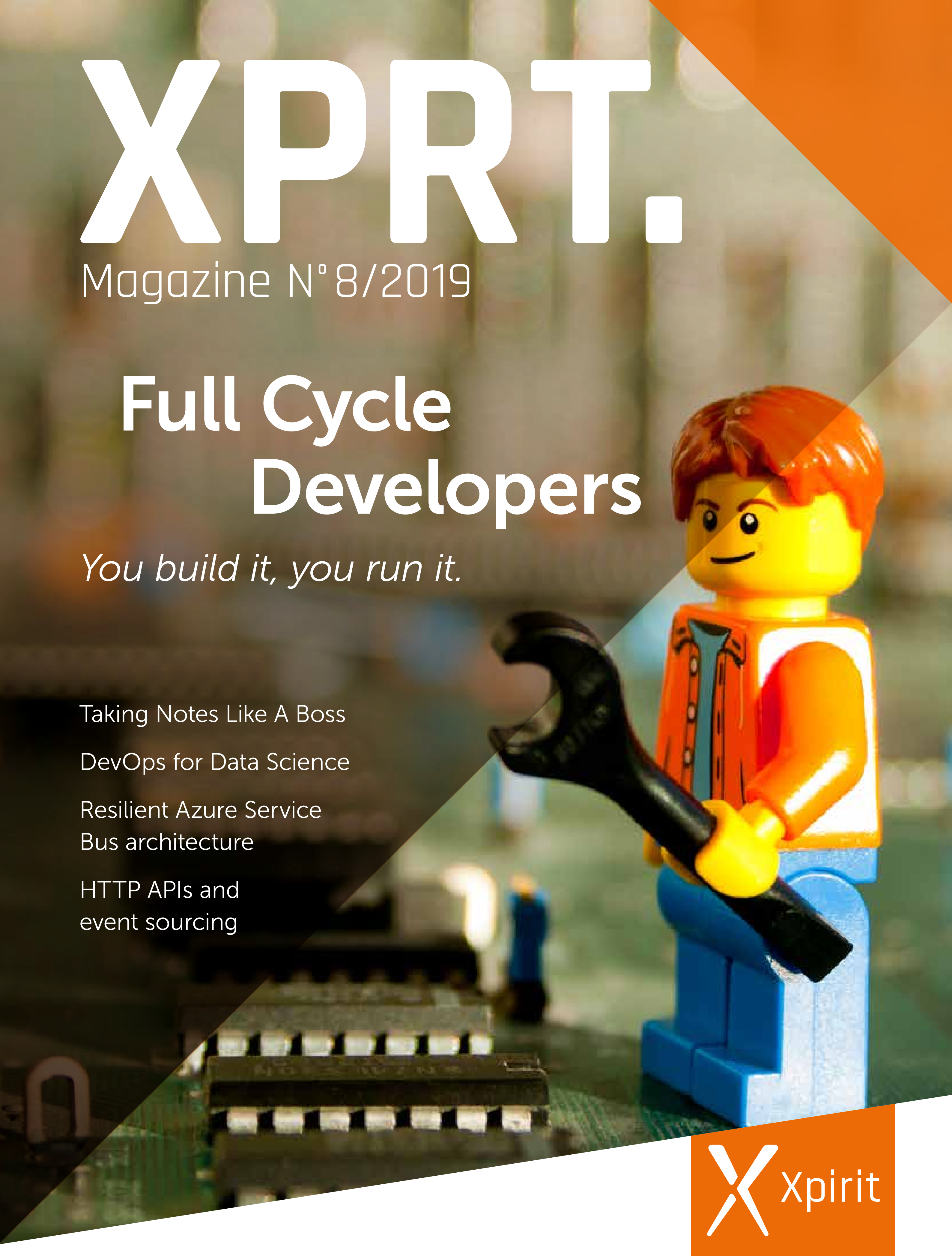
XPRT.

Magazine N° 8/2019



Full Cycle Developers

You build it, you run it.

Xpirit

PROUDLY PART OF XEBIA GROUP

Think ahead. Act now.

#GDBC

JUNE 15, 2019

YOU BUILD IT YOU RUN ~~AWAY~~ IT

GLOBAL DEVOPS

BOOTCAMP

BY XPIRIT & SOLIDIFY

GLOBALDEVOPSBOOTCAMP.COM

Colofon

XPRT. Magazine N° 8/2019
Editorial Office
Xpirit Netherlands BV

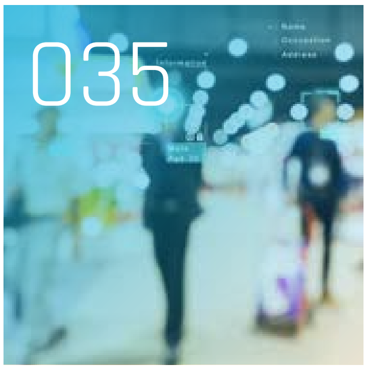
This magazine was made by
Vivian Andringa, Pascal Naber, René van Osnabrugge, Martijn van der Sijde, Loek Duys, Alex Thissen, Kees Verhaar, Geert van der Cruijsen, Sofie Wisse, Chris van Sluijsveld, Marcel de Vries, Pascal Greuter, Alex de Groot, Roy Cornelissen, Jesse Houwing, Marc Duiker, Sander Aernouts, Jasper Gilhuis, Rob Bos, Immanuel Kranendonk, Michiel van Oudheusden, Wendy van Veenendaal, Manuel Riezebosch, Marc Bruins, Reinier van Maanen, Maira Camu, Gill Cleeren, Max Verhorst, Pieter Gheysens, Jordi Borghers, Albert Starreveld

Contact
Xpirit Netherlands BV &
Xpirit Belgium
Laapersveld 27
1213 VB Hilversum
The Netherlands
+31 (0)35 538 19 21
pgreuter@xpirit.com
www.xpirit.com

Layout and Design
Studio OOM
www.studio-oom.nl

Translations
TechText

In this issue of **XPRT.** Magazine our experts share their knowledge about Full Cycle Development. You build it, you run it.



Creating impact by sharing knowledge; people first

We are proud to present the eighth edition of our XPRT magazine, designed and written by the same people who are the driving force behind the daily impact we create for and with our customers. Our magazine represents that impact, ranging from helping organizations to adopt DevOps in order to achieve digital transformation, to making architectures more resilient by using robust patterns and unit testing. Are you in control when experiencing turbulence in your production environment? Read our article about Chaos Engineering to find out.

With each new edition of our magazine, we try to share as much knowledge as we can, enabling as many organizations as we can to be more successful by using new Microsoft technology. Did you know that quite a large number of Data Science projects are executed without a valid ALM strategy? The article "DevOps for Data Science" explains how you can infuse your application with AI while still enjoying the benefits of DevOps. What are you waiting for?

Later this year, we will be celebrating our first five years of Xpirit. We would never have become what we are today without putting our people first. The great thing about letting our XPRTs create their own magazine is that a variety of topics get discussed, and this time these topics range from sketch noting, DevOps and resilience to Xamarin and other subjects.
Enjoy!

Pascal Greuter, Managing Director &
Max Verhorst, Commercial Director

Taking Notes Like A Boss

What is sketchnoting? Sketchnoting is all about capturing ideas, not about creating art. It's a way to think on paper using images and words. According to the creator of sketchnoting, Mike Rohde, when you are taking carefully hand-written notes and embellishing them with illustrations, you are sketchnoting. This way of visual thinking results in rich visual notes, mixing handwriting and drawing to create a more appealing set of notes.

Author Laurens Bonnema & Maira Camu

What is sketchnoting?
Sketchnoting is all about capturing ideas, not about creating art. It's a way to think on paper using images and words. According to the creator of sketchnoting, Mike Rohde, when you are taking carefully hand-written notes and embellishing them with illustrations, you are sketchnoting. This way of visual thinking results in rich visual notes, mixing handwriting and drawing to create a more appealing set of notes.

Why is it useful?
If you create visuals while listening and taking notes, the mind uses different capacities to process data and allows you to remember it up to 29% better. The physical action of taking notes combined with the creative action of visualization allows you to focus more and filter out the important bits.

For introverts like ourselves, creating sketchnoting is a kind of magic, especially at events. When people see you drawing, they feel invited to come over and have a chat about what you're doing.

When we work with clients, we have noticed it is usually more effective to show people what is happening rather than tell them. It tends to come across as less confrontational. This allows us to be very direct about stuff. We can be typically Dutch, without being overly blunt!

How we became sketchnoters
(Maira) "As a DevOps and test-automation consultant, I'm used to visualizing processes, pipelines, and other information, but I never considered myself good enough to be a real sketchnoter. I was going to attend the Web Summit in Lisbon and felt this was an opportunity to get started. Five days of drawing all these talks! My sister-in-law taught me the basics of sketchnoting in one afternoon, and I was off to the races! As the conference progressed, I improved my craft, and people noticed. They came over to have a chat. Some asked me whether I was hired by the conference to do this. Others just complimented me, then took a picture and shared my work online. And then, the conference organizers offered me a front-row seat at the press table, because they had seen my conference sketchnotes online and loved them."
@mairacamu

(Laurens) "When I encountered sketch-noting, it appealed to me, because as it turned out I had been doing it for the past decade, thinking I should really structure my "mind maps" more like the father of mind mapping Tony Buzan. Then I saw an awesome TEDx video by renowned graphic facilitator Rachel Smith from The Grove Consultants and encountered The Sketchnote Army. It dawned on me that what I had been doing had a name, it was not as strange and uniquely "me" as I had previously thought, and it was gaining momentum as a cool thing to do at conferences and meetings. I decided to get better at sketchnoting and went looking for ways to do so. I joined a Meetup group by Petra Hegenbart and read The Sketchnote Handbook by Mike Rohde. And then, I just started sketchnoting everywhere. At conferences of course, but also in meetings, presentations, and workshops."
@laurensbonnema

You are a sketchnoter too!

First, get some gear. Do not overthink sketchnoting, you most likely have everything you need within reach. Grab a pen to write with and take a marker to add some color and use a sheet of paper.

Start adding doodles – tiny drawings or sketches – to your notes. Use banners, headers, even draw the speaker using a stick figure. If you use color, stick to one tone or use two *complementary colors**. This will immediately make your work look more professional. Also try using different styles of writing, a calligraphy font combined with basic capital letters instantly looks good!

Favorite sketchnoter tools

If you are like us, you probably prefer to select "the best" gear for everything, including your sketchnoting gear.
If that sounds like you, these are our suggestions:

› Pens: a black fine-liner, gel-pen, or ballpoint. We use Artline drawing system fineliners size 0.3 to 0.7.
› Markers: a grey marker, and one or two complementary colors. We use Neuland Markers for this and absolutely love them. They are refillable, so not just good for you, but also for the planet!
› Paper: Moleskine, 120g A4/A3, flip-chart, plotter paper. We don't have a very strong preference here, except that the ink should not bleed on the paper, usually this means getting slightly heavier paper such as 90 or 120 grams.

We suggest getting one or two books about sketchnoting and visual thinking for inspiration and more guidelines.
Our recommendation would be Mike Rohde's The Sketchnote Handbook and The Sketchnote Workbook.

the Sketchnote WORKBOOK
→ADVANCED techniques FOR TAKING visual NOTES you CAN USE anywhere
by MIKE ROHDE author of The Sketchnote Handbook
the Sketchnote HANDBOOK
→THE ILLUSTRATED guide to visual NOTE TAKING←
by MIKE ROHDE the illustrator of REWORK

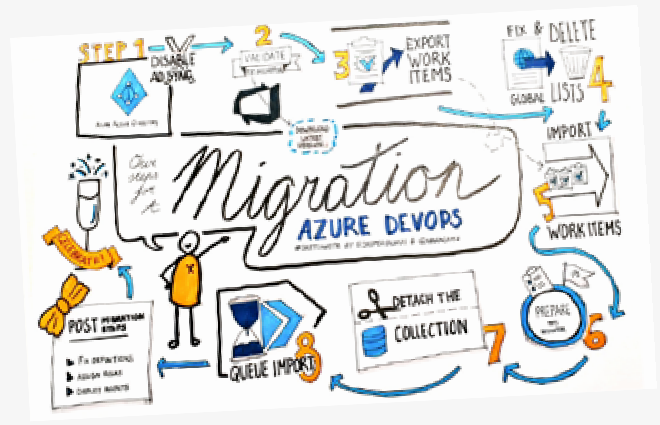
Basic Rules

› Everyone can do it, and no one draws like you.
› Embrace your mistakes, don't correct grammar or anything, just keep going and stay in the flow of drawing. (Sorry for the perfectionists out there)
› Never start over, it's about telling the story, so keep going.

Learn in public!

We have found that one of the best ways to get started is to learn in public. So grab your favorite pen, some paper, and create your first sketchnote! Then, just post it on Instagram or Twitter with the hashtags @SketchnoteArmy #sketchnote and #mairarocks to get instant feedback. You will be amazed of the positive vibe in the online community.

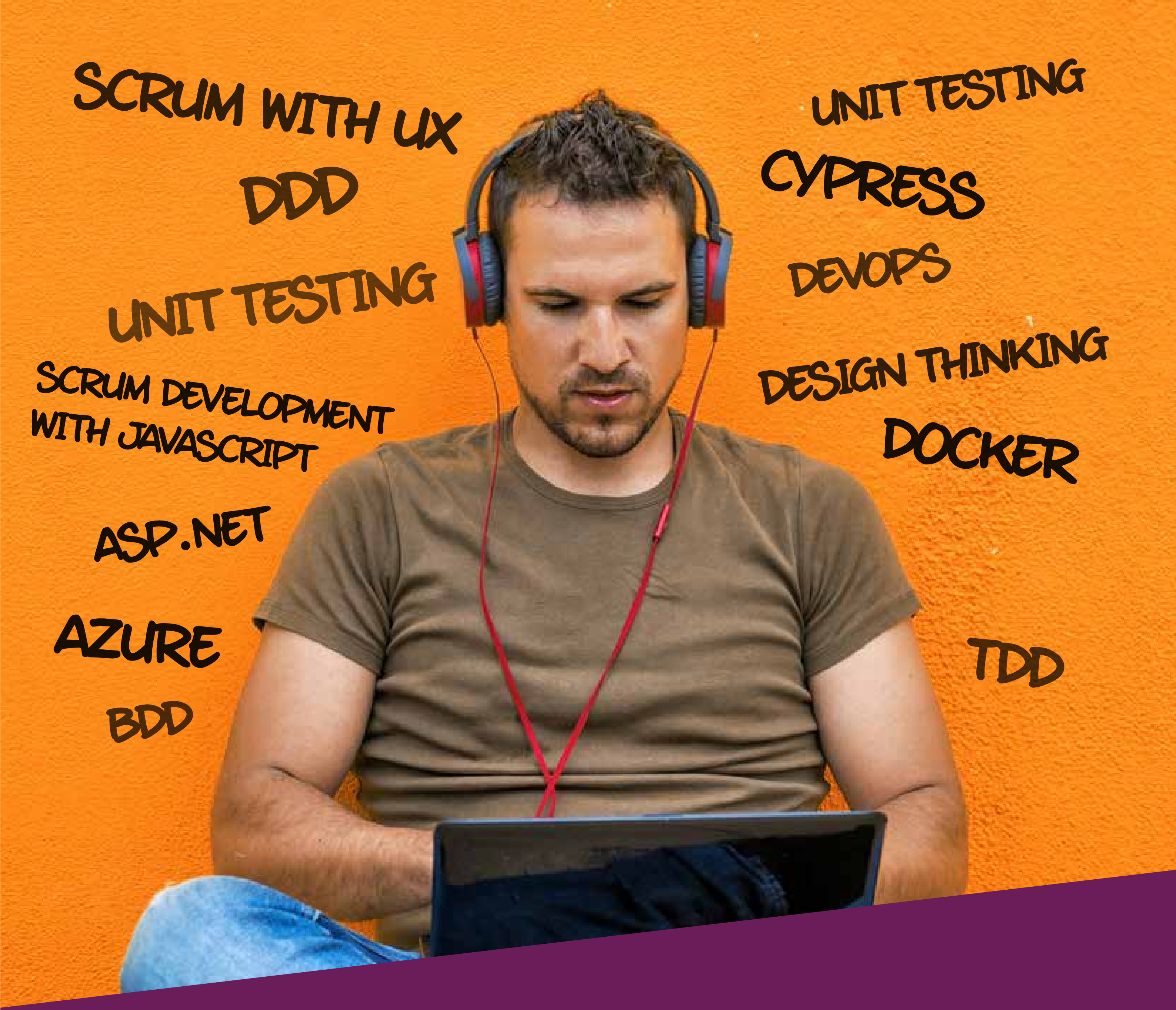
Pro-tip: People tend to take photos while you are sketch-noting, so make sure to sign your work before you start drawing with your @Twitter/Instagram handle and a *#hashtag*.



Need more encouragement?

Once you get hooked to sketchnoting and you want to learn more and get inspired, you can also advance your skills by joining us in *Agile Sketchnoting and Graphic Recording* course and we will do it together! In this training Laurens and Maira will help you learn and develop your skills! </>

* https://en.wikipedia.org/wiki/Complementary_colors
QR code link https://hubs.ly/H0gQG8Y0



Skill up for full cycle ownership

On your way to becoming a full cycle developer? There isn't just one route to full cycle ownership.

That's why Xpirit proudly joins Xebia Academy, so you can broaden your skill set from the best tools Microsoft has to offer to design, testing, deployment, and operations.

For every training you need
training.xebia.com

Xebia Academy

Enabling DevOps teams for Azure cloud solutions

Digital transformation enables companies to realize innovations and deliver products and services with higher quality in order to exceed customer expectations (better), reduce prices (cheaper) and shorten the time-to-market (faster). However, this transformation requires organizational as well as technological changes.

Authors Alex Thissen & Martijn van der Sijde

In this article we will explain an organizational and platform-agnostic technology architecture that helps in realizing these digital transformation goals. The second half of the article contains an example of an implementation of this architecture on Azure.

Adopting DevOps to enable digital transformation

The keywords for achieving better, cheaper, faster products and services are *flow* and *value*, and this is what a DevOps way of working aims to achieve. To help us focus on the creation of flow and value in the delivery of products and services, we use the DASA DevOps principles as guidance (see Figure 1).

These principles focus on organizations and the individuals in those organizations. They describe what actions, behavior and other aspects are required from these organizations in order to migrate to, or adopt a DevOps way of working. It would take too long to explain the principles in detail, but we will explain them briefly to be able to understand the reasoning behind the architectures described in this article. For more detailed information, please refer to the DASA website and resources.

The principles have been listed in their order of priority. The first one focuses on the creation of value for the customer, because this is a fundamental value for an

Principle 1

Customer-centric action
(Courage to act, innovate)

Principle 2

Create with the end in mind
(Product & Service thinking,
Engineering mindset, Collaborate)

Principle 3

End-to-End responsibility
(Live your accountability, Concept
to Grave, performance support)

Principle 4

Cross-functional autonomous
teams (T-shaped profiles,
complementary skills)

Principle 5

Continuous Improvement
(if it hurts do it more often,
experiment, fall fast)

Principle 6

Automate everything you can
(Enhance quality, maximize flow)

Figure 1: DASA DevOps principles (source: https://www.devopsagileskills.org/dasa-devops-principles/)

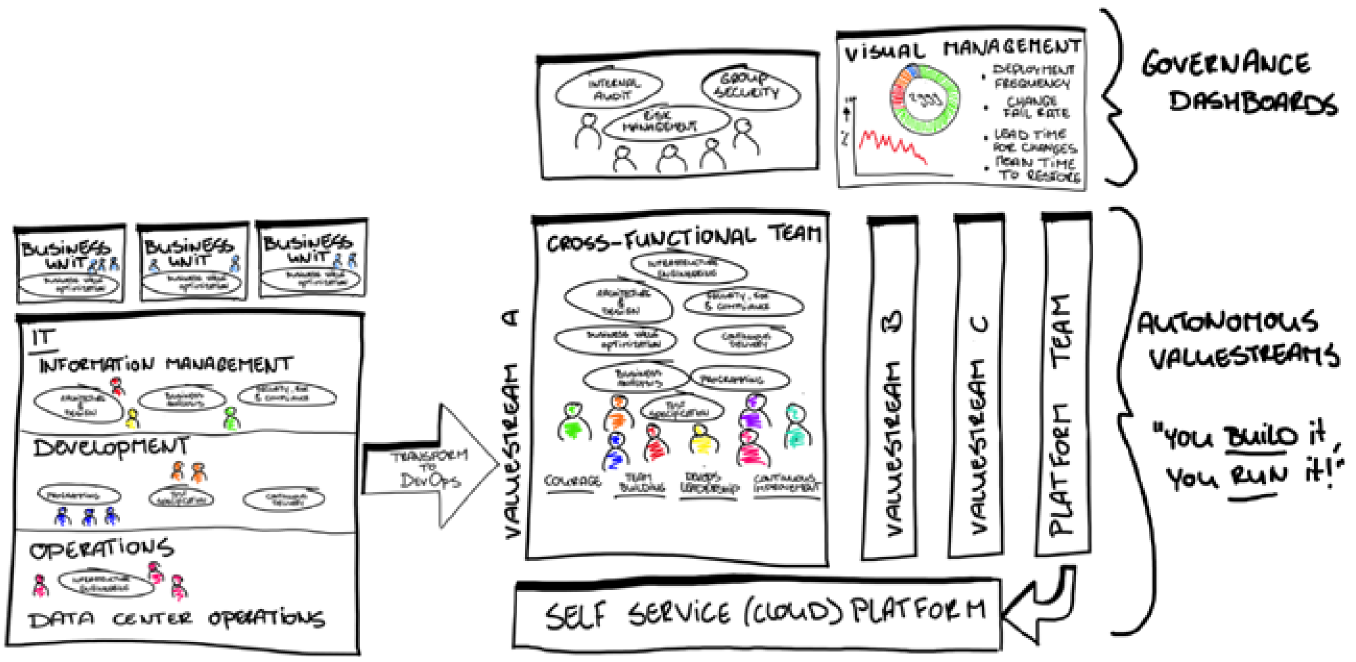


Figure 2: Organization architecture as-is and to-be

organization to be able to survive in this digital age. The second principle means breaking down siloes by structuring an organization around products and services instead of processes and subject-matter expertise. The third principle is saying: "You build it, you run it" and means that you are responsible for the products and services until they cease to exist. Achieving flow in work that needs to be done and decreasing dependencies between teams and individuals is what principle number four aims to achieve. To be able to change in incremental steps, a rigorous continuous improvement process must be adopted as stated in principle five. And last but not least, improving continually also means automating everything that is repetitive when possible. By doing this, principle six aims to increase quality and maximize flow.

In order to allow teams and individuals to apply these principles they must have, or grow towards, an organization that is geared towards increasing the flow of value to the customer. In addition, they also need the technical resources to support them in this mission. In the next two paragraphs we will zoom in on the organizational and technical architectures that enable this.

Organizationally enabling DevOps teams

When looking at an organization before a digital transformation, IT plays a supporting role to the business (see the left hand side of Figure 2). Business-units have a cross-backlog demand in order to get their required products and services to production. Development and operations are separated into a change and run organization among other siloed organization structures. This way of organizing has a negative impact on the optimal flow in the software delivery value chain because of organizational and technical dependencies. In addition, the siloes cause hand-over moments and loopbacks in the delivery process, which is also inefficient.

In the new situation (see right hand side of Figure 2), the teams are organized around autonomous business capabilities, which means that they can develop their products and services without disturbing, or being disturbed, by other developments. The teams are cross-functional, making them capable of developing and running their products and services until they are no longer required. Technically they are supported by a self-service (cloud) platform which enables them to rigorously automate and quickly innovate by incorporating new platform services to their offering. This is done in an incremental, continuous improvement way of working.

Technically enabling DevOps teams

In order to provide teams with the ability to operate and act according to the DevOps principles, a high and mature level of autonomy and agility is required. This poses requirements and constraints on the technical architecture, but also on the way governance and control is achieved.

DevOps teams will need to be able to have end-to-end responsibility for their value proposition and the corresponding implementation of architecture and applications. Nowadays most teams are very capable of doing this for traditional software development, focusing on delivering the application.

A digital transformation that incorporates a cloud platform presents new opportunities. The practices need to embrace a similar approach for the infrastructural part of the software solutions as well. DevOps teams are facilitated in this by a self-service cloud platform (see Figure 3).

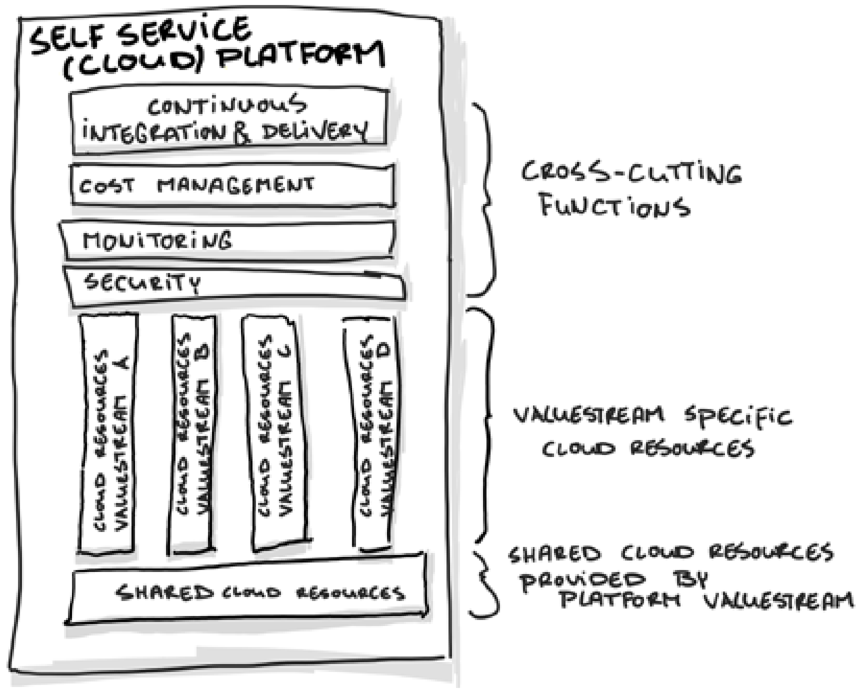

SELF SERVICE (CLOUD) PLATFORM
CONTINUOUS INTEGRATION & DELIVERY
COST MANAGEMENT
MONITORING
SECURITY
CLOUD RESOURCES VALUESTREAM A
CLOUD RESOURCES VALUESTREAM B
CLOUD RESOURCES VALUESTREAM C
CLOUD RESOURCES VALUESTREAM D
SHARED CLOUD RESOURCES
CROSS-CUTTING FUNCTIONS
VALUESTREAM SPECIFIC CLOUD RESOURCES
SHARED CLOUD RESOURCES PROVIDED BY PLATFORM VALUESTREAM

Figure 3: Self-service (cloud) platform logical architecture

Achieving full autonomy through infrastructure-as-code

Modern cloud platforms allow complete automation for provisioning the resources they offer. This automation enables a team to treat the infrastructural aspects of a software solution in the same way as the implementation of business and customer functionality. This practice of "Infrastructure as Code" creates scripts and templates in a cloud-platform specific format that is maintained, tested, and uses build and release pipelines like conventional code. This flow will allow a team to automate all aspects of the application parts they need to build and host on cloud resources. It also allows the team to continuously improve the infrastructure, because it becomes trivial to remove existing resources after changes and to reprovision them, instead of having to change and maintain previously deployed resources. A team can become completely autonomous when it is able to achieve this level of automation in cloud solutions. The team can provision and deprovision resources for these solutions as value-stream specific cloud resources (see the Cloud resources value stream A-D in Figure 3).

Maintaining governance and control with full-autonomy

The biggest challenge in delivering value after adopting the full automation of infrastructure involves the governance and control over the cloud platform. A self-service cloud platform should give autonomy and agility to teams, and also provide the appropriate level of governance and control to an organization. While the team needs to be able to be agile, move fast and be independent, the organization needs to be compliant and provide traceable processes and be in control of costs and security of the cloud platform hosting. Fortunately, cloud platforms offer various features to have this level of control while still providing self-service capabilities to the teams. The whole purpose of this approach is to enable the value stream teams to provision their own cloud resources, within the constraints offered and required by the organization. The anti-pattern to this is creating a single point in the organization, such as a team, where the teams have to request and acquire the cloud resources they need. However, having a single point of administration will block and slow down teams that want agility and speed.

The self-service cloud platform should offer cross-cutting functions (see Cross-cutting functions in Figure 3) by using the cloud intrinsic features for monitoring, cost management, and security. Each cloud platform implements these features in a different manner. From a cloud-agnostic point of view, the monitoring features should allow both teams and the organization at an aggregate level to monitor the health and security of the hosted solutions. This includes resource utilization, ownership of resources, and active security status, to name a few.

Additionally, a shareable set of cloud resources can provide a layer of structure and boundaries for the teams, on top of which they can build their solutions (see the Shared Cloud Resources in Figure 3). This can range from shared security features to networking topologies that make sure that certain quality and safety standards are being met automatically by the teams.

Limiting access to resources

Another aspect to consider is the use of authorization and role-based access control. Using these security aspects, it is possible to limit the rights of principals to create cloud resources or certain types thereof. In lieu of the full automation, compliance and security, one could go as far as removing all rights from regular user accounts, except read-only access. The rights to create and manage resources is only given to service principals (non-human accounts) that are assigned as the identities for build and release pipelines. This forces the use of the pipelines for resource management and disallows direct manual intervention, increasing compliancy and traceability of the cloud solution as well as the level of security. The strict authorization can be applied to all environments or at least the critical ones, for instance production. Since no human can make changes and everything is automated, approval processes can be simplified to check for the proper use of blessed templates and scripts. In time it might become apparent that approval is not even required anymore. At that point removing the approval altogether will increase the agility and speed for the value streams even further. The use of strict authorization should be applied with caution though, as it can severely limit the teams when applied too rigorously, and effectively take away the necessary privileges for a team to be able to self-service its cloud resources.

Transition to self-service with a cloud platform team

A dedicated team can help during the transition to the self-service cloud platform. This "cloud platform team" can accelerate the cross-team self-service features and functionalities. The purpose for this team is to implement the cross-cutting functions and the shared resources, as well as guidance for the teams and help during adoption and transitioning to the self-service platform. The team can create "blessed" templates and scripts for the teams to use in provisioning value stream-bound resources. These automation artifacts have been tested and security hardened to make sure the security baseline for the teams is met by default when they are utilized in provisioning build and release pipelines. The underlying shared resources give the teams the harness of enough freedom while maintaining a secure and compliant implementation for the cloud infrastructure.

The cloud platform team is a temporary team and should dedicate itself to delivering the cross-cutting and shared features, onboarding the value stream teams, and making themselves redundant.

The value stream teams can take over the responsibilities of the cloud platform team as a community effort. Since all provided features are treated like code, the way community contributions are made can work the same way for the delivered infrastructural artifacts.

In addition, they lead by example in showing the behavior and mindset that is required for the new way of working. Please note that the impact on cultural change should not be underestimated. To be successful, it can even be beneficial to add a dedicated coach to the platform team to accelerate this change.

Alex Thissen & Martijn van der Sijde

Implementing a self-service cloud platform on Azure

The first half of this article explained what is required organizationally and technically to maximize the creation and increase of flow of value to the customer. This second half contains an example of an implementation of the technical architecture in Azure.

The Microsoft Azure platform accompanied by Azure DevOps (previously known as Visual Studio Team Services) is well suited to implement the self-service cloud platform. Azure offers advanced resource management and monitoring capabilities. Its automation engine is called Azure Resource Manager (ARM), which can be automated by using ARM templates or the Azure Command-Line Interface (CLI). Either of these allow full automation of provisioning and managing Azure resources.

At the highest level, Azure uses the notion of an enterprise and subscriptions. The enterprise is a representation of the organization that uses Azure, and its subscriptions are administrative units of ownership and rights. The subscriptions align well with the value streams, where each team can be an owner or contributor, depending on whether full or nearly full management rights are allowed. The resource management in Azure is governed by security policies at various levels. From data plane to control plane you can define authorization at a coarse and very fine grained level. By giving the teams respective rights, they can create all resources anywhere within the subscription, or within resource groups as contributors. The latter is a way to allow teams to create resources in a more controlled way, because additional permissions can be set at a resource group level. It avoids giving the teams full administrative rights to the subscription.

For the cross-cutting functions Azure has several features offering the monitoring, compliance, security and cost management capabilities required. Azure Monitor, Azure Security Centre, and Azure Cost Management are ready-to-use features that combine information gathered from and across the subscriptions for the value streams. The governance and compliance can be taken care of at this higher aggregation level. Azure DevOps, even though not part of the Azure cloud platform per se, is the single point of arranging the build and release pipelines for provisioning. It can provide the full end-to-end traceability for compliancy reasons, from code to hosting environment. Azure DevOps combines source code management with work item tracking and pipelines to environments after approval and passing quality gates. Leveraging these features allows teams to stay compliant because every change to code and environments is tracked and audited in Azure DevOps.

The next example in Figure 5 illustrates how shared cloud resources can be used to provide a secure default self-service cloud platform. The general idea of the scenario in the example is to allow the teams to provision web resources, while still keeping control over public availability and securing their resources. The intent is to give freedom and protect against unwanted disclosure and exposure of internal network-reachable resources.

Each value stream and team is given their own subscription. Within these subscriptions virtual private networks are created to isolate value streams from each other. Hosting plans are created inside the subscriptions and the team can provision web apps as they see fit. The design of these web apps does not allow any outbound connectivity. This avoids exposing anything immediately after creation and provides a secure, default approach.

To be able to release web applications for the first time, changes need to be made at the shared resources level. While this is blocking to some extent, it does provide control in terms of which web application is allowed access to the public internet and when. This provides an opportunity to make sure that only approved and validated web resources are disclosed. It only has to happen once during the initial release, so it should not be a big nor lasting hurdle in the value stream flow. The technical implementation for retaining access over public facing web applications is the Web Application Firewall and Gateway. This Azure resource has to be configured so that it allows inbound and outbound HTTP and HTTPS traffic, all by automated scripts, and obviously after approval. By keeping this resource at a shared and governed level, the organization retains its ability to have control over web-exposed solutions, while giving the teams freedom to create any resources up to the point that they need to be released externally.

One other security measure in the example is the use of Application Service Environments (ASE).

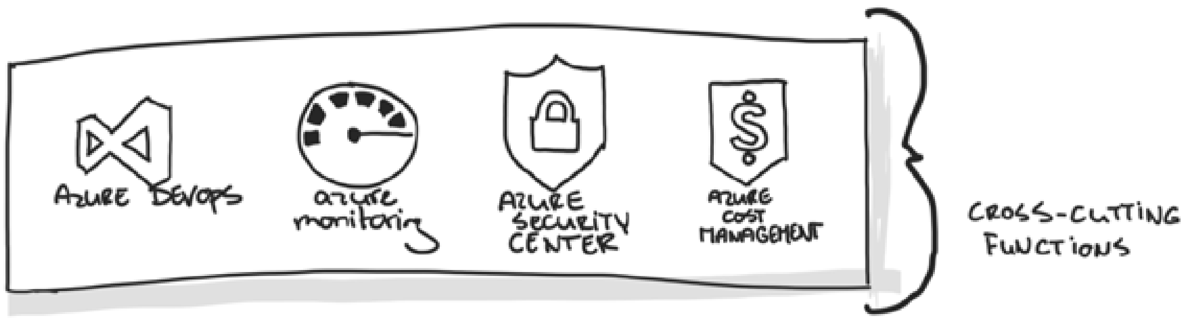


Figure 4: Cross-cutting functions in Azure

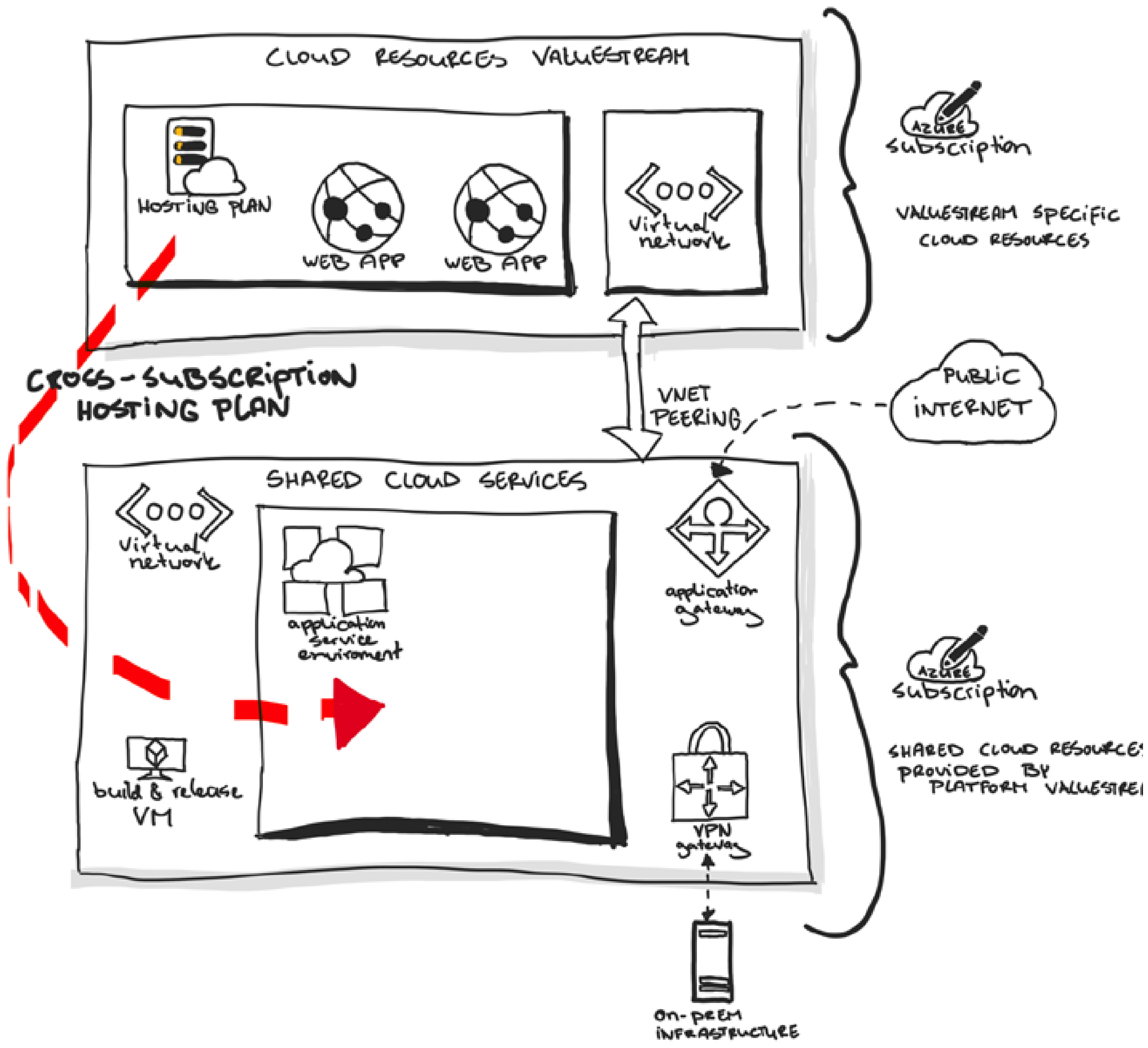


Figure 5: Example self-service platform hybrid-architecture implemented on Azure

The scenario shows connectivity to an on-premise infrastructure via a Virtual Private Network (VPN) Gateway. It is non-trivial to create a VPN tunnel to on-premises networks. Keeping the connection at a shared resource level makes it reusable over the various value streams and teams, in addition to providing a single point of entry into the on-premises network. The VPN Gateway provides another control mechanism for securing access to the on-premises resources by specifying advanced access rules for allowed network traffic to and from it.
Each value stream Virtual Network is given a peering to the shared virtual network that includes the VPN and Web Application Gateways.

The shared resources are created by the initial cloud platform team, which behaves and operates like any of the other value streams. While the team still exists, it provides a different value stream, consisting of the self-service platform's shared resources for the other teams and value streams to utilize. In a similar fashion, the underlying VNETs and peerings are also not created by the teams themselves, but by the platform team instead.

Summary

Companies with the aim to deliver better and cheaper products and services in a faster way need to make a digital transformation. They should adopt or migrate to a DevOps way of working to increase the flow of value to the customer. To achieve this, organizational and technical changes are required to enable teams and individuals. A temporary cloud platform team can help to make the transformation happen. The technical resources can be implemented in Azure, as shown in the example of a self-service platform in Azure. When both aspects (organization and technique) are applied in coherence, a company's teams and individuals are lined up to achieve the digital transformation goals. </>

DevOps for Data Science

As a reader of this magazine, you'll be familiar with the concept of DevOps: closing the gap between all disciplines involved in software engineering, and enabling continuous delivery of value to your end users. This sounds simple enough, yet it proves to be very hard in practice. In a typical software delivery environment, there are many moving parts which all need to work together – organizationally as well as technically – to be effective.

Authors Kees Verhaar & Rob Bos

To make matters worse, modern applications include Machine Learning or Artificial Intelligence components. These require a particular skillset, typically embodied in a Data Scientist. They use tools unfamiliar to the typical .NET developer and follow a development cycle that differs from what a .NET developer is used to.

In this article, we will explore the DevOps process for an app that includes an Artificial Intelligence model. In the next issue of XPRT magazine, we will implement this process in a real-world example.

Build – Measure – Learn for Application Development

A typical DevOps process entails three significant parts: Build, Measure & Learn, which comprises the typical DevOps cycle as shown in Figure 1.

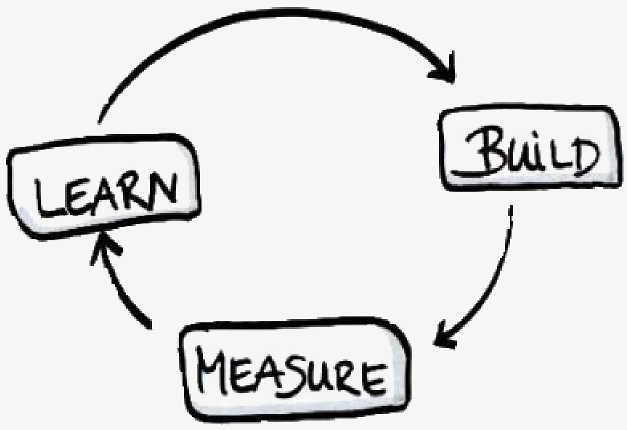


Figure 1: The DevOps cycle (source: https://innovationorigins.com/nl/startups-op-zoek-naar-een-prototype-perfect-kan-de-vijand-zijn-van-goed-genoeg/build-measure-learn/)

In "Build", we develop our application: we gather requirements, we translate them into code, we compile, deploy, test, and we release it to production. Then, we "measure": is our application running? Is it performing the way we expect? How are our users using the app? Finally, we evaluate our measurements and extract "learnings" from it. How can we improve user experience? What should be the next feature we work on? Do we need to work on stability? We feed these learnings back to the beginning of our loop (the "Build" part), and we continuously repeat this cycle to improve steadily.
In a typical .NET world, the process to implement this cycle looks similar to what is shown in Figure 2.

Now, what happens if we want to infuse a little Artificial Intelligence (AI) into our application?

Build – Measure – Learn: The Data Science way

To understand what is required to incorporate AI into our application, we must first understand the development cycle of an AI model. In "traditional" application development, you write code and an app comes out. In Data Science, this works slightly different. The result of the work of a Data Scientist is a model. This model has inputs and outputs, depending on what the model was built for. The model could be designed to detect anomalies in a continuous stream of data (for example to detect impending server outages based on operational metrics) or to recognize faces in a photograph. It could be anything. Three factors are deterministic for a model:

- **Training Features:** a set of variables that are generated from the raw data and are used to train the model.
- **Model structure:** for instance a linear regression model, a decision tree model, or a random forest model.

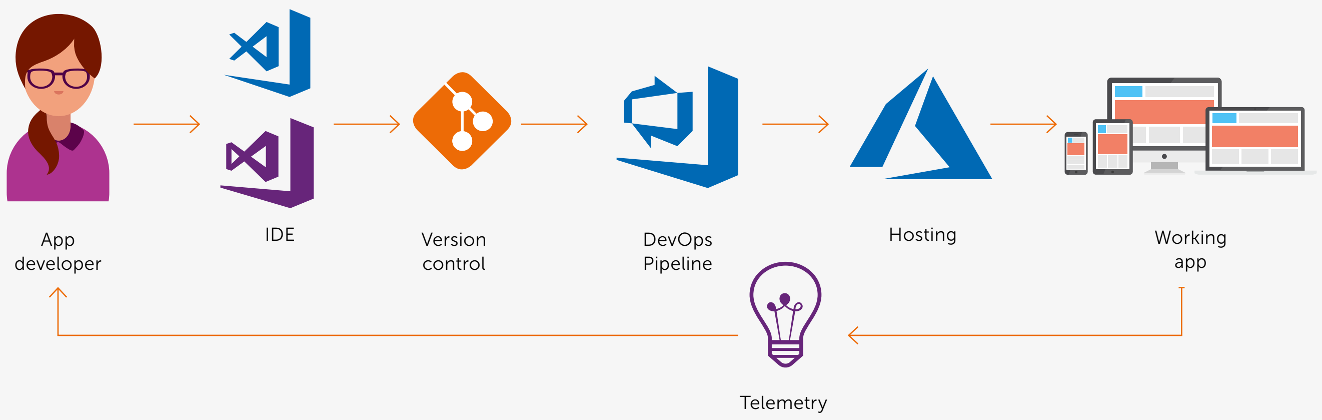


Figure 2: Implementation of the typical DevOps cycle for application development

- **Hyperparameters of the model:** for example, for a random forest model, how many trees, the maximum depth of each tree, and the minimum number of observations in each leaf node. Or for an artificial neural network model: the number of hidden layers, how many hidden nodes, the activation function, learning rate, and random seed.

These three factors are determined and tuned until the model satisfies its required performance. A typical Data Science workflow is shown in Figure 3.

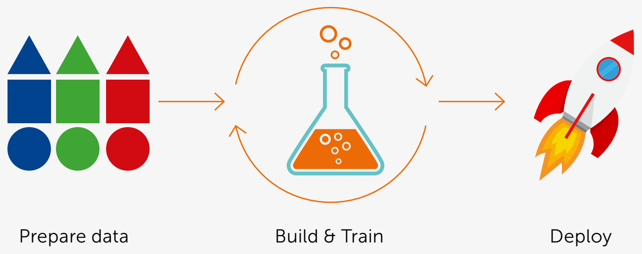


Figure 3: A typical Data Science workflow

Joining both worlds

When both workflows are combined, we arrive at a process that is similar to the process shown in Figure 4.

The key component that ties the world of the app developer and the Data Scientist together is a Model Management Service, which helps track model versions, performance, and deployments. Let's go over three critical DevOps pillars to see what should be considered there: Traceability, Automation, and Feedback.

Traceability

As we have seen before, three factors are deterministic for a model: training features, model structure, and the hyper-parameters used. The first step is to determine these factors and then tune them until the model has the performance (high enough accuracy and low enough errors) to satisfy your criteria.

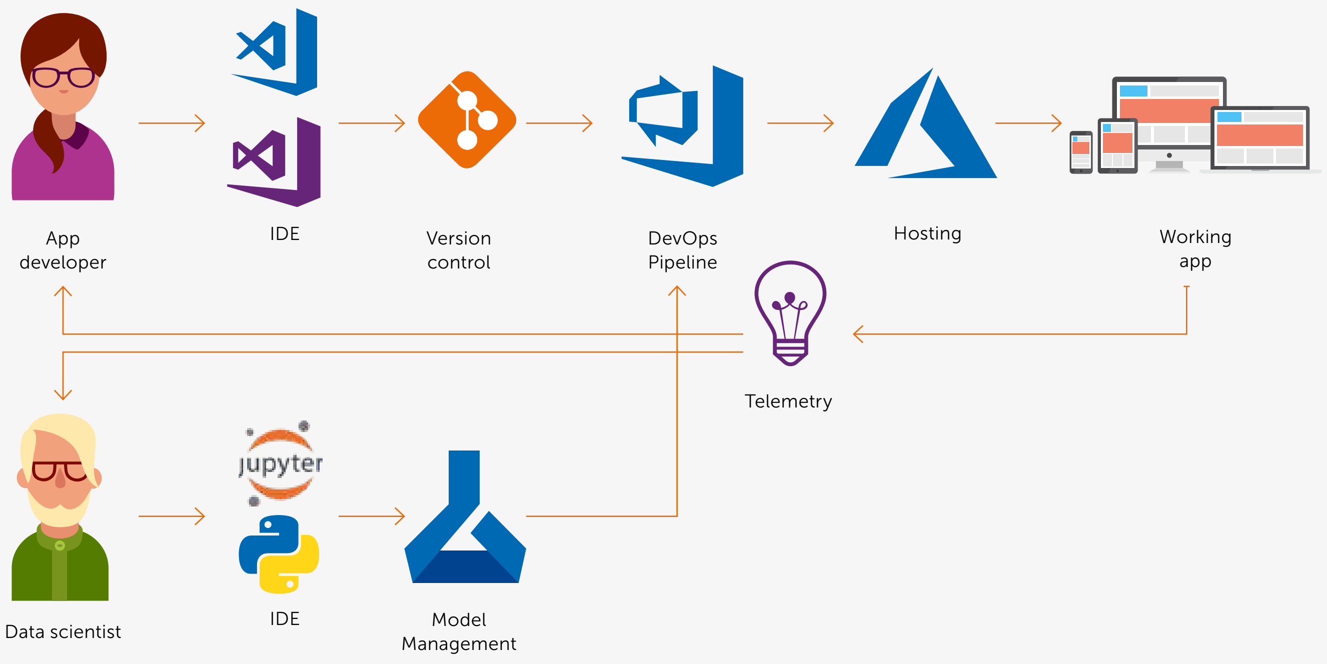


Figure 4: Implementation of the typical DevOps cycle for application development combined with Data Science

Because the steps during the modeling phase have a profound impact on the accuracy of the model, it becomes essential to have traceability of design choices. If you can't provide a clear overview of the history of the model, it will become a black box that makes some predictions without a way to validate its choices. This can have real repercussions when the model's prediction is being used to make critical decisions. Think for example of a healthcare situation, where you are predicting the probability that a patient is suffering from a particular disease, based on a set of symptoms. This could lead to a life or death decision and makes it very clear why humans still need to verify predictions and conclusions. To do so effectively, they need the entire context.

Training features

As an example, say that you are predicting the necessity of a hospital in a city and you decide to calculate that necessity based on the average age of the population in the city and the average traveling time they would have to the hospital. You split the age data into 70% training data and 30% validation data. By doing this for the entire set of residents, you didn't account for the fact that the necessity of a hospital is strongly correlated to the average age of residents in a specific area. The city in our example happens to have a very distinct set of age groups living in a particular area. The city can be split into three different areas:

- Area 1 has 10.000 residents, all in the age group of 30-40
- Area 2 has 20.000 residents, all in the age group of 20-30
- Area 3 has 30.000 residents, all in the age group of 40-50

You can see that by randomly splitting the full data set and using that data for training a model, you will skew the prediction, as half of the dataset actually has an age group of 40-50.

This example shows how easy it is to get a bias into your model by choosing to use the full dataset and forgetting to check the grouping of the features in the dataset. These decisions are usually made during a data discovery phase where you search for the properties of the dataset that are relevant for training the model. By making sure you have the setup of the model in source control with descriptive commit messages, you can keep track of the reasoning behind the choices you had to make with that dataset. This also allows you to set up checks for the dataset that you can later reuse to reevaluate those choices when new data is available. By doing so, you open up the black box, and you obtain visibility in the decisions and underlying reasons for them.

Model structure

By having the code of the model available in source control, you can also experiment with different algorithms and document their accuracy on the dataset you are using. You can record the outcomes and include them with the code you use in the final model. If you set up this process in the right way, you can use its documentation for "release notes" that contain the full research steps and reasoning behind the choices leading to this version of the model.

Hyperparameters of the model

Hyperparameters of a model are the settings you use during the training phase of the model creation. Take a decision tree algorithm for example. A regular tree will loop through the data and decide how much a specific feature will impact the desired outcome. See Figure 5 for a visualization of the steps taken to determine the income of a person, based on the features we fed to the algorithm.

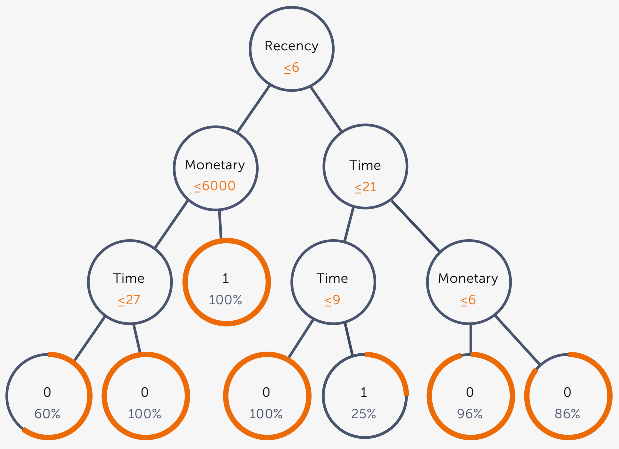


Figure 5: Example of a decision tree

For this algorithm, you can change the number of trees to use, the maximum depth of each tree and the minimum number of observations in each leaf node. Changing these settings can have a significant impact on the model, and on the time it will take to train the model. It is paramount to keep track of the parameters that were used for training and the impact they had on the resulting model. You need to store the values and the outcome in a separate store, where you can link the settings, the outcome, precision, loss values, etc. to the code used to determine this.

Automation

Automation is critical when employing DevOps. It fosters speed, greater accuracy, consistency, reliability, and increases the number of deliveries. When thinking of automating steps in building AI models, we should consider two principal parts. The first part involves the automation of preparing training data, training the model and evaluating the model's performance. The second part consists of the automation of integrating the model into your application and deploying this. The first part is implemented in a Machine Learning (ML) pipeline, while the second part is performed in a DevOps pipeline. The ML pipeline is the domain of the Data Scientist, while the DevOps pipeline bridges the gap between the Data Scientist and the app developers.
The nature of building an ML model sets some specific requirements for automation tooling in this space.

- **Compute:** an ML pipeline consists of computational steps. A lot of these steps (especially the steps of preparing data and training the model) are very computationally intensive. Automation tooling should, therefore, make it easy to distribute execution of these steps across large clusters of machines, so that execution time stays within acceptable time constraints.

Kees Verhaar & Rob Bos

- **Tools:** building an ML model requires specific toolkits and frameworks, most of which are very different from what we are used to in .NET application development. Python, TensorFlow and SciKit-learn are just a few examples of what a Data Scientist uses daily, while a .NET developer might not be so familiar with these. Automation tooling for ML models should seamlessly work with these tools, to make it easy for a Data Scientist to use without having to learn an entirely different toolset.
- **Traceability:** when automating steps in the model development process, you'll most likely produce a lot more model versions. Traceability will become more important than ever, letting you know which inputs led to which model output, and allowing you to decide (or automate) which model version should be deployed to production. Automation tools should offer seamless integration with other tools used in your development process so that traceability is guaranteed.

When selecting automation tools for automating your ML pipeline, you should carefully consider the above-mentioned three factors. When targeting the Azure platform, the Azure Machine Learning Service is the obvious choice. In our next magazine, we'll show you how to create an ML pipeline using this service.

For bridging the gap between the Data Scientist and the app developer we need a DevOps pipeline. This will be very similar to what we are used to from a .NET development world, except for the fact that it will gain one extra responsibility: integrating the correct ML model version into the application. For this, the DevOps pipeline will need to interface with the model store. The model store (part of the Model Management Service in Figure 4) contains all model versions along with the metadata describing (amongst others) model performance. With defined criteria (e.g. "model with greatest accuracy") the DevOps pipeline can select the correct model from the model store and integrate and deploy it.

Feedback

Implementing the feedback loop for Data Science looks a lot like implementing that loop for application development: you want to see how the model performs in the real world, evaluate it with the prior assumption and adjust it when necessary. Let's consider three examples: operational information, new training data, and reinforcement learning.

Operational information

Monitoring operational information answers questions like: how well are the predictions you have made followed? Or, how many times is that prediction correct? And thus, how many times is that prediction not correct? You can observe this by logging the prediction made with all of its contexts and linking this to the action that was taken based on the prediction.

New training data

You also need to send new data that is available in the system through the model training. Trends can always change over time, especially if you have a prediction that immediately influences decisions. If you tried to predict the demand for a product on the Monday of a specific week and based on that prediction your company delivers less of that product, it can very well happen that you find that your prediction had an impact on the number of sales for that product on that day. Of course, this is a self-fulfilling prophecy, since you cannot sell what you do not have.

To enable re-evaluation of your model you need to have a way to send in more data through your model, with all the necessary data preparation steps automatically executed. From there you can measure how much better or worse the performance of the model is compared to the previous dataset. It could very well be that you need to verify whether previously made assumptions and decisions are still valid.

Reinforcement training

With reinforcement training, you enable the end-users of the prediction to give feedback about it. They can indicate whether your prediction was correct or not, and how they determined this. By sending the new and updated "label" (the value that you are trying to predict) on that same information back into the build-measure-learn loop, you provide the algorithms with more information so it can adjust if necessary.

OK, so now what?

By now, you should have an idea of what it takes to incorporate Data Science model development into your DevOps cycle. Data Science is different from application development: it requires a specific skill set, model development requires a different process, and Data Scientists use different tools. However, the things that are important in DevOps are just as applicable to Data Science as they are to application development. We have shown the considerations that come into play here as well as a general direction on how to solve them.
The next step is to figure out how to implement this. What tools and techniques do we need to create a DevOps setup for an AI infused app? In the next issue of XPRT magazine, we will show you exactly that, so stay tuned! </>

Serverless and kubernetes, introduction to the virtual kubelet

If you talk about Kubernetes and serverless, there are two ways to look at this. First is the serverless programming model that is often referred to as Functions as a Service (FaaS). The second way to look at this is that we have a kubernetes cluster which has no servers that service the cluster. In this latter situation, you could use a concept like Azure Container Instances (ACI), Azure Batch, or AWS Lambda to serve the requests that come in on the Kubernetes cluster to deploy a container in a POD on the cluster.

Author Marcel de Vries

In this article, I want to introduce you to the virtual kubelet and what new capabilities this unlocks in a kubernetes cluster. This allows us to create a serverless cluster with nodes that are backed by ACI or Azure Batch.

What is a kubelet?

Let me start with a short explanation of the role of the kubelet in a kubernetes cluster. The kubelet is the agent that runs on a node to manage the lifecycle of pods. The kubelet runs as a service on a node. A pod is the unit of scheduling in the cluster and consists of one or more containers that are deployed together on one node. Most of the time a pod contains one container. The kubelet uses Docker to actually manage the lifecycle of the containers that are in a pod.

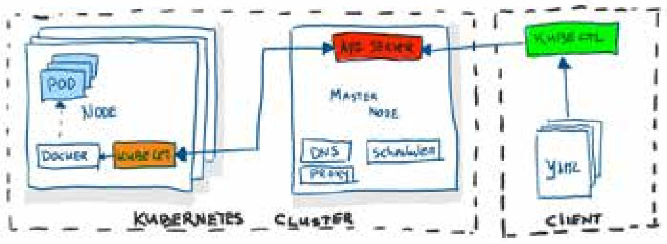


Figure 1: Kubernetes high-level architectural diagram

When a kubelet service starts on a node, it will register itself at the kubernetes API server as an available node to schedule pods on. From that moment onwards, the scheduler in the cluster can start assigning pods on that node. Scheduling a pod on a node is nothing more than assigning that pod to a node name which equals the name of the node. That name was given the moment the kubelet registered the node on the API server. In its turn, the kubelet service watches for these pod assignments by querying the API server. When it recognizes a pod with the assigned name of its node, it will start the containers which are part of that pod, using the container services running on that server. Often this is Docker.

What is a virtual kubelet?

Now that we know the role of the kubelet, let's look at what a virtual kubelet is. A virtual kubelet is a pod that contains a container which will behave as a kubelet. When you schedule this pod in the cluster it will register a node in the cluster. This is not a real node in terms of a normal virtual machine or physical server, but it serves as a virtual node on which you can schedule pods. The virtual kubelet uses a provider to do the actual scheduling of the containers that are part of a pod. The virtual kubelet project on GitHub[1] already contains different implementations of providers that can

[1] https://github.com/virtual-kubelet/virtual-kubelet

be used by a virtual kubelet implementation. For instance, providers are available for scheduling pods on AWS Fargate, HashiCorp Nomad, Service Fabric Mesh, Azure Batch, and Azure ACI. The virtual kubelet manages the lifecycle of the pods, just as a normal kubelet on a "real" node would do. The provider manages the actual lifecycle by working with the underlying infrastructure that provides the real container instances on the service it is built for.

So a virtual kubelet is a pod that you can schedule on your kubernetes cluster, which registers as a node on which you can schedule pods. The underlying provider used in the specific implementation of that virtual kubelet then manages the pods.

The following diagram shows how this all works together when you use the virtual kubelet that uses the ACI provider on Azure:

For the rest of this article, I will use the Microsoft Azure ACI provider, where the pods will be scheduled on Azure Container Instances.

How can I register the virtual kubelet with ACI as the provider?

When you have a running Kubernetes cluster like Azure AKS, it is rather easy to install the virtual kubelet. This is streamlined with the azure command line interface. Before you can install the virtual kubelet in the cluster, you need to install the tool Helm.

Helm can be seen as the package management solution for Kubernetes, just like NuGet is a package management solution for .NET application development. Helm uses a so-called Helm chart that contains the information on how the package needs to be deployed in the cluster. This means you can install a Helm Chart in a Kubernetes cluster. You can compare this to doing a NuGet install, where you download the right data, YAML files in this case, and then apply these to your project (in this case the cluster).

Helm is used to install the virtual kubelet. Hence you need to install this first. Next, you can run the command line to install the kubelet with the following command:

```
az aks install-connector --connector-name mycon --os-type
Both --resource-group <ResourceGroup> --name <ClusterName>
```

One thing to note is that the virtual kubelet is named "install-connector" in the Azure command line. This install-connector command results in the virtual kubelet pods to be scheduled on one of the available nodes in your cluster.
After running this command line you can ask the cluster which nodes are available. This is done with the following command:

```
Kubectl get nodes
```

On my kubernetes cluster this resulted in the following information:

Name	Status	Roles	Age	Version
aks-nodepool1-34126871-0	Ready	Agent	46d	V1.9.11
virtual-kubelet-mycon-linux-westeurope	Ready	Agent	1h	v1.13.1-vk-v0.7.4-44-g4f3bd20e-dev
virtual-kubelet-mycon-windows-westeurope	Ready	Agent	1h	v1.13.1-vk-v0.7.4-44-g4f3bd20e-dev

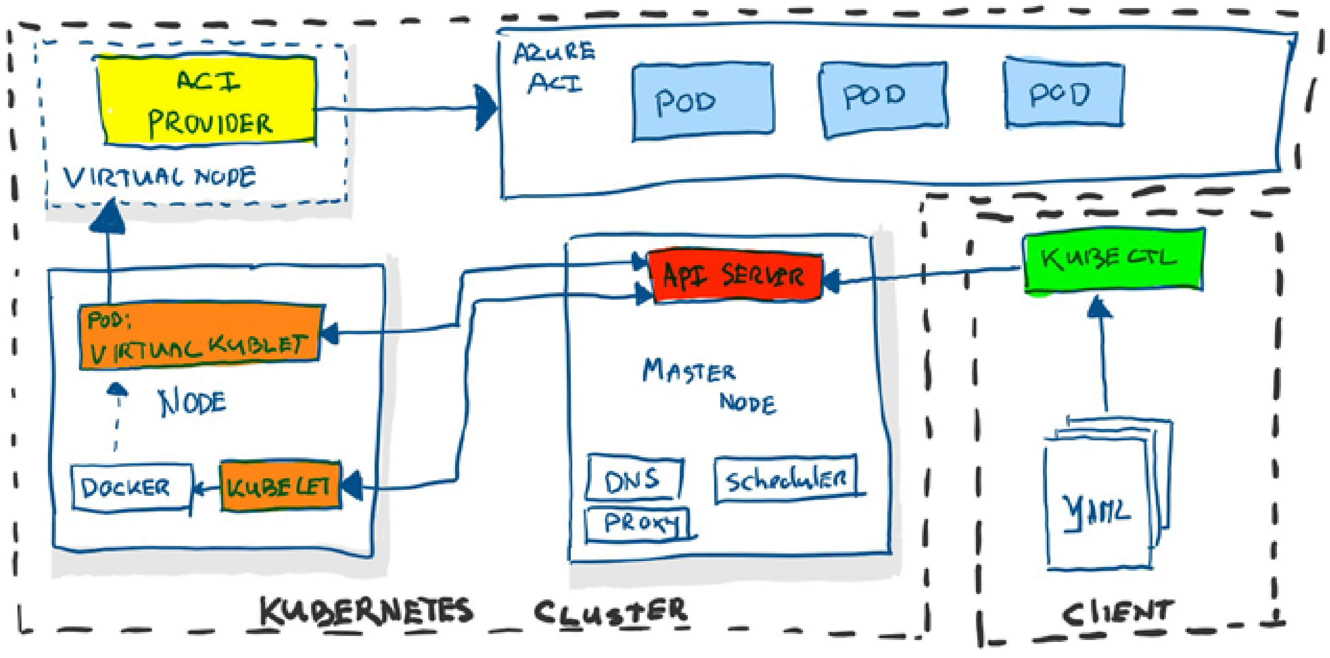


Figure 2: Virtual kubelet that uses the ACI provider for Microsoft Azure

“The advantage of running your services in a Kubernetes cluster is that you can define the desired state of your service and the cluster will try to get to this state.”

You can see that I have three nodes: one is the default node that is backed by a virtual machine and two nodes that are the virtual kubelets. The first is the virtual kubelet that ties into Linux containers on ACI; the second is the virtual kubelet that uses Windows containers on Azure ACI.

Scheduling a pod on Windows

Based on the results of querying the available nodes, you now have a Kubernetes cluster on which you can run Windows containers. This is because ACI provides the ability to schedule Windows containers. The restrictions on those Windows containers are the restrictions currently imposed by ACI. This means you can schedule containers that are based on Windows Server 2016. In the future, Windows Server 2019 will be supported.

With this configuration, you can now run e.g. Internet Information Server in the cluster. This can be done by scheduling the following deployment definition:

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: iis
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: iis
    spec:
      containers:
      - name: iis
        image: microsoft/iis
        ports:
        - containerPort: 80
        resources:
          requests:
            memory: 1G
            cpu: 1
          limits:
            memory: 1G
            cpu: 1
      nodeSelector:

            kubernetes.io/role: agent
            beta.kubernetes.io/os: windows
            type: virtual-kubelet
      tolerations:
      - key: virtual-kubelet.io/provider
        operator: Exists
      - key: azure.com/aci
        effect: NoSchedule
```

In the deployment definition you can see that we have defined a node selector. This selector indicates that we want to schedule the pod on an agent that has an OS of the type “Windows” and that the node is of the type “virtual kubelet”. This is the way we explicitly define that we want to run the pod on the Windows virtual kubelet within the cluster. The other part that is special to this deployment is the definition of the tolerations. By default, the virtual kubelet nodes are what we call tainted. Tainted means that we specify restrictions that tell the node not to schedule pods by default. You can only schedule the pods explicitly when you add a toleration to a taint. This is done to avoid scheduling just any pod on the virtual nodes. Normally you first want to fully utilize your nodes in the cluster before you start leveraging the serverless nature of ACI and scale out without creating new nodes.

You also don’t want to schedule a pod like the kube-proxy or other virtual kubelet pods on the virtual node. In this deployment we explicitly define that we accept the fact that the node is marked as NoSchedule by default and we overrule this by specifying the tolerations key ‘value pair’ that matches the taint.

After running this deployment, we can expose the scheduled IIS container in the pod via the Azure load balancer service. We can do this by running the following command:

```
kubectl expose deployment iis --port=80
--type=LoadBalancer
```

This configures the external Azure load balancer in order to make the IIS service reachable via an external IP address. If you want to know which IP address was assigned to the IIS deployment then you can query the cluster with the following command:

```
Kubectl get services
```

On my cluster this resulted in the following information:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
iis	LoadBalancer	10.0.139.238	104.40.243.220	80:32652/TCP	4m
kubernetes	ClusterIP	10.0.0.1	<none>	443/TCP	46d

Marcel de Vries

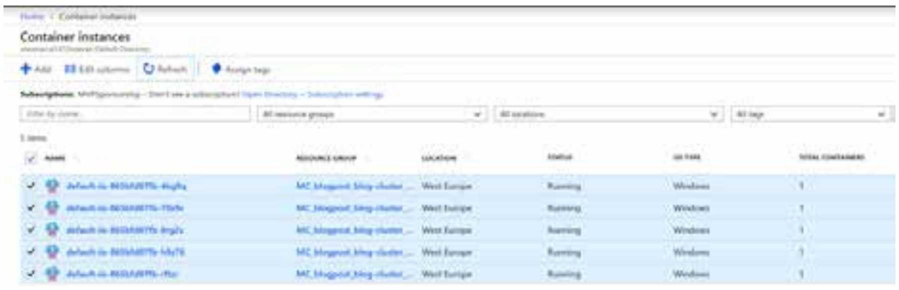
Scaling the deployment to use more replicas

The advantage of running your services in a Kubernetes cluster is that you can define the desired state of your service and the cluster will try to get to this state. All we need to do to schedule more IIS instances is to increase the number of replicas. The scheduler will then start scheduling more pods on the virtual kubelet, which in turn will delegate this to the ACI provider.

After increasing the replicas to 5 replicas with the command line:

```
kubectl scale deployment iis --replicas=5
```

you will see that 5 container instances will be scheduled on ACI, as shown below:



Container instances

NAME	RESOURCE GROUP	LOCATION	STATUS	OS TYPE	TOTAL CONTAINERS
default-iis-[illegible]	MC_blogpost_blog-cluster_...	West Europe	Running	Windows	1
default-iis-[illegible]	MC_blogpost_blog-cluster_...	West Europe	Running	Windows	1
default-iis-[illegible]	MC_blogpost_blog-cluster_...	West Europe	Running	Windows	1
default-iis-[illegible]	MC_blogpost_blog-cluster_...	West Europe	Running	Windows	1
default-iis-[illegible]	MC_blogpost_blog-cluster_...	West Europe	Running	Windows	1

Scaling up instead of out

It is also possible to schedule the containers on a more powerful container instance. ACI has the option to provide a container with 1 – 4 cpu's and you can also specify the amount of memory you want to make available for the container. This can be specified in the deployment. By defining the resource requests and limits, you define how the ACI provider will schedule the container. For example: increasing the CPU request to 2 results in a container instance scheduled with 2 CPU's.

When is a virtual kubelet useful?

The concept of a virtual node in a cluster with workloads scheduled by any type of provider allows a series of interesting scenarios. For instance, take the following use cases:

- Batch workloads
  You don't need to have VMs running in your cluster to support your batch workloads. You only need to pay for your normal workloads, and batch jobs can fan out as widely as needed to complete them in less time, while you pay per second.
- Burst loads
  If you use auto-scaling and your traffic comes in spikes, you only need to plan enough capacity for your average workload. The moment you run out of capacity in your cluster, the scheduler can start placing additional pods in something like ACI or another provider.

Conclusion

With the new capability of the virtual kubelet you can use various implementations that extend your Kubernetes cluster to be able to run your containers on a serverless infrastructure. The ACI and Azure Batch implementations allow you to leverage those parts of Azure and only pay-per-use instead of paying for the physical nodes your cluster would have otherwise. The virtual kubelet is a new way of implementing the concept of serverless, while keeping the same semantics as you already were using when running a Kubernetes cluster. It combines the best of both worlds: you can define your desired state and have the cluster manage this with a pay-per-use solution. </>

Chaos Engineering: Why you should break stuff in production on purpose

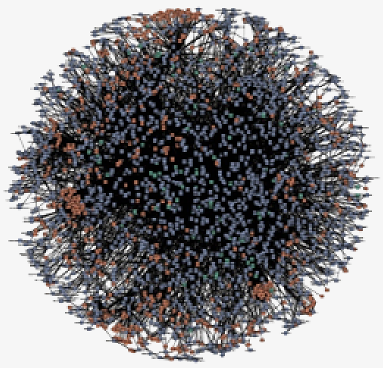
Have you ever been called out of bed because the application you work on wasn't working anymore? Or have you spent time on a Saturday doing manual failover tests from one datacenter to another? If you have, you probably are enthusiastic to learn how to avoid this. If you haven't, you're either just lucky it hasn't happened yet, or you made it somebody else's problem.

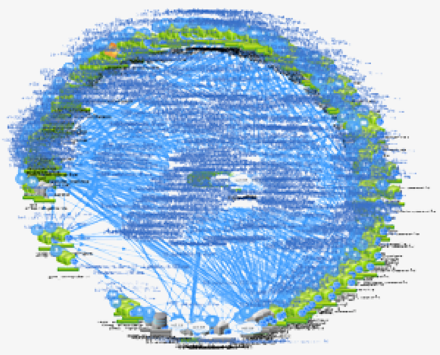
Author Geert van der Cruijsen

Measuring Complex IT landscapes
Application landscapes have evolved over the years and traditional monitoring systems are not capable of checking whether our systems are up or not. How come?

Look at the following architectures from large corporations like Amazon and Netflix. They represent all instances of microservices that run Amazon's web shop (So no AWS, this is only the online store). Do you think they have a dashboard that shows all servers and instances, showing green or red? I'll tell you now, they don't.

Microservice architectures and cloud infrastructure have changed our landscape a lot. We no longer have big servers that we care for as our pets. Instead, we have loads of smaller pieces of infrastructure that are responsible for specific parts of the application workflow. Often these pieces of infrastructure can scale horizontally running multiple instances of the same service.





What we do need to check is whether our application is operating normally? If a microservice is scaled over multiple instances, users might not even notice one of them being down.

Looking at servers being up or not is not the measurement anymore. We need to measure whether users are still able to do what they are supposed to do. Take Netflix for example. They use a great measurement for this, called "The pulse of Netflix". They use this to measure the amount of play buttons pressed. Netflix has a good understanding of the average streams started. If streams do not start, people will repeatedly press the play button to try again. As a result, the number of clicks increases. If the page with the play button does not even load, the amount of play clicks will decrease. In both cases, Netflix will get alerts of this behavior (or problem).

Monitoring user activity and success rate is of key importance when building high-availability applications. Without this you'll never know whether your application is working or not. Even if you have a small number of servers and all monitoring screens show a green status, this does not guarantee your users have a great experience in your application. A prerequisite for having a distributed, highly available application is having proper logging in place that enables you to query what users are expecting.

How to test for failure?

In the past we've tested for infra-structure failure by doing manual failover tests. Enterprises often do full datacenter failovers every 6 months or so. Most of the times these failover tests are executed during the weekend or at other times when it least impacts users.

In the age of cloud computing this feels old fashioned. We no longer have data centers and infrastructure is used as cattle instead of pets. If the infra-structure is broken or is not functioning properly, you just roll out a new one instead of nursing it back to health. We might think we've designed our sys-tems to be highly available, self-healing, auto scaling and doing fail overs, but is that working as intended?

What is Chaos Engineering?

A lot of people have heard of the term "Chaos Engineering". But when you ask them what they think it means, the most frequently heard answer is: "Killing servers randomly in production". While this certainly causes chaos, this is not what Chaos Engineering is about. This incorrect understanding comes from one of the earliest practices at Netflix. In 2010, before the term Chaos Engineering was coined, Chaos Monkey was born within Netflix. Chaos Monkey did exactly what people nowadays suspect: kill random servers at random intervals. Teams used Chaos Monkey to create applications that needed to be highly available. Surviving Chaos monkey was a great test. Later, Chaos monkey and "Failure Injection Testing" (FIT) turned into the new practice, Chaos Engineering. In 2014 this name was used for the first time for the practice of injecting failure on purpose in order to build better more highly available software. Today there is a website created by the Chaos Community to describe the principles of Chaos Engineering. You can find it at[1]. This website also contains the official description of what we currently mean with Chaos Engineering:
"Chaos Engineering is the discipline of experimenting on a distributed system in order to build confidence in the system's capability to withstand turbulent conditions in production."

Chaos engineering is all about doing controlled experiments and NOT about breaking things in production that would cause downtime or failures for your end-users.

Chaos Engineering versus regular testing

Chaos Engineering should be an addition to all the tests you are already doing. You'll need to have confidence in the quality of your application to use Chaos Engineering as an extra set of experiments to prove the resilience of your application. These kinds of tests can't be simulated by unit tests or integration tests.

But do we have to do this in production? This is a misconception that people have about Chaos Engineering. Although Chaos Engineering is often executed in production this is probably not the place to start. If you want to do your first experiments it might be possible to do this in an acceptance or test environment, depending on the experiment. As you get more confident over time, or want to test larger parts of your application landscape, production is the only place you can do this because it is often impossible to emulate a fully distributed application landscape in a test or acceptance environment.

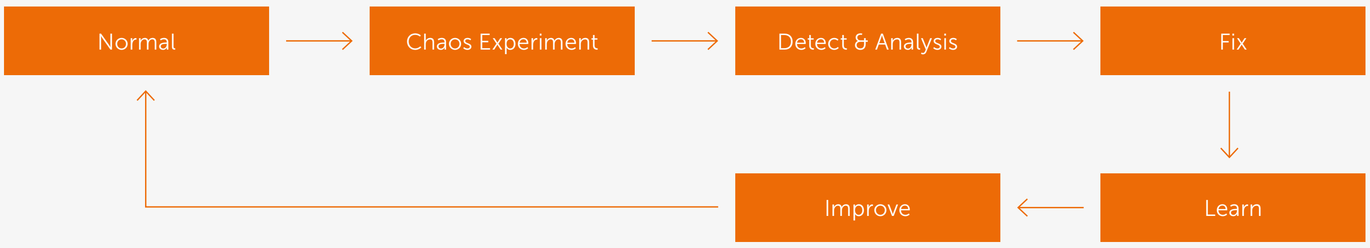
This works well in cloud environments where you have control over the infra-structure and it is possible to create an infrastructure on which to execute your experiments while the experiment takes place. If you can redirect a small number of users or specific users (maybe employees or beta testers) to this experiment infrastructure, you can run the experiments there without exposing your entire population to the risk of the experiment.

Is Chaos Engineering for me?

Who wouldn't want to add "Chaos engineer" as their job title? But is it something you really need? If you are building distributed applications (and who isn't nowadays) that need to have a high availability or are business-critical, Chaos Engineering is the only way to build this confidence for your application.

How to do your own Chaos Engineering experiments

To know how to do your own experi-ments, you need to know what to do in these experiments. It all starts with having a system that is in a steady state and that has enough observability to experiment on. No logs or monitors?



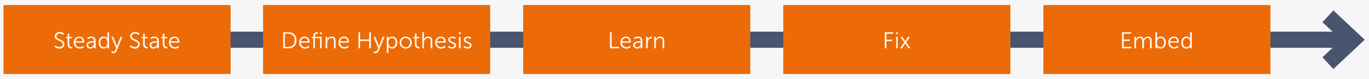
[1] https://principlesofchaos.org

No go! We can't do experiments without monitoring what is happening, so having proper logging within the application is a prerequisite.

A good way to get started with chaos experiments is to start organizing "Game Days". It's a time-boxed event where you get everyone involved in building and running your application to focus on resilience and failure by doing experiments together. The together part here is important. You are responsible together and want to avoid blaming people for things that are going wrong. Organizing a game day will embed the importance of chaos engineering into your culture and you will approve on it over time.

If you are unsure whether the failure will affect your steady state, if you are unable to come to an agreement of what will happen when failure is injected, or if you are not able to monitor this behavior, stop your experiment here. It's time to go back to the drawing board and get more information of how your application will respond to failure, or start adding more logging and monitoring.

You might think this is a bad thing but actually it's a good thing. You've learned something about your system and you're acting before something bad happens, thus making your application more resilient and ready for more experiments in the future.



Steady State
The first thing we need to do to run a chaos experiment is to define a steady state. This needs to be an indicator of your application that should work as intended for your end-users. As described earlier, Netflix uses "The pulse of Netflix" for this and you should have something similar for your experiment. This can be a lot simpler than what Netflix is using, depending on the type of experiment and the type of application.

It's important to measure a business metric instead of a purely technical metric. What we care about is whether our users are affected or not in what way they are affected. There might be a graceful degradation when certain services are down. We always want to design these changes with the end-user in mind, focusing on giving them the best experiences possible.

Hypothesis
The next thing to have is a hypothesis of what failure your application should be able to endure and what the outcome will be. The best way to create a hypothesis is by doing a brainstorm with everyone involved in that part of the application present. This should not only be the engineering team, but anyone who has a part in running your application.

Most of the time, people will have an idea of what "should" happen as part of the design, but having everyone there – from developers, operations, networking, security, architects, and of course the product owner – will allow a good discussion of what the application is really going to do in case of failure. Is there any graceful degradation, will something else take over, or will the application just stop working?

A common way to brainstorm about what failures your application should be able to endure is looking at your steady state and come up with several "What if" questions. What if the database is unavailable? What if the network latency is increased by 100 milliseconds? What if the application node restarts? Everyone can chip in with their own expertise and come up with several scenarios that will affect your steady state.

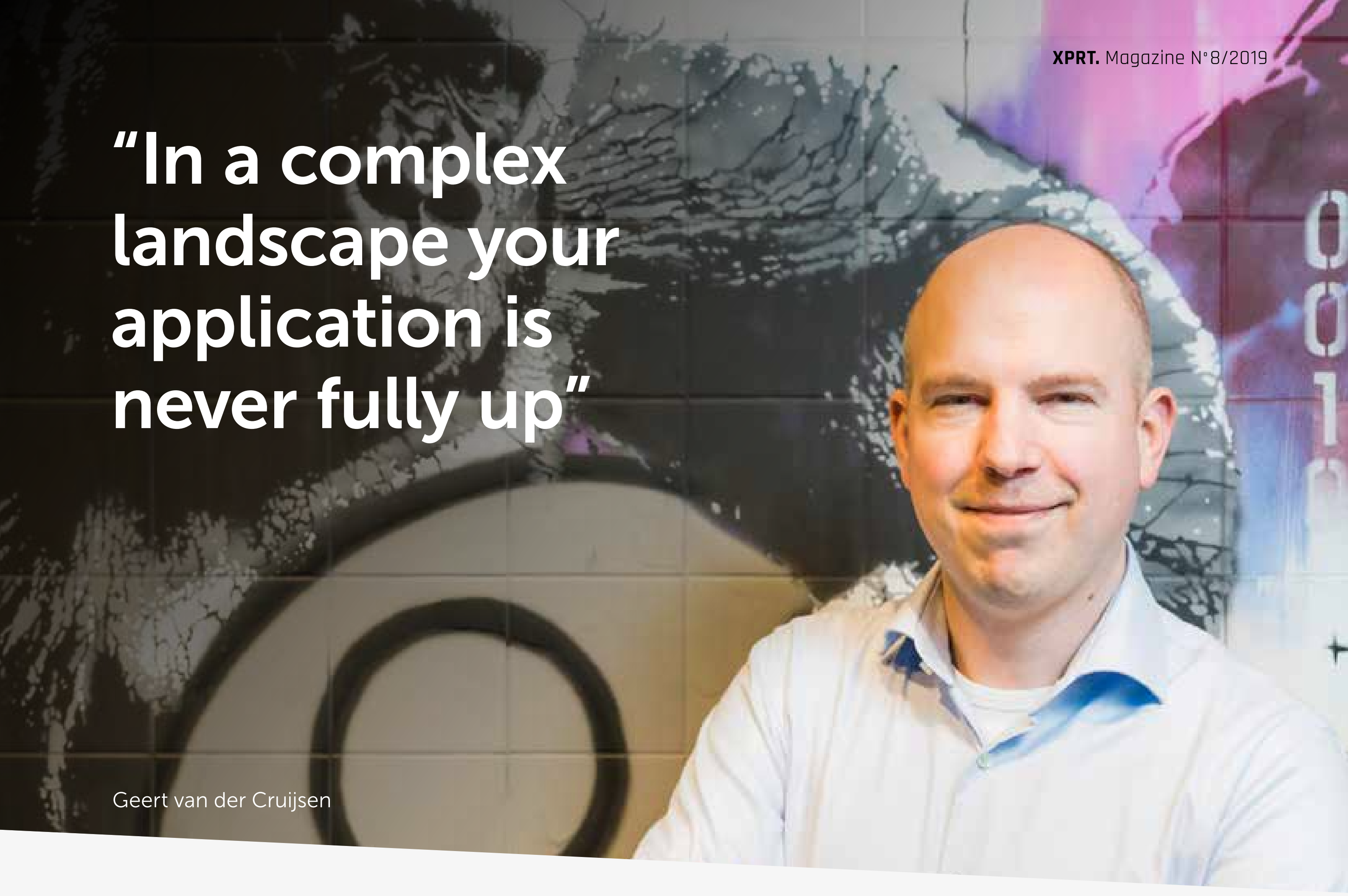
Design and execute the experiment
Once you've created a hypothesis it's time to create an experiment to test whether your hypothesis is correct. There are several things to keep in mind when designing the experiment. First of all: start as small as possible, thus minimizing the impact when things go wrong. If you are not that confident yet or this is one of your first experiments, acceptance environments might be a good place to start, but most of the times you want to do this in production because that is the only place that really gives confidence after successful experiments.

Start small so that you can minimize the blast radius. Once this is successful, you can increase the *blast radius* by adding more users or affecting a larger part of your landscape. Keep monitoring and *always have a fail-safe in place* to abort the experiment.

Cloud infrastructure is ideal for these experiments because you can spin up a second environment with ease where you do your experiments without affecting the rest of your application landscape.

Learn
After executing the experiment it's time to investigate the results and see what you can learn from your observations. It is important here to quantify your results. For example: How soon after injecting the failure were you able to see it on your monitors. How fast were you able to recover?

Fix
After quantifying the results it became easier to compare them with your assumptions or goals. If the results don't meet your expectations you can start improving your application to become more resilient to these kinds of failure. After you have made your improvements, run the experiment again to see whether the improvements are sufficient.



“In a complex landscape your application is never fully up”

Geert van der Cruijsen

Embed
If you get more familiar with these chaos experiments you might want to embed them further in your engineering culture. This can be done through continuous chaos like the original chaos monkey that keeps rebooting VM’s at random intervals. If you know that these experiments exist, and you can opt-in to them, it becomes something that is at the top of the minds of development teams right from the start.

Tools to get you started

Chaos Monkey is the original chaos engineering tool created at Netflix. It’s still being maintained and is currently integrated into Spinnaker which is Netflix’s CICD tool[2].

Gremlin is a company started by some of Netflix’s and Amazon’s Chaos Engineers who productized Chaos as a Service (CaaS). Gremlin is a paid service that gives you a CLI, agent and website that will help you set up chaos experiments. Gremlin announced a free service a month ago that offers free basic chaos experiments such as turning off machines or simulating high cpu load[3].

Chaos Toolkit is an open source initiative that tries to make chaos experiments easier by creating an open API and standard JSON format to expose experiments. They have several drivers to execute these experiments on AWS, Azure, Kubernetes, PCF and google cloud. They also offer integrations with monitoring systems and chat such as Prometheus and Slack[4].

[2] https://github.com/Netflix/chaosmonkey
[3] https://gremlin.com
[4] https://github.com/chaostoolkit/chaostoolkit

Conclusion

Making applications resilient is no longer something that is relevant only for operations. With cloud infrastructure, developers and engineering teams have become responsible for their complete applications, both at the application level and the infrastructure level. Cloud infrastructure has given us the flexibility and the agility to adapt quickly to new business requirements, but without taking care that you are fully dependent on the resilience of the cloud infrastructure itself. You’ll have to create an architecture that is resilient using these components and the only way to find out whether it is as resilient as you hoped it was is by doing controlled chaos experiments. So start experimenting yourself by organizing a game day in your own company! Are you still a bit scared to take the leap? Let me finish by this great quote from Nora Jones, Senior Chaos Engineer at Slack and co-author of the Chaos Engineering book by O’Reilly. </>

“Chaos Engineering doesn’t cause problems, it just reveals them”

Nora Jones, Chaos Engineering Lead Slack

Resilient Azure Service Bus architecture

During one of our innovation days at Xpirit, we looked at how we could make a system that we are working on more resilient against outages. Our application should be able to failover to a secondary region and failback to the primary region without much effort. For our innovation day, we specifically focused on protecting the system against Azure Service Bus outages. The messages passed to this system are critical, and we wanted to narrow the chances that we missed a message due to an outage. Another characteristic of the system is that the throughput of the total number of messages that we receive can be considered low: we process about 1000 messages every 24 hours. With these things in mind, we investigated the various approaches that we could use and worked towards a solution.

Author Marc Bruins & Sander Aernouts

The goal of this article is to explain a number of architectural patterns that we explored for Azure Service Bus.
These patterns will allow your system to cope with the fact that Azure Service Bus may go down at some point in time. None of the patterns in this article will guarantee that no messages can be lost, but most of them will reduce the chance of losing messages significantly.

We will not go into detail about queues, topics, and subscriptions. Instead, we will explain how you can use multiple Azure Service Bus namespaces in different regions to add resiliency to your systems.

As an example, we will use a system that processes fines for speeding cars. The system consists of an automatic speed trap that sends the speed and license plate to a backend service whenever it detects that a car is speeding. The fines services receives these messages and processes the fine, making sure the owner of the car receives the fine.
The messages between the automatic speed trap and the fines service are sent using an Azure Service Bus namespace. Communication is one-way; the fines service does not send any confirmation or reply to the automatic speed trap.
For simplicity sake, there are also no other parts in this system that communicate with either the automatic speed trap or the fines service. In this example, the automatic speed trap relies on the assumption that Azure Service Bus is available; it does not buffer or store messages in any way. The following figure illustrates the example system:

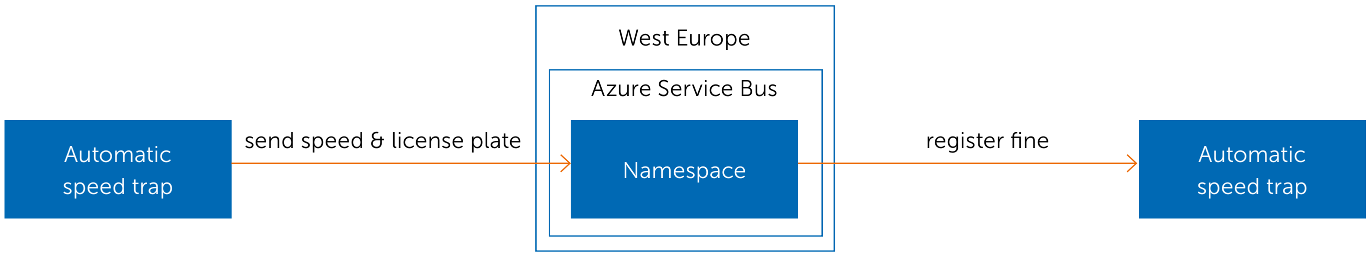


Figure 1: example system

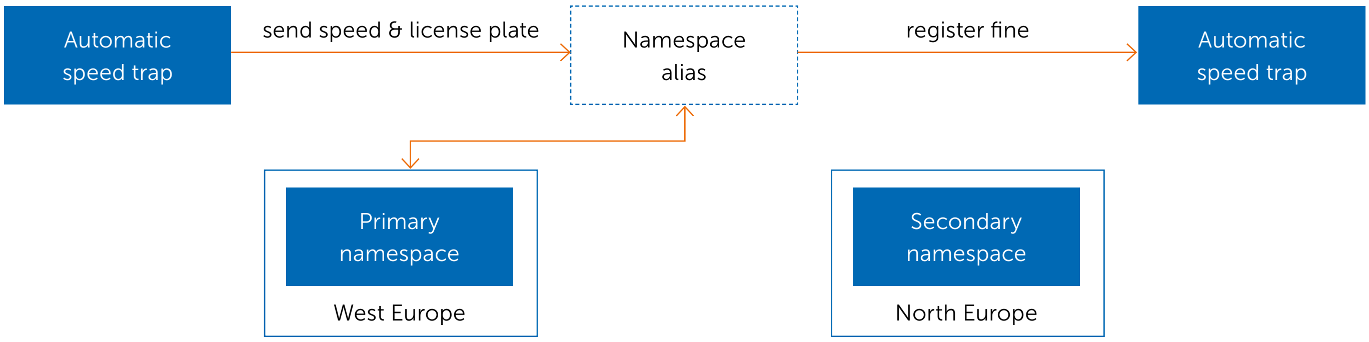


Figure 2: failover namespaces no outage

Now imagine a problem occurs, and Azure Service Bus goes down in West Europe. The automatic speed trap can no longer send messages to Azure Service Bus, and the fines service cannot process any fines. Speeding cars are not reported to the fines service, so the owner of the car will not get a fine. Let's investigate what patterns we can apply to make our system resilient to such an outage.

Failover namespaces

The first pattern we will look at is "Failover namespaces". This pattern utilizes the "Azure Service Bus Geo-disaster recovery" feature of Azure Service Bus, which is available as part of the premium SKU. When you enable this feature, you create a new namespace in a different Azure region. This new namespace will be the secondary namespace, and the other namespace will be the primary namespace. A namespace alias can also be configured that points to the primary namespace. The automatic speed trap sends messages to the namespace alias, and the fines service receives the message from the namespace alias. This way the primary and secondary name-space can be swapped easily.

As shown in figure 1, the namespace alias will point to the primary namespace until Azure Service Bus goes down in that region. Both the automatic speed trap and the fines service communicate with the namespace alias, which can be compared to a CNAME DNS record.

When Azure Service Bus goes down in West Europe, we will have to execute a failover either by pressing a button in the Azure Portal or by invoking the Azure API. When this happens, the namespace alias switches to our **secondary** namespace.

Once we've done this, the system will continue to send and receive messages.

We can only execute this failover once, and we cannot switch back to the primary namespace once Azure Service Bus comes back up in West Europe. It is possible to switch back to a namespace in West Europe, but this requires that you set up and perform another failover to move back to this region.

The downside of this pattern is that only queues, topics, subscriptions, and filters are automatically mirrored from our primary namespace into your **secondary** namespace. Messages are not mirrored, so the messages that the receiver did not read from the **primary** namespace will stay in the **primary** namespace. You will somehow have to extract them from the **primary** namespace and move them into the **secondary** namespace, otherwise they will not be processed.

Another downside is that we must initiate the failover explicitly, either by pressing a button in the Azure Portal or by somehow triggering the failover using the Azure API's. Until the failover is initiated, both the automatic speed trap and the fines service will receive errors. It is up to the automatic speed trap and the fines service to keep retrying so that communication can continue once the failover is initiated. Messages that were sent by the automatic speed trap but were not yet received by the fines service are stuck in the **primary** namespace and may be lost if the region does not fully recover from the outage. Messages that are stuck in the **primary** namespace need to be transferred to the **secondary** namespace either by a manual or automatic process, in case the region hosting the **primary** namespace fully recovers.

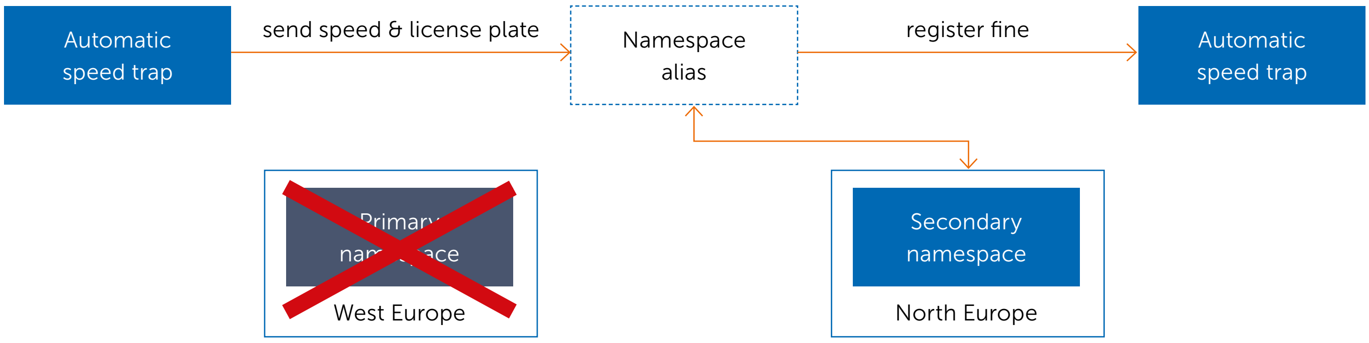


Figure 3: failover namespaces outage

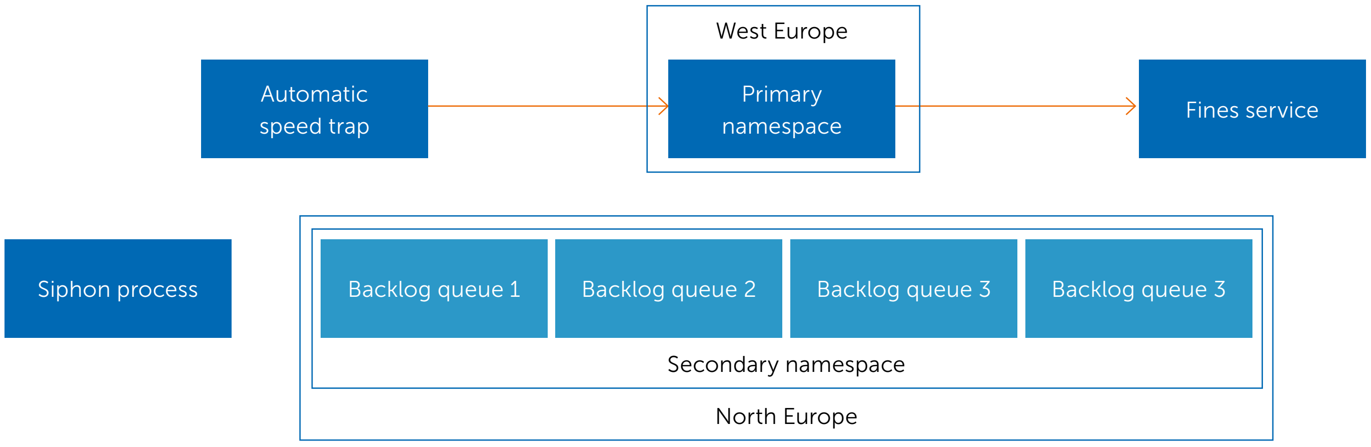


Figure 4: paired namespaces no outage

The benefit is that both the automatic speed trap and fines services are unchanged. From that perspective, there is no difference between using a namespace and using a name-space alias in your application. All you have to do is configure a different connection string. So, if we do not have control over the source code of the applications that use Azure Service Bus, this pattern allows us to add a failover option without changing any code. If required, we can even replace an existing namespace by a namespace alias, which means that you don't even have to change your configuration.

Paired namespaces
The second pattern is "paired namespaces". With this pattern, we will use one or more "backlog queues" in a **secondary** namespace that will receive and hold the messages while the **primary** namespace is down. This functionality is built into the Azure Service Bus client and can be enabled by calling the "PairNamespaceAsync" method on the "MessagingFactory" in your code.

When we pair two namespaces, the client will create one or more backlog queues in a **secondary** namespace. We can configure the number of backlog queues that are created. As long as the **primary** namespace is available, messages are sent to the **primary** namespace. When the **primary** namespace goes down, new messages are sent to one of the backlog queues in the **secondary** namespace. Messages that were already delivered to the **primary** namespace will not be resent to the backlog queues. The backlog queue is chosen randomly from the available backlog queues. The client will also continuously ping the **primary** namespace to check whether it is available again. As soon as the **primary** namespace is available again, the client will restart sending messages to the **primary** namespace. Messages that are in the backlog queues still need to be transferred back to the **primary** namespace. This process is called siphoning and is also part of the Azure Service Bus client. In this example, we have configured a s eparate process that is responsible for the siphoning process.

Figure 4 describes the normal operation: the **primary** name-space is reachable, the automatic speed trap sends all messages to the **primary** namespace, and there are no messages for the siphoning process to forward.

At some, point the primary namespace goes down. The automatic speed trap automatically sends messages to one of the four backlog queues. Also the **primary** namespace will be pinged at regular intervals until it becomes available again as shown in figure 6.

When the **primary** namespace becomes available, the automatic speed trap will start sending the messages to the **primary** namespace again. The messages that are in the backlog queues are read (received) by the siphon process and are forwarded to the **primary** namespace, as shown in figure 5. When the siphon process has read and forwarded all the messages from the backlog queues, normal operation can continue as was shown in figure 3.

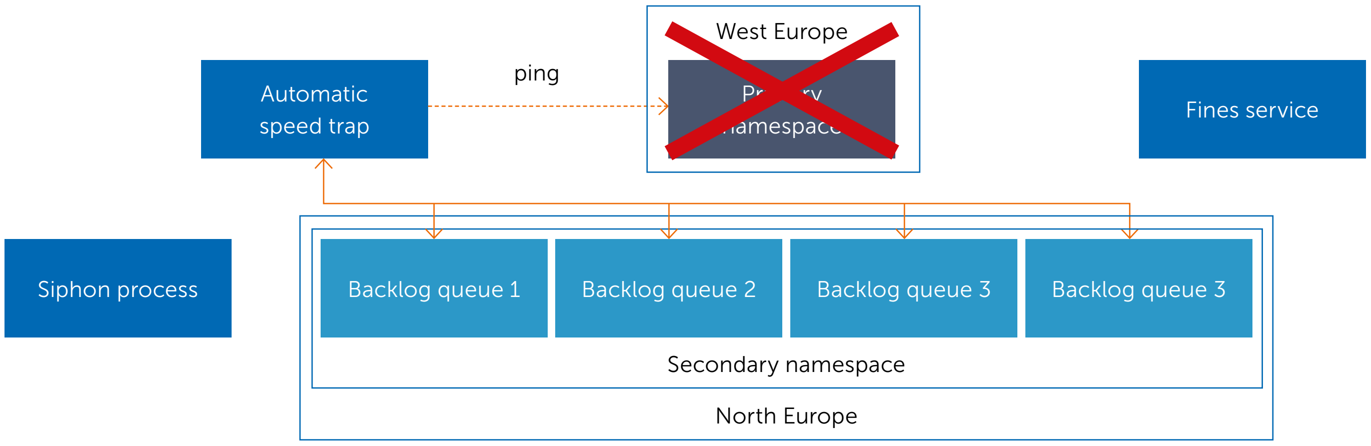


Figure 5: paired namespaces outage

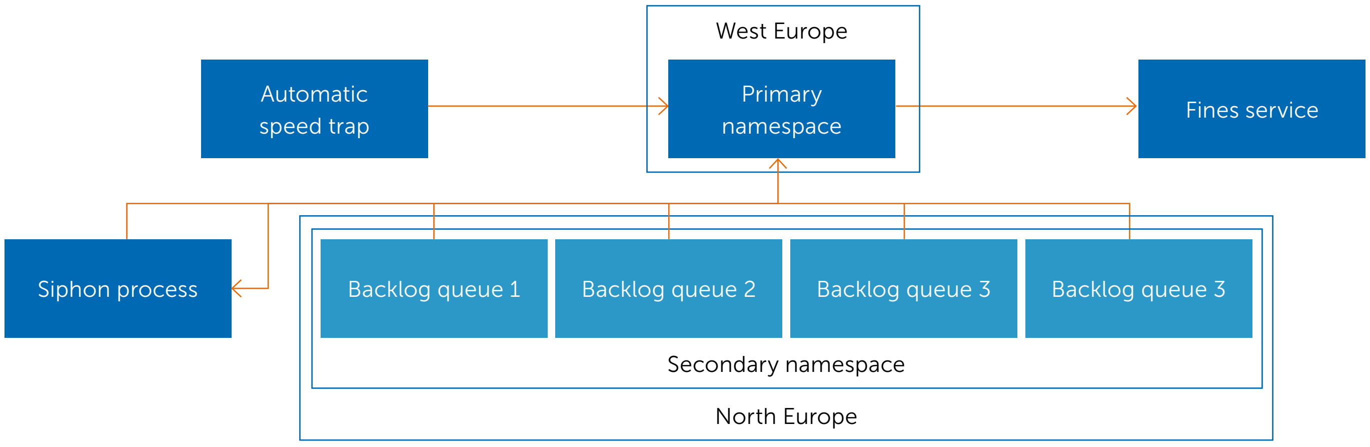


Figure 6: paired namespaces outage

In contrast to the failover namespace pattern, using the paired namespace pattern does require that we modify the code of the automatic speed trap application. Paired namespaces is a feature that is available in the Azure Service Bus client, so we only have to configure it, not write it ourselves. There is no need to change the fines services, but the downside is that no messages are delivered while the primary namespace is down. Our system will not break, and we will not lose messages, but communication between the automatic speed trap and fines service will stop until the primary namespace is available again.

The failover situation, when the automatic speed trap sends messages to backlog queues, is triggered automatically when the **primary** namespace goes down, and no manual intervention is required. Communication to the **primary** namespace will also restore on its own when the Azure Service Bus client detects the **primary** namespace is available again.

A downside is that communication between the automatic speed trap and fines service will stop until the **primary** namespace is available again. Another downside is the order in which messages are delivered. If we use multiple backlog queues, the messages are randomly delivered to one of the available queues. When the messages are then received by the siphon process and forwarded to the primary namespace, the order of messages cannot be guaranteed.

Passive-Active replication

Another option that we might want to consider is passive replication. This pattern uses two namespaces, one of which we call our **primary** namespace and the other is called our **secondary** namespace. The idea is that there is only one **active** namespace at any time which handles our messages. The automatic speed trap will send messages to the active namespace, which will be the **primary** namespace when there are no outages.

The fines service listens to both namespaces and receives all the messages it can find. When our system is running smoothly, the **primary** namespace is the **active** namespace so messages are sent through our **primary** namespace and our fines service handles those messages. Now imagine an outage where the **primary** namespace goes down. In this case we make our secondary namespace the new **active** namespace. The automatic speed trap will now send messages through the **secondary** namespace. Messages will continue to flow, so we won't experience any downtime.

To make this work, we need to build two pieces that support this pattern, one of these is the sender, and the other is the receiver. The sender is straightforward in this scenario, and it should send a message to the **primary** namespace and if that fails, it should send that same message to the **secondary** namespace. We could implement a circuit breaker here that breaks after a few attempts and checks the **primary** namespace after a specific amount of time has passed.

The receiver side is a bit trickier. It needs to know how to receive messages from both the primary and the **secondary** namespace. If the logical order in which we receive the messages is important, we need to make sure that the order in which messages are read from the namespaces is as follows: if the **primary** namespace is down, read from the secondary namespace, but when the **primary** namespace gets back up again, first drain the **secondary** namespace and then continue to read from the primary namespace, instead of immediately

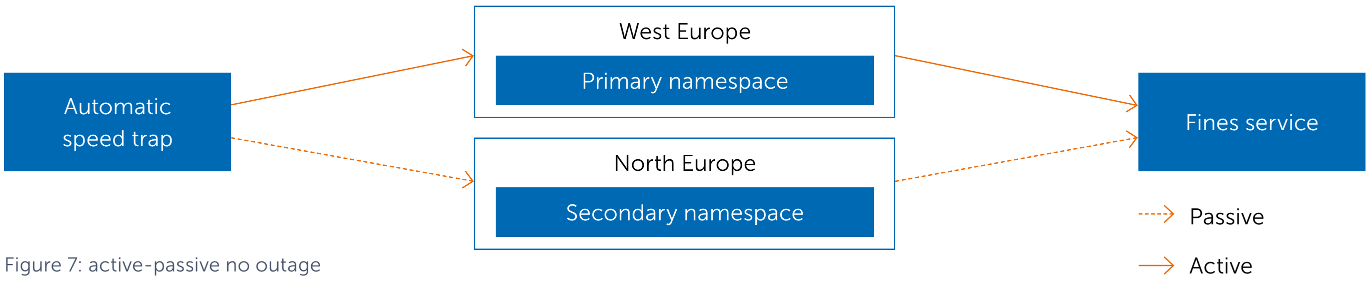


Figure 7: active-passive no outage

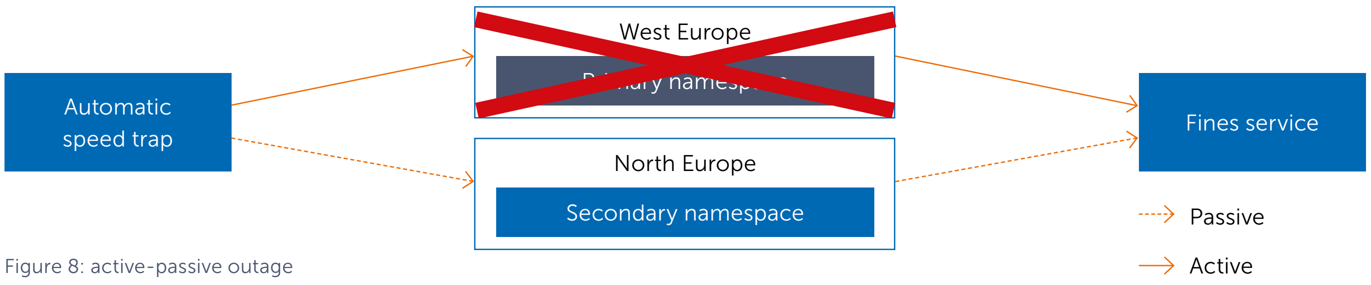


Figure 8: active-passive outage

switching back to the primary namespace. If the order is unimportant, then we don't care, and our receiver can just listen to both namespaces.

This setup gives us high availability and automatic failover without having to touch a single button. However you have to build this yourself; this pattern is not implemented by the service bus client. There is still a (very small) chance that a message gets lost. Imagine that the speed trap sends a ticket to our **primary** namespace. In this case, there are no issues and the ticket is received by the **primary** namespace. But at that moment our **primary** namespace could go down before it had a chance to deliver the message to our fines service. If that happens, that message may be lost if the region does not fully recover, and there would be one lucky speeder that doesn't receive a fine.

Active-Active replication
With the active replication pattern, we make sure that the chances of losing a message are even smaller than with passive replication. Whenever an Active-Passive namespace holds a message that the namespace received and if this namespace goes down, we lose the message. To prevent losing those messages we could use the Active-Active pattern.

To set up the active replication pattern we need to have two namespaces, a **primary** namespace and a **secondary** namespace. The automatic speed trap actively sends all messages to both namespaces. In case the **primary** namespace goes down, we still have the **secondary** namespace that has received all the messages. If there is no outage, the fines service will receive the same message from both the **primary** and **secondary** namespace. To make sure our system doesn't have duplicate entries, we must create a deduplication layer at the receiving side (fines service).

In case the **primary** namespace goes down, we still receive all the messages from the **secondary** namespace. In this case, the deduplicator just passes our messages through to the fines service. You can even add a third namespace in a different region to protect against a simultaneous outage of both the **primary** and **secondary** namespace, although this is likely to be overkill.

To make this setup work we need to modify the automatic speed trap to send all messages to both the **primary** and **secondary** namespace. If it can't deliver to one of the namespaces it doesn't matter, as long it can still deliver the message to the other namespace.

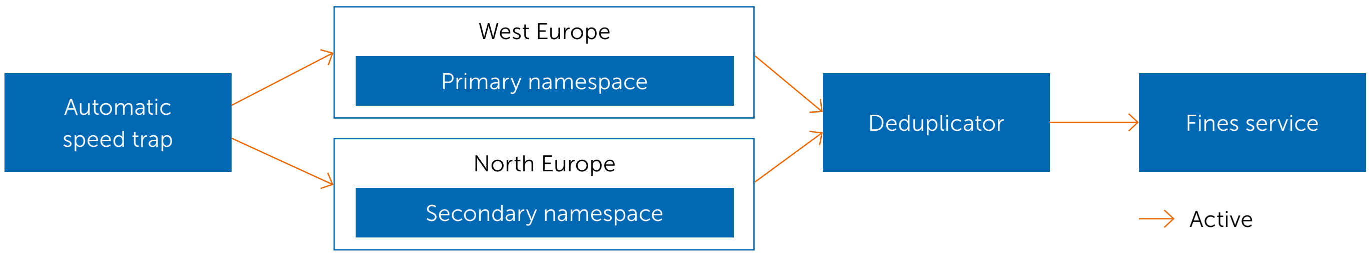


Figure 9: active-active no outage

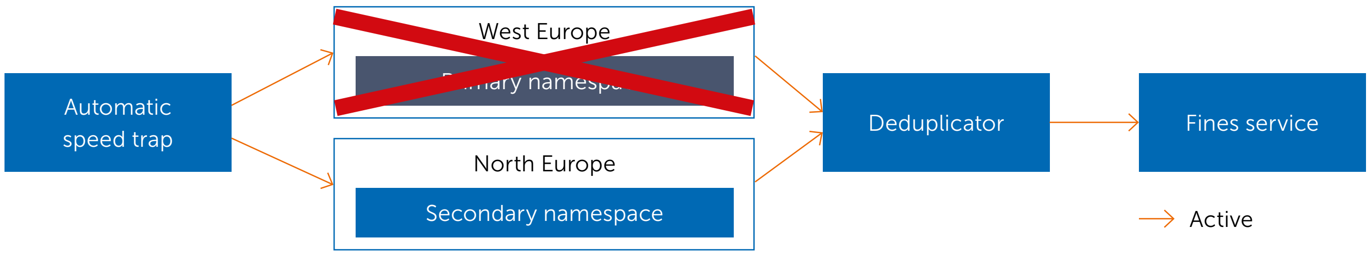


Figure 10: active-active outage

Marc Bruins & Sander Aernouts

Our fines service must be a bit more advanced. It processes duplicate messages if we don't filter the received messages in some way. Some systems don't care if they process duplicate messages, but for our system we only want one message to be processed in the fines service. To do this, we need to implement a deduplication layer that ignores messages that have already been received from another namespace. We can do this by storing the unique id of each received message in a cache. If your receiver is idempotent, you can choose to limit the number of cache items and automatically evict the oldest items (FIFO). You can be fairly certain that duplicate messages are delivered shortly after one another. If the message id already exists in the cache, we know that we can ignore the message. If the message id does not exist in the cache, we know the message can be delivered to the fines service. If your receiver is not idempotent, you must persist your processed message id's, for example in a SQL database, which means you need to protect against SQL outages as well.

This setup gives us high availability and we don't have to do anything in order to failover. The system will just continue to work. However, this setup adds complexity by having a deduplication layer that holds a record of all the messages to find out whether there are duplicates. If we choose to persist the message id's, we must think about high availability for our caching layer, and this can be a nuisance. And this is even more complex when there are multiple instances of the same receiver. The benefit of this approach compared to passive replication is that there is an even smaller chance that we lose a message. However, the costs may outweigh the benefits depending on your solution. This solution would also require you to have control over the source code from the sender and the receiver.

The pattern we chose ourselves

As stated at the beginning of this article, we explored these patterns because we wanted to protect a real system we are working on against outages of Azure Service Bus. We wanted an automatic failover and we didn't only want an Azure Service Bus failover, but instead an entire region failover, independent of the resources that are running. This narrowed our choices to Active-Passive and Active-Active.

To make this decision we looked at the value of a message. For our system, the value of a message doesn't outweigh the cost of implementing the Active-Active pattern. The system does not receive many messages, so if a region goes down, there is a small chance that we lose that message. Considering all of this we've chosen for the Active-Passive pattern. </>

"Whatever can go wrong will go wrong, at the wrong time."

Matt Mika

HTTP APIs and event sourcing

Problem Imagine you are working on a project to build a web application. In doing so, you want to use the latest and greatest in technology and implement an HTTP API. You have done this before, and based on your experience, you use the GET verb to retrieve data, either a list or a single item. In addition, you use POST to store new data and PUT for updates. Finally, you're using DELETE to remove a resource. As a result, clients can connect to your API and interact with the data. While this approach sounds simple, it can become complicated when your application uses the 'CQRS' and 'Event Sourcing' design patterns. These patterns ensure great power, but they also increase complexity.

Authors Michiel van Oudheusden & Loek Duys

Now imagine a domain in which we interact with a customer resource. You start in a simple way by including a GET request at the endpoint /api/customers that will return a list of customers from your read-model. To create a new customer, you perform a POST web request to the same endpoint. Updating of data is handled by using PUT; i.e. you send the new state to the endpoint /api/customers/{customerId}. The last part of the URL identifies a unique customer.

In your API Controller, you handle the intent of changes to a Customer by converting them to commands, e.g. a CreateCustomerCommand or an UpdateCustomerCommand. A command queue and command handlers will do the rest. Eventually, the new customer representation will appear in the read-model.

Although this looks simple, you can quickly run into a more complex situation. What if your customer entity contains related items? A customer might have one or more addresses or legal entities. Are those addresses

About CQRS and Event Sourcing

The acronym CQRS stands for Command Query Responsibility Segregation. This design pattern effectively separates read operations from write operations. This allows you to use two different models, a read model and a write model. Applications often read data more often than they write. Continuing the segregation all the way up to the data store allows you to optimize the design for both situations. For example, you can store denormalized data in a SQL database read model so it can be queried efficiently, and use a document database for the write model to get the best write performance. People often combine CQRS with Event Sourcing (ES). The easiest way to explain what Event Sourcing means is by comparing it with a bank account. Instead of storing only the latest value of the state of an object, you persist all changes. It is likely that you made your first deposit when you opened your bank account. Since then, the bank has stored all deposits and withdrawals. To know how much money is in your account, you calculate all changes. It's easy to see that this design pattern will help you create an audit trail inside your application. You can trace back all changes to a domain model. It also enables you to perform what-if scenarios by adding 'fake' events to the calculation and see how they influence the outcome. You can imagine that calculating bank statements for thousands of customers can soon become a burden for any server. In this situation, Event Sourcing and CQRS become a great combination. You store changes (events) only by adding records to a data store (the write model). But after storing the change, you can also update a read model with the current account balance. This way, your application can also respond to more complicated queries in a very efficient way.

HTTP APIs

APIs are often based on REST; REST stands for REpresentational State Transfer. An API of this type is called a RESTful API, which means an Application Program Interface (API) that uses standard HTTP requests to GET, PUT, POST and DELETE data. This enables you to expose data as resources, identified by URLs, and perform operations on these resources.

different resources, or will you consider them part of the customer graph? Besides that; a customer can go through different stages in his life cycle. Approval might be needed before a change to a customer is applied. Are these changes mere updates on the customer with a 'State' property? If so, how would you capture user comments for that specific state change or a reassignment of approval to another user?

Deleting a customer is another interesting scenario. Are you removing a customer, or do you require soft (reversible) deletes? Again, somebody might need to approve the operation before the system deletes the data.

Although some of the issues we mentioned above can be present while not using CQRS or ES, there are specific problems that pop up when using CQRS. For example, a write operation results in the creation of a command. The command will most likely be processed in the background. Although asynchronous processing is great for scalability and performance reasons, it also means that the resulting state of a resource is not yet determined during the POST or PUT web request. Also, the API consumer does not know when or how the system will process the command. In these scenarios, it is common to deliver feedback about processing back to the client via another mechanism, for instance a push channel by using technology such as Signalr.

A similar issue happens during the validation of the customer state. The command created by the HTTP action is put on a queue and handled at a later time. While you can validate the state of the resource before you create the command, the situation may have changed before the command is ready to be processed. For example, your application is used to change a customer address, while somebody else just deleted it. The HTTP request has already been returned, the client thinks all is well, but the processing of the last command will fail.

In short, does the creation of an asynchronous application also imply that we should not use a standard HTTP API? On the contrary; we believe you can do this, and we'll show you some solutions!

Possible solutions

As with every problem, there are also various solutions. If we do want to stick with HTTP verbs, then the options are as follows.

Option 1. Use PUT

Follow the REST guidelines, and perform updates by using a PUT verb. This option has all the drawbacks as mentioned before, but it is very intuitive to update resources using this verb. In this case, you would update a customer by sending the complete resource representation to the endpoint.

```
PUT /api/customers/{id}
Content-Type: application/json

{
  “name”: “customername”,
  “industry” : “name of industry”,
  “telephone” : “123 456 789”,
  “active”: true,
   etc
  “addresses”: [
     {
        “street”: “value”
     },
     {
         etc
     }
   ]

}
```

Not only can this lead to large payloads as you send the whole representation of the customer, but it also adds complexity at the server side. The server needs to discover the intent of the PUT request in order to convert it into the correct command. It all depends on how fine-grained you want it to be: building a ChangeCustomerCommand might be simple, but detecting and using state changes to trigger multiple workflows (like change approval) will be more complicated. Combine this situation with hierarchical resources, e.g. customers with addresses and you'll soon have a huge block of complex code...

Option 2. Use PATCH

You can optimize the change process by using the PATCH verb. Using PATCH indicates a partial update and allows you to update only specific fields of the resource. Instead of sending the whole resource, you only send the changes.

```
PATCH /api/customers/{id}
Content-Type: application/json

{
  "active": false

}
```

Although this makes the payload lighter, it does not solve all size-related problems. For example; adding a new address is only possible by providing the entire new set of addresses.
As a workaround for this, you can use "json-patch (RFC 6902)". This approach allows you to make very specific changes on your resource and as such, it can help reduce the payload.

```
PATCH /api/customers/{id}
Content-Type: application/json-pat-
ch+json

{
   { "op": "add", "path": "/addresses",
"value": [ { "street": "second street"
] },
}
```

The JSON-PATCH standard allows the use of operations to make targeted modifications to a resource. Adding, removing, but also replacing, copying or moving are valid operations that can be accompanied by a test condition.

Similar to the PUT method, you still need to extract and generate the command out of the data that is submitted.

Option 3. The miniput pattern

A possible shortcut is the use of the "miniput pattern", which allows partial updates to a resource by exposing child resources so we can do a complete update.

```
PUT /api/customers/{id}/telephone
Content-Type: application/json

{
    "telephone" : "123 456 789"
}
```

Note that the customer attribute named 'telephone' is now part of the URL. The server replies with the full representation of the resource, as a result of setting the content-location header value referencing the customer resource.

```
HTTP/1.1 200 OK
Content-Location: http://example.org/
api/customers/{id}
Content-Type: application/json
Content-Length: ...

{
  "name": "customername",
  "industry" : "name of industry",
  "telephone" : "123 456 789",
  "active": true,
   etc
  "addresses": [
     {
        "street": "value"
     },
     {
         etc
     }
   ]
}
```

The miniput does not target the parent resource URL, and this makes cache-invalidation of this customer resource difficult.

The above options provide some alternatives to perform updates on resources, but they still don't fit very well with the more complex scenarios like eventual consistency and complex resource graphs. However, there is a way in which we can interact with the system using a more "command-driven" approach. (Don't worry, we won't encourage SOAP web services...)

Option 4. Command endpoints

In this scenario, we make it very explicit to the caller that we expect abstract commands by exposing a single command endpoint. The consumer will send a POST request to this endpoint, and the resulting command ends up on a command queue and gets processed. You need to specify the type of command inside the body of the web request.

```
POST /api/commands
Content-Type: application/json

{
  "Name": "ChangeAddressCommand",
  "Payload": {
    "Address": { "Street": "new
street"}
  }
}
```

After we enqueue the command, we return a 202 response code. The web response also includes a location header that points to the endpoint where the status of the command processing can be retrieved. Note that it does not point to the resource itself, but to a new resource that describes the status of the command processing.

```
HTTP/1.1 202 OK
Location: http://example.org/api/com-
mand-progress/32453
```

By performing a GET on the command-progress resource, the client can see whether the command has been processed or rejected. It could also contain additional details like a webhook location, or a web-socket link for push notifications over Signalr.

This approach makes it very explicit that commands are needed, as they are the only way to make changes to the system. All other calls to resources only support the GET verb, and will query the read-model. The caller needs to be aware of the command structure available, while at the same time the types of supported commands are not easily discoverable.

Option 5. Content type

Similar to the previous solution, we need the caller to pass on the intent in the form of commands, using the content type header while operating on the resource. Queries naturally map to GET methods, while commands are mapped to POST, PUT, DELETE and PATCH. The command type is part of the content type header. For example, changing the name of a customer can be expressed as follows:

```
PUT /api/customers/242
Content-Type: application/json;
domain-model=RenameLegalEntityCommand

{
   "Name": "New Name"
}
```

Changing the name is an idempotent operation, which means that executing the same action multiple times produces the same result. The standard dictates that we must use the PUT verb in this situation. However, other commands, e.g. adding a new address, need to be expressed with a POST verb because they are not idempotent operations.

Removing a customer would be implemented using the DELETE method:

```
DELETE /api/customers/242
Content-Type: application/json;
domain-model=DeactivateCustomer
Command
```

The clear downside of this approach is that the internal domain is now partially visible on the outside. Callers need to be aware of the resources, the various operations and even different commands that can be used. However, this solution does map well to the REST principles, as it provides operations on resources and uses HTTP semantics correctly.

Option 6. POST instead of PUT
The final solution we'll discuss is to model all commands as POST actions to specific resources. As we saw in the above examples, it is very hard to map a business model to explicit resources. At the same time it moves the business logic to the client and as such creates a tight coupling, which is undesirable and can lead to errors.

"Where to PUT your POST"

Michiel van Oudheusden & Loek Duys

We can solve the low level CRUD APIs by introducing business process resources which express the intent of the operation. Consider the creation of a customer. Most likely this is not a simple process as it might require sub processes to go along, emails to be sent out, records to be created etc. The business intent is to enroll the customer, so a CustomerEnrollment endpoint can be used to actually create the customer itself.

```
POST /api/customers/CustomerEnrollment
Content-Type: application/json

{
    body
}
HTTP/1.1 202 OK
Location: http://example.org/api/customers/CustomerEnrollment/32453
Retry-After: 3
```

The resource returned is a CustomerEnrollment entity and tells the caller the state of the actual enrollment instead of the customer itself. You can also add an additional header named Retry-After that specifies the amount of seconds it will likely take to change the resource state at the server side.
Removing an address can be expressed as follows:

```
POST /api/customers/421/AddressRemoval
Content-Type: application/json

{
  "id": "3",
  "reason": "reason for removal",
  "onBehalfOf": "user name"
}
The response would look like this:
HTTP/1.1 202 OK
Location: http://example.org/api/customers/421/AddressRemoval/631
Retry-After: 3
```

Until the Address Removal command is completed, the address is still present in the customer resource. When completed, this specific instance of AddressRemoval is no longer available. As you can see, the payload is also tailored to the specific command and not to the actual entity it should alter.

This is similar to the command endpoint solution, but it is much easier to discover as it can be advertised in an API definition file like OpenAPI.

By introducing a slight variation on this solution, you can also model workflows, like the change-approval we mentioned earlier. Instead of using a command-based endpoint, you would use one based on a workflow. For example, to start an approval process you would send this web request:

```
POST /api/customers/421/CustomerChangeApproval
Content-Type: application/json

{
  "id": "3",
  "telephone": "new number",
  "onBehalfOf": "user name"
}
```

When you regard the workflow as a separate resource, it becomes easy to manage the workflow by using GET, POST, PUT and DELETE. For instance, you would use GET to retrieve a workflow, and approve the change with a PUT request, or reject it by using DELETE. You probably noticed that this re-introduces the troubles we discussed earlier, but there is a major difference... Instead of having one PUT endpoint for all workflows (and changes, and commands), we have narrowed the scope down to manage just one workflow.

Conclusion

When building a Web API, there are a lot of strategies to choose from. In our project we combined HTTP APIs with CQRS and Event Sourcing, and in that situation our choice was Option 6, the 'POST instead of PUT'-pattern. This option is particularly suitable for resources where the GET maps nicely to your data structures, but poorly to your business domain for mutations. It solves the problems introduced by applying the CQRS/ES architecture and offers the benefits of a RESTful HTTP design; it is discoverable, and it uses the correct verbs and endpoints. The fact that we return a status endpoint (instead of the new resource state) allows for asynchronous processing of commands. </>

"Too bad the post office isn't as efficient as the weather service."

Dr. Emmett Brown

Enterprise-ready Xamarin.Forms

Building mobile applications has become much easier for .NET developers since the dawn of Xamarin.Forms. Although the framework is capable of building graphically rich mobile experiences, it is often the go-to platform for line-of-business or enterprise applications. While we can all start coding our way using File → New Project, it might not be the best approach for these types of apps.

Author Gill Cleeren

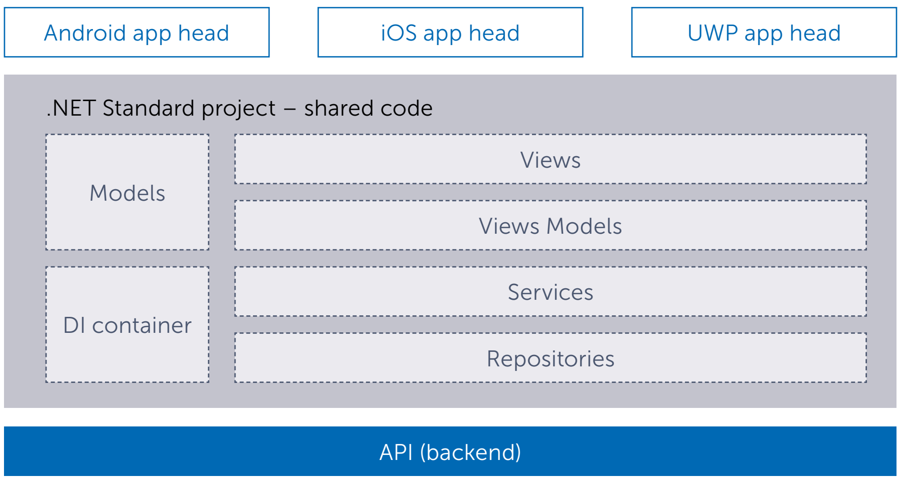
Running in an enterprise environment means that they'll often be subject to changing requirements (yes, it does seem that sometimes customers have changing demands...). Dealing with this changing environment means that we'll need to harness the apps with a decent set of unit tests so we can be confident about the changes we'll need to make. And this in turn requires that we set up an architecture for these mobile apps that lends itself to being tested easily. If we come back to the result of File → New Project, well, it's safe to say that this is not the ideal starting point. In this article, we'll talk about some of the architectural considerations we need to make when building mobile apps that are ready for the enterprise.

Layers in mobile apps

Since we were kids (OK, maybe now I'm exaggerating...), we've been taught that we should layer our software. That paradigm hasn't really changed when building mobile apps with Xamarin.Forms. The big plus of Xamarin.Forms is the huge amount of shared code we typically will get (easily up to 80% for real apps). That code is what we should focus on since this is where the action will take place. The following diagram shows a proposed approach which is definitely nothing really special if you've been using a layered architecture in other types of projects.

At the bottom of the stack we have a repository layer that will typically handle all interactions with regard to data and webservice access, so that the rest of the code doesn't get littered with these low-level details. The Service layer will typically be used for the business functionality and will interact with several repositories to combine their responses. A very important third layer in this approach is the view model layer.

Introducing a view model and thus also the MVVM pattern will be key in creating testable apps. From an MVVM point of view, the services (and the repositories that they use) act as the model. Finally, the top-most layer will be plain views, consisting of data-bound XAML that will use the view models as their



binding source. The magic of data binding and change notifications will ensure that the views will be loosely coupled to the view models. Again, while this structure is far from unique, it will introduce loose coupling in this application, thus increasing the ability to test and maintain it, which is what we set out to achieve in the first place.
Now that we have an overall view of the structure of a typical Xamarin.Forms application, let's zoom in on some of these layers in more detail and see some typical approaches used in the respective layers.

Accessing data

It's pretty hard to imagine any enterprise application that won't be working with data. Most of the data used in mobile apps will reside on the server and services will make sure that they are accessible from the app. Most apps will probably use REST services for this purpose, but other options such as WCF will work from Xamarin, albeit not always in full force. Talking with these services will typically be done using HttpClient while again other options exist. Today's REST services will most commonly exchange JSON, and in Xamarin.Forms apps this JSON can be parsed using JSON.NET.
The following code snippet shows some code that will be used to access a service.

```
var httpClient = new HttpClient();
var response = await httpClient.GetAsync
        (new Uri("https://api.github.com/events"));
if (!response.IsSuccessStatusCode)
    throw new HttpRequestException(response.ReasonPhrase);
string jsonResponse = await response.Content.ReadAsStringAsync();
var json = JsonConvert.DeserializeObject<T>(jsonResponse);
return json;
```

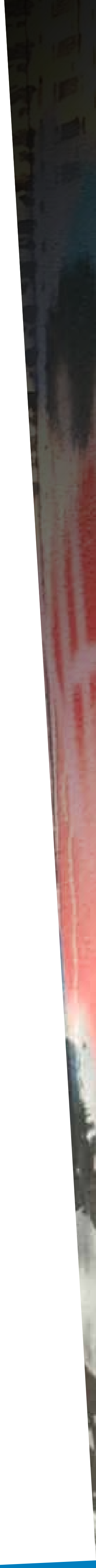
Mobile apps for the enterprise, and in fact all mobile apps, will be used in unpredictable circumstances. People use the app while on the road, inside a concrete building, in and out of a wifi-covered area and so on. Reliable network and therefore a reliable way to communicate with a backend service is often a luxury. However, apps need to be resilient to these possible network interruptions and preferably retry the service communication if possible. To solve the latter problem, we can try to code a retry-mechanism that will attempt to restore the connection after it has failed. While that's not impossible to do, it's easier if someone has already done this work for us. Polly (https://github.com/App-vNext/Polly) is a resiliency library that's commonly used in (mobile) apps to tackle possible failures in communicating with web services. Low-level stuff such as retrying the connection belongs in the repository classes. In the next snippet, you can see how we have wrapped the call to the backend using Polly, and have applied a retry-mechanism that will retry the call if the backend was unavailable for some reason. The setup of the retry mechanism is such that it uses an exponential value between different attempts.

```
var responseMessage = await Policy
    .Handle<WebException>(ex =>
    {
        Debug.WriteLine($"{ex.GetType().Name + " : " + ex.Message}");
        return true;
    })
    .WaitAndRetryAsync
    (
        5,
        retryAttempt => TimeSpan.FromSeconds(Math.Pow(2, retryAttempt))
    )
    .ExecuteAsync(async () => await httpClient.GetAsync(uri));
```

In addition to retrying, another optimization that can be done in this area is caching. Mobile apps shouldn't put load on the server to retrieve data that they may already have. Through caching, we can quite simply store data on the device. There are of course a number of options to do this. One way that I particularly like is using Akavache here. Akavache[1] is a key-value store that has many usages. The way I use it here is simply for throwing some data at it that I want to cache. The data will be stored with an expiration date and so when the data is retrieved from the cache, Akavache will check whether the locally-stored version is still valid. If so, it will be returned, if not, a new version can be fetched from the underlying data source and cached in Akavache automatically. While caching can be a lifesaver in many situations, it can also cause problems in your application. Before applying it, think whether it makes sense on that data to actually cache it. In the snippet below, you can see that we're checking whether we can find data in the Akavache cache and return it if found.

```
public async Task<Observable
Collection<Event>> GetAllEventsAsync()
{
    List<Event> eventsFromCache =
    await GetFromCache<List<Event>>
    (CacheNameConstants.AllEvents);

    if (eventsFromCache != null)//
    loaded from cache
    {
        return eventsFromCache.
        ToObservableCollection();
    }
    else
    {
        UriBuilder builder = new
        UriBuilder(ApiConstants.
        BaseApiUrl)
        {
            Path = ApiConstants.
            CatalogEndpoint
        };

        var events = await _generic
        Repository.GetAsync<List
        <Event>>(builder.ToString());

        await _cache.InsertObject
        (CacheNameConstants.AllEvents,
        events, DateTimeOffset.Now.
        AddSeconds(20));

        return events.ToObservable
        Collection();
    }
}
```

[1] https://github.com/reactiveui/Akavache)



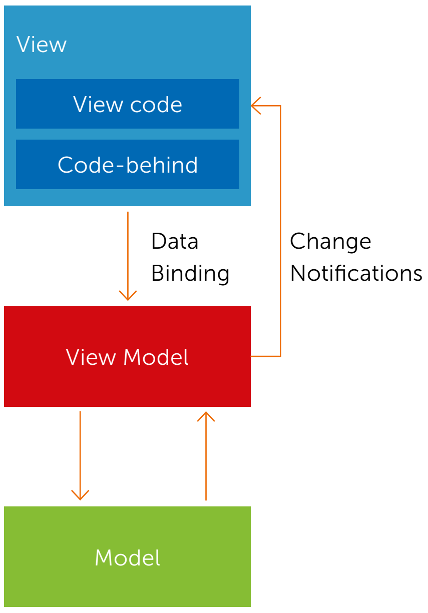
“In my opinion, the future of mobile is the future of everything.”

Matt Galligan, Co-founder of Circa

Gill Cleeren

MVVM to rule them all

In the quest to achieve loose coupling and a high(er) level of testability, we will undeniably run into UI code. This code is rather hard to test so the only option we have here is launching on an emulator and clicking/tapping through the screens. However, this is not what we intended in the first place, isn't it? The pattern that will help us here is MVVM, the Model-View-View-Model pattern. I'm sure you've already heard of it, it's a pattern that became popular at the time of WPF and (yes!) Silverlight. It's built on the foundations of XAML, data binding and commanding, and those are indeed available in Xamarin. Forms as well. The following diagram shows the structure of the involved classes. The View code is still XAML but now it contains data bound to the view model. The view model is basically an abstraction of what is presented in the view and doesn't contain actual UI elements. It will also implement the behavior, such as the interaction with the model for us. The view model will expose state (=data) and operations (=commands) to the view. Data binding and the built-in change notification system based on the INotifyProperty-Changed interface will ensure that the view is updated automatically when the data changes in the view model.



A view model is typically just a class which, as mentioned, exposes state and operations for the view to bind to. Here you can see a simple view model for a login screen, which requires a user name and password. Essentially, this is the data for that screen. Next, interactions such as clicking on a login button, which would typically be handled using an event handler in the view's code-behind, will now be wrapped inside a command in the view model instead. Commands are used to wrap functionality which can be called from other places in the application. In this case, they will wrap the behavior to handle a UI event.

```
public class LoginViewModel : ViewModelBase
{
    private ICommand _loginCommand;

    public string UserName
    {
        get { return _userName; }
        set
        {
            _userName = value;
            OnPropertyChanged(nameof(UserName));
        }
    }

    public string Password
    {
        get { return _password; }
        set
        {
            _password = value;
            OnPropertyChanged(nameof(Password));
        }
    }

    public ICommand LoginCommand => _loginCommand ?? (_loginCommand =
    new Command(OnLogin));
}
```

Simple view models

You may get the idea that a view model will simply contain all the code that originally was located inside the code-behind and that we've essentially just been moving some boxes around. That wouldn't be of much help, now would it? One of the key aspects is that view models should be as simple as possible. They are like the controller in MVC applications, and those too should remain simple. They know about the flow of the applications but they don't know how to perform navigation. They know that because of a certain event in the application, a dialog should be shown, but they don't know HOW to display that dialog.

Keeping all this knowledge outside of the view model is essential to keeping them easy to test later on. All this "external" knowledge about how to navigate, how to show a dialog, how to check whether we are connected with the internet and so on should be pushed into a separate service class, which in essence is nothing more than a simple class that is capable of just one single piece of functionality. It's a good example of using the Single Responsibility Principle. Think of a navigation service, a dialog service, a connection service, and many others. In a real-life application, you'll end up with quite a few of these. Below, you can see (part of) a dialog service. To display dialogs, we use another library called ACR Dialogs and that's wrapped inside this simple service.

```
public class DialogService : IDialogService
{
    public Task ShowDialog(string message, string title, string buttonLabel)
    {
        return UserDialogs.Instance.AlertAsync(message, title, buttonLabel);
    }

    public void ShowToast(string message)
    {
        UserDialogs.Instance.Toast(message);
    }
}
```

Services are commonly registered in the application by means of a dependency injection container such as Autofac or TinyIOC. These containers work perfectly fine in Xamarin.Forms apps and allow us to register service classes during the bootstrapping of the application. In the following snippet you will see that we're using the container and registering some of the classes we'll typically have in this type of applications, such as a view model and a service class.

```
public class AppContainer
{
    private static IContainer _container;

    public static void RegisterDependencies()
    {
        var builder = new ContainerBuilder();

        //ViewModels

        builder.RegisterType<LoginViewModel>();
        builder.RegisterType<DialogService>().As<IDialogService>();

        _container = builder.Build();
    }
}
```

Once registered, view models will get an instance of these services injected through dependency injection. These instances are then invoked to perform the actual functionality such as showing the actual dialog. Note that indeed it's the view model that will know that a dialog needs to be shown, but it doesn't know how to do this. That's the responsibility of the DialogService class.

"Hello, is this View Model? Yes, this is View Model"

Remember that at the beginning of this article we set out to create a loosely coupled architecture that's easy to test? Well, we have another problem to solve. Very often, view models will need to interact with other view models. Think of a Settings View Model that needs to let other view models know that the user has switched the currency. Our first thought might be that we would have a direct reference from this Settings View Model to all interested view models. While that would work, we would end up with references from one view model to the next, and this brings tight coupling with it, which is not what we were aiming for! This means that the view models need another way of communicating, and the preferred way of doing so is through a messenger using a pub-sub model. In this model, a view model will register to send messages to the messenger, and other view models will register to receive updates from that messenger. Xamarin.Forms comes with support for this pattern, built-in through the Messaging-Center class. You can see an example of this below.

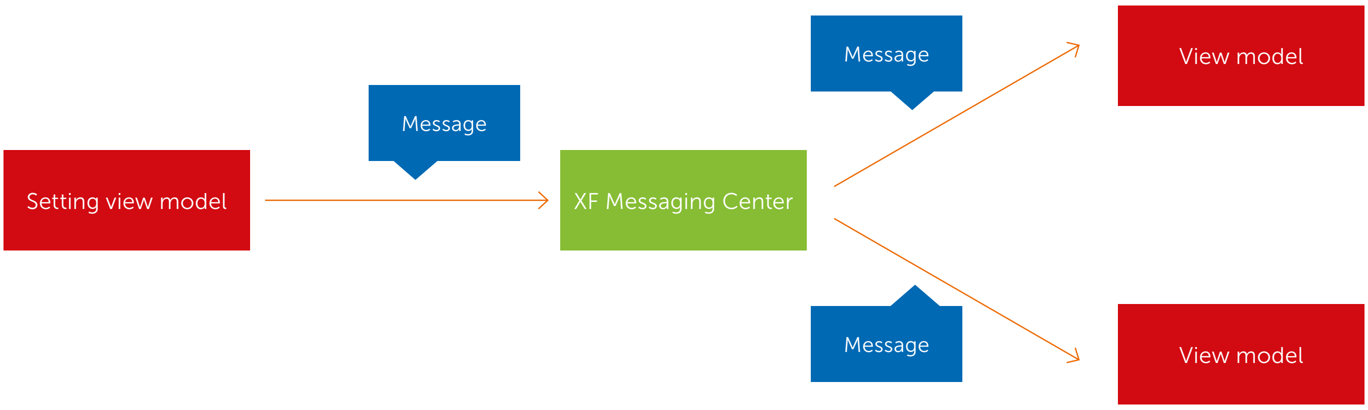
Registering to receive the message:

```
public async override Task Initialize
Async(object data)
{
    MessagingCenter.Subscribe
    <Currency>(this, Messaging
    Constants.
    CurrencyChanged, OnCurrency
    Changed);
}
Sending the message:
private async void OnChangeCurrency()
{
    MessagingCenter.Send(this,
    MessagingConstants.Currency
    Changed, SelectedCurrency);

}
```

Putting things to the test

Now that we have separated everything nicely, can we actually test the view models and thus create a more robust code base? Well, the answer is a definite YES! Take a look at the following snippet in which we are creating a unit test for one of the view models in the application.

```
[Fact]
public void LoginCommandIsNot
NullTest()
{
   var authenticationService = new
   AuthenticationMockService();
   var loginViewModel = new Login
   ViewModel(authenticationService);
   Assert.NotNull(loginViewModel.
   LoginCommand);
}
```



Summary

Creating loose coupling and testable applications is definitely applicable for mobile applications with Xamarin.Forms. The patterns we've described here definitely put us on the right track to create Xamarin.Forms apps that will be easier to test and maintain in the long run. And that's exactly what enterprises are looking for right now for their mobile endeavors. </>

.NETCore.With(“vsCode”).Should().Have(“Unit Tests”).Part(“II”).

In our previous article[1], in XPRT Magazine #7, we showed how to get started with unit testing .NET Core projects using VS Code (an excellent code editor, in our opinion). In this follow-up article we'll continue with:

- What is new in the latest releases of .NET Core and VS Code related to unit testing?
- How to run unit tests across multiple projects.
- How to collect test coverage results across multiple projects.

Authors Reinier van Maanen & Marc Duiker

What's new?

.NET Core / ASP.NET Core

Among the many updates of .NET Core 2.x and ASP.NET Core, the most notable change of the last few months related to testing is the Microsoft.AspNetCore.Mvc.Testing[2] package in ASP.NET Core 2.1.

This package streamlines integration test creation and execution and handles the following tasks:

- Copies the dependency file (*.deps) from the tested app into the test project's bin folder.
- Sets the content root to the tested app's project root so that static files and pages/views are found when the tests are executed.
- Provides the WebApplicationFactory class to streamline bootstrapping the tested app with TestServer.

Example usage of the WebApplicationFactory in an integration test:

```
public class BasicTests
    : IClassFixture<WebApplicationFactory<RazorPages
      Project.Startup>>
{
    private readonly HttpClient _client;

    public BasicTests(WebApplicationFactory<RazorPages
    Project.Startup> factory)
   {
        _client = factory.CreateClient();
    }

    [Fact]
    public async Task GetHomePage()
    {
        // Act
        var response = await _client.GetAsync("/");

        // Assert
        response.EnsureSuccessStatusCode();
        // Status Code 200-299
        Assert.Equal("text/html; charset=utf-8",
            response.Content.Headers.ContentType.
            ToString());
    }
}
```

So while the above isn't about 'pure' unit testing, it's still a valuable addition to your testing arsenal, which enables you to write integration tests for Razor Pages with minimal effort.

If you want to write end-to-end tests for web apps, then use a tool such as Selenium[4]. Do not use Coded UI tests for this, as this is deprecated[3].

[1] https://xpirit.com/netcore-withvscode-should-haveunit-tests/
[2] https://docs.microsoft.com/en-us/aspnet/core/release-notes/aspnetcore-2.1?view=aspnetcore-2.2#integration-tests
[3] https://www.seleniumhq.org/
[4] https://docs.microsoft.com/en-us/visualstudio/test/use-ui-automation-to-test-your-code?view=vs-2017

Reinier van Maanen & Marc Duiker

Unit testing Libraries & Frameworks
The mocking framework **NSubstitute** had a major release to version 4.0.0. The breaking change is related to argument matchers, Arg.Is, Arg.Any etc., which now use ref returns, a C# 7.0 feature. This change allows proper support for working with ref and out arguments.

VS Code extensions
The **.NET Core Test Explorer** extension had six new releases[5] since the previous article and is currently at version 0.6.3. It has been improved to support multiple workspaces and includes numerous bug fixes.

The **Coverage Gutters** extension had four new releases[6] and is now at version 2.3.1. It contains dozens of bug fixes and performance improvements by making better use of async operations.

Coverlet, the cross platform code coverage tool for .NET Core has had eight new releases[7], including two major versions, and is now at version 4.1. The changes include several performance enhancements and a feature to compute cyclomatic complexity.

Running tests across multiple projects
In the previous article, we showed a simplified situation of one .NET Core console project with one corresponding XUnit test project. In real life however, you will have dozens of projects, with many of them having test projects as well. This requires a different set up of your VS code files in order to run tests across multiple projects. A couple of examples are shown below, and you'll probably end up combining a few of them.

Tasks.json
An easy way to build multiple projects is by extending the tasks.json file, making use of the dependsOn property of a task:

```
{
  "label": "build Project.A",
  "command": "dotnet build
  Project.A /property:Generate
  FullPaths=true",
  "dependsOn": "clean Project.A",
  "problemMatcher": "$msCompile",
  "type": "shell",
  "group": {
    "isDefault": true,
    "kind": "build"
  },
},
{
  "label": "clean Project.A",
  "command": "dotnet clean
  Project.A",
  "dependsOn": "build Project.B.
  UnitTests",
  "problemMatcher": "$msCompile",
  "type": "shell"
},
{
  "label": "build Project.B.
  UnitTests",
  "command": "dotnet build
  Project.B.UnitTests /property:-
  GenerateFullPaths=true",
  "dependsOn": "clean Project.B",
  "problemMatcher": "$msCompile",
  "type": "shell"
},
{
  "label": "clean Project.B",
  "command": "dotnet clean
  Project.B",
  "dependsOn": "clean Project.B.
  UnitTests",
  "problemMatcher": "$msCompile",
  "type": "shell"
}
```

When you execute tasks 'build Project.A', it will first try to execute 'clean Project.A' because it depends on that step. 'clean Project.A' has a dependency on 'build Project.B', which depends on 'clean Project.B'. This means that the tasks will be executed in the following order:
- Clean Project.B
- Build Project.B
- Clean Project.A
- Build Project.A

[5] https://github.com/formulahendry/vscode-dotnet-test-explorer/releases
[6] https://github.com/ryanluker/vscode-coverage-gutters/releases
[7] https://github.com/tonerdo/coverlet/releases

Of course, this only cleans and builds. You can add dotnet test as well, but in our experience running all unit tests on each build isn't very effective. Create a separate task for that, or use the Test Explorer. Another way to do this, is described in a blog post by Scott Hanselman[8]. He uses dotnet watch[9] to trigger tests whenever source code changes. This still runs all tests, and while it isn't like Visual Studio's awesome Live Unit Testing, it's a step.

The downside to this approach is that the tasks.json can become quite big and uneasy to maintain. Read on for some ways around this.

A last interesting bit is the 'Generate-FullPaths' property. This doesn't have anything to do with building multiple projects, but without this, any compiler errors in VSCode aren't clickable in the error window which degrades usability.

PowerShell Script

Another way to build multiple projects is by combining PowerShell and the tasks.json. Create a buildsolution.ps1 file and add the following and anything else you require:

```
dotnet clean Project.B
dotnet build Project.B
/p:GenerateFullPaths=true
dotnet clean Project.A
dotnet build Project.A
/p:GenerateFullPaths=true
```

You can then call this script from the tasks.json file in a custom task:

```
{
  "label": "build",
  "command": "powershell",
  "args": [
    "-ExecutionPolicy",
    "Unrestricted",
    "-NoProfile",
    "-File",
    "${cwd}/buildsolution.ps1"
  ],
  "type": "shell",
  "problemMatcher": "$msCompile",
  "group": {
    "isDefault": true,
    "kind": "build"
  }
}
```

The advantage here is that this results in an easier to maintain tasks.json file, you can do anything you want in the PowerShell script, and you can even use that same scripts in a build pipeline, making sure the build on a buildserver runs the same way it's run locally. It will require you to use a PowerShell task in your build. As with the tasks.json, the same remarks and suggestions about running tests apply here.

Solution file

You can create a solution file with dotnet new sln and refer to the solution file with the dotnet CLI: `dotnet build ProjectsAplusB.sln`. Ofcourse, this helps clean up the tasks.json as well as you can see below. Using a solution file also has the added benefit that if you have Visual Studio IDE and need one of its features, a switch can be made easily. Also, just as with referencing a Power-Shell script, this gives you the option to create a build pipeline on Azure DevOps which behaves more like a local build. Unlike the PowerShell solution, you can just use the standard dotnet task for that.

Running `dotnet new sln` will just create an empty solution. Adding and removing projects can be done with `dotnet sln add ProjectA` and `dotnet sln remove ProjectB`. You can list all projects in the solution with `dotnet sln list`.

The tasks.json will end up looking like this:

```
{
  "label": "build",
  "command": "dotnet build
  ProjectsAplusB.sln /property:
  GenerateFullPaths=true",
  "dependsOn": "clean",
  "problemMatcher": "$msCompile",
  "type": "shell",
  "group": {
    "isDefault": true,
    "kind": "build"
  },
},
{
  "label": "clean",
  "command": "dotnet clean
  ProjectsAplusB.sln",
  "problemMatcher": "$msCompile",
  "type": "shell"
}
```

Test Explorer

In the previous article, we mentioned the Test Explorer extension, which gives you a GUI for running all your unit tests. Making sure the Test Explorer picks up tests from all projects is very easy. Just change the value of `dotnet-test-explorer.testProjectPath`, making use of wildcards: Change `"/ProjectA.Tests"` to `"/*.Tests"` and you're done. There is a problem with this if you also use Test Explorer to generate coverage files with Coverlet as we showed you in the previous article. Read on to learn more!

Collecting test coverage results across multiple projects

Configuring Test Explorer to run tests from multiple projects and also configuring it so that Coverlet writes its output to disk results in an issue: for every unit test project a separate coverage file is written, and Coverage Gutters won't merge the results. Simply configuring Coverlet to write the results to 1 file also doesn't work, the file is overridden for every project so, after the entire run, only the coverage of the last project is visualized by Coverage Gutters. Luckily, there is a way to configure Coverlet to merge the results but, it's not easy:
Supply these arguments as value for `dotnet-test-explorer.testArguments`:

```
"dotnet-test-explorer.testArguments":
"--filter Category!=Integration
/p:CollectCoverage=true \"/p:Coverlet
OutputFormat=\\\"json,lcov\\\"\"
/p:CoverletOutput=..\\lcov /p:Merge
With=..\\lcov.json" (yes including
all the escaping and extra quotes)
```

When running the tests, this will create a lcov.json and lcov.info in the root of the workspace. The json file is in an coverlet specific format and is just a simple JSON file, which has some benefits like being able to use it in the MergeWith parameter. The lcov file is still needed, because this is used by Coverage Gutters. What happens with the above configuration is that the lcov.json is merged for each unit test project and then a new lcov.info is generated, based on the merged file.

[8] https://www.hanselman.com/blog/AutomaticUnitTestingInNETCorePlusCodeCoverageInVisualStudioCode.aspx
[9] https://docs.microsoft.com/en-us/aspnet/core/tutorials/dotnet-watch?view=aspnetcore-2.2

The end result is one big lcov.info file with coverage from all test projects. It's not in any project directory, so Coverage Gutters won't detect it, and a workaround is needed to enable proper detection.

There is another issue here, because running the tests will merge any existing lcov.json file, also those from a previous run. So there's still some work to do at the Coverlet plugin. Other issues are that this will not update the merged file properly when tests are removed (it seems only to add coverage lines and not remove lines which aren't covered anymore) and last but not least, running the tests with these arguments will crash if there isn't a file to merge with (which is troublesome with the first project you'll run tests on). Read on for some workarounds!

Workaround for Coverage Gutters not picking up the coverage file

At the moment Coverage Gutters only looks for lcov files in project directories. As mentioned before, it won't merge results if it finds multiple lcov files in multiple project directories, so we used Coverlet to merge. The workaround is quite trivial, just write the merged lcov file to one of your project directories:

```
"dotnet-test-explorer.testArguments":
 "--filter Category!=Integration
/p:CollectCoverage=true \"/p:Coverlet
OutputFormat=\\\"json,lcov\\\"\"
/p:CoverletOutput=..\\Project\\lcov
/p:MergeWith=..\\Project\\lcov.json",
```

There is a GitHub issue[10] here about being able to direct Coverage Gutters to a specific coverage file. As soon as this has been implemented this workaround shouldn't be needed anymore.

Workaround for resetting the coverage for each run and preventing a crash when running for the first time

This workaround is a bit dirty (as workarounds always tend to be). You can alter the csproj of one of your **test-projects**. Don't pick a project that contains the implementation, depending on your build configuration that one is built more than once: one time by itself and one time as a dependency of your testproject. Include this:

```
  <Target Name="ResetCoverageFile"
AfterTargets="Build">
    <Copy SourceFiles="..\Project\
    lcov.empty" DestinationFiles="..\
    Project\lcov.json" />
  </Target>
```

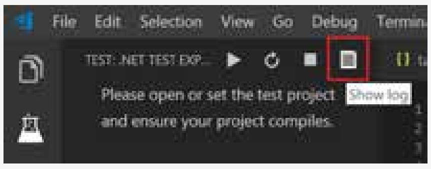
lcov.empty is an empty json file, so this results in a clean slate each time Coverage Gutters runs, builds all projects and executes all tests, resulting in up-to-date coverage and fixing the problem with the first test run.
Of course all of the above: from running the tests with coverage to clearing previous results can also be added to a powershell or any other script and then bound to a build task. You can then check coverage without the Test Explorer in a fairly easy way.

A small issue that remains is that coverage won't be updated when executing a single test, but that's fine for most people.

Debugging / viewing total coverage percentage

If you're someone who likes to measure code quality by total test coverage, the easiest way to see the coverage percentage is by viewing the log of the Test Explorer. Of course, we don't have to tell you that coverage by itself doesn't mean much!

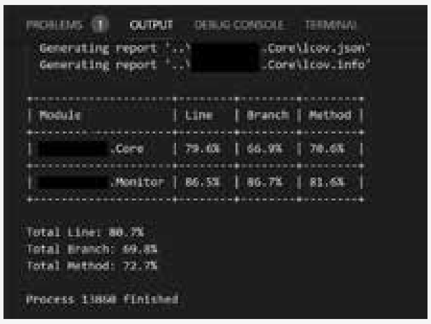
An interesting alternative, which is still very much under development, is Stryker[12]. It alters your code right before tests run, and checks whether at least one unit test fails. Read more on their website. For now, to check the coverage percentage:



Please open or set the test project
and ensure your project compiles.

Show log button

The output will be something like this:



```
 Generating report '..\[redacted].Core\lcov.json'
 Generating report '..\[redacted].Core\lcov.info'

+-------------------+--------+--------+--------+
| Module            | Line   | Branch | Method |
+-------------------+--------+--------+--------+
| [redacted].Core    | 79.6%  | 66.9%  | 70.6%  |
+-------------------+--------+--------+--------+
| [redacted].Monitor | 86.5%  | 86.7%  | 81.6%  |
+-------------------+--------+--------+--------+

Total Line: 80.7%
Total Branch: 69.8%
Total Method: 72.7%

Process 13868 finished
```

Show log output

These logs can also be very useful when debugging issues, so if things aren't working have a look here.

Conclusion

As shown by the large number of VS Code extension releases the tooling landscape related to unit testing is evolving and improving at a rapid pace. For running tests across multiple test projects, it appears that using a sln file would still be the easiest way and this also allows developers to use both VS Code and Visual Studio IDE.

We hope this article has given you a better understanding of how to configure unit testing for .NET Core projects in VS Code. If you have any further questions or comments, don't hesitate to contact us. </>

"Being proud of 100% test coverage is like being proud of reading every word in the newspaper. Some are more important than others."

Kent Beck on Twitter[11]

[10] https://github.com/ryanluker/vscode-coverage-gutters/issues/178
[11] https://twitter.com/KentBeck/status/812703192437981184?s=09
[12] https://github.com/stryker-mutator/stryker-net

Source Server Indexing and Symbol Server Management with Azure DevOps

Developers debug their applications on a daily basis and everyone must have experienced the power of debugging. But what if you want to debug a crash dump or what if you want to debug a NuGet package in your application? The concept of Source Server Indexing and Symbol Server Management is still not a widely known practice in the field, but setting up a Source Server and Symbol Server in an enterprise development environment can be extremely valuable. If you see how easy it is to set things up with Azure DevOps, it should be mandatory for every software application you are working on. It can make your life so much easier, and not only yours, but also the lives of many other developers.

Authors Pieter Gheysens

There are two main reasons why you should embrace the concept of Source Server Indexing and Symbol Server Management:

1. Live Debugging
   During development, it will help anyone who is referencing assemblies that are built with a source-server-enabled build, to debug those assemblies with the original source code. Think of NuGet packages for example. How annoying can it be when you are using a NuGet package and you cannot step into the original source code?

2. Debugging Crash Dump Files or Snapshots
   Every application which is pushed to production should allow easy troubleshooting and easy debugging when something bad happens. This can simply not be done when you cannot rely on source server indexing and a central symbol server. Rest assured, something bad will eventually happen and you want to be ready for this when users are sending you crash data, Snapshots, or IntelliTrace log files.

Why are pdb files so important?

Let's get to the basics first and start with the importance of pdb files (also known as symbol files). Every developer in the Microsoft ecosystem probably has already seen these files, but to my surprise not a lot of developers actually know how important these files can be and how they work.

Program database (PDB) is a proprietary file format (developed by Microsoft) for storing debugging information about a program (.dll / .exe) and is created from source files during compilation. It stores a list of all symbols in a module with their addresses, together with the name of the source file and the line on which the symbol was declared. These files are only created once during the compilation process and are uniquely matched with the binaries. This process cannot be forged afterwards.

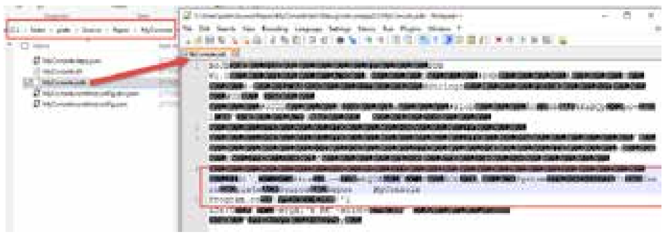
In essence, the pdb files help developers to load all debugging information (variables, function names, source line numbers) in the development environment (Visual Studio) while "debugging". In addition, they provide the capability to step into the original source code files via breakpoints, watch variables, and perform many other useful tasks related to the art of debugging. WinDbg (The Windows Debugger) can also be used to debug application code and analyze crash dumps.

[1] https://en.wikipedia.org/wiki/Program_database

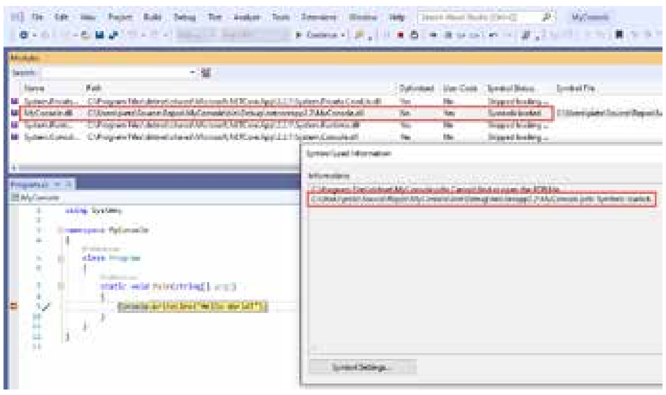
In both scenarios, you always must obtain the proper symbols for the code you wish to debug, and load these symbols into the debugger. In short: no debugging without a matching pdb file. With .NET Core we can now do similar things on Linux (LLDB Debugger, ProcDump or SOS plugin), but this will be out of scope for this article.

Creating a simple Console Application in Visual Studio and compiling/building the project will drop this pdb file next to the assembly file (exe/dll).

Looking more closely at the content of the pdb file, you will notice that somewhere the file path to the Program.cs source file can be found.



So, when debugging the MyConsole application in Visual Studio you will notice that the symbols are loaded from the pdb file and this allows the editor to dive into the source code while running the application.



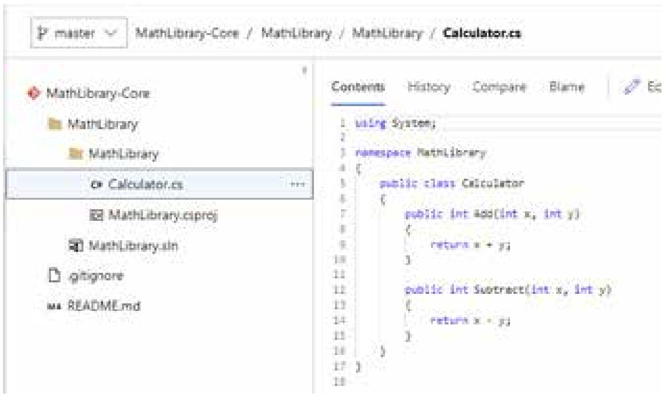
The editor can find a valid pdb file by means of the file name and the location of the pdb file (probing). What's also key is that it must be the exact pdb file that was composed during the compilation process, and the handshake is done through a GUID that is embedded in the assembly file (.dll) and the pdb file. If the GUID of the assembly and the pdb file do not match, the editor won't be able to debug the module at the source code level, and there's no way to override this.
This emphasizes the importance of storing your pdb files because without these files, you are losing control over the entire debugging process.

Of course, this always automatically works for local development (private builds), and there won't be a mismatch between the local running application and the underlying pdb files. But what about public builds where the sources are compiled on an independent build server, and where the output assemblies are stored as artifacts?

Pieter Gheysens

Consuming NuGet packages from Azure DevOps Artifacts

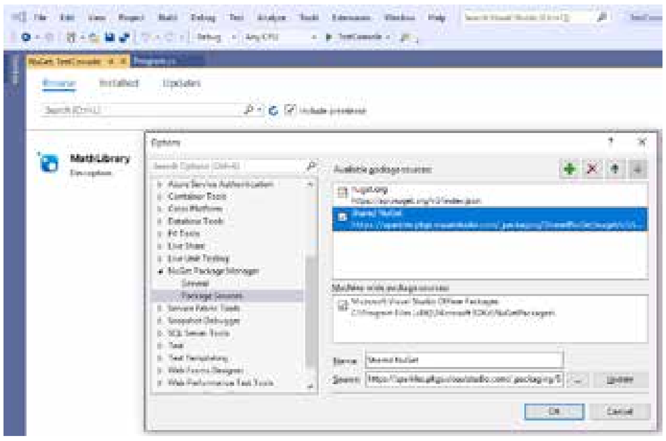
A good example to show the need for Source Server Indexing and a Symbol Server is the use of (internal) NuGet packages. I have created a new Git repository in Azure DevOps and added a .NET Core Class Library with a Calculator class that provides two basic methods (Add and Subtract).



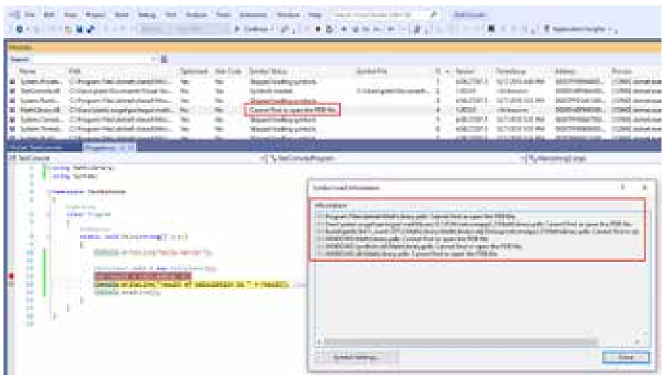
```
using System;

namespace MathLibrary
{
    public class Calculator
    {
        public int Add(int x, int y)
        {
            return x + y;
        }

        public int Subtract(int x, int y)
        {
            return x - y;
        }
    }
}
```

I also have created a Build Pipeline in Azure DevOps to create a NuGet package from this assembly and to publish the NuGet package to a feed in Azure Artifacts. This NuGet package now becomes available for consumption by all teams with access to this feed.

Adding the package feed url in Visual Studio (NuGet Package Manager > Package Sources) offers developers the option to select the appropriate NuGet package and add it to the current application. No big deal and business as usual, but imagine the functionality inside the NuGet package is a bit more complex and you want to understand how the logic has been implemented while debugging.

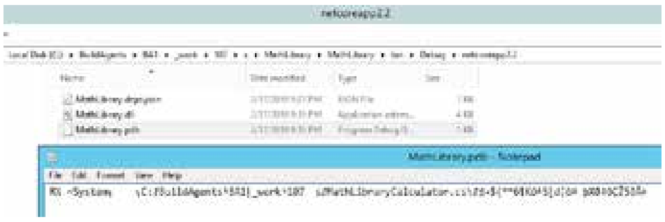


Setting a breakpoint at one of the Calculator methods won't allow you by default to step into the code and see what happens under the hood. This is because Visual Studio doesn't have access to the exact pdb file that was created during the Azure DevOps build process on the build server.



The NuGet package used in Visual Studio delivers the binary file (.dll) but the matching .pdb file is nowhere to be found on the local machine where I'm trying to debug the Add method of the Calculator class. And even worse, the pdb file can't be recovered because the build process didn't take care of storing the file into a shared location (Symbol Server), and a new build will potentially override the old version of the build output in case the same private build agent was used.

However, there's another problem that must be solved in order to provide seamless support for debugging. Let's look at the content of the latest pdb file I could retrieve in the workspace of the private build agent.



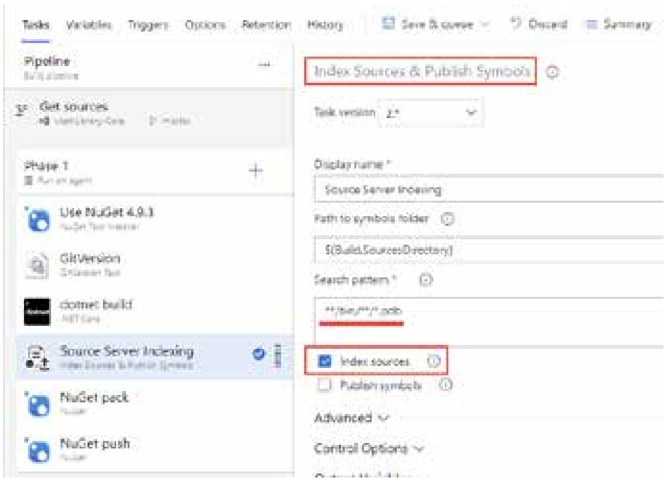
```
RS <System;    \C:\BuildAgents\5A1\_work\107  s\MathLibrary\Calculator.cs
```

If I were able to use the matching version of the pdb file in my debugging session, Visual Studio would be redirected to fetch the exact source file (Calculator.cs) from the hardcoded file path that was used in the build process on the build server. However, it is not our goal to define a similar file structure on your local machine to fake the retrieval of source files, and it can never guarantee that you are providing the exact same source files that were used during the build process.
Let's zoom into the solution to solve the issues above.

Source Server Indexing

When compiling sources on the build agent and producing the pdb files, we must find a way to avoid pointing to a fixed file path of the source files being used in the build process. And that's exactly what Source Server Indexing will do. It's a simple and efficient process to embed a version-control path (including the version identifier) into the pdb file, and ensure that it is readable by Visual Studio or Windbg. This technique allows the editor to retrieve the exact source file directly from the version control system instead of the fixed file path on the build agent.

Since TFS 2010, the build system provides an out-of-the-box solution for embedding this information into the pdb file. I remember using a Perl script in the past to accomplish this manually for TFS 2008, but luckily this has become a simple build task in Azure DevOps and TFS.



Index Sources & Publish Symbols

Task version: 2.*

Display name *: Source Server Indexing

Path to symbols folder: $(Build.SourcesDirectory)

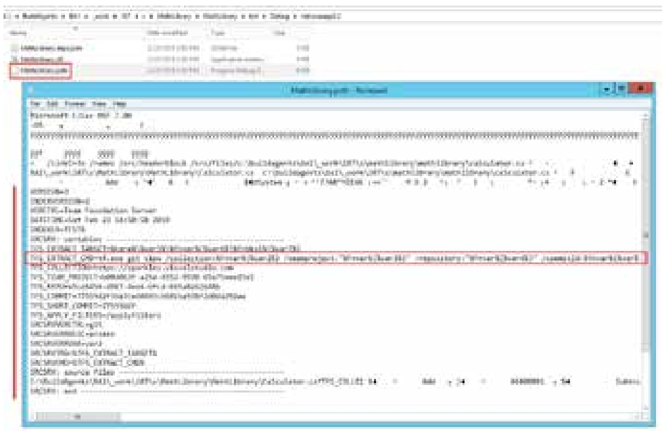
Search pattern *: **/bin/**/*.pdb

[✓] Index sources

[ ] Publish symbols

The above image shows the required "Index Sources & Publish Symbols" build task that will scan for pdb files, and this task will eventually inject extra information into the pdb file to link towards the exact versioned source files being used at compilation time.

Running this build and looking for the content inside the pdb files reveals the magic that was being done inside the build process.



```
[illegible]
VERCTRL=Team Foundation Server
[illegible]
SRCSRV: variables ------------------------------------------
[illegible]
SRCSRV: source files ---------------------------------------
[illegible]
SRCSRV: end ------------------------------------------------
```

A big chunk of extra data has been injected into the pdb file and it now contains a tf.exe command to dynamically extract the source file from a Git repository (via the commit id) inside a Team Project from Azure DevOps or TFS. Note that the variables can still be overridden via a srcsrv.ini file in case the collection url changes for TFS or Azure DevOps.

Another method to enable a similar debugging experience is to use Source Link[2], which is a language-control and source-control agnostic system. Microsoft libraries such as .NET Core and Roslyn have enabled Source Link. For this article I have chosen to explain Source Server Indexing, which doesn't require extra properties in the .NET project.

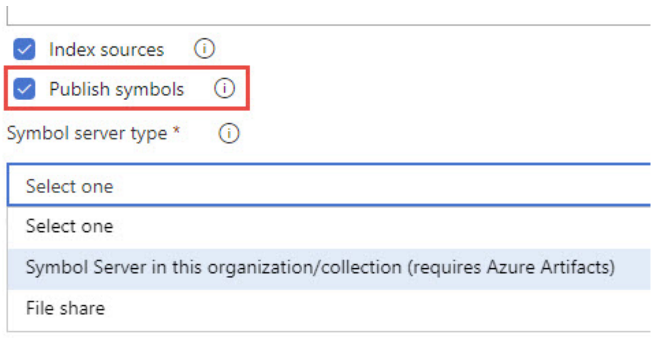
[2] https://github.com/dotnet/sourcelink/blob/master/README.md

Symbol Server Management

Source Server Indexing is only one part of the solution, because storing this pdb file only in the workspace on the build server does not make any sense. We need this particular pdb file in a central location that can be easily searched when starting a debugging session.

This is where Symbol Server Management can play a valuable role. A symbol server enables a debugger to automatically retrieve the correct symbol file (pdb). This is based on the unique GUID that was used in the compilation process on the build server to mark the assembly file and the pdb file. Remember that this linking is a one-time operation and cannot be reproduced after the facts. Losing the pdb file means that you lose the opportunity to debug the output assembly. For ever!

Support for Symbol Server Management is now provided by the same "Index Sources & Publish Symbols" build task. Until now it is only possible to publish the pdb files to Azure DevOps, which is a full-blown Symbol Server. Older versions of TFS or Azure DevOps Server only allows you to push the pdb files to a network share.

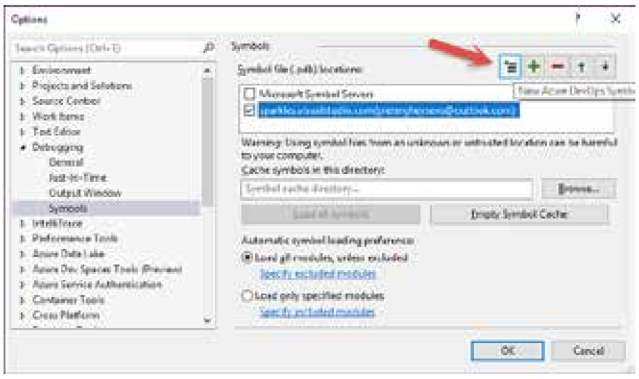

☑ Index sources
☑ Publish symbols
Symbol server type *
Select one
Select one
Symbol Server in this organization/collection (requires Azure Artifacts)
File share

Back to Visual Studio to activate debugging with symbols

In my TestConsole application in Visual Studio I already picked up the latest NuGet package from the Azure Artifacts feed, which was made available via the latest build that included Source Server Indexing and the publication of the symbols to Azure DevOps.

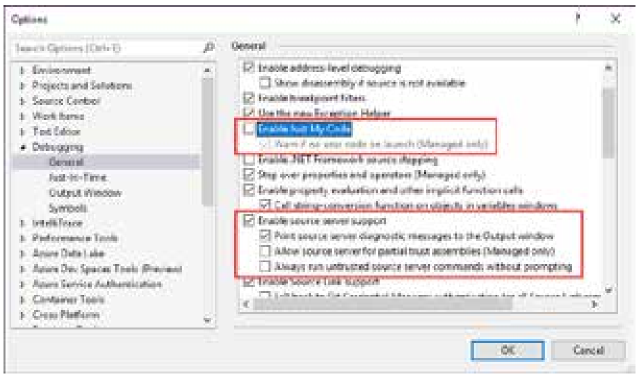
To get the full debugging experience with symbols, you must verify a number of settings that are not turned on by default in Visual Studio.

› Connect/Register the Azure DevOps Symbol Server


Symbol file (.pdb) locations:
☐ Microsoft Symbol Servers
☑ [illegible]
Warning: Using symbol files from an unknown or untrusted location can be harmful to your computer.
Cache symbols in this directory:
Automatic symbol loading preference:
● Load all modules, unless excluded
○ Load only specified modules

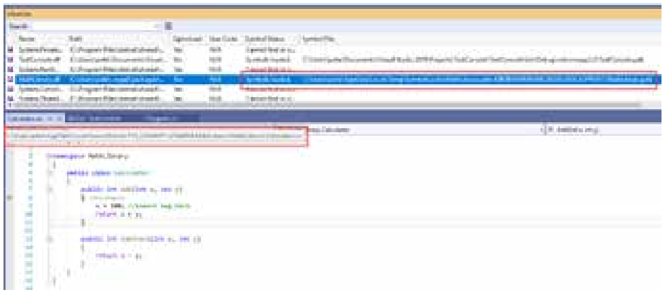
Your Azure DevOps organization will be just another symbol server next to the Microsoft Symbol Servers and you can choose to enable/disable it at any time.

› Disable "Just My Code" and enable "Source Server Support"


☑ Enable address-level debugging
☐ Show disassembly if source is not available
☑ Enable breakpoint filters
☑ Use the new Exception Helper
☐ Enable Just My Code
☑ Warn if no user code on launch (Managed only)
☐ Enable .NET Framework source stepping
☑ Step over properties and operators (Managed only)
☑ Enable property evaluation and other implicit function calls
☑ Call string-conversion function on objects in variables windows
☑ Enable source server support
☑ Print source server diagnostic messages to the Output window
☐ Allow source server for partial trust assemblies (Managed only)
☐ Always run untrusted source server commands without prompting
☑ Enable Source Link support

The first toggle is important to not only debug the sources you manage inside your solution, and the second option is required to fetch the original source files from the pdb file when the debugging process needs the source code.

When trying to step into (F11) the Add method of the Calculator class, Visual Studio will now help to search for the matching pdb file in the Azure DevOps Symbol Server, and the content of the pdb file will instruct Visual Studio to download the Calculator.cs file from the Git repository inside Azure Repos. The pdb file and the Calculator.cs file are now locally available in the cache folder of your computer, ready for live debugging actions.

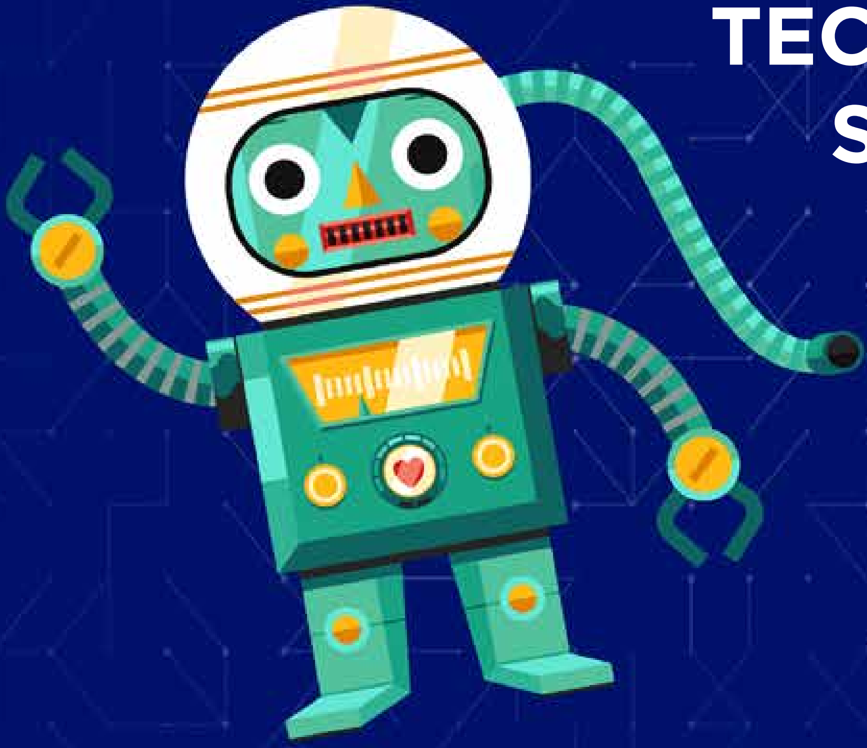


Conclusion

This article will help you solve the issue of not being able to attach the debugger in certain scenarios and step into the original source code, It should provide you with enough information to assess why it's so important to treat your pdb files in the same way as you treat your assembly files that might go to production. Source Server Indexing and the publication of the symbols (pdb files) go hand-in-hand and should always be enabled in your automated build processes that produce output for production. Azure Pipelines provides the right build task to accomplish this for cross-platform applications and the rest of the magic is done inside your favorite debugging tool. </>

TECHORAMA

DEEP KNOWLEDGE IT CONFERENCE



TECHORAMA 2019
SPACE EDITION

WORKSHOPS: SEP 30
CONFERENCE: OCT 1-2

TICKETS ON SALE STARTING MAY 6

WWW.TECHORAMA.NL

www.xpirit.com

Think ahead.
Act now.

If you prefer the digital version of this magazine, please scan the qr-code.