, say Method1. Then, Method1 passes it to Method2 which passes it to Method3, and so on. Any of these methods can modify the `TextDocument` object passed to it.
Now, in order to answer question 3, we need to understand how all of these methods modify the document.

By the way, if we make this function *pure*, we can also remove Q1.b and Q2. This would make the function even easier to understand. To learn more about pure functions, see the Functional Programming for C# Developers article (bit.ly/dnc-fsharp) here at DotNetCurry.

Modifying immutable data objects

Immutable data objects cannot be modified. If we need to have a modified version of a data object, we have to create another one that has the same data, except for the part that we want to change. For example, if we want to append "Thank you" to the content of a document, we would create a new document like this:

```
var newDocument =
  new TextDocument(
    existingDocument.Identifier,
    existingDocument.Content + Environment.NewLine + "Thank you");
```

What if a data object has ten properties and we just want to "modify" one property?

We need to invoke a constructor and pass ten arguments. Nine arguments from these will just be simple property access expressions like *existingDocument.Identifier*. That code would look ugly.

To fix this problem, we can create `With` methods. Consider these `With` methods for the TextDocument class:

```
public TextDocument WithIdentifier(String newValue)
{
  return new TextDocument(identifier: newValue, content: Content);
}

public TextDocument WithContent(String newValue)
{
  return new TextDocument(identifier: Identifier, content: newValue);
}
```

Each of these methods allows us to "modify" a specific property of an object. Here is how to use one:

```
var newDocument =
  existingDocument
    .WithContent(existingDocument.Content + Environment.NewLine + "Thank you");
```

The DataObjectHelper Visual Studio extension

Writing `With` methods for large data objects is a hard and error-prone task. To facilitate this, I have created a Visual Studio extension that enables the auto generation of such methods.

You can download the DataObjectHelper extension (bit.ly/dncm35-dohelper) via the Visual Studio's Extensions and Updates tool from the Tools menu.

Once installed, you can use it by doing the following:

1. Create the following class anywhere in your code base:

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
public class CreateWithMethodsAttribute : Attribute
{
  public Type[] Types;

  public CreateWithMethodsAttribute(params Type[] types)
  {
    Types = types;
  }
}
```

2. Create a `static` class (of any name) to contain the `With` methods and apply the `CreateWithMethods` attribute to this class like this:

```
[CreateWithMethods(typeof(TextDocument))]
public static class ExtensionMethods
{
}
```

Note that when applying the attribute, we provide the type of data objects that we wish to generate `With` methods for. A single applied attribute can be given many types.

3. Hover over the `CreateWithMethods` attribute (where it is applied on the `ExtensionMethods` class). The icon for Roslyn based refactoring should show up. Look out for a refactoring with title "Create With methods", and click on it.

Figure 2: Creating With methods using the DataObjectHelper Visual Studio extension

Figure 2 shows how Visual Studio gives us a preview of refactoring with the `With` methods that the extension is about to make.

Please note that the *extension* creates `With` methods as *extension* methods, and not as *instance* methods inside the data object. One advantage of this is that you can create such methods for data objects that you don't have the source code for.

Make invalid states unrepresentable

This means that the structure of the data objects should be designed in a way that only valid states can be represented by such objects. Consider this data class:

```
public enum ResultType { Success, PartialSuccess, Failure }

public class DocumentTranslationResult
{
  public DocumentTranslationResult(
    ResultType result,
    Document translatedDocument,
    ImmutableDictionary<int, ImmutableArray<string>> pageErrors,
    string error)
  {
    Error = error;
    PageErrors = pageErrors;
    TranslatedDocument = translatedDocument;
    Result = result;
  }

  public ResultType Result { get; }

  public string Error { get; }

  public ImmutableDictionary<int, ImmutableArray<string>> PageErrors { get; }

  public Document TranslatedDocument { get; }
}
```

This class represents the result of translating a text document (e.g. from English to Spanish). It has four properties.

The `Result` property can contain one of three values:

1. **Success**: the translation is 100% successful. In this case, the `TranslatedDocument` property would contain the translated document. All other properties would be null.

2. **PartialSuccess**: the translation is partially successful. In this case, the `PageErrors` property would contains a list of errors for each page in the source document. This is represented by a dictionary where the key is an integer representing the page number, and the value is an array of error messages for that page. The `ImmutableDictionary` and `ImmutableArray` types are similar to the standard Dictionary class in .NET and arrays in C#, except that they are immutable. For more information, see this article from Microsoft: https://msdn.microsoft.com/en-us/magazine/mt795189.aspx.
   Also, the `TranslatedDocument` property would contain the translated document. We should expect to have some untranslated words or statements in this document. All other properties would be null.

3. **Failure**: Nothing could be translated. This can happen for example when a translation web service that we use, is offline. The `Error` property would contain an error message. Other properties would be null.

Note that the constructor of this class allows us to specify the values of the four properties when we construct the object. The problem with such design is that we can create an instance of `DocumentTranslationResult` that has an invalid state. For example, we can create an instance where `Result` has a value of `Success` but `TranslatedDocument` is null:

```
var instance =
  new DocumentTranslationResult(
    result: ResultType.Success,
    translatedDocument: null,
    pageErrors: null,
    error: serviceError);
```

This is clearly a programmer mistake.

Either the programmer wanted to set `Result` to `Failure` but set it to `Success` by mistake, or she wanted to set `error` to `null` and `translatedDocument` to a specific value but forgot to do so (maybe as a result of copying and pasting some other code?).

One attempt to fix this problem is to throw exceptions on invalid states. Consider this updated constructor:

```
public DocumentTranslationResult(
  ResultType result,
  Document translatedDocument,
  ImmutableDictionary<int, ImmutableArray<string>> pageErrors,
  string error)
{
  if(result == ResultType.Success && translatedDocument == null)
    throw new ArgumentNullException(nameof(translatedDocument));

  if(result == ResultType.Failure && error == null)
    throw new ArgumentNullException(nameof(error));

  if (result == ResultType.PartialSuccess && pageErrors == null)
    throw new ArgumentNullException(nameof(pageErrors));

  if (result == ResultType.PartialSuccess && translatedDocument == null)
    throw new ArgumentNullException(nameof(translatedDocument));

  Error = error;
  PageErrors = pageErrors;
```

```
  TranslatedDocument = translatedDocument;
  Result = result;
}
```

Now, if we try to construct the same invalid object as the example above, an exception will be thrown.

This is better since we will get an exception when we construct the object, instead of getting a NullReferenceException when we try to use the object later.

Still, we can only detect this problem at runtime. It would be better if we could detect and prevent this problem at compile-time. That is, if we get a compilation error trying to construct an invalid object.

Let's now consider another example:

```
var instance =
  new DocumentTranslationResult(
    result: ResultType.Success,
    translatedDocument: document,
    pageErrors: null,
    error: serviceError);
```

Again, this is clearly a programmer mistake.

This code will compile and will not give us a runtime exception when executed. When debugging the application, the programmer might hover over the instance variable and see a non-null value for the `Error` property and incorrectly think that this result represents a failed result.

We can mitigate this problem by throwing an exception when the *result* argument is `Success` and the *error* or the *PageErrors* arguments are non-null. Such invalid state checking in the constructor acts as documentation that helps programmers understand which states are valid and which are not.

There is a better option!

Let's remove the constructor and add these three constructors instead:

```
public DocumentTranslationResult(Document translatedDocument)
{
  Result = ResultType.Success;
  TranslatedDocument = translatedDocument;
}

public DocumentTranslationResult(
  Document translatedDocument,
  ImmutableDictionary<int, ImmutableArray<string>> pageErrors)
{
  Result = ResultType.PartialSuccess;
  TranslatedDocument = translatedDocument;
  PageErrors = pageErrors;
}

public DocumentTranslationResult(
  string error)
{
  Result = ResultType.Failure;
  Error = error;
}
```


This is better. Now it is harder for the programmer to make the mistakes he/she made before. With this change, only the following three ways of constructing the object will result in a successful compilation:

```
var instance1 = new DocumentTranslationResult(document);

var instance2 = new DocumentTranslationResult(document, pageErrors);

var instance3 = new DocumentTranslationResult(serviceError);
```

Now, these constructors act as the documentation for valid states.

Still, one can argue that by looking at the four properties of this class alone (without looking at the constructors), one cannot easily understand which combinations of values are valid and which are not.

Consider what happens if you want to use IntelliSense to view the properties of a variable of type `DocumentTranslationResult`. Here is how it looks like in Figure 3:

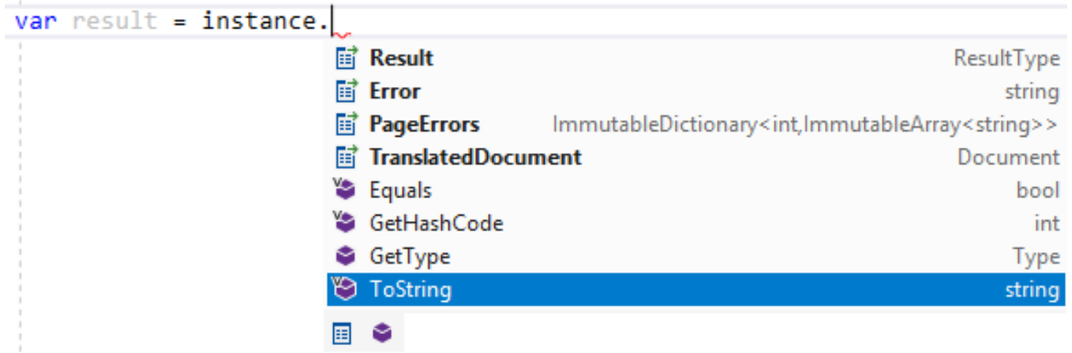


Figure 3: IntelliSense for DocumentTranslationResult

All that the programmer can determine from this is that this object has four properties.

This doesn't tell her/him which combinations are valid and which are not. The programmer has to go and see the constructors to understand which combinations are valid and which are not.

Let's say that the programmer wants to write a method to generate a formatted message about the result of translation. Consider this `FormatResultForUser` method that the programmer writes:

```
public string FormatResultForUser(DocumentTranslationResult result)
{
  switch (result.Result)
  {
    case ResultType.Success:
      return
        "Success. Translated document length: "
         + result.TranslatedDocument.Content.Length;

    case ResultType.Failure:
      return "Failure. Error: " + result.Error;

    case ResultType.PartialSuccess:
      return
        "Partial success. Number of errors: "
```

```
        + result.PageErrors.Values.Sum(x => x.Length)
        + “, Translated document length: “
        + result.TranslatedDocument.Content.Length;

    default:
      throw new Exception(“Unexpected result type”);
  }
}
```

In this code, the programmer is switching on the `Result` property to handle each case. One problem with this code is that when the programmer accesses the *result* parameter, all the properties are available. So, for example, in the Success case, the programmer should only access the `TranslatedDocument` property, however, nothing prevents her from accessing the `Error` property.

Can we find a better design that better communicates how to interpret and use the `DocumentTranslationResult` class?

Consider this alternative design:

```
public abstract class DocumentTranslationResult
{
  private DocumentTranslationResult(){}

  public sealed class Success : DocumentTranslationResult
  {
    public Success(Document translatedDocument)
    {
      TranslatedDocument = translatedDocument;
    }

    public Document TranslatedDocument { get; }
  }

  public sealed class Failure : DocumentTranslationResult
  {
    public Failure(string error)
    {
       Error = error;
    }

    public string Error { get; }
  }

  public sealed class PartialSuccess : DocumentTranslationResult
  {
    public PartialSuccess(
    ImmutableDictionary<int, ImmutableArray<string>> pageErrors,
    Document translatedDocument)
    {
      PageErrors = pageErrors;
      TranslatedDocument = translatedDocument;
    }

    public ImmutableDictionary<int, ImmutableArray<string>> PageErrors { get; }

    public Document TranslatedDocument { get; }
  }
}
```

In this updated design, the `DocumentTranslationResult` class is abstract and contains no properties whatsoever. It has three derived classes that are nested in it; Success, Failure, and PartialSuccess.

The Success class has only one property - `TranslatedDocument`, the Failure class also has one property - `Error`, and the PartialSuccess class has two properties: `PageErrors` and `TranslatedDocument`.

Like before, we have three constructors, one constructor per subclass. But now, each subclass has only some of the properties of the original class. It is easier to tell which properties are related with which result type.

Note also that the `ResultType` enum is no longer needed. The type of the result is encoded in the type system, i.e., as a derived class of `DocumentTranslationResult`. This has the additional benefit of preventing invalid enum values.

In C#, enum variables are not guaranteed to hold a valid value. By default, the underlying type behind an enum is `int`. This means that any `int` value can be converted to `ResultType`. For example, *(ResultType)100* is a valid value to be assigned to a variable of type ResultType.

Note also that there is a parameterless constructor of `DocumentTranslationResult` that is private. This gives us better control over which classes can inherit from `DocumentTranslationResult`. Usually, if the base class constructor is private, no classes can inherit from it. However, nested classes are special in that they can access the private members of the classes they reside in.

Now, if we want to modify this data object to add another result type, we can create another nested class inside `DocumentTranslationResult` that derives from it. Such a class has to be a nested class in `DocumentTranslationResult` or the code will not compile.

Now, if we have a variable of type `DocumentTranslationResult.PartialSuccess` and we try to use IntelliSense to view its properties, we can see the two relevant properties, as seen in Figure 4.

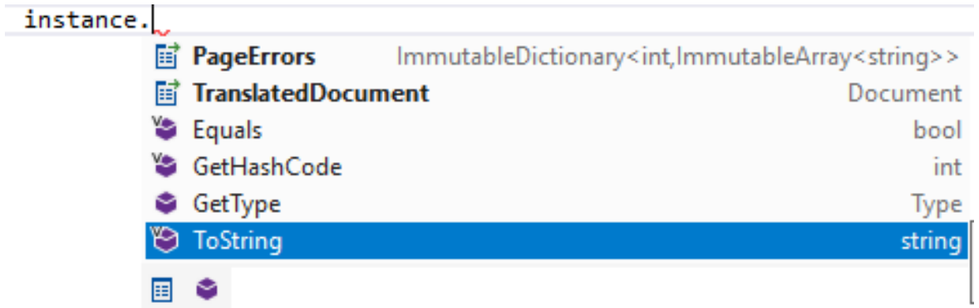


Figure 4: IntelliSense for DocumentTranslationResult.PartialSuccess

However, if we have a variable of type `DocumentTranslationResult` (which can hold an instance of any of the three subclasses), we get the following in IntelliSense (see Figure 5).

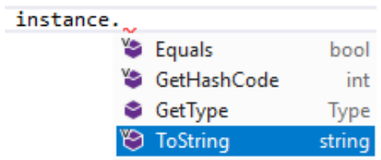


Figure 5: IntelliSense for DocumentTranslationResult abstract class

This is very bad!

Now, we have no idea how to work with this data object. We now have to go to the class definition and read it. It is only after we have read it, do we know about the three subclasses and the properties they contain.

Before I propose a fix for this, let's see how the `FormatResultForUser` method looks like now:

```
public string FormatResultForUser(DocumentTranslationResult result)
{
  switch (result)
  {
    case DocumentTranslationResult.Success successResult:
      return
      "Success. Translated document length: "
      + successResult.TranslatedDocument.Content.Length;

    case DocumentTranslationResult.Failure failureResult:
      return "Failure. Error: " + failureResult.Error;

    case DocumentTranslationResult.PartialSuccess partialSuccessResult:
      return
      "Partial success. Number of errors: "
      + partialSuccessResult.PageErrors.Values.Sum(x => x.Length)
      + ", Translated document length: "
      + partialSuccessResult.TranslatedDocument.Content.Length;

    default:
      throw new Exception("Unexpected result type");
  }
}
```

Similar to the `FormatResultForUser` method from before, we use a `switch` statement to switch on the type of the result. The ability to switch over the type of a specific variable is only available as of C# 7, as a result of introducing the pattern matching feature.

Notice that for each case, we are defining a new variable. For example, for the Success case we have a variable called `successResult` of type `DocumentTranslationResult.Success`. This means that we can only access properties related to the success case via this variable. Contrast this with the `FormatResultForUser` method from before. There, we had a single variable result that gave access to all the properties.

For more information about the pattern matching feature of C# 7, read the C# 7 - What's New article here at DotNetCurry (bit.ly/dnc-csharp7).

Still, as I showed earlier, a variable of type `DocumentTranslationResult` does not tell us anything useful in IntelliSense. One way to fix this is to add a `Match` method to the `DocumentTranslationResult` class like this:

```
public TResult Match<TResult>(
  Func<Success, TResult> caseSuccess,
  Func<Failure, TResult> caseFailure,
  Func<PartialSuccess, TResult> casePartialSuccess)
{
  switch (this)
  {
    case Success success:
      return caseSuccess(success);
    case Failure failure:
      return caseFailure(failure);
    case PartialSuccess partialSuccess:
      return casePartialSuccess(partialSuccess);
    default:
      throw new Exception(
        "You added a new subtype of DocumentTranslationResult without updating the
Match method");
  }
}
```

The name of the method `Match` comes from pattern matching.

Now if we use IntelliSense to access a variable of type `DocumentTranslationResult`, we see the Match method as shown in Figure 6:

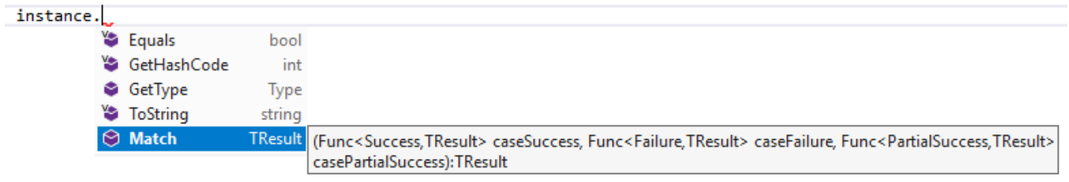


Figure 6: IntelliSense of the DocumentTranslationResult.Match method

Using IntelliSense, we know that the translation result can be of three types; Success, Failure, and PartialSuccess. Here is how the updated `FormatResultForUser` method looks like:

```
public string FormatResultForUser(DocumentTranslationResult result)
{
  return result.Match(
    caseSuccess: success =>
      "Success. Translated document length: "
      + success.TranslatedDocument.Content.Length,
    caseFailure: failure =>
       "Failure. Error: " + failure.Error,
    casePartialSuccess: partialSuccess =>
       "Partial success. Number of errors: "
       + partialSuccess.PageErrors.Values.Sum(x => x.Length)
       + ", Translated document length: "
       + partialSuccess.TranslatedDocument.Content.Length);
}
```

For each result type, we provide a lambda that takes the specific subtype and gives back a value. This means that in each case, we only have access to the properties defined in the corresponding subtype.

The `Match` method is a generic method that has a generic type parameter `TResult`. This allows us to use it to return a value of any type. Of course, all the lambdas passed to the `Match` method must return a value of the same type.

The `Match` method enforces the matching to be *exhaustive*, that is, the code using the `Match` method has to handle all cases or otherwise it won't compile. Compare this with when we used a `switch` statement. When using a `switch` statement, we can forget to include a specific case and the compiler will not complain.

Sometimes, we need to deal with an instance of `DocumentTranslationResult` in a way such that we

don't want to extract or generate some data from it, instead we might want to simply invoke some code for each case.

For example, we might want to write some data from the object to the console. The following `Match` method overload helps us achieve this:

```
public void Match(
  Action<Success> caseSuccess,
  Action<Failure> caseFailure,
  Action<PartialSuccess> casePartialSuccess)
{
  switch (this)
  {
    case Success success:
    caseSuccess(success);
      break;
    case Failure failure:
    caseFailure(failure);
      break;
    case PartialSuccess partialSuccess:
    casePartialSuccess(partialSuccess);
      break;
    default:
      throw new Exception(
        "You added a new subtype of DocumentTranslationResult without updating the
Match method");
  }
}
```

Instead of taking `Func` delegates, this method takes `Action` delegates. The method will call the appropriate action based on the type of the result.

Using the DataObjectHelper extension to create Match methods

The DataObjectHelper extension can also help with generating `Match` methods. Here is how to use it to do so:

1. Create the following class anywhere in the codebase:

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple =true)]
public class CreateMatchMethodsAttribute : Attribute
{
  public Type[] Types;

  public CreateMatchMethodsAttribute(params Type[] types)
  {
    Types = types;
  }
}
```

2. Create a `static` class (of any name) to contain the `Match` methods and apply the CreateMatchMethods attribute to this class like this:

```
[CreateMatchMethods(typeof(DocumentTranslationResult))]
public static class ExtensionMethods
{
}
```

3. Hover over the `CreateMatchMethods` attribute (where it is applied on the ExtensionMethods class). The icon for Roslyn based refactorings should show up. A refactoring with title "Create Match methods" should show up, click on it.

Like `With` methods, `Match` methods are created as extension methods.

Other issues

Let's look at the `DocumentTranslationResult.PartialSuccess` class. This class has a property of type `ImmutableDictionary<int, ImmutableArray<string>>` called `PageErrors`. This property has these issues:

Issue 1: The page number is represented using an integer. Integers can have the value 0 and even negative values. However, a page number cannot be zero or negative. The current design allows us to create an instance of `DocumentTranslationResult.PartialSuccess` that has errors for page -5 for example.

Of course, we can fix this problem by throwing exceptions in the constructor of the `DocumentTranslationResult.PartialSuccess` class. However, a better way to communicate that page numbers have to be positive, is to create a type for positive integers like the following:

```
public class PositiveInteger
{
  public int Value { get; }

  public PositiveInteger(int value)
  {
    if (value < 1)
      throw new ArgumentException("value has to be positive", nameof(value));

    Value = value;
  }
}
```

And change the property like this:

```
public class PositiveInteger
{
  public int Value { get; }

  public PositiveInteger(int value)
  {
    if (value < 1)
      throw new ArgumentException("value has to be positive", nameof(value));

    Value = value;
  }
}
```

And change the PageErrors property like this:

```
public ImmutableDictionary<PositiveInteger, ImmutableArray<string>> PageErrors {
get;}
```

At runtime, we are still protecting against negative page numbers by throwing exceptions. However, this new design communicates in a better way which page numbers are valid and which are not. Also, instead of having an exception-throwing-code duplicated in all places that needs to accept page numbers, we centralize this check in the `PositiveInteger` class.

There are ways to get better compile-time invalid state checking here. For example, we could write an analyzer (bit.ly/dnc-diag-analyzer) that prevents the code from compiling if we construct a `PositiveInteger` class using a non-positive integer literal.

Additionally, we could make the analyzer prevent construction of this class using a non-literal integer. A special static `TryCreate` method can be used to allow the construction of `PositiveInteger` instances using a non-literal int:

```
public static bool TryCreate(int value, out PositiveInteger positiveInteger)
{
  if (value < 1)
  {
    positiveInteger = null;
    return false;
  }

  positiveInteger = new PositiveInteger(value);
  return true;
}
```

This has the advantage of making it very clear to the developer who is creating an instance of a `PositiveInteger` that not all instances of `int` can be converted to `PositiveInteger`. The developer has to try and check if such conversion was successful.

A better signature for this method would use the `Maybe` monad (more on this later):

```
public static Maybe<PositiveInteger> TryCreate(int value)
```

To make the design even more descriptive, we can use a `PageNumber` class (that contains a PositiveInteger) for the key of the dictionary to represent the page number.

Issue 2: String is used as the type of error. A better option is to create an `ErrorMessage` class to better communicate that this represents an error message.

Issue 3: The dictionary and the array can be empty. However, if the document is partially successful, then at least one page must contain at least one error.

Again, we can throw exceptions at runtime in the `DocumentTranslationResult.PartialSuccess` class constructor if the dictionary is empty or the error array for any page is empty. However, it is much better to create a `NonEmptyImmutableDictionary` and `NonEmptyImmutableArray` classes in the same way we created the `PositiveInteger` class.

Also, to make the design more descriptive, we can create a `PageErrors` data object to represent page errors. Such object would simply contain the dictionary property.

Note: *The tendency to use a primitive type (e.g. `int`) instead of a special class (e.g. `PageNumber`) to represent a page number is called primitive obsession.*

Don't use null to represent optional values

If you are reading a data class and you see a property of type `string`, can you tell whether it is optional or required? In other words, is it valid for it to be null?

We sometimes use the same type, e.g. `string`, to represent required and optional values, and this is a mistake.

Model your data objects in a way that makes it clear which values are optional and which are not. The Maybe monad (also called the Option monad) can be used to mark a value, e.g. a property, as optional. Consider this Customer data object:

```
public class Customer
{
  public Address PrimaryAddress { get; }

  public Maybe<Address> SecondaryAddress { get; }

  //..
}
```

It is very clear from this design that the primary address is required, while the secondary address is optional.

A very simple implementation of the `Maybe` monad looks like this:

```
public struct Maybe<T>
{
  private readonly bool hasValue;

  private readonly T value;

  private Maybe(bool hasValue, T value)
  {
    this.hasValue = hasValue;
    this.value = value;
  }

  public bool HasValue => hasValue;

  public T GetValue() => value;

  public static Maybe<T> OfValue(T value)
  {
    if (value == null)
      throw new Exception("Value cannot be null");

    return new Maybe<T>(true, value);
  }
```

```
  public static Maybe<T> NoValue()
  {
    return new Maybe<T>(false, default);
  }
}
```

There are many open source C# implementations available for the `Maybe` monad. When creating the DataObjectHelper extension, I wrote one implementation. You can find it here: https://github.com/ymassad/DataObjectHelper/blob/master/DataObjectHelper/DataObjectHelper/Maybe.cs

Sum types and Product types

The `DocumentTranslationResult` class I used in the example (as is finally designed) is `sum` type. A `sum` type is a data structure that can be any one of a fixed set of types. In our example, these were the Success, Failure, and PartialSuccess types.

These three sub types themselves are product types. In simple terms, a product type defines a set of properties that an instance of the type needs to provide values for. For example, the `DocumentTranslationResult.PartialSuccess` type has two properties; `PageErrors` and `TranslatedDocument`. Any instance of `DocumentTranslationResult.PartialSuccess` must provide a value for these two properties.

Sum types and product types can be composed to form complex data objects. The `DocumentTranslationResult` example in this article has shown a *sum* type whose possible subtypes are product types. But this can be even deeper. For example, the Success case can hold a property of type `TranslationMethod` which can be either Local or ViaWebService. This `TranslationMethod` type would be a *sum* type. Again, the Local and ViaWebService types can be product types each containing details specific to them. For example, the ViaWebService type can have a property to tell which web service was used.

Design your data classes so that they are either sum types or product types. In C#, this means that a data objects should either be abstract and have no properties and have derived types to present cases for this type, or it should be a sealed class and contain properties.

If you design your data objects this way, then you only need `Match` methods for sum types and `With` methods for product types.

By the way, the `Maybe` monad is a sum type because it can either have a value or not have a value. In my implementation, I use a struct for technical reasons. However, I have seen implementations of it that use two classes to represent the two cases.

Sum types and product types and combinations of these are called *Algebraic Data Types*. These types have very good support in functional programming languages. In the next section, I will show you how you can easily create your data objects in F# and use them from your C# projects.

Note: *The phrase "make illegal states unrepresentable" was introduced by Yaron Minsky. For more information, see this video: https://blog.janestreet.com/effective-ml-video/*

Using F# to create data objects and use them in C# projects

F# is a .NET functional programming language. Although it is a functional language, it also has object-orientation support. Because it compiles to MSIL as any .NET language does, code written in F# can be consumed in C#, and vice-versa.

F# has great support for algebraic data types. Consider this code:

```
module MyDomain =
  type Document = {Identifier : string; Content: string;}

  type imdictionary<'k,'v> = System.Collections.Immutable.
  IImmutableDictionary<'k,'v>
  type imarray<'e> = System.Collections.Immutable.ImmutableArray<'e>

  type DocumentTranslationResult =
    | Success of TranslatedDocument : Document
    | Failure of Error: string
    | PartialSuccess of PageErrors : imdictionary<int, imarray<string>> *
  TranslatedDocument : Document
```

The code starts by creating a module.

A module in F# is simply grouping of F# code. This module contains a Document *record*. A record in F# defines a set of named values. In C# terms, this is like a data class which contains a set of properties. A record in F# is a product type.

The `Document` record has two values (properties in C#); Identifier of type string and Content of type string.

Next, we define two type aliases; `imdictionary` and `imarray`. This is simply to make it easier to use the ImmutableDictionary and ImmutableArray types.

Next, we define what is called a *discriminated union* in F#. This is a sum type.

A discriminated union allows us to create a type that can be one of a set of fixed types. In this case we define the `DocumentTranslationResult` discriminated union that can either be Success, Failure, or PartialSuccess. Notice how we define the fields (or properties in C#) for each case.

Now compare the amount of code we write here versus the amount of code we need to write in C#.

We can write our data objects as F# records and discriminated unions and then write our units of behavior in C#.

Note: *Types created with F# are somewhat similar to the types I described in this article when viewed from C#.*

There are some differences though.

The DataObjectHelper library can be used to create `With` and `Match` methods for types created in F#. The `static` class that will contain these methods must exist in a C# project, but the types themselves can

come from F#. The DataObjectHelper library supports creating `With` and `Match` methods for all the types in a specific module. A module in F# is compiled as a `static` class in MSIL. If you specify this class when applying the `CreateWithMethods` and the `CreateMatchMethods` attributes, `With` and `Match` methods for all types within this module will be generated.

By the way, in the future, C# is expected to have records similar to F# records. You can see the proposal here: https://github.com/dotnet/csharplang/blob/master/proposals/records.md

A data object should represent one thing only

Don't be tempted to use a single data class for many purposes. For example, if there is another translation process in your application that can either succeed or fail (cannot be partially successful), don't reuse the same `DocumentTranslationResult` class because it models a result that can also be partially successful. In this case, create another data object to model this new result type.

Non-closed data type hierarchies

There are cases where it makes sense to have something similar to a sum type but whose subtypes are not fixed. That is, other cases can be added later by a developer who is not the author of the type. I chose to make this topic out of scope for this article, but I might discuss it in a future article.

Conclusion:

In this article, I have given some recommendations on how to design data objects in C#.

I talked about making the data objects immutable to make multithreading safer and to make the program easier to understand. I also talked about making invalid states unrepresentable. We make invalid states unrepresentable by modeling data objects carefully as sum types or product types.

I talked about making it easier to "modify" immutable objects by creating `With` methods and how the DataObjectHelper Visual Studio extension helps us generate such methods.

Also, for *sum* types, the DataObjectHelper extension can help us generate `Match` method to easily and safely switch over the different cases represented by sum types.

I also talked about how we can use F# to concisely create our data objects, and use them in C# projects.

Yacoub Massad

Author

Yacoub Massad is a software developer who works mainly with Microsoft technologies. Currently, he works at Zeva International where he uses C#, .NET, and other technologies to create eDiscovery solutions. He is interested in learning and writing about software design principles that aim at creating maintainable software. You can view his blog posts at criticalsoftwareblog.com

Thanks to *Damir Arh* for reviewing this article.

Rahul Sahasrabuddhe

Clear the deck for – Kubernetes!

It's all Greek to you? Well, Kubernetes is indeed coined from a Greek word. In this article, although we will not learn the Greek language, we will learn about what containerization is, how it is rapidly changing the cloud-based deployment landscape and where Kubernetes fits in. We will also touch upon how various cloud providers are dealing with this change.

Introduction

TL;DR: Kubernetes is containers + orchestration.

*image courtesy: pixabay.com

Confused? Ok; let's get more technical and understand it better.

Kubernetes (also called as k8s) is an open source system (or a platform) that is used for orchestration of application deployments based on Linux's container-based approach across a variety of cloud offerings.

Quite a loaded definition isn't it?

Kubernetes essentially allows you to effectively manage application's deployments lifecycle viz. handling dependencies, packaging, auto-scaling them up or down, rolling in and out the versions of deployment, managing high availability aspects and so on.

Ok so why is it called Kubernetes?

It literally means pilot or captain in Greek. So if you want to ship your app, then you need a right pilot or captain – and that, my fellow passengers, is Kubernetes for you! If you look closely at the logo of Kubernetes, it is actually similar to a ship's wheel with spokes.

Figure 1: Kubernetes Logo

Kubernetes was founded by Joe Beda, Brendan Burns and Craig McLuckie for Google as an internal project based on Linux containerization and was called as Borg. This was eventually made as an open source project by Google and was donated to Cloud Native Computing Foundation.

Let's look at what Kubernetes is and what it is not:

Kubernetes..

1. provides deployment and rollout configurations that allow you to specify how you want your apps to be deployed initially and updated later on. This also ensures that rollbacks are easier to manage. The deployment configurations are called manifests and are written in YAML or JSON.

2. supports automatic bin-packing. It ensures that containers run optimally in terms of their resource requirements like memory, CPU etc. You can define min and max resource requirements for your containers and Kubernetes will fit them into a given infrastructure to optimize the workload. This optimizes underlying infrastructure without compromising performance metrics of your application.

3. has built-in service discovery. Kubernetes exposes your containers to other containers across boundaries of physical hardware (i.e. over internet as well). This allows you to use other services or allows your apps services to be consumed by other apps. This is a truly micro-services based model in the works.

4. provides auto-scaling: It allows your apps to be up-scaled based on increased demand and down-scaled based on reduced demand. This can be done at configuration level.

5. nodes are self-healing. The high availability aspect is owned by Kubernetes services. It also ensures that containers restart and reschedules nodes if they go down or are not responsive to user requests. When it brings up the containers again, they are not available to the user until they are in an appropriate state for containers to serve user requests.

6. It has a built-in mechanism to manage secret information securely.

7. Storage orchestration allows Kubernetes to provide persistent storage out of the box. Under the hood it can connect to any cloud provider's storage service.

8. Batch execution is out of the box for managing CI workloads.

Kubernetes is not..

1. It is not just some open source science project. It is much more serious than that and it is evident by the fact that it was initially developed up by Google and now it is owned and maintained by CNCF (Cloud Native Computing Foundation).

2. It is not just limited to running applications. It can also run "applications" like Spark, RabitMQ, Hadoop, MongoDB, Redis and so on. Basically if an app can run in a container, Kubernetes can manage it.

3. It does not own CI/CD aspects for your app. You will still need to own and take care of those.

4. Using Kubernetes does not absolve you from not worrying about application architecture and multi-tenancy. It takes no responsibility of what is inside the container. Kubernetes only ensures that the infrastructure used by apps is container-based. Kubernetes is not the silver bullet.

5. Lastly Kubernetes is not some scary and complicated technology. Anyone who can invest some time in understanding the concepts behind containerization well can work with Kubernetes on any cloud platform confidently and efficiently.

In order to understand the value proposition that Kubernetes brings in, we will need to take a step back first and understand as to how application deployment in general has evolved over the last few years and why containerization has caught all the attention of late.

Its show time folks – an app release!

Be whichever role you are in – developer, tester, project manager or product manager – the real acid test of all your day's (or sprint's) work is a successful deployment of your application.

You can pretty much compare this experience with a movie release.

After days and months of efforts of conceiving, scripting, acting, recording, editing and doing whatever it takes to make a movie, the anxiety of a movie release can be well felt by all of us who work hard in building software.

Not so many years ago, an application or product release used to be an elaborately-planned and a once-in-a-3-to-6-months kind of a phenomenon. There was a careful process of verification of features developed and corresponding regressions by thorough testing through various environments like Dev > Test > Staging > Prod.

However nowadays, companies release an application directly on production systems at times and that too every hour (or every second sometimes). And hence Continuous Integration (CI) and Continuous Deployment (CD) have become critical aspects of product (application) development & deployment methodology.
So what has caused this change?

Following are some key factors that have a role to play in this shift of deployment strategy:

1. **Move to cloud:** These days most applications are deployed to private or public clouds. On-premise deployment is now considered as a special case. In fact, until the containerization arrived on the scene, cloud-based deployment nearly has been synonymous to a virtualized hyper-visor based deployment. So you would end up installing or deploying your applications on VMs allocated to you as a part of private/public/hybrid cloud infrastructure.

2. **Time-to-market:** The technology advances and WYSIWYG JavaScript based UIs for web or mobile have reduced time to develop apps significantly. Additionally, fierce competition in product space has resulted into pushing deadlines from months to weeks and from weeks to days. Product releases have to be fast enough to maintain user stickiness. So CI/CD has pushed the bar to deploying apps as you develop.

3. **Microservices:** This is more of an architecture level change forced by points #1 and #2 together. Applications have evolved into loosely coupled service or function based components that are mashed up to get desired results. So your applications should no more be tightly coupled monolithic components. And to support that model, the deployment of application should also be modular and flexible enough.

So the application deployment has now become a fast-paced process that needs to run reliably and optimally irrespective of the application development methodology.

Making a "K"ase for Kubernetes

Let's look at current and past application deployment options with an analogy to understand containerization, and thereby Kubernetes.

Consider your application as an individual that wants to live in a specific community or area (i.e. the infrastructure in which it gets deployed) and while doing so, he/she will consume resources like water, electricity and so on (i.e. the services it uses like storage, CPU, memory etc.).

The following figure depicts three scenarios of living viz. a house/bungalow colony, an apartment complex and finally a hotel.

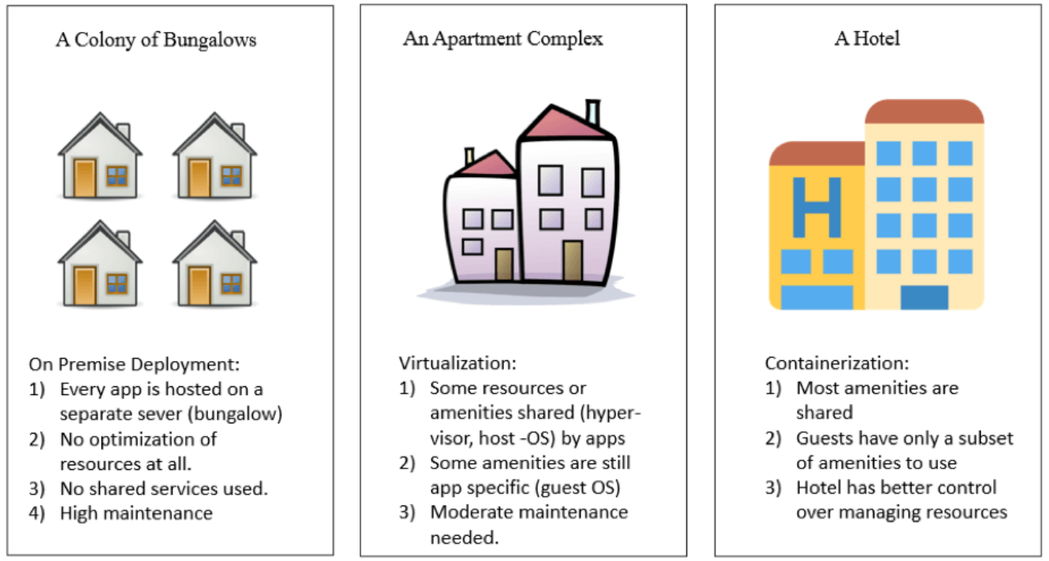


Figure 2 : App Deployment Analogy

Here are some salient points:

1. This is used as an analogy to build a case and illustrate the point. Please do not stretch it too far

2. In context of the living in bungalow (on premise):

   - When you are living in a bungalow, you get desired freedom but at the same time, you have to be responsible for managing and maintaining various services yourself (electrical wiring within the bungalow, repairs to water line etc.). This maps to OS upgrades, app upgrades and so on in context of application deployment from IT perspective.
   - Each house may have services used and developed differently. So plumbing layout for one house is not the same as plumbing layout for another. From a regulatory authorities' perspective, they do not have much control over such situation. This can be related to challenges faced in managing various applications on premise from IT perspective.

3. For the apartment complex (virtualization):

   - Some things are clubbed together as common services. So the drainage line, water-line are same. However, some things are still required to be taken care of by the owner (internal electrical wiring or plumbing or bath fittings). This is similar to having a guest OS installed on a VM that runs on a hyper-visor with its own OS.
   - The apartment owner would still incur some cost for maintaining various services.
   - Any internal change or update needed for each apartment needs to be handled by accessing each apartment. So app updates are not really seamless in a virtualized environment.

4. For the hotel (containerization):

   - More number of services are clubbed together in this model. Consequentially, you are in control of only a few services in your hotel room (like hot/cold temperature control of water in your room). This model allows the executioner to optimize various parameters.
   - You are allowed to bring in only specific things as per hotel policies. So in a container-based ecosystem, the app can only have specific privileges.
   - You can rely on specific services that are available for common use. For example laundry service. So a containerized app can rely on external services and can reuse certain libraries and frameworks.

5. From bungalow to apartment to hotel, there is a trade-off made in terms of having to worry less about managing the services as against the individual (i.e. the app or app provider) losing "control" over services. While the individual can do whatever he/she wants in his/her bungalow, it certainly will not be acceptable in a hotel room. You know what I mean!

So from the perspective of an infrastructure service provider, bungalow to hotel is a great transition to have, since the resources are optimized across consumers and more control is gained ensuring a reliable service.

From the consumer perspective, it works out well too because the consumer does not have to worry about the nitty-gritties of managing services.

In a nutshell, containerization is the way to go when it comes to reliable, scalable and cost-effective application deployment.

Let's now look at visualization vs. containerization and understand the nuances better.

Virtualization vs. Containerization

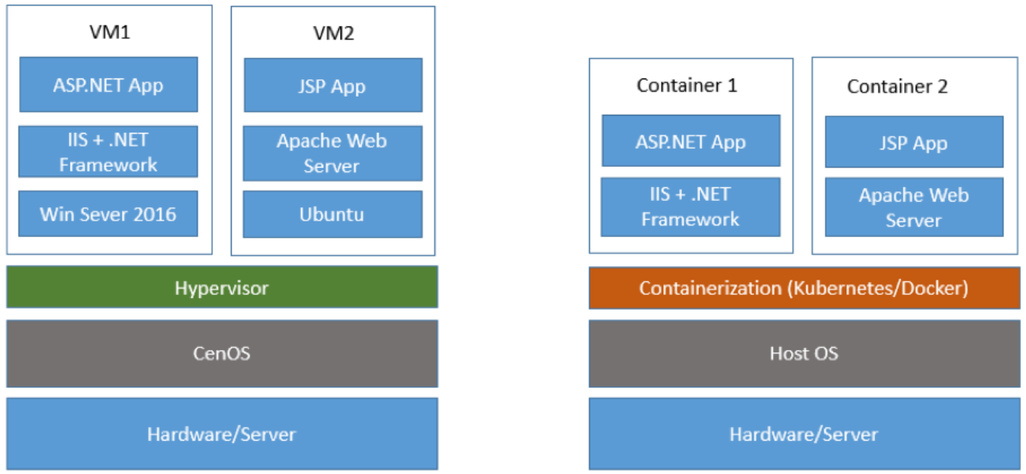


Figure 3 : Virtualization vs. Containerization - how do they stack up?

Following are some key aspects:

1. The basic premise of virtualization is optimization of hardware whereas containerization focuses more on an application that needs to be hosted or handled within the scope of container. VMs are abstracting the hardware whereas containers are abstracting the services at OS level.

2. In a virtualized set up, hypervisor runs on one OS (and machine) and then each VM has a guest OS that it runs on. In containerization, the containers use base OS services like kernel, hardware and so on. This implies immediate cost saving in OS (and in some case framework/packages) licenses. It also means that same OS license can be used for managing multiple containers having multiple applications. That results in further saving of cost of hardware.

3. VMs usually would take significant time to boot up; whereas containers can be brought up fairly quickly. Hence, overall performance of bringing up the apps is faster in containerization as compared to a virtualized environment since the unit being handled in case of containerization, is an app, as against a VM.

4. The unit of operation – VM – in case of virtualization limits the scalability of infrastructure across private/public/hybrid clouds. You have to bring up a new VM to scale up (only applicable to hardware-based scaling or horizontal scaling).

5. For very fast and very frequent deployments (which is a business requirement now), containerized model is far more efficient, scalable and reliable as compared to virtualized model.

6. It can be argued that containers are less secured as compared to VMs because they share the kernel of the OS (as against a VM that is well insulated so to say). But this has been handled by relevant container solutions by using signed containers for deployment or scan contents of container before/during deployment.

Let's Talk Greek – the Kubernetes Language

Following is a schematic representation of core components of Kubernetes. It does not include ALL the components; but nonetheless, this will give you a good enough idea of the components involved.

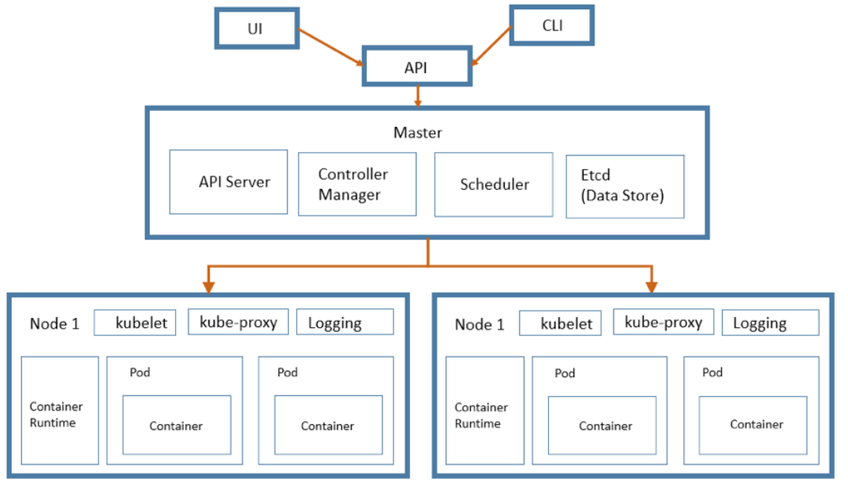


Figure 4: Kubernetes Architecture Schematic

Kubernetes works on the basic principle of grouping or clustering the resources that are needed by applications to be deployed. The unit of work or measure is the application. So the application that we want to deploy is the focal point of all the operations or actions.

Kubernetes architecture consists of following key components:

1. **Cluster:** It is basically a group of nodes that can be physical or virtual servers. A cluster houses Kubernetes set up. A cluster has a master node.

2. **Node:** A node (used to be called as minion) is given/assigned tasks by Master and is controlled by Master. This is the single unit of operation that handles containers. It is a worker machine and it might be a VM or a physical machine based on the underlying cluster. Each node runs the following two components:

   a. Kubelet: Each node runs an agent called as "kubelet". It keeps a watch on the master for the pods that are assigned to the node on which it is running. It performs operations on nodes with a continuous update provided to the Master for all the operations. This is how the master would know the current status of containers (and hence apps) and can take corresponding actions.

   b. kube-proxy or proxy: It handles various network aspects for node and plays a vital role in network load-balancing. It is accessed by Services.

   c. Each node runs a container runtime (like docker runtime) and that is the binding glue between the nodes and Master when it comes to execution of tasks.

d. The nodes also have an elaborate logging mechanism in place to log all the activities. Fluentd, Stackdriver or even Elasticsearch.

3. **Pod:** A pod is nothing but a group of containers deployed to a single node. Pods create the required abstraction for containers to work across nodes. A pod plays an important role in scaling of apps. You can scale up/down components of app by adding/removing pods respectively. Pods are created and destroyed by replication controllers based on the template provided to them. Pods have labels assigned to them. Labels hold pod-specific data in key-value pairs.

4. **Master:** This is the controller that holds together all the services of Kubernetes and hence controls all the activities of nodes in a cluster. This is also called as the Control Plane of Kubernetes. The master is hosted on one of the nodes. It has the following sub-components:

   a. API Server (kube-apiserver): It provides access to various RESTful APIs that handle various aspects like manipulation of states of objects and persisting their state in data store. The API server allows the UI and CLI to access the APIs through the API layer shown in Figure 4.

   b. Controller Manager (kube-controller-manager): It manages various controllers like node controller, endpoint controller, replication controller and token controllers.

   c. Scheduler (kube-scheduler): It simply manages schedules for newly created pods and assigns them to nodes.

   d. Data Store (etcd): It serves as single source of truth when it comes to all the data used by various Kubernetes components. The data is stored in a key-value pair.

5. **Service:** This is an abstraction on top of pods and provides a "virtual" IP for nodes to communicate with each other. It defines logical set of pods and policies to access them. They are called as micro-services.
   kubctl: This is a command-line tool used for various operations between API service and master node.

6. **Volumes:** these are directories that carry data used by pods for operations. A volume provides an abstraction for the underlying data and data source to be accessed. For example, an Azure file or Azure directory can be accessed through volumes and so on.

7. **Deployment:** It is a declarative way of managing deployments of apps. It is used by deployment controller for managing deployments.

8. **Secrets:** A secret object is used to hold sensitive information like passwords, OAuth tokens etc.

The list can go on forever and will have more updates after every release of Kubernetes. This link has a detailed glossary of terms used in context of Kubernetes.

There is a Kubernetes for that!

Since Kubernetes is based on containerization principle, it is not essentially tied to any specific IT infra model or any cloud provider. And that is why it is termed as a "platform" that can run on various configurations.

The following link gives a complete run-down of all possible scenarios and corresponding Kubernetes offerings. Please do go through this link to understand the versatility of the platform.

Having gone through what all is on offer, let us now focus only on specific key cloud platforms (not necessarily in any order of popularity or preference – before you begin to take sides!)

All the cloud providers already have a container service developed (CaaS or Containerization as a service) and now they have come up with a Kubernetes-based container service (i.e. managed Kubernetes service – wherein some basic aspects of Kubernetes are already owned/managed by underlying cloud provider).

Following is a short summary:

Cloud Provider	Container Service	Managed Kubernetes Service	Relevant Links	Comments
Azure	ACS (Azure Container Service)	AKS	How-to (goo.gl/U6s5A7) Quotas and region availability (goo.gl/534RGD)	• Service is free but you need to pay for VM infra. • Starts with 3 nodes.
AWS	ECS (Elastic Container Service)	EKS	How-to (goo.gl/gda57R) ECS vs. EKS (goo.gl/R4yGkn)	• Kops is the tool used for handing ops aspects • Fargate is preferred over EKS for execution
GCP	GCE (Google Container Engine)	GKE	How-to (goo.gl/BwPP3h) GCE vs. GKE vs. GAE (goo.gl/S8ThPe)	• Billing is pay per use. • Much faster support on GCP for newer features

Now that we know so much about Kubernetes, it would be interesting to know about real products that are deployed using Kubernetes. Head over to the case studies page here and you will find a lot of real-world products using Kubernetes.

And an interesting fact for game lovers – the application logic for Pokemon Go runs on GKE and the auto-scaling abilities of Kubernetes were put to real test (and use) when the popularity of the game sky-rocketed in certain timeframes.

The Competition

Loving Kubernetes? Well, let's see at what the competition of Kubernetes looks like.

Usually Kubernetes gets compared with Docker. However, the key difference is Docker is a way of defining the containers for the application, whereas Kubernetes is the container orchestration platform or engine (COE). So you will create a Docker image of your application and then you will use Kubernetes to deploy it onto one of the cloud platforms. A Kubernetes pod would contain a Docker image for an app. So it is not Kubernetes vs. Dockers but it is Kubernetes and Dockers.

Docker has come up with a service called Docker Swarms and that essentially is comparable to Kubernetes when it comes to functionalities offered by it.

Apache Mesos is another name that usually comes into picture when it comes to comparing COEs. It is a DC/OS (Datacenter Operating System). Marathon is the COE based on Mesos.

Following is a short and sweet comparison of all three offerings:

Key Aspects	Kubernetes	Docker Swarm	Apache Mesos /Marathon
Released in*	July-2015	Released with Docker 1.12 version	July-2016
Design Principles	Resource-grouping based on pods	Docker-based	Based on "cgrouops" or control groups concepts in Linux
Stability	Fairly matured and stable. And is continuously being updated.	Fairly new and hence evolving.	Fairly matured.
Container Images Supported	Supports Docker and rkt up to certain extent	Tightly coupled with Docker image format	Supports mainly Docker
Learning Curve	Steeper	Easier to work with if Docker knowledge	Steeper
Deployment	YAML based	YAML based	Own format

* Dates might vary based on sources.

Conclusion

Containerization is probably where the application deployment world is having more focus of late than virtualization. And various cloud providers are giving due attention to this euphoria as well.

As far as container-based application deployment space is concerned, Kubernetes has made quite a splash and is following that up with good action in terms of rolling out new features and versions.

Does it all make virtualization irrelevant? Will Kubernetes be the clear winner?
Well, time will tell.

You must have realized that Kubernetes is no child's play! However, here is an excellent video tutorial that pretty much discusses all the concepts we talked about in a much simpler kindergarten story-telling way. Have a look.

Φιλοσοφία Βίου Κυβερνήτης – ok that's actually in Greek and it means – philosophy (or wisdom) is the governor of one's life!

• • • • • • • •

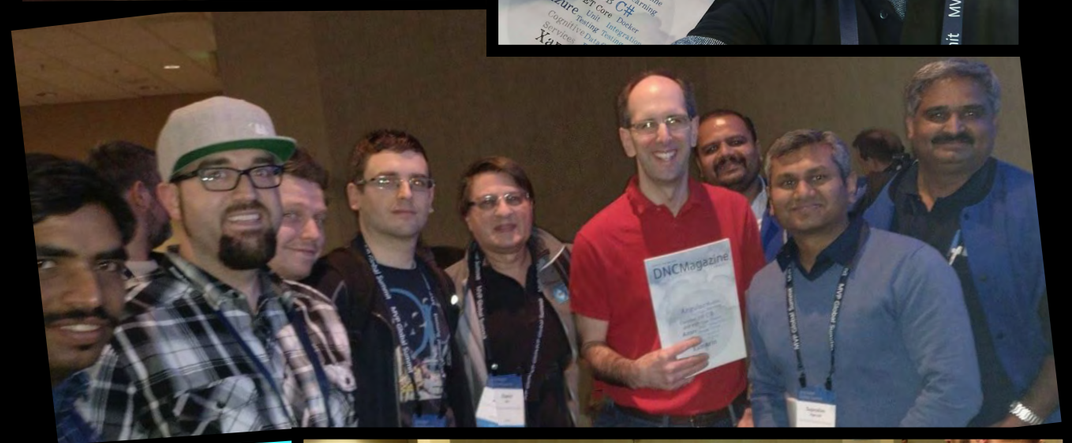
Rahul Sahasrabuddhe

Author

Rahul Sahasrabuddhe has been working on Microsoft Technologies since last 17 years and leads Microsoft Technology Practice at a leading software company in Pune. He has been instrumental in setting up competencies around Azure, SharePoint and various other Microsoft Technologies. Being an avid reader, he likes to keep himself abreast with cutting edge technology changes & advances.

Thanks to Subodh Sohoni and Abhijit Zanak for reviewing this article.

MVP Global Summit 2018



Shreya Verma

New SQL Graph Features in Azure SQL Database and SQL Server 2017

> BE IT SOCIAL NETWORKS, TRANSPORTATION AND FLIGHT NETWORKS, FINANCIAL TRAFFIC ROUTES, FRAUD DETECTION TO WEBSITE TRAFFIC ANALYSIS OR PREDICTING THE LIKELIHOOD TO PURCHASE; GRAPHS ARE UBIQUITOUS.

Graph Databases are emerging as a strong technology to solve problems in these areas, where relationships are the first-class entities for the application. These workloads comprise of highly connected data and one must navigate these data points to generate results like recommendations, fraud detections, shortest paths or predicting the best time to run a deal on a retail website.

But, do you need to install a new graph database product to solve these problems?

In this article, we will look at the graph database features introduced with SQL Server 2017 and Azure SQL Database. We will discuss the benefits of using SQL Server Graph, typical graph database use-cases and how you can start using the graph features in your application today.

Editorial Note: *If you haven't already, download SQL Server 2017 or start with a Free Trial of Azure SQL Database subscription (goo.gl/SgCzR3) to explore the concepts explained in this tutorial.*

What is a graph database?

A graph database is a collection of nodes (or vertices) and edges (or relationships). A node represents an entity (for example, a person or an organization) and an edge represents a relationship between the two nodes that it connects (for example, likes or friends).

Both nodes and edges may have properties associated with them.

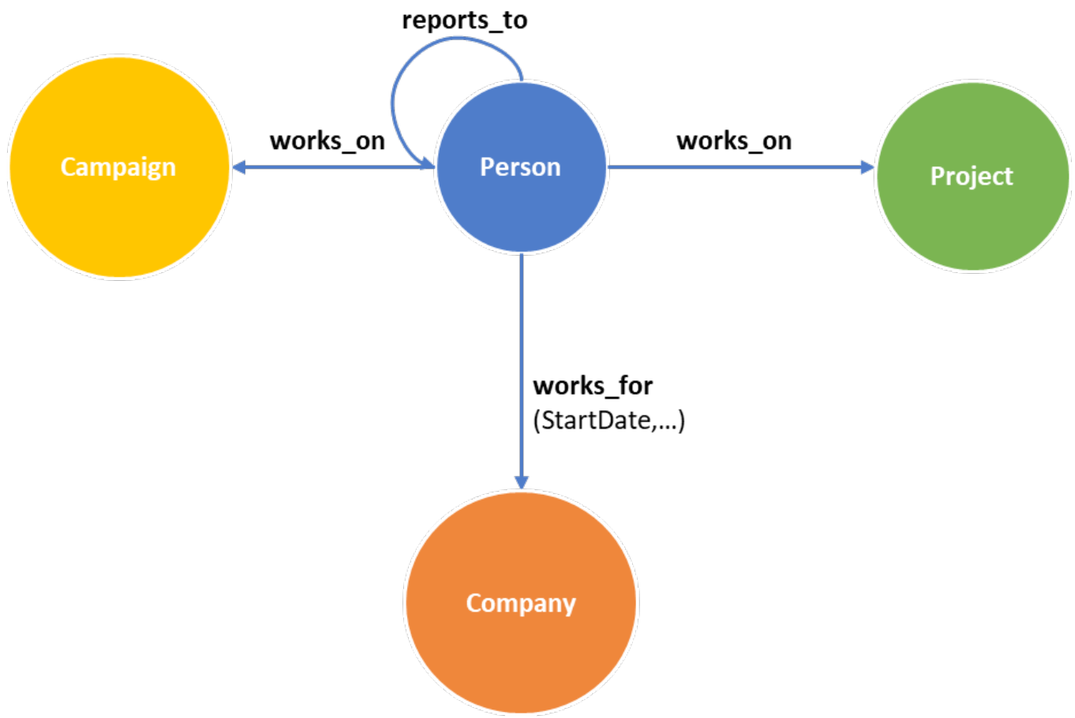


Figure 1: A property graph database with entities and relationships.

One may ask, **if a graph database is also a collection of entities and relationships, then how is it different from a relational database?**

Graph Database vs Relational Database

Here are some features that graph databases offer, which make them unique:

- Unlike a relational database, where foreign keys represent *one-to-many* relationships, an edge in a graph database can represent *many-to-many* relationships. An edge can also be heterogeneous in nature, that is, you can connect different type of nodes with each other using the same edge. For example, in the property graph shown above, you can connect a 'Person' to a 'Project' or a 'Campaign' via the *works_on* edge.

- Relationships become first class entities in a graph database and may or may not have any properties associated to them. For example, in the graph shown in Figure 1, a 'Person' *works_for* a 'Company' and you can store the StartDate (the date since when the employee started working for the company) as an attribute of the *works_for* relationship.

- Graph databases provides you a query language which allows you to express pattern matching and graph traversal queries easily.

- You can heterogeneously connect to various data points and express arbitrary length traversal queries easily.

What is SQL Graph?

SQL Server 2017 and Azure SQL Database now support graph data processing.

SQL Server implements a property graph model and can natively store nodes and edges. The T-SQL extensions also let you write join-free pattern matching and traversal queries for your graph database. In this section, we will review the features that were introduced with SQL Server 2017 and are also available on Azure SQL Database.

Creating a Node or Edge table

Transact-SQL (T-SQL) extensions allow users to create node or edge tables. Both nodes and edges can have properties associated to them.

Since, nodes and edges are stored as tables, all the operations that are supported on relational tables, are also supported on a node or edge table.

Additionally, all types of constraints and indexes are also supported. That is, along with edges, you can also create foreign key relationships between your node and edge tables. Clustered and non-clustered Columnstore indexes are also supported, which can be very useful in analytics-heavy graph workloads.

The following is an example of the new DDL that has been introduced. Note the two new keywords 'AS NODE' and 'AS EDGE'.

```
-- DDL for creating node and edge tables.
CREATE TABLE Person (ID INTEGER PRIMARY KEY, Name VARCHAR(100), Age INT) AS NODE;
CREATE TABLE works_for (StartDate DATE, EmploymentStatus VARCHAR(100)) AS EDGE;
```

For every NODE table, the engine adds one implicit column `$node_id` to the table, which uniquely identifies each node in the database. Values for this column are automatically generated by the engine every time a row is inserted into the edge table.

For every edge table, the engine adds three implicit columns `$edge_id`, `$from_id` and `$to_id` to the table.

`$edge_id` uniquely identifies every edge in the database and just like `$node_id`, values for `$edge_id` are automatically generated. The `$from_id` and `$to_id` implicit columns hold the `$node_id`s of the nodes that a given edge is connecting to. Users are expected to insert `$node_id` values into the `$from_id` and `$to_id` columns to identify the nodes that the edge is connecting.

Check out some SQL Graph insert examples and best practices.

Traversing or querying the graph

The new MATCH built-in function is introduced to support pattern matching and join-free multi-hop navigation or traversal through the graph.

MATCH uses ASCII-art style syntax for pattern matching. For example, if a user wants to find the names of the people who work for Microsoft, they can write the following query.

```
-- Find the people who work for Microsoft
SELECT Person.Name
FROM Person, works_for, Company
WHERE MATCH(Person-(works_for)->Company)
AND Company.Name = 'Microsoft';
```

In a MATCH pattern, you go from one node to another via an edge. Entities appearing at the two ends of an arrow are nodes and the one appearing inside parenthesis is an edge. The direction of the arrow in MATCH corresponds to the direction of the edge.

Note: *Edges in SQL Graph are directed and they always go from one node to another node.*

Native to the SQL Server engine

Graph extensions are fully integrated into the SQL Server engine. It uses the same SQL Server storage engine, metadata, and query processor to store and query graph data. This enables users to query across their graph and relational data in a single query.

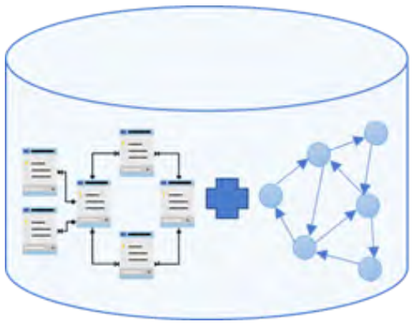
Users can also benefit from combining graph capabilities with other SQL Server technologies like Columnstore, Availability Groups, R services, and more.

Tools and ecosystem

Users benefit from SQL Server's existing tools ecosystem. Tools like backup and restore, import and export, BCP and more work out of the box. Other tools or services like SSIS, SSRS or Power BI work with graph tables, just the way they work with relational tables.

Benefits of using SQL Server Graph

1. Relational and Graph on a single platform



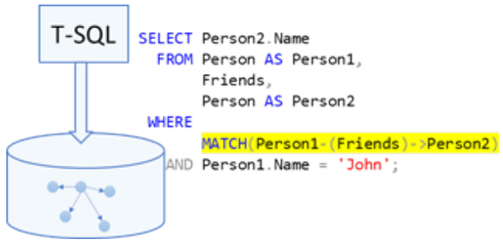
SQL Server now provides you the best of both relational and graph databases on a single platform.

Customers do not have to turn to a different product, deal with multiple licensing costs or perform complex ETL operations to take data out of one system and move it to another system for analysis. A single platform can now hold both the relational and graph data.

You can also query across your graph and relational data in the same query.

2. T-SQL extensions for graph traversal queries

The T-SQL extensions let you create and traverse the graph using a known language and in a known environment. You do not need to learn a new language, tools or ecosystem just to process different kinds of data. Your application, which is already talking to the relational database, can query the graph database using the same set of tools.

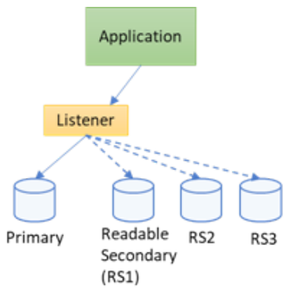


3. Performance and Scale

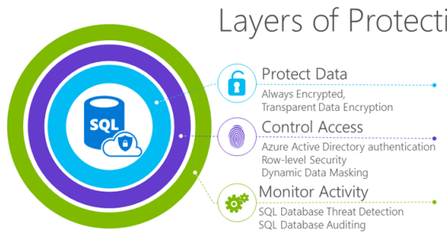
As per the TPC-H 10TB benchmark result by Hewlett Packard Enterprise (HPE), SQL Server attained a new world record in database performance for both analytical and transactional workloads? You get the same performance and scale benefits when you use the graph features in SQL Server.

4. High Availability

SQL Server can be easily configured for high availability. On Azure SQL Database, you can ensure high availability with three hot replicas and built-in automatic failover that guarantees a 99.99% availability SLA. You can accelerate recovery from catastrophic failures and regional outages to an RPO of less than five seconds with active-geo replication.



5. Security and Compliance



A defense-in-depth strategy, with overlapping layers of security, is the best way to counter security threats.

SQL Server provides a security architecture that is designed to allow database administrators and developers to create secure database applications and counter threats.

SQL Database helps you build security-enhanced apps in the cloud by providing advanced built-in protection and security features that dynamically mask sensitive data and encrypt it at rest and in motion.

With physical and operational security, SQL Database helps you meet the most stringent regulatory compliances, such as ISO/IEC 27001/27002, Fed RAMP/FISMA, SOC, HIPPA, and PCI DSS.

6. Fully managed on Azure SQL Database

Azure SQL Database is intelligent, fully-managed cloud database service built for developers.

You can accelerate your graph database application development and make the maintenance easy using the SQL tools that you already know. You can also take advantage of built-in intelligence that learns app patterns and adapts to maximize performance, reliability, and data protection.

When to use SQL Graph?

Graph databases shine when relationships between various data points are equally or more important than the actual data itself.

For example, an online retailer may decide to run a marketing campaign based on the connections that people share with other people, locations or weather conditions. Or a bank may study the links between various account holders to identify fraud rings formed by these account holders.

Up until few years ago, graph databases were considered suitable for networking problems, like social networks, transportation networks, and computer networks. But, recently with the advent of big data and evolving modern applications, graph databases are garnering lot of attention in traditionally relational applications like banking, insurance, CRM and more. Organizations running these type of applications need access to both relational and graph databases.

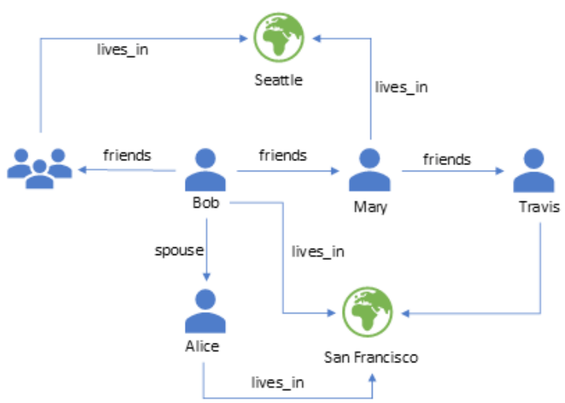
SQL Graph helps keep your graph and relational data on a single platform.

In this section we will discuss some scenarios where graph databases can be useful.

Graph Database Examples

Social Networks

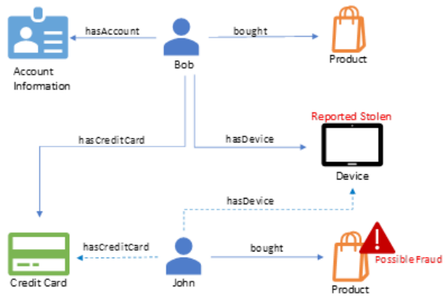
Social Networks are inherently graphs. It is only natural to model them as a graph.



Graph databases not only allow modeling of social networks as graphs, they also let you traverse and query the graph in an intuitive way.

There are many modern applications like online dating, job websites and social media marketing, which depend largely on the connections that people share with each other for better data analysis or to provide services to their customers like providing recommendations. Graph databases make it easy to answer questions like **"find me all my friends who are 2-5 hops away from me and who live in Seattle".**

Fraud Detection



Insurance and banking companies lose millions of dollars to fraud every year. SQL Graph can make it easier to identify fraud transactions that connect back to a stolen device or credit card.

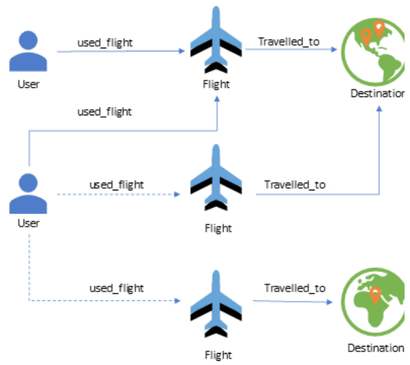
Using SQL Graph, you can also more easily identify fraud rings formed by a group of people who do not share direct connections with each other, but do share some common information like address, phone or SSN in their account details.

Recommendation Engines

SQL Graph can be easily used to generate recommendations.

For example, an airline company may want to generate flight recommendations for a customer booking tickets to a destination, based on ***"people who travelled to this destination also travelled to these other destinations"***, they can use SQL Graph and simply develop a recommendation engine for such queries.

The graph queries are not only easy to understand, but also very easy to maintain and extend to meet the requirements of an evolving application. Here are some recommendation engine examples:



- Product Recommendations
- Song Recommendations on Million song dataset

CRM

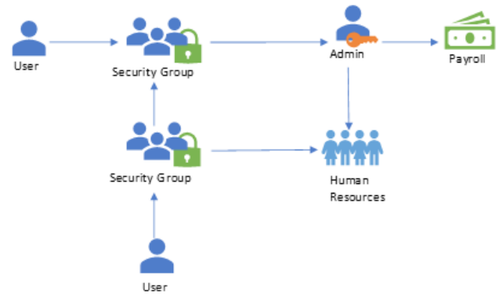
CRM software helps with managing the relationships between sales people, accounts and opportunities. The schema can be modeled as a graph to uncover complex relationships between various entities.

SQL Graph can help with analyzing highly interconnected data and understand the correlation between different entities in the database, for example, ***"which campaign was most effective in converting opportunities into accounts"***.

SQL Graph can also help the sales people find the shortest path that exists between them and another sales person in the organization who can help them run a campaign for an account that they work with.

Identity Management

Identity access management is hierarchical in nature and hierarchies can be easily implemented in a relational database. However, with the increase in complex access management rules in the modern organizations, modelling identity access management rules as a perfect tree structure is not enough.



Often times, finding answers to simple questions like, ***"find users who have access to a given security group"*** requires recursively traversing multiple levels of access rules. This is done by going deep into the multiple levels and traversing the complex connections that exist between users and access roles.

Graph databases help model this type of information more intuitively and can make deep traversal queries much easier to write.

How can I get started with SQL Graph?

Download SQL Server 2017 or start with a **Free Trial of Azure SQL Database subscription** (goo.gl/SgCzR3). Here are some resources that will help you in getting started:

- SQL Graph Overview
- SQL Graph Architecture

Other Resources

Blogs

- Graph Processing with SQL Server 2017 and Azure SQL Database
- SQL Graph – quick overview (YouTube)

Samples

- Product Recommendations
- Song Recommendations on Million song dataset
- Bulk Insert best practices

Conclusion

Graph databases are garnering a lot of interest across many industry domains.

If you are already using SQL Server to store your relational data, there is no need to turn to a new product for your graph data requirements. SQL Server 2017 and Azure SQL Database can help you solve your graph problems easily with the help of new features and T-SQL extensions.

You get the same performance, security and data durability guarantees on your data as you get with the relational data.

Shreya Verma

Author

Shreya Verma is a Program Manager on the SQL Server team. She has over 12 years of experience in the database industry, working on both SQL and NoSQL products. As a Program Manager in the SQL Server team, Shreya focusses on query processor and graph extensions in SQL Server and Azure SQL DB. Prior to this, Shreya has worked on Amazon DynamoDB and ANTs Data Server products.

Thanks to Suprotim Agarwal for reviewing this article.

Ravi Kiran

BUILDING ANGULAR APPLICATIONS USING ASP.NET CORE ANGULAR CLI TEMPLATE

ASP.NET core and Angular are created to build the next generation of web applications. When used together, they bring in a capable ecosystem to build end-to-end applications. This article will show how to build a simple CRUD application using the ASP.NET Core Angular CLI template.

Microsoft built ASP.NET core to make modern application development easier and efficient. ASP.NET core makes it possible to build and deploy .NET based server web applications across any of the major platforms available.

On the other hand, front-end web is evolving very rapidly. The tooling to build front-end applications is getting matured to meet the developers' needs. This brings up a challenge to the server platforms to support the front-end ecosystem inside their boundaries.

The ASP.NET team has built Angular and React templates to support development of front-end based applications with ASP.NET core as the backend.

The Angular team built Angular CLI to address the difficulties that developers face while setting up the environment to build a sophisticated web application. It reduces the time and effort required to generate a new Angular application. Using the commands it provides, one can generate an application, generate different code blocks of Angular, add unit test spec files, run tests, run the applications in development mode and generate build files to deploy the application.

These features are hard to recreate. Anyone would love to have these features while working on Angular. So, the ASP.NET team released a new template by including Angular CLI. This template is still in RC at the time of this writing and is expected to be ready for production in a few months.

This tutorial will show you how to create a simple CRUD based application using the ASP.NET core Angular template.

System Setup

To follow along with this article and build the sample, your system should have the following software installed:

- Visual Studio 2017
- .NET core 2.0 or above
- Node.js
- Angular CLI: To be installed as a global npm package using the following command

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

Angular CLI is installed using npm, so the system should have Node.js installed before the npm command is executed.

The ASP.NET core Angular template to be used in this article is still a release candidate and it is not available on Visual Studio's template dialog box. The template has to be installed using the following command:

```
> dotnet new --install Microsoft.DotNet.Web.Spa.ProjectTemplates::2.0.0-rc1-final
```

Once these installations are complete, you can start using the new template to build applications.

Understanding the Application Structure

We are going to build a video tracker application. Using this application, one can enter his or her favorite

set of videos with their links and can watch the video by clicking the link whenever he or she wants to watch it.

At first, we need to get this project generated using the template. Open a command prompt, go to the folder where you want to place your code and run the following command:

```
> dotnet new angular -o VideoTracker
```

This command will generate the web application containing ASP.NET core and Angular. The code gets added to a new folder named *VideoTracker*. This folder contains the *VideoTracker.csproj* file, open the project in Visual Studio 2017 using this file.

You will find the folder structure similar to the following:

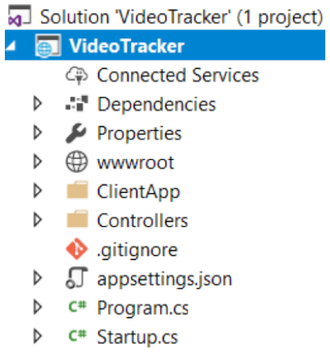


Figure 1 – Folder Structure

The following listing describes the files and folders in the above screenshot:

- *ClientApp* folder contains the Angular CLI based front end application
- *Controllers* folder contains a sample controller with an API endpoint
- The file *appsettings.json* contains some settings used by the ASP.NET core application
- The *Program.cs* file has the *Main* method to start the application
- The *Startup.cs* file sets up the services and middlewares for the ASP.NET core application

The following screenshot shows structure of the *ClientApp* folder:

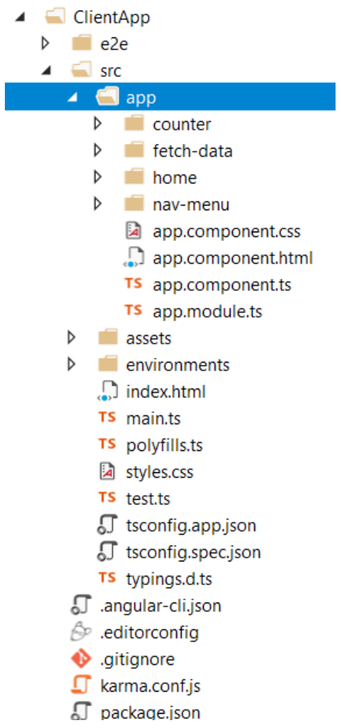


Figure 2 – Structure of ClientApp folder

If you are familiar with the files generated by Angular CLI, most of the files in Figur 2 would look familiar to you. If not, read the Angular CLI introduction article to understand the structure.

The *src* folder contains the front-end code that gets rendered on the browser. To build the Video Tracker application, we will modify the files inside the *src* folder.

The following listing describes the vital parts of the *src* folder:

- The file *main.ts* bootstraps the application
- The file *app.module.ts* defines the Angular module of the application and configures the routes. The following snippet shows the modules imported and the route configuration added to *AppModule*:

```
imports: [
  BrowserModule.withServerTransition({ appId: 'ng-cli-
  universal' }),
  HttpClientModule,
  FormsModule,
  RouterModule.forRoot([
    { path: '', component: HomeComponent, pathMatch:
'full' },
    { path: 'counter', component: CounterComponent },
    { path: 'fetch-data', component: FetchDataComponent },
  ])
]
```

- Notice the *BrowserModule* in the imports array. Call to the method *withServerTransition* is made to support server-side rendering of the Angular application. The application has three routes configured.

- The *AppComponent* in the file *app.component.ts* contains the *router-outlet* component, which is the placeholder to display the routes. It uses the *NavMenuComponent* to show the menu items that help to navigate between the routes.

- The components supplied to the routes carry out simple tasks. The *HomeComponent* has a static HTML template displaying some information about ASP.NET core, Angular, Bootstrap and the template. The *CounterComponent* has a basic data binding example. The *FetchDataComponent* makes a call to the sample API added in the project and displays the result in a table.

To run any Angular CLI commands to generate the new angular code files, you can open a command prompt in the *ClientApp* folder and run the command there. This will be covered later in this article.

The *Startup.cs* file contains the configuration to start the client application from the *ClientApp* folder. To run the application in development mode, it uses the "npm start" script configured in the client application. The "npm start" script uses the "ng serve" command in turn to start the Angular application in development mode. The following snippet in the *Configure* method does this:

```
app.UseSpa(spa =>
{
  // To learn more about options for serving an Angular SPA from ASP.NET Core,
  // see https://go.microsoft.com/fwlink/?linkid=864501

  spa.Options.SourcePath = "ClientApp";

  if (env.IsDevelopment())
  {
    spa.UseAngularCliServer(npmScript: "start");
  }
});
```

While running the application in production mode, it takes the static files from *ClientApp/dist* folder. This is because the Angular CLI spits out the bundled static files inside this folder. The following snippet in the *ConfigureServices* method of the *Startup* class does this:

```
// In production, the Angular files will be served from this directory
services.AddSpaStaticFiles(configuration =>
{
  configuration.RootPath = "ClientApp/dist";
});
```

Build the project using Build menu or by pressing Ctrl+Shift+B. This will take a couple of minutes, as it is going to install the npm packages added to the Angular application. Once the build completes, run it using Ctrl+F5. You will see a bootstrap based Angular application loaded in the browser.

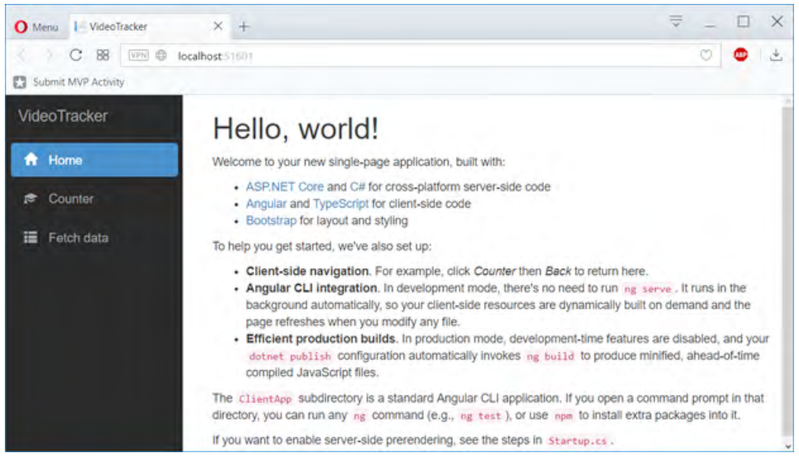


Figure 3 – Running instance of generated application

As you see, the generated application has three pages. They are registered as Angular routes. The Counter page shows how data binding works in Angular with a number incrementing on click of a button. And the Fetch Data page calls the sample API in the application to fetch some data and displays it in a table.

Adding Database and Building API

Now that we understood the structure and the way the application works, let's modify it to build the video tracker. Let's get the database and the API ready to serve the data. We will create the database using Entity Framework Core.

Add a folder named *Model* to the project. Add a new C# class file to this folder and name it *Video.cs*. Change contents of the file as follows:

```
namespace VideoTracker.Model
{
  public class Video
  {
    public int ID { get; set; }
    public string Title { get; set; }
    public string Link { get; set; }
    public bool Watched { get; set; }
    public string Genre { get; set; }
  }
}
```

Add another file to the *Model* folder and name it *VideoTrackerContext*. This class will be the DbContext class for the database we are going to create. Open this file and replace it with the following code:

```
using Microsoft.EntityFrameworkCore;

namespace VideoTracker.Model
{
  public class VideoTrackerContext: DbContext
  {
    public VideoTrackerContext(DbContextOptions<VideoTrackerContext> options)
      :base(options)
    {
    }

  public DbSet<Video> Videos { get; set; }
  }
}
```

Open the file *appsettings.json* and add the following connection string to it:

```
"ConnectionStrings": {
  "VideoTrackerContext": "Server=(localdb)\\
mssqllocaldb;Database=VideoTrackerDb;Trusted_
Connection=True;MultipleActiveResultSets=true"
}
```

The DbContext class has to be registered in the services of the application. To do so, add the following snippet to the *ConfigureServices* method of the *Startup* class:

```
services.AddDbContext<VideoTrackerContext>(options =>
  options.UseSqlServer(Configuration.GetConnectionString("VideoTrackerContext")));
```

Now we have the connection string, model class and the context class ready.

To create the database and seed some data into it, we need to add EF Core migrations. Migrations need a context factory class. Add a new C# class to the *Model* folder and name it *VideoContextFactory.cs*. Place the following code inside this file:

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Design;
using Microsoft.Extensions.Configuration;
using System.IO;

namespace VideoTracker.Model
{
 public class VideoContextFactory : IDesignTimeDbContextFactory<VideoTrackerContext>
 {
  public VideoTrackerContext CreateDbContext(string[] args)
  {
   IConfigurationRoot configuration = new ConfigurationBuilder()
    .SetBasePath(Directory.GetCurrentDirectory())
    .AddJsonFile("appsettings.json")
    .Build();
   var optionsBuilder = new DbContextOptionsBuilder<VideoTrackerContext>();
    optionsBuilder.UseSqlServer(configuration.
GetConnectionString("VideoTrackerContext"));

   return new VideoTrackerContext(optionsBuilder.Options);
  }
 }
}
```

To access the database created, we need *Microsoft.EntityFrameworkCore.Tools* package and need to run migration commands in the package manager console. Open the Package Manager Console in Visual Studio and run the following commands in the given order:

```
> Install-Package Microsoft.EntityFrameworkCore.Tools
> Add-Migration Initial
> Update-Database
```

Once these commands are successful, you will have access to the database and the table created. You can see the database in the SQL Server Object Explorer. It will have the Videos table created with the five columns as specified in the *Video* class earlier.

Let's add some data to this table, so that we can see the data in response to the APIs.

Add a new file to the *Model* folder and name it *Seed.cs*. Add the following code to it:

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using System;
using System.Linq;

namespace VideoTracker.Model
{
 public class Seed
 {
  public static void Initialize(IServiceProvider serviceProvider)
  {
   using (var context = new VideoTrackerContext(
    serviceProvider.GetRequiredService<DbContextOptions<VideoTrackerContext>>()))
    {
     // Look for any movies.
     if (context.Videos.Any())
     {
      return;   // DB has been seeded
     }

     context.Videos.AddRange(
new Video
{
 Title = "Raghuvamsa Sudha",
 Link = "https://www.youtube.com/watch?v=l89ZaBqYiTA",
 Genre = "Music",
 Watched= true
},

new Video
{
 Title = "Kapil finds a match for Sarla",
 Link = "https://www.youtube.com/watch?v=GfHEPKgkkpw",
 Genre = "Comedy",
 Watched = true
},

new Video
{
 Title = "Arvind Gupta TED Talk",
 Link = "https://www.youtube.com/watch?v=KnCqR2yUXoU",
 Genre = "Inspirational",
 Watched = true
},

new Video
{
 Title = "Event Loop - Philip Roberts",
 Link = "https://www.youtube.com/watch?v=8aGhZQkoFbQ",
 Genre = "Tech",
 Watched = true
     }
     );
    context.SaveChanges();
   }
  }
 }
}
```

The *Initialize* method of the *Seed* class has to be called as soon as the application starts. Modify the *Main* method in the *Program.cs* file as follows:

```
public static void Main(string[] args)
{
  // Creating a host based on the Startup class
  var host = CreateWebHostBuilder(args).Build();

  // Creating a scope for the application
  using (var scope = host.Services.CreateScope())
  {
    // Getting service provider to resolve dependencies
    var services = scope.ServiceProvider;

    try
    {
      // Initializing the database
      Seed.Initialize(services);
    }
    catch (Exception ex)
    {
      var logger = services.GetRequiredService<ILogger<Program>>();
      logger.LogError(ex, "An error occurred seeding the DB.");
    }
  }
  host.Run();
}
```

If you run the application after saving the above changes, the data would be inserted to the Videos table.

As the database is ready with some data, let's add the API. Right click on the *Controllers* folder and choose `Add > Controller`. From the menu, choose API Controller with actions using Entity Framework.

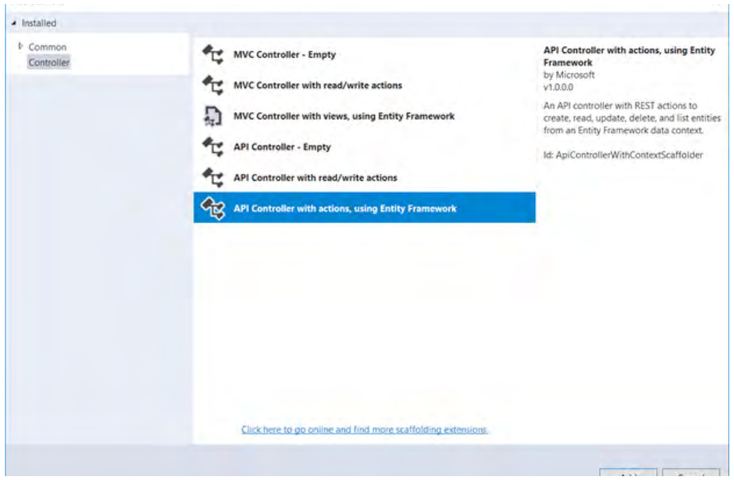


Figure 4 – Add control dialog

Click on the add button after choosing the option. This will show another prompt asking for the details of the model and the context class. Change the values as follows:

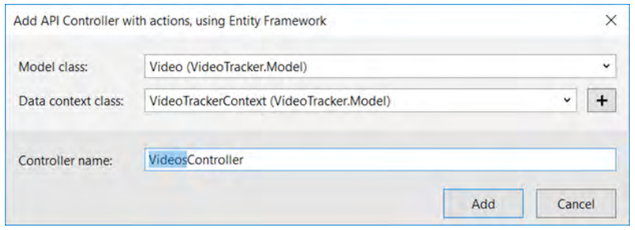


Figure 5 – Controller with actions dialog

Once you hit the add button, it will create the *ValuesController* performing CRUD operations on the *Videos* table. We don't need to make any changes to this file, we can use it as is. Run the application and change the URL in the browser to http://localhost:<port-no>/api/Videos. This will display the set of videos added to the table. Figure 6 shows this:


```
(4)[
- {
    id: 1,
    title: "Raghuvamsa Sudha",
    link: https://www.youtube.com/watch?v=189ZaBqYiTA,
    watched: true,
    genre: "Music"
  },
- {
    id: 2,
    title: "Kapil finds a match for Sarla",
    link: https://www.youtube.com/watch?v=GfHEPKgkkpw,
    watched: false,
    genre: "Comedy"
  },
- {
    id: 3,
    title: "Arvind Gupta TED Talk",
    link: https://www.youtube.com/watch?v=KnCqR2yUXoU,
    watched: false,
    genre: "Inspirational"
  },
- {
    id: 4,
    title: "Event Loop - Philip Roberts",
    link: https://www.youtube.com/watch?v=8aGhZQkoFbQ,
    watched: false,
    genre: "Tech"
  },
]
```


Figure 6 – Output of the videos API

Building Video Tracker Client App

As stated earlier, we need to modify the *ClientApp/src* folder to build the client part of the application. The application needs two views, one to display the list of videos and the other to add a new video. The Angular components required for these views can be generated using the Angular CLI commands. Open a command prompt and move to the *ClientApp* folder. Run the following commands to get the components generated:

```
> ng generate component video-list
> ng generate component add-video
```

Angular CLI provides shorter versions of these commands to ease the usage. The command to generate the *video-list* component can be replaced with the following:

```
> ng g c video-list
```

These commands add components with their basic skeleton and register them in the *AppModule*. These components have to be registered as routes, so that we can access them as Angular views.

As we will be using these components going forward, it is better to remove the components that were added by the template. Open the file *app.module.ts* and change the code in the file as shown in the following snippet:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule, InjectionToken } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpClientModule } from '@angular/common/http';
import { RouterModule } from '@angular/router';

import { AppComponent } from './app.component';
import { NavMenuComponent } from './nav-menu/nav-menu.component';
import { VideoListComponent } from './video-list/video-list.component';
import { AddVideoComponent } from './add-video/add-video.component';

@NgModule({
  declarations: [
    AppComponent,
    NavMenuComponent,
    VideoListComponent,
    AddVideoComponent
  ],
  imports: [
    BrowserModule.withServerTransition({ appId: 'ng-cli-universal' }),
    HttpClientModule,
    FormsModule,
    RouterModule.forRoot([
      { path: '', component: VideoListComponent, pathMatch: 'full' },
      { path: 'addvideo', component: AddVideoComponent }
    ])
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Now this file is free from the components added by the template and it registers the new components in the routes.

The navigation menu still points to the old routes. Let's change it to use the new routes there. Open the file *nav-menu.component.html* and change the code of this file as follows:

```
<div class='main-nav'>
 <div class='navbar navbar-inverse'>
  <div class='navbar-header'>
   <button type='button' class='navbar-toggle' data-toggle='collapse' data-
   target='.navbar-collapse' [attr.aria-expanded]='isExpanded' (click)='toggle()'>
    <span class='sr-only'>Toggle navigation</span>
    <span class='icon-bar'></span>
    <span class='icon-bar'></span>
    <span class='icon-bar'></span>
```

```
    </button>
     <a class='navbar-brand' [routerLink]='["/"]'>VideoTracker</a>
   </div>
   <div class='clearfix'></div>
    <div class='navbar-collapse collapse' [ngClass]='{ "in": isExpanded }'>
     <ul class='nav navbar-nav'>
      <li [routerLinkActive]='["link-active"]' [routerLinkActiveOptions]='{ exact:
      true }'>
       <a [routerLink]='["/"]' (click)='collapse()'>
        <span class='glyphicon glyphicon-home'></span> Video List
       </a>
      </li>
      <li [routerLinkActive]='["link-active"]'>
       <a [routerLink]='["/addvideo"]' (click)='collapse()'>
        <span class='glyphicon glyphicon-plus'></span> Add Video
       </a>
      </li>
     </ul>
    </div>
  </div>
</div>
```

Save these changes and run the application. You will see the new menu items in the menu bar as shown in Figure 7:

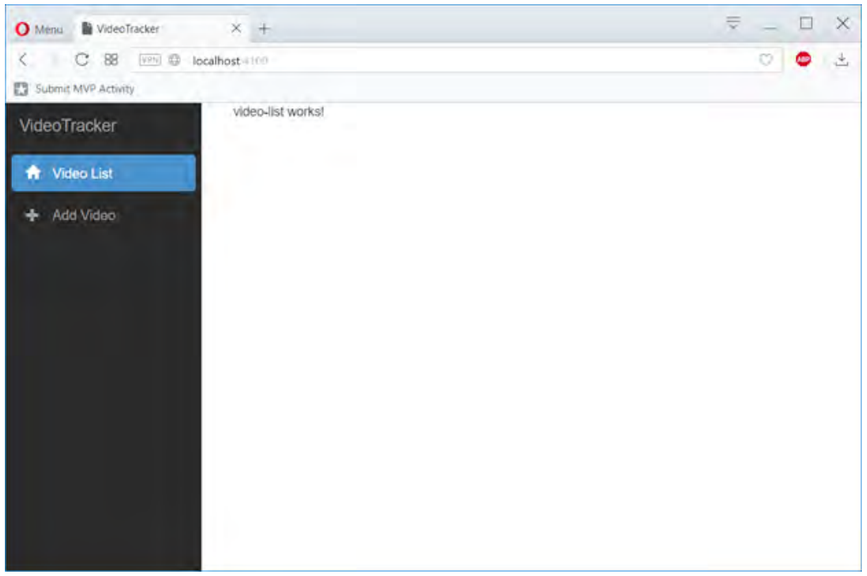


Figure 7 – Application with new menu items

Before modifying the components to show the new views, let's create a service to interact with the backend. The reason behind creating a service to perform the data operations is to keep the components focused on building data required for its view alone. Also, as the service would be available to all the components in the application, anyone can use it without having to repeat the logic. Run the following command to generate the service:

```
> ng g s videos -m app.module
```

This command adds the *VideosService* and registers in the *AppModule*. This service will interact with the web APIs created in the *VideosController*. Open the newly added file *videos.service.ts* and modify the code in this file as shown below:

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from "rxjs/Observable";
import { Video } from "./video";

@Injectable()
export class VideosService {

  constructor(private httpClient: HttpClient) { }

  // Getting the list of videos
  getVideos(): Observable<Video[]> {
    return this.httpClient.get<Video[]>('/api/Videos');
  }

  // Adding a new video by calling post
  addVideo(video: Video) {
    return this.httpClient.post('/api/Videos',video);
  }

  // Marking a video as watched using the PUT API
  setWatched(video: Video) {
    video.watched = true;

    return this.httpClient.put(`/api/Videos/${video.id}`, video);
  }
}
```

The *videos-list* component will use the *getVideos* method to fetch the videos and show them in a table. It needs to update the *watched* property of the video when link of a video is clicked. It is better to write a method in the component to perform both these tasks. To open the link programmatically, the *window* object has to be used.

The application will be hard to test if the *window* object is used from global scope. Even if we are not going to use unit testing in this article, it is always good to use a better way to access the global objects. For this, we need to create a token using *InjectionToken* to register the object and inject it using the token. Add a new file to the *src* folder and name it *window.ts*. Add the following code to it:

```
import { InjectionToken } from '@angular/core';

const WINDOW = new InjectionToken<any>('window');

export const windowProvider = { provide: WINDOW, useValue: window };
```

The object *windowProvider* can be used to register the injectable. Open the file *app.module.ts* and import the *windowProvider* object.

```
import { windowProvider } from './window';
```

Modify the *providers* array of the module metadata as shown in the following snippet to register the token:

```
providers: [
  VideosService,
  windowProvider
]
```

Now we have all the objects ready to be used in the components. Open the file *video-list.component.ts* and modify the code as shown below:

```
import { Component, OnInit, Inject } from ‘@angular/core’;

import { VideosService } from ‘../videos.service’;
import { Video } from “../video”;
import { windowProvider } from ‘../window’;

@Component({
  selector: ‘app-video-list’,
  templateUrl: ‘./video-list.component.html’,
  styleUrls: [‘./video-list.component.css’]
})
export class VideoListComponent implements OnInit {
  public videos: Video[];

  constructor(private videosService: VideosService,
    @Inject(windowProvider.provide) private window: Window) { }

  ngOnInit() {
    this.videosService.getVideos()
      .subscribe(videos => {
        this.videos = videos;
      });
  }

  watch(video: Video) {
    this.videosService.setWatched(video)
      .subscribe(v => console.log(v));
    this.window.open(video.link, ‘_blank’);
  }
}
```

Notice the way the *window* object is injected in the above component. It Is injected using the *Inject* decorator and the token is passed into the decorator.

The template has to be modified to display the data in the table. Place the following code in the file *video-list.component.html*:

```
<table class=”table”>
 <thead>
  <tr>
   <th>Title</th>
   <th>Genre</th>
   <th>Watched</th>
   <th>Watch</th>
  </tr>
 </thead>
 <tbody>
  <tr *ngFor=”let video of videos”>
   <td>{{video.title}}</td>
   <td>{{video.genre}}</td>
   <td>
    <span *ngIf=”video.watched”>Yes</span>
    <span *ngIf=”!video.watched”>No</span>
   </td>
   <td>
    <a href=”{{video.link}}” target=”_blank” (click)=”watch(video)”>Watch Video</a>
   </td>
  </tr>
 </tbody>
</table>
```

Save these files and run the application. Figure 8 shows the home page displaying a list of videos.

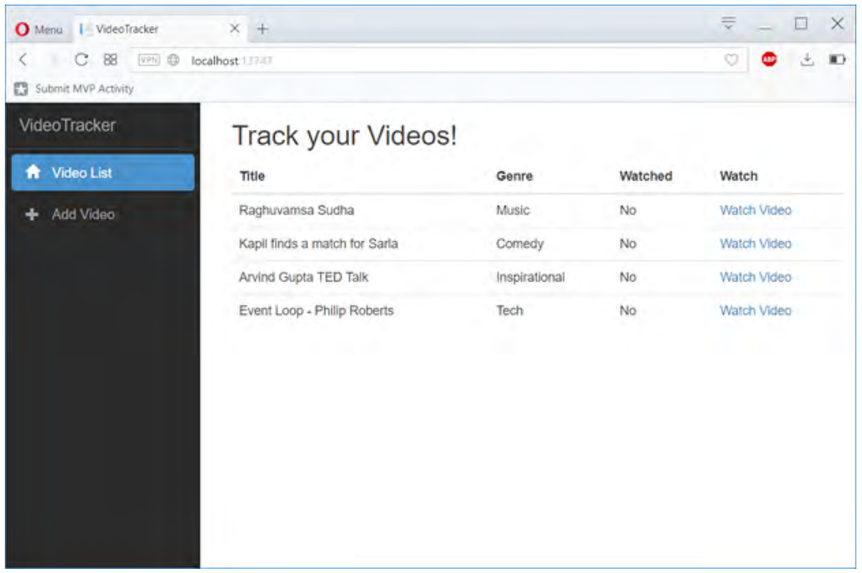


Figure 8 – Video tracker with videos list

Let’s get the *add-video* component ready to add new videos to the application. Open the file *add-video.component.ts* and replace it with the following code:

```
import { Component, OnInit } from ‘@angular/core’;
import { Router } from ‘@angular/router’;

import { VideosService } from ‘../videos.service’;
import { Video } from “../video”;

@Component({
 selector: ‘app-add-video’,
 templateUrl: ‘./add-video.component.html’,
 styleUrls: [‘./add-video.component.css’]
})
export class AddVideoComponent {
 public genres: string[];
 public video: Video = {};

 constructor(
  private router: Router,
  private videosSvc: VideosService) {
  this.genres = [
   “”,
   “Music”,
   “Comedy”,
   “Inspirational”,
   “Tech”,
```

```
    "News",
    "Sports"
  ];
 }

 addVideo() {
  this.videosSvc.addVideo(this.video)
   // The subscribe method here adds a listener to the asynchronous add
   // operation and the listener is called once the video is added.
   // Here, we are switching to the video list page after adding the video
   .subscribe(() => {
   this.router.navigateByUrl('/');
  });
 }
}
```

The last thing to do is to modify the template of the *add-video* component to add a form. The form will have basic required field validations to ensure that the fields have values when the form is submitted.

Place the following code in the file *add-video.component.html:[.[.πl-ºp[*

```
<div>
 <form #videoForm="ngForm" (ngSubmit)="addVideo()">
  <div class="form-group">
   <label for="name">Title</label>
    <input type="text" class="form-control"
     name="title"
     required
     [(ngModel)]="video.title"
     #title="ngModel">
  </div>

  <div class="form-group">
   <label for="link">Link</label>
    <input type="text" class="form-control"
     name="link"
     required
     [(ngModel)]="video.link"
     #link="ngModel">
  </div>

  <div class="form-group">
   <label for="genre">Genre</label>
    <select class="form-control"
     name="genre"
     required
     [(ngModel)]="video.genre"
     #genre="ngModel">
     <option *ngFor="let genre of genres" [value]="genre">{{genre}}</option>
    </select>
  </div>

  <button type="submit" class="btn btn-success"
   [disabled]="!videoForm.form.valid">Submit</button>
  <button type="reset" class="btn"
   (click)="videoForm.reset()">Reset</button>
 </form>
</div>
```

Now the application is complete. Run it and you can add a new video. And then you can see it in the video list page.

Conclusion

As we saw, ASP.NET core now has good support for Angular development using Angular CLI. The template doesn't try to hide any of the features of Angular CLI and provides a good support for building and deploying Angular applications with ASP.NET core. We can use this template to build and deploy dynamic web applications using the goodness of both the technologies.

Download the entire source code from GitHub at
bit.ly/dncm35-aspcorecli

Microsoft MVP Most Valuable Professional

Ravi Kiran
Author

Ravi Kiran (a.k.a. Ravi Kiran) is a developer working on Microsoft Technologies at Hyderabad. These days, he is spending his time on JavaScript frameworks like AngularJS, latest updates to JavaScript in ES6 and ES7, Web Components, Node.js and also on several Microsoft technologies including ASP.NET 5, SignalR and C#. He is an active ***blogger****, an author at SitePoint and at DotNetCurry. He is rewarded with Microsoft MVP (ASP.NET/IIS) and DZone MVB awards for his contribution to the community.*

Thanks to Keerti Kotaru for reviewing this article.

Daniel Jimenez Garcia

Testing ASP.NET Core Applications

Part III: End-to-End (E2E) Tests

This is the third entry in a series of articles taking a look at all the different strategies for testing web applications in general and ASP.NET Core applications in particular.

A testing strategy comprising unit, integration and E2E tests was introduced, with earlier articles covering both unit (bit.ly/dnc-aspcore-unit) and integration testing (bit.ly/dnc-aspcore-integration). In this article, we will now take a look at E2E (end to end) tests. The source code is available on its GitHub project.

Logo Courtsey: docs.microsoft.com

The role of End-to-End (E2E) tests

In the previous articles, we discussed the idea of defining a testing strategy involving unit, integration and E2E tests that roughly followed a pyramidal structure with unit tests at the bottom and end-to-end tests at the top:

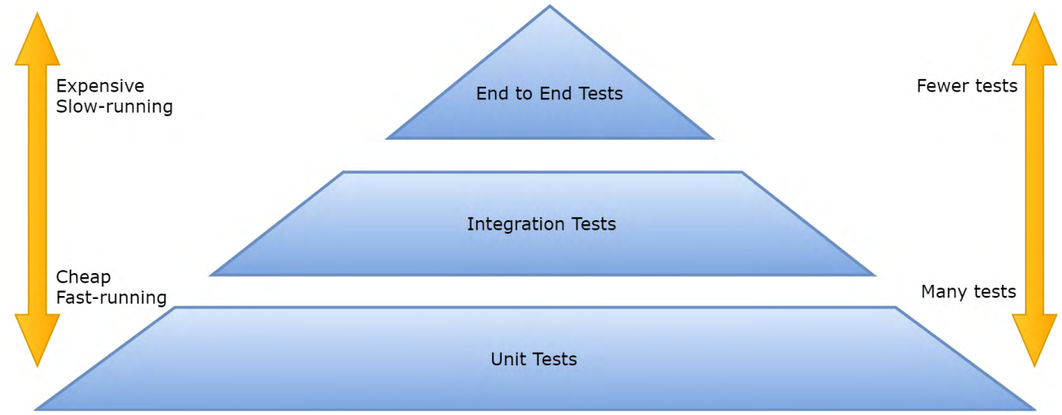


Figure 1, the testing pyramid

By their very nature, E2E tests are the closest to how a user interacts with the application. This makes them an expensive but powerful tool:

- They are expensive to write and maintain as they become tightly coupled with the user interface. Since the UX is one of the more volatile aspects of the application, these tests are under constant danger of being broken by UX changes.
- They involve a real browser running against your application, so they are the slowest tests in your strategy. They also depend on UX behavior which means they often have to wait for actions to complete or the interface to update
- However, they are also very valuable as we get feedback closer to what a user might experience when using our application. At the end of the day, our application is intended to be used by our users and we need to make sure things are working from their perspective.

Given these characteristics, we need to strike the proper balance between cost and value when adding E2E tests. Having no such tests would be a risk but having too many of them would be very expensive! (In some cases you can ignore E2E tests altogether, for example applications without UX that just publish an API for other systems to integrate with).

Let's ensure we have proper testing coverage for our business logic, corner cases and component interactions with unit and integration tests.

We can then use the E2E tests to ensure the most important actions users would take from their perspective. Applied to our running example of the BlogPlayground application, we can use E2E tests to ensure:

- Users can login and logout

- Users can see the list of articles
- Users can open an article
- Users can add an article
- Users can remove an article

Basically, we are smoke testing the basic functionality of the application in its most simple form, without worrying much about corner cases or business rules.

Selenium and the Web Driver API

Selenium is a tool that lets you automate browsers whereas the Web Driver API is a standard for automating browsers. It so happens that Selenium implements the Web Driver API and can abstract your tests from many different target browsers.

While these tools can be used to automate any tasks which usually involve a human interacting with a browser, they are most widely used for testing web applications.

When using Selenium, you need to stand up a Selenium server which connects to a browser via a web driver. Your test interacts with the selenium server through the JSON Web Driver API, which in turn controls the browser through a browser-specific driver that also implements the Web Driver API.

In order to write tests, you can use pretty much any language since Selenium has client libraries available for most languages like C#, JavaScript, Python, Ruby or Java.

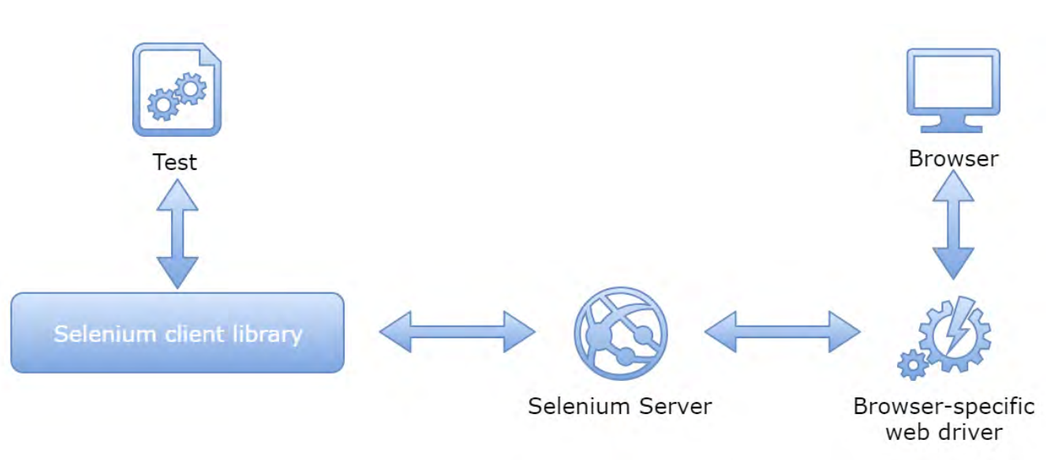


Figure 2, Selenium based tests

While your tests could directly use the browser-specific web driver, introducing the Selenium server in the middle is a powerful abstraction that makes your tests independent of the target browser and environment.

You can write your test once using the Selenium client library and run against any target environments such as different desktop browsers on different operating systems or even browsers in mobile devices, simply letting the Selenium server know where to find the appropriate drivers and which one to use during that particular test run.

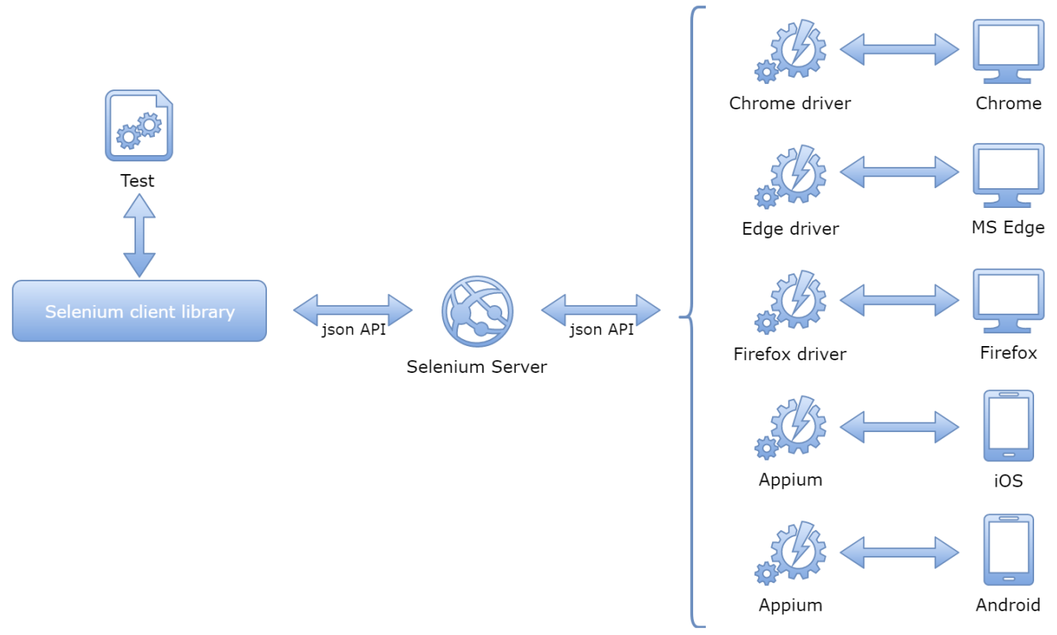


Figure 3, Selenium drivers allow the same test to run against different environments

It is now so common to use Web Driver based tests for verifying your application that there are many cloud services that let you run your tests against many different browsers/devices, both simulated or on real hardware. These include platforms like BrowserStack, Sauce Labs or CrossBrowserTesting just to name a few.

In this article we will use Nightwatch.js, a framework that lets you write Selenium Web Driver tests using a JavaScript library and we will see how to run these tests from the command line (either as a developer or from a CI server).

Preparing the application for E2E tests

In order to make any automated test stable, we need to keep as many factors under control as possible. Amongst other things, this means being able to control the state of the system our tests will run against. For example, if we want our tests to be able to login, there needs to be an account created with the right username and password. We might also want to make sure test data shows up on different pages for which we would need to seed the system with some predefined test data.

This greatly depends on how we are going to manage the server, our E2E tests run against:

- We can deploy a test environment (separated from our development/production environments) which is used for the sole purpose of E2E testing, with tests directing the browsers to the URL of said environment. Our CI/testing process needs to make sure the state of the system is the right one before the tests can run, using methods like seeding scripts, tests API and/or backups.

- We can stand up a new environment from scratch, run our tests against it and shut it down at the end. We can include seeding scripts as part of the startup process and we don't need to worry about any previous state nor cleaning afterwards as the whole environment is shutdown.

In this article, I will follow the second option as it is very similar to what we did for the integration tests. In fact, we can reuse the same predefined data and seeding script. We will basically use a Kestrel server with a SQLite in-memory database, which will be started at the beginning of the test run and shutdown at the end.

Note: *There are very little changes in terms of how the tests are written or executed when following a different option. Your tests would just use a different URL and you would come up with a different set of steps/scripts/ processes to ensure the state of the system is the desired one.*

Let's start by adding a new empty ASP.NET Core project to our solution, named **BlogPlayground.E2ETest**. Add a reference to the existing **BlogPlayground** and **BlogPlayground.IntegrationTest** projects.

Now we can replace the predefined `Startup` class with one that basically extends the regular Startup class by setting up a SQLite in-memory database and running the database seeder script that inserts the predefined data:

```
public class TestStartup: BlogPlayground.Startup
{
  public TestStartup(IConfiguration configuration) : base(configuration)
  {
  }
  public override void ConfigureDatabase(IServiceCollection services)
  {
    // Replace default database connection with SQLite in memory
    var connectionStringBuilder = new SqliteConnectionStringBuilder { DataSource =
    ":memory:" };

    var connection = new SqliteConnection(connectionStringBuilder.ToString());
    services.AddDbContext<ApplicationDbContext>(options => options.
    UseSqlite(connection));

    // Register the database seeder
    services.AddTransient<DatabaseSeeder>();
  }

  public override void Configure(IApplicationBuilder app, IHostingEnvironment env,
  ILoggerFactory loggerFactory)
  {
    // Perform all the configuration in the base class
    base.Configure(app, env, loggerFactory);

    // Now create and seed the database
    using (var serviceScope = app.ApplicationServices.
    GetRequiredService<IServiceScopeFactory>().CreateScope())
    using (var dbContext = serviceScope.ServiceProvider.
    GetService<ApplicationDbContext>())
    {
      dbContext.Database.OpenConnection();
      dbContext.Database.EnsureCreated();

      var seeder = serviceScope.ServiceProvider.GetService<DatabaseSeeder>();
      seeder.Seed().Wait();
    }
  }
}
```

Next, we need to update the predefined `Program` to launch a Kestrel server using the `TestStartup` class we have just created. This also needs to include the `UseContentRoot` workaround that allows the Razor views declared on the BlogPlayground project to render correctly when launching this server:

```
public class Program
{
  public static void Main(string[] args)
  {
    BuildWebHost(args).Run();
  }

  public static IWebHost BuildWebHost(string[] args)
  {
    // To avoid hardcoding path to project, see: https://docs.microsoft.com/en-us/
    aspnet/core/mvc/controllers/testing#integration-testing
    var integrationTestsPath = PlatformServices.Default.Application.
    ApplicationBasePath; // e2e_tests/bin/Debug/netcoreapp2.0
    var applicationPath = Path.GetFullPath(Path.Combine(integrationTestsPath,
    "../../../../BlogPlayground"));

    return WebHost.CreateDefaultBuilder(args)
     .UseStartup<TestStartup>()
     .UseContentRoot(applicationPath)
     .UseEnvironment("Development")
     .Build();
  }
}
```

This process should result very similar to the one we followed for standing up a server, our integration tests could run against. The only difference is that now we will be running a real Kestrel server in its own process (instead of the TestServer) and a SQLite database (instead of the limited EF in-memory one).

Once compiled, you should be able to run this project from either Visual Studio or the command line with `dotnet run`. Double check you can `cd` into the new project folder and run it from the command line as this is exactly what we will do at the beginning of our test run!

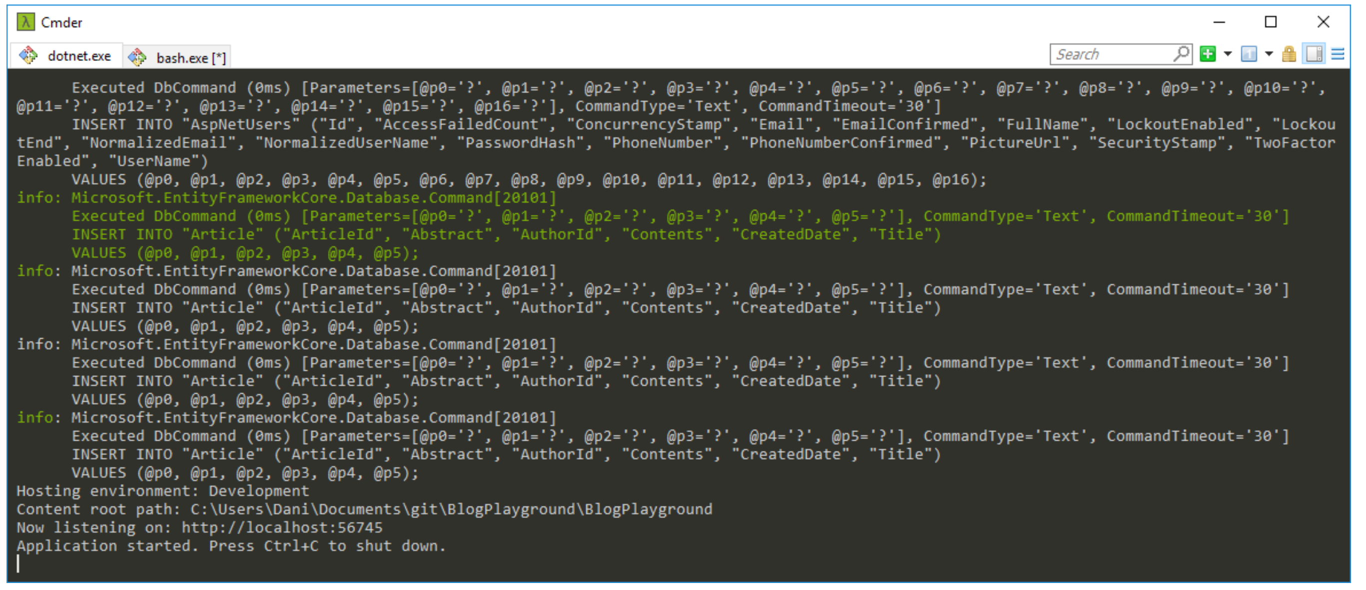


Figure 4, making sure the test server starts with dotnet run

Writing and running Selenium tests with Nightwatch.js

Now we have everything ready to start writing our Selenium-based tests.

I have chosen Nightwatch.js, a popular JavaScript framework that allows you to write your tests using a nicer API, as compared to barebones Selenium.

Note: *Nightwatch allows you to have access to the underlying Selenium API when needed.*

There are several reasons why I have ended up using Nightwatch.js and JavaScript for this task:

- Its dynamic nature makes it easier to write and maintain tests that depend on something as volatile as the UX, while at the same time keeping your test code clean with custom commands, assertions and page objects.
- Nightwatch syntax makes it really well suited for browser-based tests, cleanly managing CSS/XPath selectors and asynchronous actions.
- Nightwatch is able to start/stop the Selenium process if required. It is also easy to integrate with cloud providers like Sauce Labs or BrowserStack.
- It is closer to the code used by the UX itself which means no context switching between writing frontend and test code. It also makes it a breeze to write JavaScript code that can be run in the browser as part of your test.
- The scripting capabilities of Node make it easier to automate pre-processing steps like standing up the server while ensuring it runs from the command line and can be easily added to any CI process.

You could argue that writing XUnit tests in C# that use the Selenium library is equally valid, and if that works better for you and your team, please use that by all means. The underlying principles are the same.

Use the tools that work for you as long as you reach the same test coverage and are satisfied with the code quality and maintainability of your tests.

Prerequisites to run Selenium based tests

In order to run Selenium based tests, we need to run a selenium server, which is written in Java. This means we need to install Java on our development environment (and the CI server which will run the tests unless you use a cloud service).

Since Nightwatch.js is a JavaScript framework, we will also need to install Node.js on the same machines.

Once installed, let's start by adding a **package.json** file to our BlogPlayground.E2ETest project. This is the file that Node's npm uses to manage dependencies and define scripts. You can use VS "Add New Item..." command for that, or run the `npm init` command from the project folder.

Next, let's add the required Node dependencies for Nightwatch.js and Selenium by running the following command on the project folder:

```
npm install --save-dev nightwatch selenium-server-standalone-jar
```

We will be running the tests against Chrome and Edge, which means we are going to need the web drivers for these two browsers.

Add a new folder named **selenium-drivers** to the project and download the latest versions of the drivers (assuming you have the latest versions of the browsers installed). The binaries can be downloaded from their official pages:

- Chrome driver
- Edge driver

Now we need to add a configuration file that lets Nightwatch know where to look for test files, where to find the Selenium server, which browsers to use, and so on. Although you can read more about the configuration of Nightwatch on its documentation, I have annotated the code so it should be easy to follow:

```
const seleniumJar = require('selenium-server-standalone-jar');
const settings = {
 // Nightwatch global settings
 src_folders: ['./tests'],
 output_folder: '.test-results/',

 // Nightwatch extension components (commands, pages, assertions, global hooks)
 globals_path: './globals.js',
 //custom_commands_path: './commands',
 //page_objects_path: './pages',
 //custom_assertions_path: './assertions'

 // Selenium settings
 selenium: {
  start_process: true,
  server_path: seleniumJar.path,
  start_session: true,
  log_path: '.test-results/',
  port: process.env.SELENIUM_PORT || 4444,
  host: process.env.SELENIUM_HOST || '127.0.0.1',
  cli_args: {
   'webdriver.edge.driver': './selenium-drivers/MicrosoftWebDriver.exe',
   'webdriver.chrome.driver': ''
  }
},

test_settings: {
 default: {
  // Nightwatch test specific settings
  launch_url: process.env.LAUNCH_URL || 'http://localhost:56745',
  selenium_port: process.env.SELENIUM_PORT || 4444,
  selenium_host: process.env.SELENIUM_HOST || 'localhost',
  screenshots: {
  enabled: true,
  on_failure: true,
  on_error: true,
  path: '.test-results/screenshots'
 },
 // browser related settings. To be defined on each specific browser section
```

```
  desiredCapabilities: {
  },
  // user defined settings
  globals: {
   window_size: {
   width: 1280,
   height: 1024
  },
  start_app: process.env.LAUNCH_URL ? false : true,
  login_url: (process.env.LAUNCH_URL || 'http://localhost:56745') + '/Account/
  Login',
  user_info: {
   email: 'tester@test.com',
   password: '!Covfefe123',
  },
  navigation_timeout: 5000,
  initial_load_timeout: 10000
 }
},
// Define an environment per each of the target browsers
chrome: {
 desiredCapabilities: {
  browserName: 'chrome',
  javascriptEnabled: true,
  acceptSslCerts: true,
  chromeOptions: {
   args: ['--headless'],
   }
  },
 },
 edge: {
  desiredCapabilities: {
   browserName: 'MicrosoftEdge',
   javascriptEnabled: true,
   acceptSslCerts: true
   }
  }
 }
};

//make output folder available in code
//so we can redirect the dotnet server output to a log file there
settings.test_settings.default.globals.output_folder = settings.output_folder;

//Set path to chromedriver depending on host OS
var os = process.platform;
if (/^win/.test(os)) {
  settings.selenium.cli_args['webdriver.chrome.driver'] = './selenium-drivers/
chromedriver-2.35-win.exe';
} else if (/^darwin/.test(os)) {
  settings.selenium.cli_args['webdriver.chrome.driver'] = './selenium-drivers/
chromedriver-2.35-mac.exe';
} else {
  settings.selenium.cli_args['webdriver.chrome.driver'] = './selenium-drivers/
chromedriver-2.35-linux.exe';
}

module.exports = settings;
```

There are a few points worth highlighting:

- In the global setting section, you can see the path to a folder where we can define custom commands, assertions and pages. These should exist, so we will uncomment them once we add one of those.
- A number of settings have a default value that can be overridden by setting environment variables, more specifically the LAUNCH_URL, SELENIUM_HOST and SELENIUM_PORT. This allows the tests to be dockerized and even target different servers from a CI process by setting these ENV variables.
- The section annotated as "user defined settings" lets you define any data you later want to use in any of your tests, for example I have added there the credentials of one of the test users that our seeding scripts will create.
- The `--headless` argument for Chrome allows it to run in headless mode. You won't see any new Chrome window appearing in your system while the test runs, which is desirable on any kind of CI server. If you are curious and want to see the Chrome window while running, comment that line!

Starting the server as part of the test run

Now we need to add another file called **globals.js** where we can add global test hooks which are functions that run:

- Before and after the entire test run
- Before and after each test

This file is conveniently referenced from the configuration file as the `globals_path` setting and has the following rough structure (before we start adding our code):

```
module.exports = {
  // External before hook is ran at the beginning of the tests run, before creating
  the Selenium session
  before: function (done) {
   done();
  },

  // External after hook is ran at the very end of the tests run, after closing the
  Selenium session
  after: function (done) {
   done();
  },

  // This will be run before each test file is started
  beforeEach: function (browser, done) {
   done();
  },

  // This will be run after each test file is finished
  afterEach: function (browser, done) {
  //close the browser window then signal we are done
  browser
   .end()
   .perform(() => done());
  }
};
```

Basically, what we need is to run the `dotnet run` command before the test run, and to kill that process at the end of the test run. We will use Node's `childProcess` module, in concrete its spawn method, piping its output to a log file:

```
const childProcess = require('child_process');
const path = require('path');
const fs = require('fs');

let dotnetWebServer = null;
let dotnetWebServerStarted = false;

function startWebApplication(outputFolder, done) {
 const logFile = path.join(outputFolder, 'server.log');
 console.log(`Starting web application. Log found at: ${logFile}`);

 // Start web app in separated process.
 dotnetWebServer = childProcess.spawn("dotnet", ["run"]);

 // Fail test run startup if the server dies before it got properly started
 dotnetWebServer.on('close', (code) => {
  if (code !== 0 && !dotnetWebServerStarted) {
   console.error(`Could not start dotnet server. Exited with code ${code}. Check
   log at ${logFile}`);
   process.exit(-1);
    }
  });

  // Do not start the test until we see the "Application started" message from
  dotnet
  dotnetWebServer.stdout.on('data', (chunk) => {
   if (chunk.toString().includes("Application started")) {
    dotnetWebServerStarted = true;
    done();
   }
  });

  // Redirect the standard output of the web application to a log file
  const appLogPath = path.join(__dirname, logFile);
  const childServerLog = fs.createWriteStream(appLogPath);
  dotnetWebServer.stdout.pipe(childServerLog);
  dotnetWebServer.stderr.pipe(childServerLog);
}
```

Now we can update the before and after functions to ensure the dotnet server is started/stopped at the beginning/end of the test run:

```
before: function (done) {
 startWebApplication(this.output_folder, done);
},

after: function (done) {
 const os = process.platform;
 if (/^win/.test(os)) childProcess.spawn("taskkill", ["/pid", dotnetWebServer.pid,
 '/f', '/t']);

 else dotnetWebServer.kill('SIGINT');
 done();
},
```

Finally, we might want to run our test against an environment which isn't started by the tests themselves, for example a test environment deployed by CI or running in a separated docker container.

We can use our configuration file to setup a `start_app` flag which will depend on whether a specific target URL is provided as an environment variable or not, and then check that variable in our before/after global functions:

```
// user defined settings in nightwatch.config.js
globals: {
  …
  start_app: process.env.LAUNCH_URL ? false : true,
  …
}

// Global hooks in global.js
before: function (done) {
  if (this.start_app) {
    startWebApplication(this.output_folder, done);
  } else {
    done();
  }
},

after: function (done) {
 if (this.start_app) {
  const os = process.platform;
  if (/^win/.test(os)) childProcess.spawn("taskkill", ["/pid", dotnetWebServer.pid,
  '/f', '/t']);
  else dotnetWebServer.kill('SIGINT');

  done();
 } else {
    done();
 }
},
```

Writing and running our first test

It is time to add our first test.

Create a folder named **tests** and a new file inside it named **0.smoke.test.js**. Now let's write a simple test that simply verifies if the browser loads the home page and we get the right title:

```
module.exports = {
 '@tags': ['smoke-test', 'home-page'],
 'home page can be opened with default url': function (browser) {
   browser
    .url(browser.launchUrl)
    .waitForElementVisible('body', browser.globals.initial_load_timeout)
    .assert.title('Home Page - BlogPlayground')
    .assert.containsText('.body-content #myCarousel .item:first-child', 'Learn how
    to build ASP.NET apps that can run anywhere.');
  },
};
```

I hope the test is easy to understand, but basically we are using Nightwatch's commands and assertions to write our test.

These are convenience methods provided on top of the Web Driver protocol, but you also have access to the underlying Web Driver using the same browser object. Feel free to read around the documentation and see other test examples.

Before we can run the test, we are going to define npm commands that will allow us to easily trigger a test run against Chrome or Edge:

```
“scripts”: {
 “test:chrome”: “nightwatch --config ./nightwatch.config.js --env chrome”,
 “test:edge”: “nightwatch --config ./nightwatch.config.js --env edge”
},
```

You can now switch to the command line and run any of the following commands:

```
npm run test:chrome
npm run test:edge
```

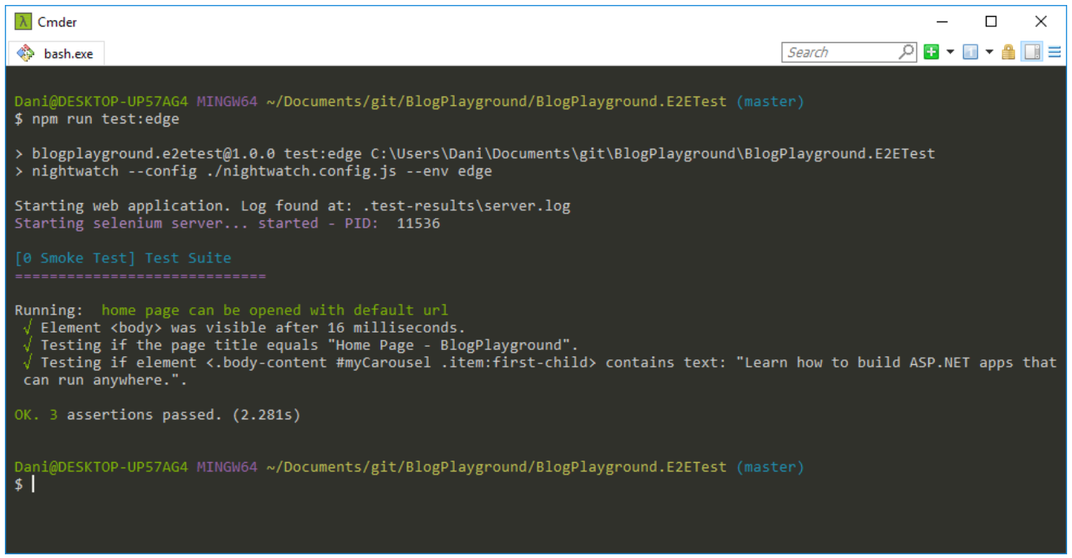


Figure 5, running the first test

Before we move on, you might be wondering how to run specific tests, or exclude certain tests, etc.

Nightwatch allows several command line options for such purposes which can be added to any of those commands, for example (the double pair of dashes is intentional):

```
# runs only tests with the ‘smoke-test’ tag
npm run test:edge -- --tag smoke-test

# runs only test with specific file name
npm run test:edge -- --test tests/0.smoke.test.js
```

This might not only be useful for developers, but also to run different sets of tests on different stages of a complex CI process. For example, a server might be started exclusively for the test run and the full suite is run, while a much smaller set of smoke tests might be run against a prod/staging environment after each deployment.

Handling login through a custom command and a global test hook

Some of the pages we would like to test in our sample project like adding/removing an article requires to be logged in as a valid user. By starting the server and seeding the database, we know certain user accounts will exist.

Now, we can add those credentials to the nightwatch’s global settings and create a custom login command. Let’s create a **commands** folder with a file **login.js** inside. Remember to uncomment the `custom_command_path` settings inside nightwatch.config.js.

The structure of a custom command in nightwatch is the following:

```
exports.command = function (userInfo) {
 const browser = this;

 // Use default nightwatch API
 browser
  .url(…)
  .waitForElementVisible(…)
  .click(…);

 return this; // allows the command to be chained.
};
```

As the file name is login.js, the command will be available in test code as `browser.login(userInfo);`

Let’s implement that command by navigating to /Account/Login, filling the login form with the test user credentials and verifying we are redirected back to the login page, but now logged in as the expected user:

```
exports.command = function (userInfo) {
 const browser = this;
 browser
  // go to login url
  .url(browser.globals.login_url)
  .waitForElementVisible(‘.form-login’, browser.globals.initial_load_timeout)
  // fill login form
  .setValue(‘input[name=Email]’, userInfo.email)
  .setValue(‘input[name=Password]’, userInfo.password)
  .click(‘.form-login button.btn-default’)
  .pause(1000)
  // After login, we should land in the home page logged in as the test user
  .assert.title(‘Home Page - BlogPlayground’)
  .assert.containsText(‘.navbar form#logout-form’, ‘Hello Tester!’);

  return this; // allows the command to be chained.
};
```

Let’s update our smoke test to verify users can login and logout from our application, taking the user credentials from the global settings defined in nightwatch.config.js. In Nightwatch, each test file is considered a test which can contain multiple steps which run in sequence.

This way, we can update the smoke test with an initial step that tries to login, a second step that verifies that the home page can be loaded, and a final step that verifies we can logout:

```
module.exports = {
 '@tags': ['smoke-test', 'home-page'],

 'can login with test user': function (browser) {
   browser.login(browser.globals.user_info);
 },

 'home page can be opened with default url': function (browser) {
   browser
    .url(browser.launchUrl)
    .assert.title('Home Page - BlogPlayground')
    .waitForElementVisible('body', browser.globals.navigation_timeout)
    .assert.containsText('.body-content #myCarousel .item:first-child', 'Learn how
    to build ASP.NET apps that can run anywhere.');
  },

  'can logout': function (browser) {
    browser
     .assert.containsText('.navbar form#logout-form', 'Hello Tester!')
     .click('.navbar form#logout-form button[type=submit]')
     .waitForElementVisible('.navbar .navbar-login', browser.globals.initial_load_
     timeout)
     .assert.containsText('.navbar .navbar-login', 'Log in')
     .assert.attributeContains('.navbar .navbar-login .login-link', 'href',
     browser.globals.login_url);
  }
};
```

Finally, we can update the global `beforeEach` function in **globals.js** to automatically run the login command before starting any test, other than the smoke test. This way, we can assume we are logged in as the test user on any test:

```
beforeEach: function (browser, done) {
  // Every test will need to login with the test user (except in the smokeTest
where login is part of the test itself)
  if ( !browser.currentTest.module.endsWith("smoke.test")) {
    browser.login(browser.globals.user_info);
  }

  //Once all steps are finished, signal we are done
  browser.perform(() => done());
},
```

With these changes, we are now ready to write the final set of tests for the articles pages.

Testing the Articles pages

Start by adding a new test file **articles.test.js** inside the tests folder. Inside this test, we will add several steps that verify the following process:

- The articles list page loads and displays several predefined articles
- A new article can be added
- New articles appear on the list page and the recent articles sidebar
- An article can be opened by navigating to its details page
- An article can be removed

Implementing these steps is a matter of using the right CSS selector to find the appropriate elements from the page so you can click on them, set their value or verify their contents are as expected.
A simple implementation is shown here for completeness:

```
const testArticle = {
 title: 'Testing with Nightwatch',
 abstract: 'This is an automated test',
 contents: 'Verifying articles can be added'
}

module.exports = {
 '@tags': ['articles-page'],

 'Articles can be opened with its url': function (browser) {
   browser
   // Open the articles list page
    .url(`${browser.launchUrl}/Articles`)
    .assert.title('Articles - BlogPlayground')
    .waitForElementVisible('body', browser.globals.navigation_timeout)
    // Verify at least the 2 default articles show up in the list
    .expect.element('.body-content .article-list li:nth-child(1)').to.be.present;
 },

  'New Articles can be added': function (browser) {
    browser
     // Go to the create page
     .url(`${browser.launchUrl}/Articles/Create`)
     .assert.title('Create - BlogPlayground')
     .waitForElementVisible('body', browser.globals.navigation_timeout)
     // Enter the details and submit
     .setValue('.body-content input[name=Title]', testArticle.title)
     .setValue('.body-content textarea[name=Abstract]', testArticle.abstract)
     .setValue('.body-content textarea[name=Contents]', testArticle.contents)
     .click('.body-content input[type=submit]')
     // Verify we go back to the articles list
     .pause(browser.globals.navigation_timeout)
     .assert.title('Articles - BlogPlayground');
},

'New Articles show in the Articles page': function (browser) {
  browser
   .assert.containsText('.body-content .article-list li:first-child', testArticle.
   title)
   .assert.containsText('.body-content .article-list li:first-child', testArticle.
   abstract)
   .assert.containsText('.body-content .article-list li:first-child .author-name',
   'Tester');
},

'Articles can be read in their details page': function (browser) {
  browser
    // Open the article from the lisdt
    .click('.body-content .article-list li:first-child h4 a')
    // Verify navigation to the article details and the right contents are
    displayed
    .pause(browser.globals.navigation_timeout)
    .assert.title(`${testArticle.title} - BlogPlayground`)
    .assert.containsText('.body-content .article-summary', testArticle.title)
    .assert.containsText('.body-content .article-summary', testArticle.abstract)
    .assert.containsText('.body-content .article-summary .author-name', 'Tester')
```

```
      .assert.containsText('.body-content .markdown-contents', testArticle.
      contents);
  },

  'Articles can be removed': function (browser) {
    browser
      // Click remove on article details
      .click('.body-content .sidebar button.dropdown-toggle')
      .waitForElementVisible('.body-content .sidebar ul.dropdown-menu', browser.
      globals.navigation_timeout)
      .click('.body-content .sidebar ul.dropdown-menu li:last-child a')
      // Verify navigation to the confirmation page and click delete
      .pause(browser.globals.navigation_timeout)
      .assert.title('Delete - BlogPlayground')
      .click('.body-content input[type=submit]')
      // Verify navigation to articles list and that it disappeared from the list
      .pause(browser.globals.navigation_timeout)
      .assert.title('Articles - BlogPlayground')
      .assert.containsText('.body-content .article-list li:first-child', 'Test
      Article 2');
  }
};
```

Go ahead and run the tests again, you should see them all passing in your console output:

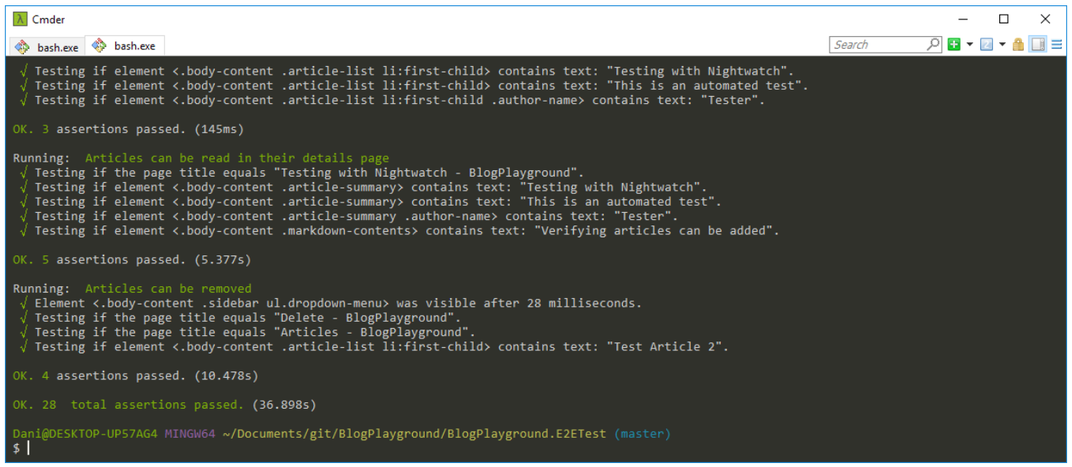


Figure 6, full test run

A note on CSS selectors and tests maintainability

As you might notice, the tests are hugely dependent on the HTML structure and attributes of the application. You really want your selectors as vague as possible to avoid them breaking with the simplest of refactoring. Sometimes this will mean adding ids or class names to elements, so your tests can find them without having to assume too much about the HTML structure of your page.

For example, instead of a selector like '`.body-content .sidebar button.dropdown-toggle`' it might be a good idea to add an `id` to the dropdown with article actions and simply use '`#articles-actions-dropdown`'.

This allows you to redesign your article details page and the test won't care as long as there is a dropdown with such an `id`, while with the first selector, your test might break if the dropdown uses a link tag or is moved away from the sidebar.

Another consideration is the repetition of selectors inside a test file like articles.test.js.

You might want to extract constants with those selectors or even better define Nighwatch custom pages that explicitly provide elements to interact with and encapsulate methods to perform actions on those pages. This allows you to use Page Models in your test, instead of directly interacting with the browser API. As you write more tests and your test suite grows, you really want to leverage Nightwatch's extensibility and create custom commands, pages and assertions so your test code stays clean and maintainable.

This is one of the reasons for using a framework like Nightwatch over directly using the Web Driver API.

Conclusion

Testing is hard, there is no escape from this. And within the different types of tests, End-to-end (E2E) tests are the ones that present the biggest challenge. Not only are they closely dependent on the UI (which makes them very brittle), they also involve the largest number of moving parts.

These tests require your application to run on a real (or as close as possible/feasible) environment and involve a real browser (most likely multiple different browsers), a driver for that browser and a Selenium environment. That's a fairly large number of processes and services, all of them required to work in order for your tests to run and remain stable!

There are many ways you can write E2E tests using the Web Driver protocol, of which we have seen one possible approach with Nightwatch and Selenium.

While my experience with these tools has been positive and made my life easier managing this particular challenge, you might prefer to use other tools. That's absolutely fine, since the principles are the same and your main challenge will be finding the right balance between coverage and the creation/maintenance cost of the tests.

It is an interesting challenge but remember you can leverage the work you did with unit and integration tests. This way, it should be easier for you to keep a reduced number of E2E tests while maintaining a good overall coverage of your application.

Daniel Jimenez Garcia

Author

Daniel Jimenez Garcia is a passionate software developer with 10+ years of experience. He started as a Microsoft developer and learned to love C# in general and ASP MVC in particular. In the latter half of his career he worked on a broader set of technologies and platforms while these days is particularly interested in .Net Core and Node.js. He is always looking for better practices and can be seen answering questions on Stack Overflow.

Thanks to Damir Arh for reviewing this article.

Damir Arh

Writing Applications for Different .NET Framework Versions

As .NET developers we're used to running the latest version of the operating system on our computers and having the latest updates installed. However, that's not always the case in enterprise environments where IT personnel are often a lot more conservative. They make sure that all critical business software will keep running without issues, before doing any large-scale updates.

As a result, the software we write can't always take advantage of the latest .NET framework features.

In this article, we'll take a look at the major differences between the .NET framework versions and explore how we can still target older versions of .NET framework even when using the latest development tools.

The Evolution of .NET Framework

As of this writing, it's been over fifteen years since the original release of .NET framework 1.0 and looking at it today, it seems much less impressive than it did at that time. Most of the features we are casually used to, were not even available then and we would have a hard time if we wanted to use it for development of modern applications today.

Here's a quick rundown of the major features in every .NET release.

CLR version	.NET framework version	Most important changes
1.0	1.0	Initial release: console, Windows Forms, Windows Services, Web Forms, Web Services
1.1	1.1	IPv6, ASP.NET mobile controls, ADO.NET ODBC
2.0	2.0	Generics, edit and continue, 64-bit runtime
	3.0	WPF, WCF, WF
	3.5	LINQ, ASP.NET AJAX, WF services, cryptography
	3.5 SP1	.NET Client Profile, EF
4	4	DLR, PCL, TAP, ASP.NET MVC
	4.5	Async I/O, ASP.NET Web API, ASP.NET Web Pages
	4.5+	64-bit edit and continue, async debugging, HTTP/2, 64-bit JIT

Table 1: .NET framework version history

.NET framework 1.0 included a very limited set of application frameworks. Only the following types of applications were supported:

- Console and Windows Forms applications for the desktop,
- Windows Services,
- ASP.NET Web Forms based web applications, and
- Web services using the SOAP protocol.

.NET framework 1.1 didn't add too much. Although it featured a newer version of CLR (the Common Language Runtime), it was mostly to enable side-by-side installation of both versions on the same computer.

The most notable new features were support for IPv6, mobile controls for ASP.NET Web Forms and an out-of-the-box ADO.NET data provider for ODBC. There wasn't a particularly convincing reason to migrate to v1.1 unless you had a specific need for one of these features.

.NET framework 2.0 was a different story, altogether. It built on the recognized weaknesses of the previous two versions and introduced many new features to boost developer productivity. Let's name only the ones with the biggest impact:

- The introduction of **generics** allowed creation of templated data structures that could be used with any data type while still preserving full type safety. Generics was widely used in collection classes, such as lists, stacks, dictionaries etc. Before generics, only objects could be stored in the provided collection classes. When used with other data types, code had to cast the objects back to the correct type after retrieving them from the collection. Alternatively, custom collection classes had to be derived for each data type. Generics weren't limited to collections, though. Nullable value types were also relying on them, for example.
- Improved debugging allowed **edit and continue** functionality: when a running application was paused in the debugger, code could be modified on the fly and the execution continued using the modified code. It was a very important feature for existing Visual Basic 6 programmers who were used to equivalent functionality in their development environment.
- **64-bit runtime** was added. At the time, only Windows Server had a 64-bit edition and the feature was mostly targeted at server applications with high memory requirements. Today, most of Windows installations are 64-bit and many .NET applications run in 64-bit mode without us even realizing it.

All of these, along with countless other smaller improvements to the runtime and the class library made .NET framework 2.0 much more attractive to Windows developers using other development tools at the time. It was enough to make .NET framework the de-facto standard for Windows development.

.NET framework 3.0 was the first version of .NET framework which couldn't be installed side-by-side with the previous version because it was still using the same runtime (CLR 2.0). It only added new application frameworks:

- **Windows Presentation Foundation (WPF**) was the alternative to Windows Forms for desktop application development. Improved binding enabled better architectural patterns, such as MVVC. With extensive theming support, application developers could finally move beyond the classic battleship -gray appearance of their applications.
- **Windows Communication Foundation (WCF)** was a new flexible programming model for development of SOAP compliant web services. Its configuration options made it possible to fully decouple the web service implementation from the transport used. It still features the most complete implementation of WS-* standards on any platform but is losing its importance with the decline of SOAP web services in favor of REST services.
- **Windows Workflow Foundation (WF)** was a new programming model for long running business processes. The business logic could be defined using a graphical designer by composing existing blocks of code into a diagram describing their order of execution based on runtime conditions.

While WF wasn't widely used, we can hardly imagine .NET framework without WCF and WPF and its derivatives.

.NET framework 3.5 continued the trend of keeping the same runtime and expanding the class library. The most notable addition was **Language Integrated Query (LINQ)**. It was a more declarative (or functional) approach to manipulating collections. It was highly inspired by SQL (Structured Query Language) widely used in relational databases. So much so that it offered a query syntax as an alternative to the more conventional fluent API.

By using **expression trees** to represent the query structure it opened the doors to many LINQ providers, which could execute these queries differently based on the underlying data structure. Three providers were included out-of-the box:

- LINQ to Objects for performing operations on in-memory collections.
- LINQ to XML for querying XML DOM (document object model).
- LINQ to SQL was a simple object-relational mapping (ORM) framework for querying relational databases.

Web related technologies also started to evolve with this version of .NET framework:

- **ASP.NET AJAX library** introduced the possibility of partially updating Web Forms based pages without full callbacks to the server by retrieving the updates with JavaScript code in the page. It was the first small step towards what we know today as single page applications (SPA).
- **WF services** exposed WF applications as web services by integrating WF with WCF.
- Managed support for **cryptography** was expanded with additional algorithms: AES, SHA and elliptic curve algorithms.

Looking at the name of **.NET framework 3.5 SP1** one would not expect it to include features, but at least two of them are worth mentioning:

- **Entity Framework (EF)** was a fully featured ORM framework that quickly replaced LINQ to SQL. It also heavily relied on LINQ for querying.
- **.NET Client Profile** was an alternative redistribution package for .NET framework which only included the parts required for client-side applications. It was introduced as a mean to combat the growing size of the full .NET framework and to speed up the installation process on the computers which did not

need to run server-side applications.

.NET framework 4 was the next framework that included a new version of the CLR and could therefore again be installed side-by-side with the previous version.
Note: *As mentioned previously, you cannot run versions 2.0, 3.0, and 3.5 side by side on a computer. Apps built for versions 2.0, 3.0, and 3.5 can all run on version 3.5. Starting with the .NET Framework 4, you can use in-process side-by-side hosting to run multiple versions of the CLR, in a single process.*

The most notable new additions were:

- **Dynamic Language Runtime (DLR)** with improved support for late binding. This simplified certain COM interop scenarios and made porting of dynamic languages to CLR easier. IronRuby and IronPython (Ruby and Python ports for .NET framework) took advantage of that.

- **Portable Class Libraries (PCL)** were introduced for creating binary compatible assemblies that could run on different runtimes (not only .NET framework, but also Windows Phone and Silverlight).

- **Task-based Asynchronous Pattern (TAP)** was a new abstraction for writing multi-threaded and asynchronous applications which hid many of the error-prone details from the programmers.

- **ASP.NET MVC** was an alternative programming model to Web Forms for creating web applications. As the name implies, it implemented the model view controller (MVC) pattern which was gaining popularity on other platforms. It was first released out-of-band, but .NET framework 4 was the first version to include it out-of-the-box.

.NET framework 4.5 was the last major new release of .NET framework with several new important features:

- Support for **asynchronous I/O operations** was added as the new `async` and `await` syntax in C# made it much easier to consume them.

- **ASP.NET Web API** and **ASP.NET Web Pages** were first included in the framework after they were originally released out-of-band. They allowed development of REST services and provided an alternative Razor syntax for ASP.NET MVC views, respectively.

- PCL was expanded to support **Windows Store** applications for Windows 8.

Also, .NET Client Profile was discontinued as optimizations to the size and deployment process of .NET framework made it obsolete.

Since .NET framework 4.5, the release cadence increased. The version numbers (4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.x) reflected that. Each new version brought bug fixes and a few new features. The more important ones since .NET framework 4.5 were:

- Edit and continue support for 64-bit applications.

- Improved debugging for asynchronous code.

- Unification of different ASP.NET APIs (ASP.NET MVC, ASP.NET Web API and ASP.NET Web Pages).

- Support for HTTP/2.

- Improved 64-bit just-in-time (JIT) compiler for faster startup time of smaller applications.

The focus of new development has now shifted to .NET Core, therefore new major improvements to .NET framework are not very likely. We can probably still expect minor releases with bug fixes and smaller improvements, though.

Backward Compatibility

Backward compatibility has been an important part of .NET framework since its original release. With rare exceptions, any application built for an older version of .NET framework should run in any newer version of .NET framework without any issues.

We can group.NET framework versions into two categories:

- In the first category are .NET framework versions which include a new version of CLR. Some breaking changes were introduced in these versions which could affect existing applications. To keep all applications working in spite of that, these .NET framework versions are installed side-by-side, i.e. they keep the previous version intact and allow applications that were built for it to still use it. There were four versions of CLR released: 1.0, 1.1, 2.0, and 4.

- In the second category are .NET framework versions *without* a new version of CLR. These only expand the class library, i.e. add new APIs for the applications to use. They install in-place and therefore affect existing applications, forcing them to run on the updated .NET framework version.

Since Windows 8, .NET frameworks 1.0 and 1.1 are not supported any more. The operating system comes preinstalled with the version of .NET framework that is current at the time of release. Newer versions are installed as part of Windows Update. The preinstalled version is based on CLR 4. .NET framework 3.5 (the latest version running on CLR 2.0) can be optionally installed as a Windows feature.

Applications built for .NET framework 4 or newer will run on the preinstalled CLR 4 based .NET framework version. They might still fail to run properly if they use an API from a newer version of .NET framework that is not yet installed on the target computer.

Applications built for older versions of .NET framework will by default try to run on .NET framework 3.5. If it is not installed on the target computer, the application will fail to run and prompt the user to install .NET framework 3.5. This behavior can be changed by adding a `supportedRuntime` entry to the application configuration file (named `MyApplication.exe.config`, where `MyApplication.exe` is the affected executable filename):

```
<?xml version=”1.0” encoding=”utf-8” ?>
<configuration>
  <startup>
    <supportedRuntime version=”v2.0.50727”/>
    <supportedRuntime version=”v4.0”/>
  </startup>
</configuration>
```

By specifying multiple runtimes, they will attempt to be used in the order given, i.e. in the above example if .NET framework 3.5 is installed on a machine (v2.0.50727 indicates CLR 2.0), the application will use it, else it will run on the latest .NET framework version installed.

Alternatively, only v4.0 could be listed in the configuration file to force the application to run using the latest version of .NET framework even if .NET framework 3.5 is also installed. If the application is known not to have any issues with the latest version of .NET framework, this will allow it to take advantage of optimizations in CLR 4 without recompiling it.

Multi-Targeting in Visual Studio

To help with development of applications for the .NET framework version of choice, Visual Studio features full multi-targeting support. When creating a *new project*, there is a dropdown available in the New Project dialog to select the .NET framework version.
This will not only affect how the new project will be configured, but will also hide any project templates that are not supported in the selected .NET framework version (e.g. *WPF App* cannot be created when .NET framework 2.0 is selected because it was only introduced in .NET framework 3.0).

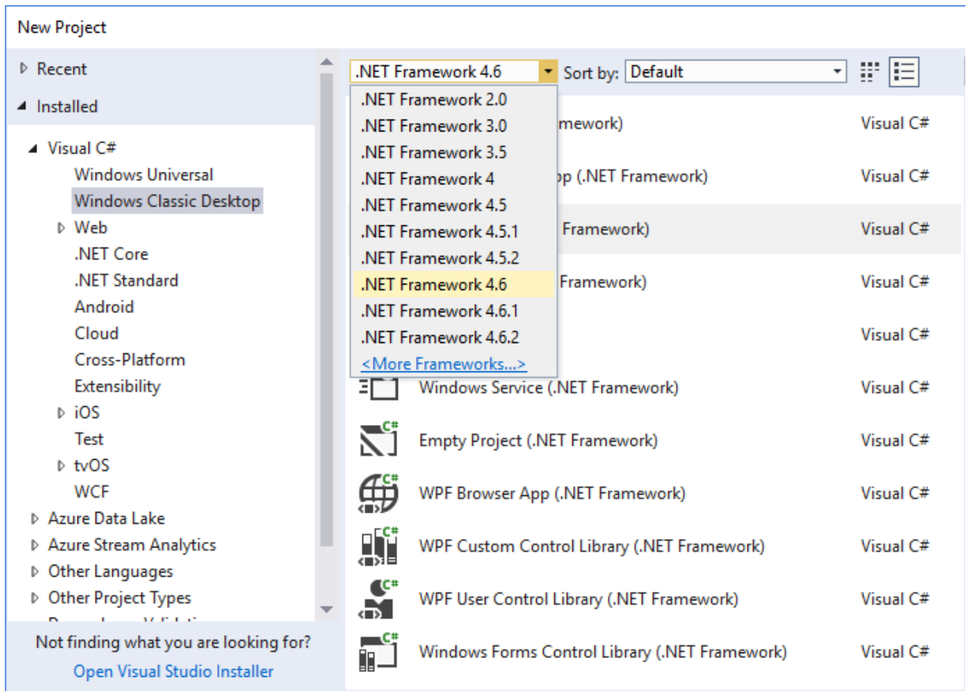


Figure 1: .NET framework dropdown in New Project dialog

Visual Studio will make sure that only APIs included in the selected.NET framework version will be available to you. If you attempt to use any that are not supported, they will be marked accordingly in the code editor.

```
using System.Collections.Generic;
using System.Linq;

namespace Console
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            var list = new List<string>();
            var sublist = list.Where(s => s.Length > 5);
        }
    }
}
```

The type or namespace name 'Linq' does not exist in the namespace 'System' (are you missing an assembly reference?)

Using directive is unnecessary.

Show potential fixes (Ctrl+.)

Figure 2: Unsupported APIs marked as errors in the code editor

The target framework for an existing project can be changed at a later time on the *Application* tab of the project properties window. This will have the same effect as if that target framework was already selected when the project was created. If the existing code used APIs, which are not available after the change, the project will fail to build until the code is fixed or the target framework is changed back. The compiler will also warn about any referenced system assemblies that are not available so that they can be removed from the project references.

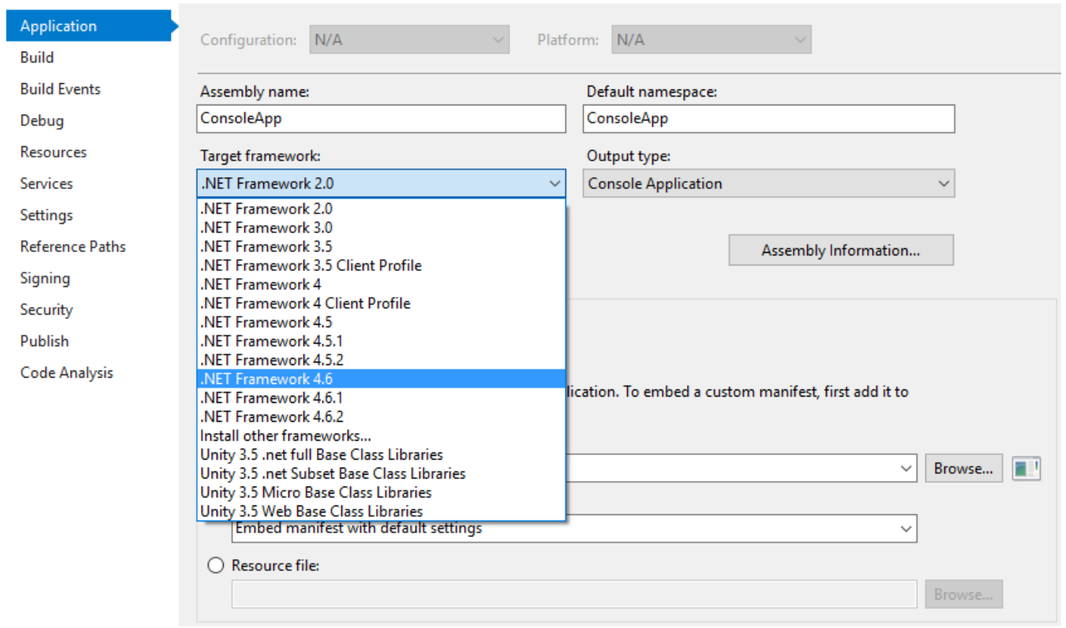


Figure 3: Changing the target framework for an existing project

Which Version of C# to Use?

No matter which .NET framework version we target in the project, the C# language version in use will not change. That's fine because the vast majority of language features that were introduced in later versions of the language don't depend on the CLR or specific APIs. They are only syntactic sugar and the bytecode generated by the compiler will still work in .NET framework 2.0. There are a couple of exceptions, though.

For example, the `async` and `await` keywords are syntactic sugar for Task-based asynchronous pattern, which was only introduced in .NET 4, therefore they cannot be used in earlier versions of .NET framework. They also depend on some additional classes, which are only available in .NET framework 4.5. However, these can be added to the project by installing *Microsoft.Bcl.Async* NuGet package. This way, you can start using `async` and `await` keywords in a project targeting .NET framework 4.

Similarly, some compiler features depend on specific attributes in their implementation:

- To compile code taking advantage of extension methods, `System.Runtime.CompilerServices.ExtensionAttribute` is required. It was only introduced in .NET framework 3.5.

- Caller information attributes were only added in .NET framework 4.5 and without them we have no way of telling the compiler our intention although the generated code would still work in older versions of .NET framework.

```
public void LogMessage(string message,
  [CallerMemberName] string memberName = "",
  [CallerFilePath] string sourceFilePath = "",
  [CallerLineNumber] int sourceLineNumber = 0)
{
  // logging implementation
}
```

Although there is no official NuGet package to make these two features available for use in older versions of .NET framework, the compiler only requires the presence of attributes with the correct fully qualified name to make them work. You can either declare them in your own code or install one of the unofficial NuGet packages.

The above-mentioned limitations don't prevent you from using the latest version of compiler when targeting an older version of .NET framework, though. The code editor will immediately warn you if you try to use an unsupported feature so that you can avoid it while still taking advantage of all the other supported features.

With that being said, there is still a legitimate use case when you might want to use an older version of the compiler in your project.

Each compiler version is only available from the Visual Studio version which it was originally released with. If some developers on the project are still using an older Visual Studio you will need to avoid using newer language features or they won't be able to compile the code. To safely achieve that, you can specify the language version in the *Advanced Build Settings* dialog, accessible via the *Advanced...* button on the *Build* tab of the project properties.

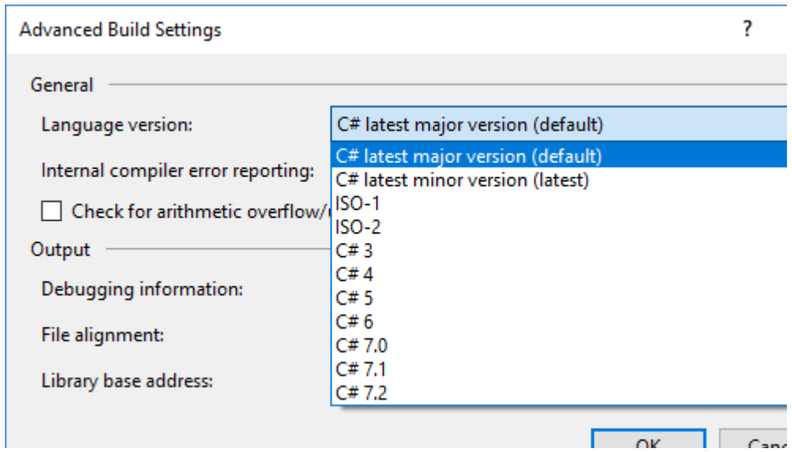


Figure 4: Language version selection in Advanced Build Settings dialog

Conclusion:

When deploying applications to more strictly managed environments, you might encounter restrictions on what versions of .NET framework can be made available.

Thanks to backward compatibility most of applications developed for older versions of .NET framework will run just fine, if not better on the latest version of .NET framework. They can run even without recompilation as long as you declare that the runtime is supported in the application configuration file.

On the other hand, when targeting an older version of .NET framework, you will need to avoid using APIs which were not yet available. Visual Studio can help you with that if you correctly set the .NET framework version you are targeting in the project properties. You will be able to keep using the latest version of C# though, unless you have team members who are still working on an older version of Visual Studio.

Microsoft® MVP Most Valuable Professional

Damir Arh
Author

Damir Arh has many years of experience with Microsoft development tools; both in complex enterprise software projects and modern cross-platform mobile applications. In his drive towards better development processes, he is a proponent of test driven development, continuous integration and continuous deployment. He shares his knowledge by speaking at local user groups and conferences, blogging, and answering questions on Stack Overflow. He is an awarded Microsoft MVP for .NET since 2012.

Thanks to Suprotim Agarwal for reviewing this article.

Gerald Versluis

A lap around VISUAL STUDIO APP CENTER

About a year ago (Nov 2016), Microsoft introduced a new application suite which was named Visual Studio Mobile Center. In November 2017, this solution was rebranded to Visual Studio App Center and made generally available. So, I think it's about time we have a look at what App Center is, and what it can do for you! Spoiler alert: it has everything to do with all your apps, and not just Xamarin apps!

I was actually attending the MVP Global summit in November 2016 when Microsoft introduced the Azure Mobile Center, which was what it was called back then.

I also remember slightly panicking when they demoed it. I was just starting off my first book which handles continuous integration and delivery for your Xamarin apps, based on Visual Studio Team Services (VSTS). For a second there I thought that App Center, as it is called today, would replace building apps in VSTS altogether and my book would become redundant instantly.

Luckily, that is not the case, but we will get to that a little later on.

What Is Visual Studio App Center? – A Brief History

App Center is a lot of things.

Microsoft calls it: "mission control for apps", and that is exactly what it is. Almost everything that has to do with an app lifecycle, can be found in App Center.

As you might know, Microsoft has been acquiring some companies that are mobile related, as well as developing some mobile specific services, in-house. The most prominent one is of course Xamarin, which enables you to write cross-platform apps with all the .NET goodies. But HockeyApp, CodePush and Azure App Services are also on this list.

Let's quickly go over each of them, so it gives you a better overview of what functionality came in App Center, and from where.

HockeyApp

The HockeyApp solution was primarily targeting the distribution of apps outside of the regular App Stores. Its name was derived from the Ad-Hoc profile to distribute apps. By the means of HockeyApp, you were able to create groups of users, gather their device details and distribute a version of your app to this group, in a very easy manner. Later they also added some basic crash reporting and analytics, but before that ever really took off, Microsoft came in 2014 and bought the service as a whole.
When Microsoft made App Center generally available, HockeyApp became a free service to match the pricing of App Center. HockeyApp will be retired at some point but it is unclear as to when that will exactly happen.

Xamarin

In early 2016, Microsoft and Xamarin announced that they have reached an agreement and Xamarin became part of Microsoft.

This is where things started to get interesting.

With the acquisition of Xamarin they didn't just gain a solution to build cross-platform apps, but also some great products and services like Test Cloud and Insights.

If you do not know it yet; Xamarin Test Cloud is a unique service which lets you run automated UI tests on actual physical devices. Running and testing your app has never been easier and in more resemblance to real world usage.

Xamarin Insights was another service that focused on retrieving crash reports and providing analytics, just like HockeyApp. However Xamarin did it a better job in my opinion. But that doesn't really matter anymore right now because one of the first things that got merged was HockeyApp and Insights.

Actually, Xamarin Insights will be retired as of March 2018.

CodePush

As an open-source project, CodePush has been around since 2015. This service focuses on releasing new code to an already released app, without the need to update. Basically, what this does is push a newer version of your app to users' device, without going through the App Store review process.

This works for hybrid-based apps like React Native and Cordova and such. While this solution does have its purposes, it should be used sparingly. The App Stores will have strict rules in place when you want to update your app without going through their review process. Moreover, I can imagine that a user can be very confused when the whole app is just changed overnight, while they didn't update anything themselves.

These services are now also integrated into App Center and will be retired as an individual service in October 2018. This functionality will not be described in detail in this article.

Visual Studio Team Services

What began as Team Foundation Server (TFS) has now evolved into a fully-featured, cloud hosted solution: Visual Studio Team Services. By using VSTS, you gain a lot of functionality at once: hosting code with Git, automated builds, automated releases, tracking work items and even more! This product (TFS) has been developed by Microsoft for over a decade now and is pretty sweet.

I already mentioned at the beginning of this article that you can setup a fully automated pipeline for your apps with VSTS, and you will be able to continue to do so.

In future, it should be possible to use some more advanced features of VSTS when starting from App Center, with just a click of a button. This is possible because VSTS APIs are used under the hood. That's right, you're just seeing another view on top of the VSTS powerhouse, especially for us mobile developers.

As you can see, Microsoft now had a lot of awesome solutions at their disposal, some of which were overlapping. Basically, what they did is take the best of each of these services, rebrand them under one umbrella (Microsoft) and integrate them in a very effective way.

App Center is very focused on helping developers who are getting started with mobile development, as well as to make the life of an established mobile developer much, much easier. For instance: setting up a build takes you as little as 10 minutes! To see this in action, have a look at my Gur.Codes(); vlog here: https://www.youtube.com/watch?v=MDzyNSUvCOs.

App Center Today

Now that we have a pretty good understanding of what the App Center is, let's just dive in and take a look at what we can work with today. To access App Center, just head over to https://appcenter.ms. There you have several options to create an account: GitHub, Microsoft Account, even Facebook and Google or just signup for a App Center account.

With whatever option you chose, you will land at your *personal dashboard*, mine is shown in Figure 1.

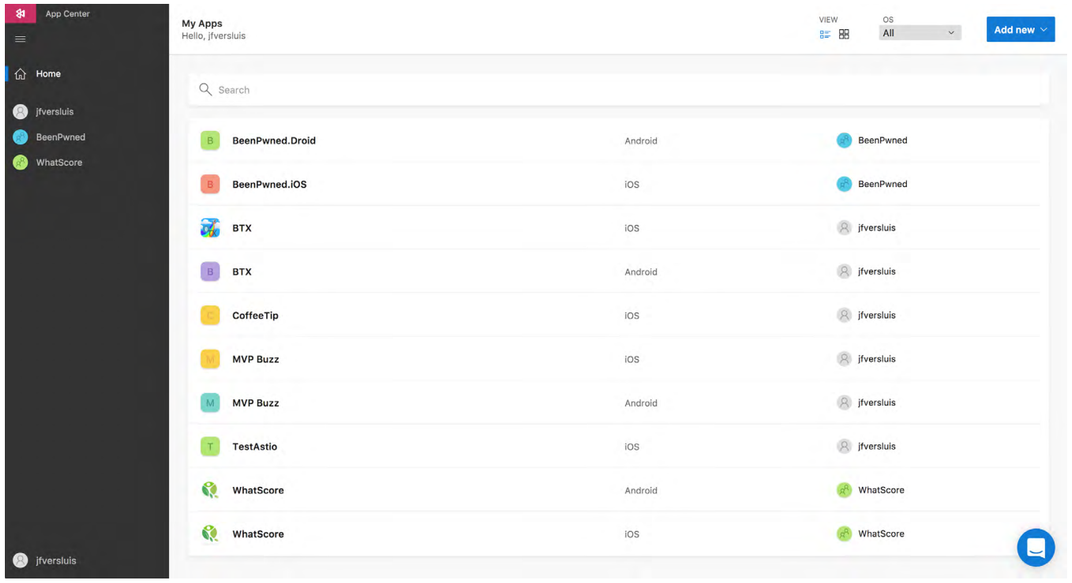


Figure 1: My App Center personal dashboard with the overview of my apps

The first thing you need to do is create a new app definition. You can do this by clicking the blue **Add New** app button in the upper-right corner.

To add an app, you will have to specify some basic information like name, the platform that you are targeting and the technology that it is built in. Note here that the technologies also list native technologies like Objective-C, Java and even Cordova! This means, you can also use it for any legacy apps that you might have lying around.

For this example, I will be sticking to a Xamarin iOS app.

After saving it, you will be presented with an *app dashboard*. Here you can find everything that is specific for this app. Because we have configured the targeted platform and also the technology for the app, it will already show some custom messages that are relevant to it. This also means that if you have one app that targets multiple platforms, each platform will have a separate app definition.

Looking at the welcoming page, we see that there are several NuGet packages available that we can drop into our project to enable crash reporting and analytics. More on that later on.

Let's quickly go over the different sections that we see on the left (not shown in the figure), listing all the functionalities that we can use from within the App Center.

Build Configuration

One of the main functionalities of App Center is the Build section. This enables you to setup an automated build pipeline within a matter of minutes, literally. If you have been into automated builds before, you might know that when building for iOS, you still need some form of Mac hardware!

Well, Microsoft has now taken care of this for you. They have provisioned Macs in the cloud, for you to use when building your app.

Want to hear something better? These can also be used in VSTS!

When we go into the Build section for the first time, we need to connect to a repository provider. It needs to get the code from somewhere. The providers supported right now are GitHub, Bitbucket and VSTS (with Git).

To connect, simply click the option of your choice and connect to that service. Once done, you will see all the branches currently in that repository. For each of these branches you can create a separate build definition. To do so, simply click on that branch and a configuration screen will come up. The looks of this screen will depend a little bit on what platform you are targeting and which platform you are using, but the gist should be the same.

You can see the screen in Figure 2.

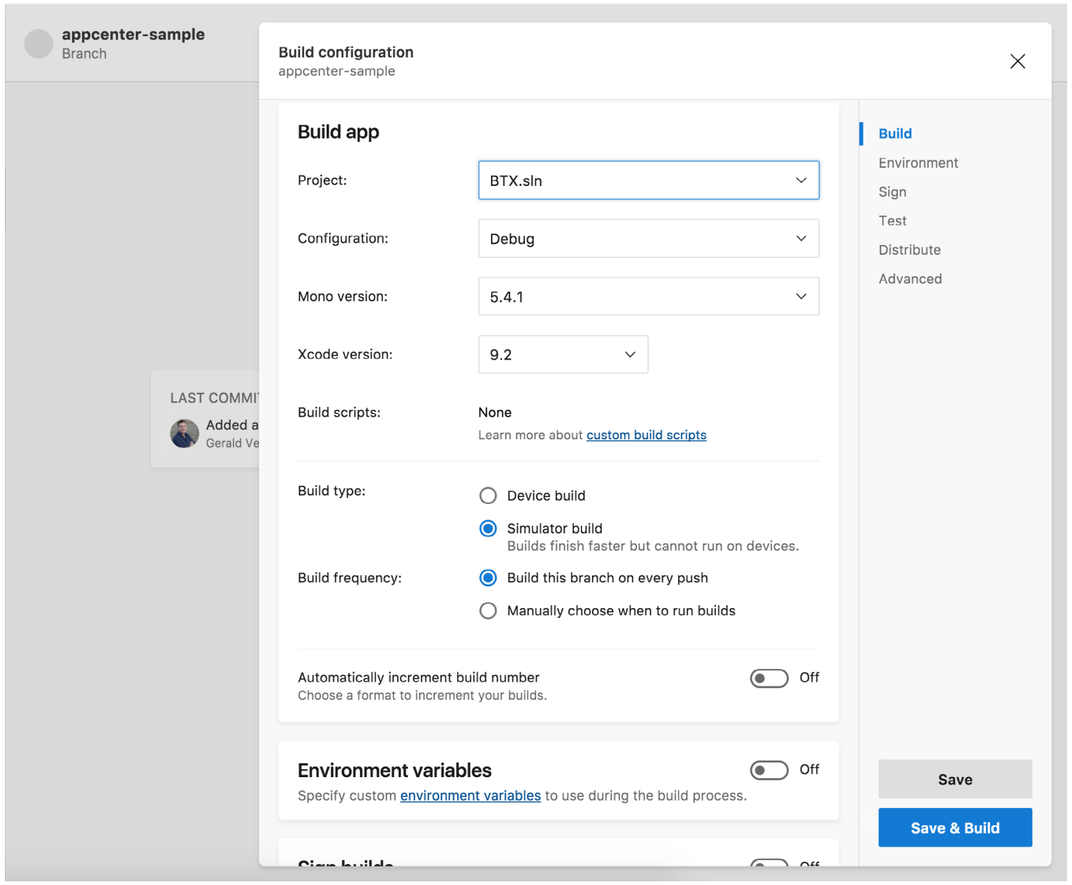


Figure 2: The screen where you configure the build for this branch

At the top of this screen, you will notice that the most important stuff is already filled in!

In case of my Xamarin iOS app, it already selected the solution to build, the configuration is set to Debug, and you can choose a Mono and Xcode version if you want to. But basically, you can leave it like this and just trigger a build immediately.

There are some options here for you to configure. Most notably, if you want to distribute your build, you need to sign it. Depending on the platform that you are targeting, you will need to provide some certificates or a keystore or similar. You can also incorporate Bash/Powershell build scripts to do some more advanced stuff and choose when a build should be triggered.

If you chose to sign your build, you can also enable the option to distribute this build - another great integration point; directly from build to release. Here you can define a group that you want to notify whenever there is a new build. Whenever that happens, these users can go into a web portal on their phone and download the new version right away.

There is one other option 'Test on real device', we'll talk about that a bit more in a minute.

Test on a real device

This section is basically about Xamarin Test Cloud incorporated in App Center.

Here you will see an overview of the tests that you ran on this app. You can go into a certain test to run and inspect the results for all the devices that you specified. This proves to be a very powerful tool. Not only can you see a screenshot of every step along the way, but you will also be provided with data like device logs, stack traces (where applicable) and statistics like memory and CPU usage.

We won't go into much details because I could write a separate article about it (and I did in an earlier edition of DNC Magazine), but I just what to show you how nicely this integrates with the automated builds.

If you recollect the steps in creating the build definition, there was a setting called 'Test on real device'. With this little switch, you turn on a small sanity test. Without writing a single line of code, your app will be put through the Test Cloud on a physical device, just to see if it boots correctly, free of charge.

Pretty awesome, right?

Distribute

The section under Distribute is mostly the part where HockeyApp comes in. This functionality enables you to configure certain users and organizes them into groups. From there, you can start distributing versions. These versions can either come from the App Center itself, or you can upload a binary. Another option would be to provide a binary from another build system through the App Center APIs.

The distribute functionality also helps you collect the device IDs which are required to be included in your test version of the app. It can even bring your app all the way to the actual App Store by integrating with the Apple and Google APIs for the same.

Crash Reporting

Another helpful feature is crash reporting. This is extremely easy to setup. To use the most basic functionality, all you need to do is drop in a NuGet package and use one line of initialization code. From then on, whenever your app crashes in an unhandled exception, the stack trace will automatically be reported here for you to see. This way, you can easily and quickly identify problems without having to rely on your users.

It also helps you categorize them and analyze the data to see if it happens on a specific OS version or

version of your app, etc.

Although not available right now as of this writing, a future feature will give you the ability to report handled exceptions. That way you can get some more insights on errors that were handled in an orderly fashion, but might still be good to know of.

Analytics

Analytics is somewhat related to crashes and like Crash Reporting, this is a mix of HockeyApp and Xamarin Insights functionality.

You can get some basic functionality by just dropping in the analytics NuGet package with just one line of initialization. This will help you identify where your users are using the app geographically, on what device, what version OS, what version of your app and much, much more.

You can also identify custom events, so you can track users as they go through your application. In Xamarin Insights, the analytics and crashes were very tight, and you could follow a user's session right up to the crash, which can be very insightful.

Of course, be very careful with crashes and even more so, with analytics while collecting personal data the users' permissions, as otherwise you might get into trouble with the new GDPR rules very easily.
Under the analytics section, there is a powerful little feature called Log Flow. When you go to it, you will be presented with an empty screen and a message telling you that it is waiting for data from your app. Then when you start your app with the analytics package installed, you will start seeing near real-time data flowing in from your app session.

Push

If you think about mobile, you can't get around push notifications. There are some services within Azure that help you with this, but of course App Center is the perfect place for this.

Incorporating push notifications can be done easily with a NuGet package, although it needs a bit more work than the crash and analytics one. When implemented, App Center lets you build different audiences easily. Besides sending notifications to everyone, you can also send something to a subset based on geographic location, version of the app, OS version and more.

At this point, it is probably also good to mention that App Center offers a lot of great services, but it is up to you which services you want to use and how. You can mix-and-match all the services to accommodate for your needs. This way, it is very easy to transition gradually into App Center for instance.

App Center is built API-first, which means that the APIs are actually a bit ahead of the UI in terms of features. To make it even more easier to reach these APIs, Microsoft has also introduced a command-line interface tool. With this tool you can easily integrate calls to the App Center APIs from your existing build pipeline.

You can see the full API definition here: https://openapi.appcenter.ms/ and the command-line tooling can be found here: https://github.com/Microsoft/appcenter-cli.

Pricing

Another important aspect that you might be interested in is the pricing of all this goodness. While everything was in preview, it was free of charge. At the time of writing, some parts are still in preview and thus for free, but you can expect this to change.

The pricing right now can be seen in Figure 3 and is designed to increase, as your app grows.

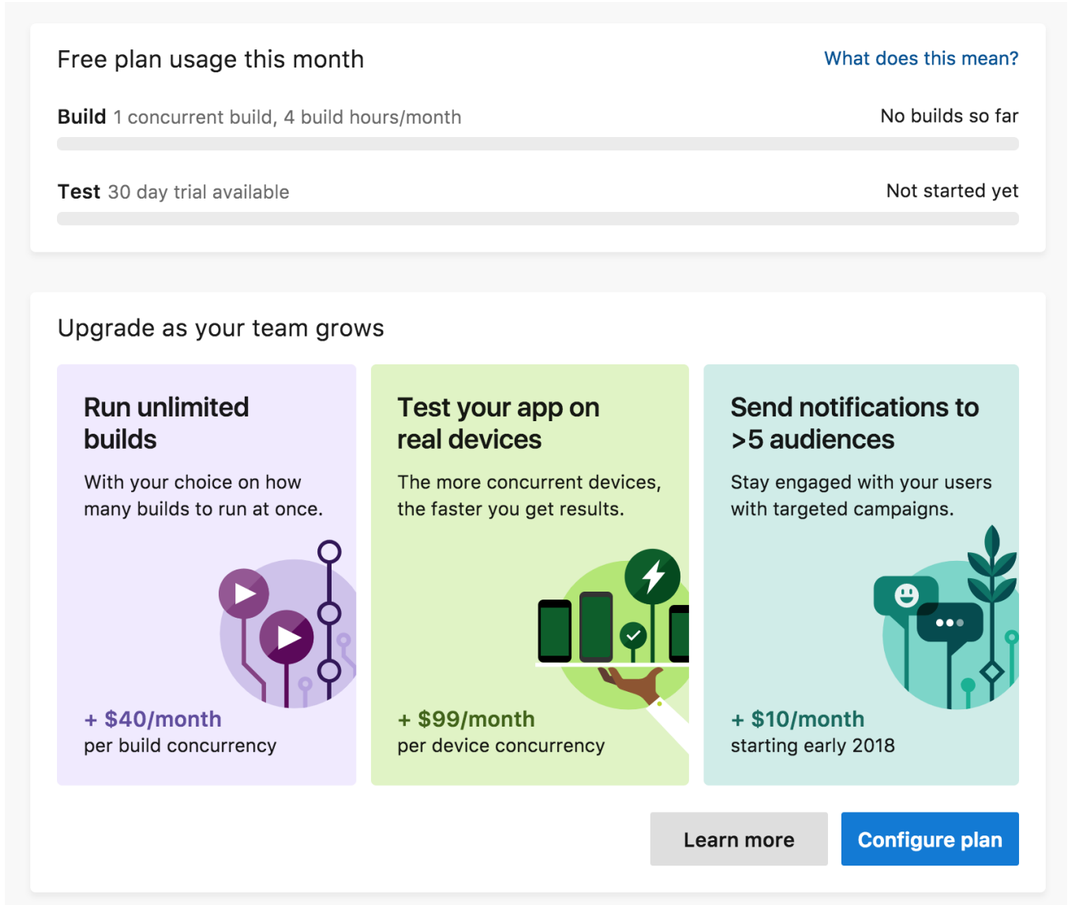


Figure 3: Pricing for the different functionalities of App Center

As mentioned, you can start for free with building (limited to 240 build minutes per month), distributing (unlimited releases to unlimited users), crash reporting and analytics. For the automated UI tests, a 30-day trial is available.

So, as you can see, there is nothing stopping you from trying it out right now!

Let's Sum It All Up

App Center proves to be a great tool for everything that has to do with your mobile app development. You can start with the basic functionalities for free, so there should be no reason to do any more 'right-click deploys'.

Now go and set up that automated build and distribution pipeline, it's a breeze!

There are some rough edges right now in terms of missing functionalities that was available earlier as separate services (e.g. HockeyApp of Xamarin Insights), but these will be brought back eventually.

It is awesome that all the tools are available through APIs and command-line tools, this way you can incorporate it in a variety of ways and start working with it, whichever way you like.

Nevertheless, the App Center has a great looking UI too which can help you get started in no time.

All in all, App Center is a great product to keep your eye on, I think there is a lot more to come. For now, all the tight integrations with Azure seem to be put on the backlog, but I'm sure it will return eventually.

To stay informed about the future plans, have a look at the road map: https://docs.microsoft.com/en-us/appcenter/general/roadmap.

If you have any questions, please don't hesitate to reach out to me in any way. Thank you for reading!

Microsoft® MVP Most Valuable Professional

Gerald Versluis
Author

Gerald Versluis (@jfversluis) is a full-stack software developer and Microsoft MVP (Xamarin) from Holland. After years of experience working with Xamarin and .NET technologies, he has been involved ina number of different projects and has been building several apps. Not only does he like to code, but he is also passionate about spreading his knowledge - as well as gaining some in the bargain. Gerald involves himself in speaking, providing training sessions and writing blogs (https://blog.verslu.is) or articles in his free time. Twitter: @jfversluis Email: gerald@verslu.is Website: https://gerald.verslu.is

Thanks to Suprotim Agarwal for reviewing this article.