XPRT.

Magazine N°12/2022

Together we build an Engineering Culture

Launching Xpirit IoT: Smart & Connected Services

What's what with WebAssembly?

Shift left using Bicep

Customizing Codespaces

Xpirit as an IT Beehive

Xpirit

Together we drive change.

DEVOPS BOOTCAMP

GLOBAL, LOCAL & VIRTUAL

ACCELERATE DEVOPS ADOPTION WITH THIS EXCLUSIVE DEVOPS EXPERIENCE

HAVE YOU EVER WANTED TO EXPERIENCE WHAT IT'S LIKE TO WORK IN A TEAM THAT PRACTICES REAL DEVOPS? DO YOU WANT TO RUN A DEVOPS BOOTCAMP?

Then this is the event for you! You learn how to build software with immediate feedback loops and push it to production, multiple times a day, without hesitation. You will be able to translate everything into your daily practices and initiate your DevOps transformation based on experience instead of text-book examples.

DO YOU WANT TO RUN A DEVOPS BOOTCAMP? CONTACT MAX FOR ALL OPTIONS.

Max Verhorst / +31 (0)6 13 46 80 02 /
mverhorst@xpirit.com

In this issue of **XPRT.** Magazine, our experts share their knowledge about building an Engineering Culture.

INTRO

004 Together we build an Engineering Culture

XPIRIT

006 Bringing the Xpirit quality and services to our German customers

008 Launching Xpirit IoT: Smart & Connected Services

INNOVATION

011 What's what with WebAssembly?

016 Getting Your IoT Projects Off The Ground By Building On Azure

018 Azure container Apps: The future of Microservices in Azure?

INFRASTRUCTURE

024 Beacons create safe routes for Maersk's voyages through the cloud

026 Stop wrestling with ARM Templates, work on your Biceps

031 Shift left using Bicep

CULTURE

037 Never stop learning – Thoughts after four years with our epic team

039 Xpirit as an IT Beehive

041 The epic story of Blinky

DEVELOPMENT

046 The value of your development toolchain

050 Customizing Codespaces

055 Preparing for a security assessment

059 Embrace Chaos to Achieve stability

Colophon

XPRT. Magazine N°12/2022

Editorial Office
Xpirit Netherlands BV

This magazine was made by
Alex de Groot, Alex Thissen, André Geuze, Anne Meijer, Arjan van Bekkum, Bas van de Sande, Benny van der Poel, Casper Dijkstra, Chris van Sluijsveld, Davy Davidse, Diederik Tiemstra, Dennis Thie, Duncan Roosma, Erick Segaar, Erik Oppedijk, Erwin Staal, Loek Duys, Geert van der Cruijsen, Hans Bakker, Hindrik Bruinsma, Immanuel Kranendonk, Jasper Gilhuis, Jesse Houwing, Jesse Swart, Kees Verhaar, Maarten Blok, Marc Bruins, Maira Duijst - Camu, Manuel Riezebosch, Marcel de Vries, Mark Foppen, Max Verhorst, Martijn van der Sijde, Matthijs van der Veer, Michiel van Oudheusden, Natascha Former, Niels Nijveldt, Patrick de Kruijf, Reza Atlaschi, Patrick van Kleef, Reinier van Maanen, René van Osnabrugge, Reda Fakirmohamed, Rob Bos, Robert de Veen, Roy Cornelissen, Rutger Buiteman, Sander Aernouts, Sander Trijssenaar, Sofie Wisse, Gill Cleeren, Suraj Sewbalak, Thijs Limmen, Wesley Cabus, Tijmen van de Kamp, Pieter Gheysens, Kristof Van Hees, Annemie Vandenberghe, Jasper Van Mensel, Laurenz Ovaere, Wouter Van der Auwera, Kristof Riebels, Michael Kaufman, Thomas Tomow, Tobias Mackenroth, Olena Borzenko, Andreas Läubli

Contact
Xpirit Netherlands BV
Laapersveld 27
1213 VB Hilversum
The Netherlands
Call +3135 538 19 21
mverhorst@xpirit.com
www.xpirit.com

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

Translations
Mickey Gousset (GitHub)

© Xpirit, All Right Reserved
Xpirit recognizes knowledge exchange as prerequisite for innovation. When in need of support for sharing, please contact Xpirit. All Trademarks are property of their respective owners.

Gold
Microsoft Partner
Microsoft

GitHub Verified Partner

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

Together we build an Engineering Culture

At Xpirit, we believe every company is an IT company, no matter what product or service it provides. Today, no company can make, deliver, or market its product efficiently without IT technology. Whether it is banks, insurance companies, logistics companies, or retailers, IT and software are critical to their success. Many companies embrace this fact, and are insourcing software developers to be better, faster, and cheaper. They understand when they adopt new technology and implement it successfully, they gain a stronger foothold on the market. Companies that wait for a second or third wave stay at the back of the pack and will have a very hard time to become a leader in their market.

Authors Marcel de Vries and René van Osnabrugge

Engineering Culture

State of the Art Software Engineering

Smooth Delivery

Move the business needle

Empowering operating model

Power through platforms

Epic Work Environment

Appropriate continuity

Knowledge Driven

Since our start in 2014 this is the premise we have worked from. Helping companies to become an IT company. We have always done that by leveraging the potential and knowledge of our people, each one from within their own area of expertise. And together, we drive change at customers by introducing concepts that are part of an Engineering Culture.

And this change is pursued in multiple areas, because we believe you cannot be successful if you focus on 1 area alone. It is not enough to write good software, build a great cloud foundation, have a nice workplace, or automate everything. It is everything, together, that will make you great. Not only in the development teams but also at the leadership level.

We call this combined set of capabilities and behaviors a company should have "an Engineering Culture". To be an IT company, you need to act and behave like an IT company. And everything we do as Xpirit adds to this vision.
This Engineering Culture can be seen from many different perspectives that we categorize in a number of distinct pillars that together will help you become successful and can be used to drive change.

One of the pillars is "**State of the art software engineering**". In this magazine, we talk about the latest innovation in software development and the Azure cloud platform and talk about the use of IoT, Azure Container Apps, and Web Assembly.

We also cover the "ops" side of development. Although Infrastructure as Code is already becoming the new normal, there is still a lot to be discovered and learned about it. In this edition, we will cover Bicep and ARM and how these can help you to speed up your delivery of infrastructure in the cloud.

Another pillar is "**Smooth Delivery**". This has been our bread and butter since we started Xpirit. This is crucial to be successful as an IT company and therefore part of an Engineering Culture. We talk about the development toolchain and supply chain, that has become the heartbeat of product delivery, and an attack vector if you look at security. We also explore how you can lower the barrier of entry in a product development team with the use of Codespaces, that is introduced by GitHub as a means to improve the developer experience. As an early adopter we worked with this technology for quite a while and we can share what we have learned so you can speed your adoption of this new feature.

With the increase of automation and the fact cyber criminals are adopting the cloud and DevOps practices faster than the average enterprise, we see an increase in the number of threads we need to deal with as an industry. Business Continuity, Reliability, and Security is essential. We cover these topics in our Engineering Culture pillar "**Appropriate continuity**". We want to be "secure and compliant by default", while increasing your speed of delivery and the stability of the products you deliver. In this magazine you'll find an article that can help you prepare for a security assessment and we introduce you to the concept of Chaos Engineering to validate all the hypotheses you make during the development of your applications and infrastructure.

The fourth pillar we'll touch upon in this magazine is "**Epic Work Environment**". Does your culture match your ambition to become an IT company? DevOps is all about People, Process and Tools, so we also cover a lot of the cultural aspects that come with becoming an IT company. A new approach to knowledge sharing and a learning mindset. Because every profession changes over time. Including ours.

This magazine is a reflection of the wide spread of knowledge that is present at Xpirit. We love to share our knowledge throughout this magazine and this is also part of our own internal Engineering Culture under a fifth pillar "**Knowledge Driven**". You might already have seen that we are expanding in Europe. We introduce Xpirit Germany that will help drive change at companies in Germany, and we broadened our capabilities into the IoT business.

Last but not least, you can read about how platforms can help you to accelerate your business. You can read about how the implementation of a self-service cloud portal helped Maersk to effectively respond to the log4j vulnerability, and we talk a bit about how you can use the Azure IoT platform and how to migrate your AKS workloads to Azure Container Apps. These articles are part of the sixth pillar "**Power Through Platforms**".

We hope you enjoy this magazine and would like to challenge you to take a step back and look at your own company. Are you an IT company? And what do you need to build your own Engineering Culture? </>

Marcel de Vries
Chief Technical Officer

xpirit.com/marcel

René van Osnabrugge
ALM, DevOps, Continuous Delivery, Initiator and Inspirator

xpirit.com/rene

Bringing the Xpirit quality and services to our German customers

In Germany, we'll focus on the core Xpirit offerings: DevOps with GitHub, Cloud-Native Development, and Cloud Transformation. But we also plan to expand our portfolio and establish new practices important to our customers in Germany, like Microsoft 365, Managed Services, and Low-Code development.

Authors Michael Kaufmann, Thomas Tomow and Tobias Mackenroth

With our main office in Frankfurt, we offer a 100% hybrid working environment. To stay connected, we meet once a month in the office.

Our projects, customers, and members are located across Germany. Our members can choose to work in their home office or in a shared office space we rent for them close to their home.

Our management team has many years of experience in Microsoft consulting in Germany - helping their clients succeed in adopting the cloud, developing enterprise scale applications, and implementing DevOps practices.

Michael is responsible for the vision and execution in Germany. He is a Microsoft Regional Director and Microsoft MVP, a book author, and regular speaker at international conferences. He has a strong focus on the GitHub partnership, our Managed DevStack offering, and GitHub and DevOps Training Services. Thomas is responsible for the technical strategy and will also take care of operational concerns. He is a Microsoft MVP for Azure focusing on IoT, cloud-native development and artificial intelligence. He also provides training and consulting around GitHub, DevOps, and Azure. Tobias takes care of clients, contracts, and marketing. He's responsible for a healthy, intimate, and long-lasting relationship with our clients and helps them by ensuring the delivery of high-quality services from the entire Xebia Group.

We have known each other for many years, and we trust each other completely. Doing “epic shit” has always been one of our key drivers and reflects our DNA, together with the principles "people first", "sharing knowledge", "quality without compromise", and "customer intimacy".

Although Xpirit Germany is in early days, our aspiration is to grow our team to over 12 by the middle of 2022.

Why we are the perfect fit

The Netherlands embraced the cloud earlier than we did in Germany. But now, with the cloud transformation at full speed, we believe we can bring many of the learnings from the existing Xebia customer base to Germany, and help our customers succeed in their journey.

With more than two decades of consulting experience, we know that a partnership with clients consists of trust and value. Therefore, we will help, support, and consult with focus on pure, honest, and reliable consulting. Clients can expect quality without compromise from us. </>

Michael Kaufmann
CEO

xpirit.com/mkaufmann

Thomas Tomow
CTO

xpirit.com/thomas-tomow

Tobias Mackenroth
CCO

xpirit.com/tobias-mackenroth

Launching Xpirit IoT: Smart & Connected Services

The impact of Internet of Things (IoT) is made clear by its moniker as the third wave of computing – right after computers and smart phones. All three waves are still having massive impact on all aspects of society. Embedding sensors and compute power in consumer products and industrial equipment has unlocked a new world of insights, optimizations, and autonomous behavior. We expect IoT to continue to transform many industries in the coming years, and we see interesting challenges ahead that fit our core expertise: continuous integration extending to the embedded domain, and an overall need for higher code quality delivered in ever-shortening cycles.

Author Tijmen van de Kamp

In this new and exciting domain, we can utilize our Cloud and DevOps expertise, knowing that Microsoft is adding IoT-specific features and services to the Azure cloud on a regular basis. End-to-end security and robustness are must-have ingredients in the IoT domain, and we are ready to deliver.

IoT is here to stay

Regardless of which analysis report on IoT you open, the trend is clear: IoT is on a strong growth trajectory, with no signs of slowing down anytime soon. Even our personal lives are touched by IoT: from smart lighting to intelligent thermostats, more and more consumer products are becoming "smart", meaning they can share data to ultimately deliver new services and new experiences. And as we are becoming accustomed to having real-time insights in key data such as our home's energy footprint, it only makes sense that those same types of immediate insights drive major digital transformations in the corporate world.

At its heart, IoT is about connecting "things", an intentionally broad term that encompasses an entire world of physical devices: machines, vehicles and buildings often come to mind, but "things" also include human- and animal-wearable devices, sensor networks deployed across farms or nature reserves, and many, many more. Connecting all these devices together gives unparalleled insights in what is happening in the real world, in real-time. Those insights alone can help transform businesses, since understanding and the ability to measure is at the heart of making improvements. Taking actions based on the insights generated by connecting things is the next step in creating an IoT-powered value chain, regardless of whether those actions are performed by those same things or somewhere else entirely.

Predictive Maintenance is but one of the strong recipes for success IoT can bring to the table: by analyzing equipment sensor data & behavior, looming failures can be detected and corrected before service is interrupted. This not only applies to major industrial equipment but can be just as valuable for keeping your domestic heating equipment up and running: instead of sending a service engineer every year, IoT makes it possible to only send someone if and when your boiler is in need of maintenance. This drives down cost, improves quality of life, and prevents unnecessary travel, all in a single stroke.

IoT touches many industries

IoT is sometimes labeled as the "real-time revolution": a major change driven by data that gives insight in the here and now. This availability of immediate data has impact on almost every industry. Precision farming uses sensors to monitor crops, weather, and livestock, allowing farmers to respond in the right way, at the right time. Valuable assets in construction can be tracked and secured but can also be used to their fullest potential by connecting real-time insights to planning and resource availability. Inventory tracking in warehouses and stockpiles improves efficiency and effectiveness of supply chains, and monitoring spaces in office buildings and stores can dramatically improve safety whilst reducing the CO_2 footprint.

Although there is a huge number of potential use cases in each industry,

there are a set of common denominators that can help us to understand where opportunities lie. Most use cases fall in one of three main categories: core optimizations, improving performance and experience, and transforming businesses. As an example, reducing inventory and optimizing logistics can help reduce cost, whereas real-time insights can dramatically improve customer experience. IoT can also be a driver to develop new business models in existing industries, and help companies shift from building products to delivering digital services.

In addition to classifying IoT initiatives in terms of their impact to the business, we identify six domains across the multitude of industries, use cases and generic applications like predictive maintenance and remote inspections, to further reduce complexity of the IoT landscape. These domains help us create focus in building our expertise and our portfolio of services, and we have found them to be helpful in assessing customer realities as well:

- **Connected assets:** owned stationary equipment
- **Connected vehicles:** tracking location and many other aspects of vehicles on the road
- **Connected products:** (durable) consumer goods
- **Connected spaces:** includes smart buildings, but also any other facility where automation revolves around optimizing occupancy and space utilization
- **Connected inventory:** non-connected goods tracked by external control mechanisms
- **Connected people:** wearables and other tools to augment workers and individuals

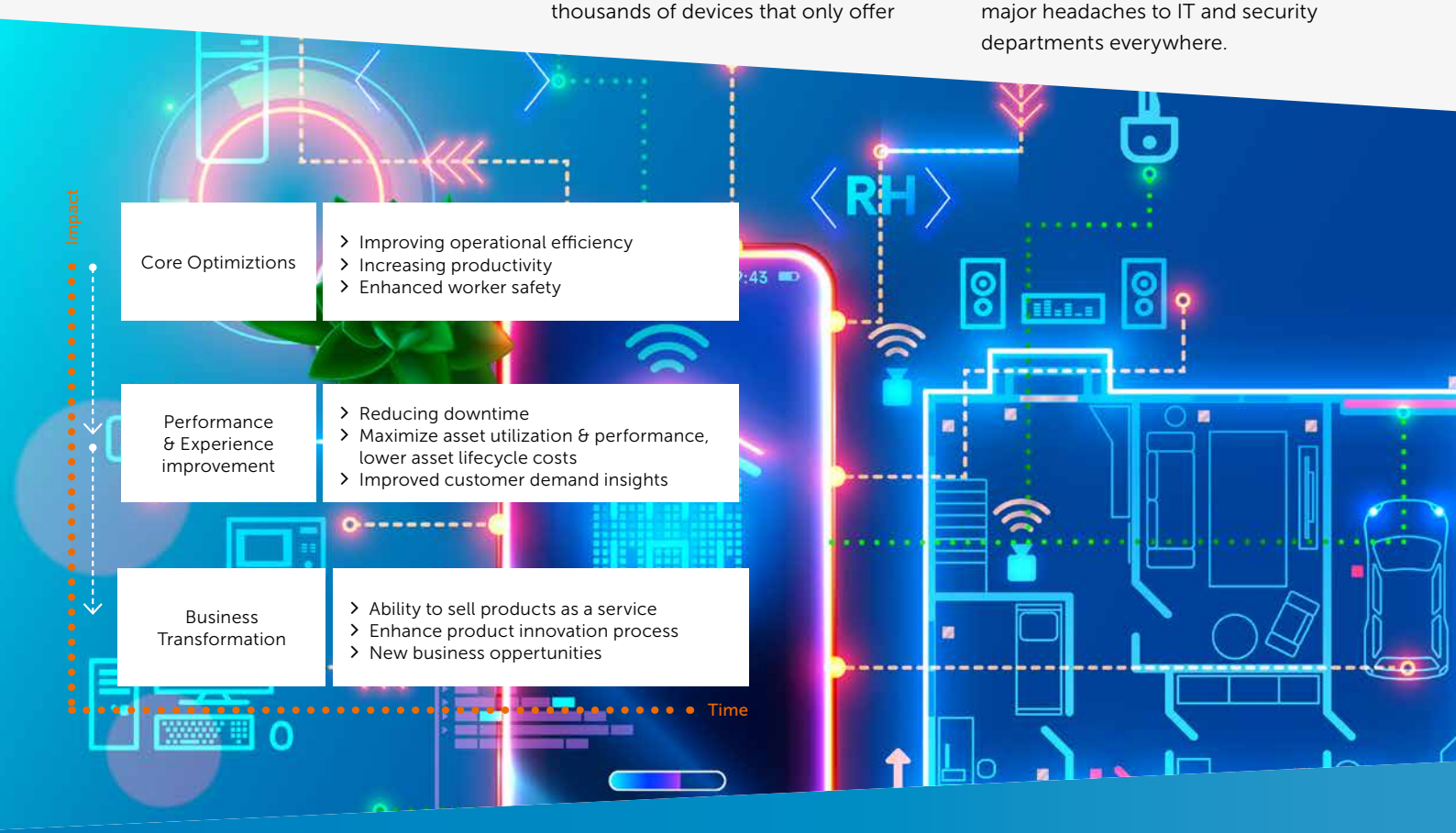
IoT needs a team to drive it

Doing IoT "right" takes more than connecting a few devices and calling it a day. IoT adds complex aspects to the already complicated world of enterprise architecture and IT management: it comes with a whole new class of embedded IT devices that need to be managed in terms of connectivity and updates, but that also have their specific challenges including power management, battery status and physical security and maintenance.

When deploying data platforms or applications, there are usually easy fallback scenarios if something goes wrong. In IoT, there is the added challenge of not only deploying to thousands of devices that only offer very limited ability for interaction, but also to devices that are often hard or impossible to reach if something goes wrong. Every IoT developer has experienced some version of sending out an update to a device, only to be rewarded with the deafening silence of a no-longer-responding piece of equipment.

In addition to the complexity brought on by the volume of data that an IoT solution can generate, many developers have only had limited experience in working with time-based data streams that typically come with IoT scenarios. Although the Microsoft Azure cloud platform has several easy to leverage components tailored to this, managing real-time data flows is just another aspect that drives up the complexity of an IoT project.

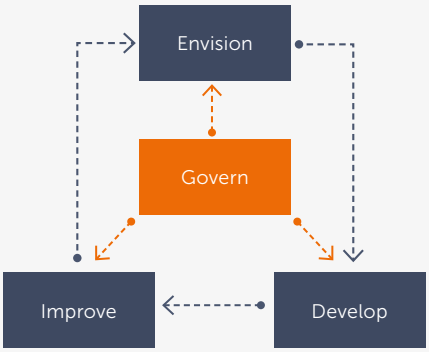
An overview of key IoT challenges would not be complete without mentioning the elephant in the room: security. Every IoT device deployed increases the attack surface of an organization, potentially with poorly secured devices that rarely get updated. Stories of casinos being hacked through their fish tank automation system make for good headlines, but also bring major headaches to IT and security departments everywhere.



The news coverage of IoT in the market comes with additional side effects: not only will the expectations for success of an IoT journey in a typical organization be very high, but you can almost guarantee there will be a commercially available platform or solution out there that can cover at least some of what that organization is trying to achieve. The "build versus buy" decisions in IoT can become very complex, very quickly, especially in domains where the end state is a landscape of built and bought components that are expected to work together seamlessly. A typical example is a "smart building": there will be different systems from different vendors for lighting, climate, and access control, but the end user expects a smart space that just works. In practice, we often implement point solutions for various subdomains and then add a "platform of platforms": a control layer to make a system of smart components act as one entity.

Ingredients for a successful IoT journey

To tackle IoT successfully, it is imperative to embed four major disciplines in your to-be organization: envisioning, developing, continuous improvement, and governance. Envisioning is all about establishing and guarding the vision for the IoT journey, and keeping the higher-level goals top of mind: what is the purpose and expected outcome of an IoT initiative, and how do ongoing development and new insights influence the overall direction and priorities?



Building the ability to run, maintain and further improve IoT solutions into the supporting organization is often postponed, but needs to be there before the first launch of a product, even if it is only in pilot or MVP stage.

Solutions that are built with a "you build it/you run it" DevOps mindset easily morph from pilot to production stages, whereas quick-and-dirty prototypes end up costing much more in the long run. Especially in IoT, observability and instrumentation are paramount, since it is much harder to see what is going on in the field.

Developing IoT solutions right requires a proper blueprint and a consistent architecture for both cloud and devices, where all aspects of software craftsmanship and cloud-native software development come into play. Quality and standardization must be enforced across integrations, cloud platform, and embedded devices, and these environments should be treated as a continuum, rather than as separate universes.

Governance ties the other three disciplines together in terms of architecture, security and compliance, cost control, standardization, and interoperability. These aspects apply to all elements on an IoT initiative, ranging from connectivity and gateways all the way up to data management and advanced visualizations.

Your IoT journey, powered by Xpirit

We help clients that have not started on IoT yet, as well as clients that are already (well) underway on their IoT journey. Whether the primary driver to start on IoT is to innovate or to catch up with competition, our North Star Vision and IoT roadmap propositions help chart viability, establish priority, and build an IoT master plan. Often, the next step is to do several quick prototyping sprints where we dive deep, to quickly address challenges and uncertainties.

We introduce aspects of our IoT Reference Architecture to build things first-time right where appropriate.

In parallel, we use our IoT Center of Excellence blueprint to help you build an organization that can nurture and grow your IoT initiatives. With those building blocks in place, we help you move from prototype via pilot to the launch of the Minimum Viable Product, and beyond.

With customers who are already underway with IoT we usually start with an IoT assessment, which yields a broad and objective view of the status quo. Together with selected partners, we look at software, hardware, data, security, the validity of the underlying business case, and many more aspects. Depending on the outcome, we bring in the right propositions to help tackle any challenges we may have uncovered. We aim to move all four disciplines forward over time but focus where attention is most needed.

What sets us apart is our ability to deliver according to your needs: our "us/together/you" approach means we want to empower you and your teams. Xpirit can quickly boost existing IoT teams with the required expertise, provide ongoing support and mentoring to teams over a longer period, or deliver an IoT journey end-to-end for our customers. We are continuously evolving our portfolio of services and our IoT accelerators, including an IoT reference architecture and a blueprint for an IoT center of excellence. We augment our own expertise as needed with help from our trusted partners, especially where it comes to hardware, embedded software, and connectivity.

Interested in learning more?

Have a cool project to talk about? Want to work with amazing people on IoT projects? Reach out to IoT CTO Tijmen van de Kamp. </>

Tijmen van de Kamp
Xpirit IoT: smart & connected services

xpirit.com/tijmen
tvandekamp@xpirit.com

What's what with WebAssembly?

You have probably heard of Blazor and that it uses WebAssembly to run .NET code inside a browser. But did you know you can use WebAssembly for much more? In this article, we will show you a few cool things to do with WebAssembly.

Authors Chris van Sluijsveld and Loek Duys

What is WebAssembly?

WebAssembly was invented as a language to run binary code inside a web browser. Applications running in WebAssembly run isolated, just like Docker containers. The use of virtualization allows a WebAssembly program to be portable across operating systems and different processor without modification. It runs on Windows, Mac, Linux, and devices like the Raspberry Pi equally well. This is a big difference from containers, which are created for specific operating systems and processor types.

WebAssembly, or 'Wasm,' was invented by Mozilla and is now pushed forward by the ByteCode Alliance, a group of companies including Microsoft, Intel, and Google. Currently, Chrome, Edge, Safari, and Firefox support running WebAssembly. Because it's a compact binary format, it runs with little overhead at near-native speed. When compared with JavaScript, WebAssembly applications usually run much faster. Many popular programming languages can be compiled into WebAssembly and run on the web. Supported languages include Rust, Python, Go and C. You can even run the .NET Mono CLR in WebAssembly and use it to run regular .NET DLLs. Blazor currently works like that. Microsoft is also experimenting with a .NET runtime that compiles C# to Wasm.

Mozilla designed WebAssembly to co-exist with JavaScript and work together to deliver a good web experience. A file containing WebAssembly code is called a Module. An instance of a Module runs inside a sandboxed execution environment. Sandboxing ensures WebAssembly cannot access sensitive data like files and network resources without explicit consent from the hosting environment, which is usually your web browser. WebAssembly does that by importing and exporting functions from and to its host.

Beyond the browser

WebAssembly does not make any assumptions about the host environment it runs on. This means that WebAssembly can also run outside of a browser. The 'WebAssembly System Interface' or 'WASI' enables this. WASI is an API that provides WebAssembly with operating-system functionality like access to the file system and communication over networks.
This works using the function imports mentioned earlier. Most modern browsers implement the WASI interface, and a few stand-alone implementations are also available. Having a stand-alone WASI runtime means that a browser is no longer required to run WebAssembly code. As with any good OSS technology, there are quite a few WASI runtimes to choose from: wasmtime, WAMR, Wasmer, WasmEdgeRuntime, Wasm3 and others. For this article, we selected a host called wasmtime.

Developing with WebAssembly

By now, you're probably anxious to get started running some WebAssembly code. We will start by building and running a simple 'hello world' WebAssembly program inside a browser, using the Rust programming language on Linux. Please note that if you are running Windows, you can use WSL2 to follow the steps in this article.

Running WebAssembly in your browser

We will create some Rust code and compile it into WebAssembly. If you don't have the Rust Tools installed yet, run this command to install them:

```
curl --proto '=https' --tlsv1.2 -sSf
https://sh.rustup.rs | sh
```

We want to create an interactive program that displays a message box to the end user when executed. To do this inside a browser, we will import JavaScript functions.

To do this, we need a way to interact from WebAssembly to JavaScript. We can use wasm-pack to do this. Run this command to install wasm-pack:

```
curl https://rustwasm.github.io/wasm-pack/installer/
init.sh -sSf | sh
```

Now, let's create a simple Hello-World library in Rust by running:

```
cargo new --lib hello-wasm
```

Change the directory to the newly created program folder:

```
cd hello-wasm/
```

The 'hello-wasm' folder contains a file named Cargo.toml that describes how to build the project and a folder 'src' with file named lib.rs which we will change now. Modify the file to create a program that outputs 'Hello world!' when run.
Use your favorite text editor to change the code as displayed in Figure 1:

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
extern "C" {
    pub fn alert(s: &str);
}

#[wasm_bindgen]
pub fn greet(name: &str) {
    alert(&format!("Hello, {}!", name));
}
```

Figure 1. lib.rs

The first line imports an existing library named 'prelude', so we can use the features in the code following the 'use' statement. The *wasm-pack* tool uses the attribute '#[wasm_bindgen]' which enables Rust to invoke JavaScript. The function 'alert' exists in JavaScript and is exposed to Rust by using the 'extern' keyword. The other way around works as well; the final section of the code exposes a Rust method to JavaScript by using the 'pub' keyword on the function named 'greet'.
After compilation, we will have a WebAssembly program that exposes a function 'greet. We will use JavaScript on a web page to invoke that method. When executed, it will call the JavaScript 'alert' function to create a message box.

Finally, change the *Cargo.toml* file to include the dependency to the wasm-pack tool with the content from Figure 2.

```
[package]
name = "hello-wasm"
version = "0.1.0"
edition = "2021"
[lib]
crate-type = ["cdylib"]
[dependencies]
wasm-bindgen = "0.2"
```

Figure 2. Cargo.toml

We are now ready to run wasm-pack to compile the project using this command:

```
wasm-pack build --target web
```

After about half a minute, you should see output similar to this:

```
[INFO]: :-) Done in 27.08s
[INFO]: :-) Your wasm pkg is ready to publish at /home/
user/xpirit/magazine/rust/hello-wasm/pkg.
```

Let's run the JavaScript file inside an HTML file to show it works. Create a file named 'index.html' inside the pkg folder, with the content of Figure 3:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>hello-wasm example</title>
</head>
<body>
    <script type="module">
        import init, { greet } from "hello_wasm.js";
        init()
            .then(() => {
                greet("World")
            });
    </script>
</body>
</html>
```

Figure 3. index.html

As you can see, the web page uses JavaScript to call the 'greet' method in WebAssembly. To see how that works, inspect the file 'hello_wasm.js'.

You will need a webserver to run this page and its JavaScript. The script won't run if you open the local file directly from disk. If you are using VS Code, you can use the 'Live Server' extension by Ritwick Dey. When it runs, you should see something similar to the screen from Figure 4.

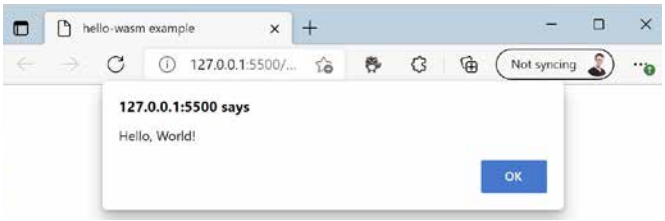


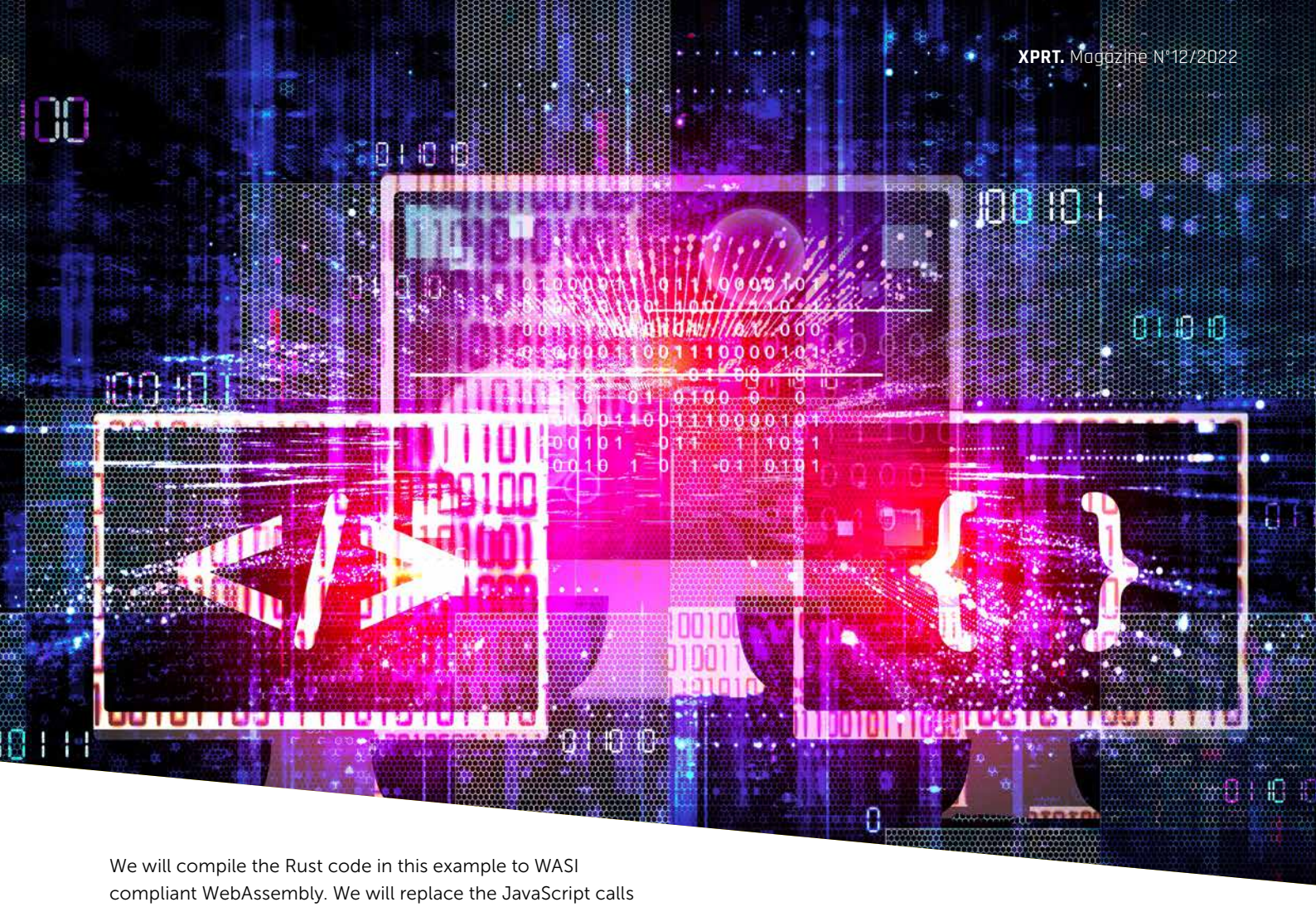
Figure 4. Hello World in browser

Congratulations! You have just built your first WebAssembly module, hosted inside the browser!

Running WebAssembly without a browser

WebAssembly can also be run locally on your machine, using a stand-alone runtime. This is because WebAssembly relies on the WebAssembly System Interface (WASI) to talk to the underlying operating system. The WebAssembly System Interface does not depends on browsers and does not have a requirement for JavaScript to run.

So for the next sample we will not use Javascript anymore but create a console application that will run directly on your operating system using WASI.



We will compile the Rust code in this example to WASI compliant WebAssembly. We will replace the JavaScript calls with console input and output, using the *stdin* and *stdout* streams. Earlier, we installed Rust, this doesn't install the tools needed to compile to WASI compliant WebAssembly. For that, we will need a new library named 'wasm32-wasi'. Install it using this command:

```
rustup target add wasm32-wasi
```

Now that you have installed the correct build target, we will create a new project. You can do that by running the following command:

```
cargo new hellowasi
```

Change the directory to the newly created program folder:

```
cd hellowasi/
```

Next, compile the program to WASI compliant WebAssembly, by running:

```
cargo build --release --target wasm32-wasi
```

To run the compiled WebAssembly, we will need a stand-alone host. We will use *wasmtime* for that. Install wasmtime using:

```
curl https://wasmtime.dev/install.sh -sSf | bash
```

Run the h*ellowasi* program with:

```
wasmtime ./target/wasm32-wasi/release/hellowasi.wasm
```

The output should say *"Hello, World!"*.

Congratulations! You have just built and run your first native WebAssembly program.

Running WebAssembly with WAGI

We now have a nice console program writing its output to the stdout stream. But what if you want to develop APIs in WebAssembly and host them on any platform? This is where the current experimental WebAssembly Gateway Interface (WAGI) comes in. To quote the WAGI website, "**WAGI allows you to run WebAssembly WASI binaries as HTTP handlers.**"

"Did you know Deis Labs used to be Deis which are the founders of Helm for Kubernetes. It became Deis Labs after the acquisition by Microsoft in April 2017. Deis Labs currently runs many experimental programs revolving around WebAssembly and Kubernetes."

> "If WASM+WASI existed in 2008, we wouldn't have needed to created Docker. That's how important it is. Webassembly on the server is the future of computing. A standardized system interface was the missing link. Let's hope WASI is up to the task!"
>
> – Solomon Hyke, Co-founder of Docker

Similar to how wasm-pack acts as a bridge between JavaScript and WebAssembly, WAGI connects HTTP requests and responses to your program's stdin and stdout streams.

To get started with WAGI, download the latest release from https://github.com/deislabs/wagi/releases and unpack it. Next, we will need to add a configuration file to tell WAGI where the WebAssembly program we want to run is located for each requested URL. In our example, we will use the root path at "/".

Create a wagi.toml file containing the content from Figure 5.

```
[[module]]
route = "/"
module = "target/wasm32-wasi/release/hellowasi.wasm"
```

Figure 5. wagi.toml

Because WAGI acts as an HTTP server, we need to make sure to write required content-type headers to the output streams in our WebAssembly program. Make sure your main.rs file looks like Figure 6.

```
fn main() {
    println!("Content-Type: text/plain\n");
    println!("Hello, world!");
}
```

Figure 6. main.rs

Recompile the program using:

```
cargo build --release --target wasm32-wasi
```

After that, you should be able to WAGI to run your WebAssembly program with the following command:

```
wagi -c wagi.toml
```

When WAGI is running, you can visit http://localhost:3000 to see the output. The result should look similar to Figure 7.

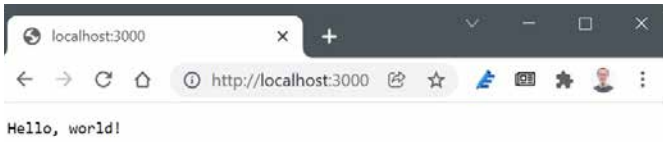


Figure 7: Hello World WAGI API in browser

Congratulations! You have just built and run your first WebAssembly API server using WAGI.

Moving to the Cloud

Running code in a light-weight isolated sandbox may seem familiar to you. The way that code is executed in WebAssembly bears a lot of resemblance with the way we run code inside containers. So, what if we want to run our WebAssembly API at scale in the Cloud? We can do this using Azure Kubernetes Service and an experimental service called Krustlet, developed by Deis Labs. Krustlet acts as a Kubernetes node that allows you to run WebAssembly programs on Kubernetes. Similar to how Kubernetes nodes run containers based on images stored in an OCI registry, Krustlet runs WebAssembly programs based on OCI artifacts (in our case the wasm file).

OCI artifacts support many different formats. Aside from container images, you can also store your WebAssembly program as OCI artifacts in a suitable registry (for example, Azure Container Registry). To do this, you will need another program called wasm-to-oci.

Download the latest release from https://github.com/engineerd/wasm-to-oci/releases and then install it using the following commands:

```
mv linux-amd64-wasm-to-oci wasm-to-oci
chmod +x wasm-to-oci
sudo cp wasm-to-oci /usr/local/bin
```

Make sure to have Azure Container Registry (ACR) available t o test things on and enable the 'anonymous pull' feature to allow WAGI to access it without logging in.

Login to ACR and push your WebAssembly program to ACR with:

```
az acr login --name acrwasmwasi
az acr update --name acrwasmwasi --anonymous-pull-enabled
true
wasm-to-oci push ./target/wasm32-wasi/release/
hellowasi.wasm acrwasmwasi.azurecr.io/hellowasi:v1
```

Note: make sure to replace acrwasmwasi with the name of your container registry.

Out of the box, WAGI, which we used earlier in the previous paragraph, supports OCI artifacts as a source for WebAssembly code.

You can try this by updating the wagi.toml file created earlier to look like Figure 8.

```
[[module]]
route = "/"
module = "oci://acrwasmwasi.azurecr.io/hellowasi:v1"
```

Figure 8. wagi.toml

The run WAGI again using this command:

```
wagi -c wagi.toml
```

If this is successful, the output in the browser on http://localhost:3000 should look the same as it did in Figure 7.

Running WebAssembly on AKS

Because Deis Labs is part of Microsoft, preview support for Krustlet nodes is already available in Azure. If you want to learn more and try it for yourself, follow the walkthrough on https://docs.microsoft.com/en-us/azure/aks/use-wasi-node-pools

Conclusion

You have seen a few interesting applications of WebAssembly. It could be considered a highly secure way of running code in an isolated sandbox locally, in the browser, in Kubernetes etc.

But there are some things you need to be aware of...

WebAssembly code in a binary format is hard to read. It's difficult to know what code in a wasm file does from the outside, especially when it is also obfuscated. Combine this with high execution speed and browser support, and it becomes easy to see how WebAssembly can be used for naughty things like running crypto-miners inside an unwitting user's web browser. At the time of writing, there are no security scanning platforms to scan images for malware and vulnerabilities.

Most, or all, of the current hosts are vulnerable to attacks like SPECTRE, so at this time the examples shown here should not be used in a production environment.

In the future, if these security risks are mitigated, WebAssembly could become the next big thing after containers. </>

Relevant links

- Describes the current state and future plans for WebAssembly https://hacks.mozilla.org/2018/10/webassemblys-post-mvp-future/
- Website of the ByteCode Alliance https://bytecodealliance.org/
- Wasmtime for .NET https://github.com/bytecodealliance/wasmtime-dotnet
- Website of Deis Labs part of Microsoft https://deislabs.io/
- Microsoft experimental runtime that compiles C# to wasm. https://github.com/dotnet/runtimelab/tree/feature/NativeAOT-LLVM

Chris van Sluijsveld
Digital disruptions using Microsoft Cloud Technology

xpirit.com/chris

Loek Duys
Cloud software architecture

xpirit.com/loek

Getting Your IoT Projects Off The Ground By Building On Azure

With the popularity of the Internet of Things, new proof of concepts and prototypes are starting everywhere. If you're contemplating getting started with IoT or need a nudge in the right direction, this article will highlight some great options to get you started. Now, some projects go nowhere, with others end up being very successful. But even in the latter case, a new IoT platform will still fail if the wrong choices were made in the technology selection, right at the project's inception.

Author Matthijs van der Veer

An IoT solution will have a couple of key components, even for a prototype. There needs to be a robust messaging system in place. This allows you to have bidirectional communication with your devices. You also need to store a collection of devices you can trust and their configuration to run correctly. These devices live at "the edge", a collective term for anywhere from a factory, train tracks, or someone's home. These devices could range from a tiny microcontroller to more powerful computers running artificial intelligence workloads. Both will send messages to the platform, and when these messages are received, they are transformed, analysed, and visualised to extract insights from the data. The information is then stored, preferably in different ways for long-term storage vs storage that needs to be readily available. With so many moving parts, choosing the right technology becomes critical because it will impact your project's future.

One example of a project I've seen came to a grinding halt through the weight of its own complexity. A small IoT prototype was a success and became part of the company's core business. But it simply wouldn't scale any further than a couple of devices. The technologies used to develop this project seemed "fine" at the time. Surely you can come back to fix this, right? But a couple of years later, they're running a custom message broker and a handful of databases and spreadsheets to tie everything together. IT doesn't want to host their platform, and it's now running in your "private cloud" in the attic.

The software described above is not an exaggeration, and I'm sure there are many more platforms out there like it. And who can blame the authors of these projects? They might have been trying something new, using the skills they had at that moment. When the

prototype became a success, it was put into production instead of turning it into a scalable solution first. So how can you avoid making the same mistake?

Build for success with Azure

Instead of building and designing everything from scratch, you can get a head start by using Azure platform as a service (PaaS) components. These are the same components used for global-scale IoT platforms, managing millions of devices. While at first, this might sound like an excessive measure for a prototype with just a few devices, the PaaS components in Azure scale remarkably well. The best place to start for most platforms is Azure IoT Hub. You can get started with a fully-featured IoT Hub for about 20 Euros per month, and with 400.000 IoT messages per day, it will be a long time before you have to scale it up. So even for a proof of concept, you can spin up your own IoT Hub and save yourself the trouble of hosting custom message broker, identity management, and message routing solutions.

When using IoT Hub, you have many options to transform and analyse the data you're receiving. A typical scenario with new projects is starting with Azure Functions to transform and bring data from one place to another. Moving to Azure Stream Analytics can be a great choice when the requirements become more complex or need to consider time windows. It allows you to run analytics over data streams and extract the most critical insights. It also has built-in anomaly detection, a complex feature to build from scratch.

Another great place to start is Azure IoT Central. This software as a service (SaaS) product builds on top of IoT Hub and other Azure components to offer a highly scalable product. You can be the proud owner of an IoT Central instance in minutes for a few cents per month, so pricing shouldn't be a limiting factor. It has dashboarding, device registration, a ruling engine, and even some new multi-tenancy features built-in. This means you can start to impress your organisation with a complete IoT platform without reinventing the wheel. And if there are features you need that aren't in Azure IoT Central, you can stream the device data to your own software. Your IoT prototype became so successful that the organisation wants to include the data into their CRM platform? No problem, stream the data to Service Bus or Event Hub for further processing or send it directly to an HTTP endpoint of your choice.

In both cases, you get a huge jump-start in functionality and can get started with something much more important: building the features only you can create. You know your business better than anyone else, so build on these world-class components and focus on what you do best. Building an IoT platform shouldn't be about making all the plumbing, time and again. It's about realising value.

Cloud logic, at the edge

Following the advice above, you will have a great start in the cloud, but IoT also involves devices. Your project could use off-the-shelf hardware, but you might need a device that doesn't exist yet. Creating IoT devices is usually done by professional device builders. Combining electronics and writing code for microcontrollers is not a skill every developer has. But that doesn't mean you can't build simple prototypes. You can get started by building devices with Arduino or .NET nanoFramework. The latter gives you a subset of the .NET CLR to write software for micro-controllers in C#. Getting started with nanoFramework is blazingly fast, and the different applications you can write with it deserve their own article. The most important thing is both Arduino and nanoFramework have many libraries available to do the heavy lifting, so even on the edge, you're able to get started quickly.

But you might need more robust hardware. If you're running AI at the edge or need to go beyond the constraints of a microcontroller, Azure IoT Edge will accelerate your device solution. It allows you to write device software in a language that you probably already use in your day job. If you know .NET, Python, Node.js, Java or C, and have some experience creating Docker containers, you have what it takes to be an Azure IoT Edge developer. Another benefit of Azure IoT Edge is you can use CI/CD to deploy updates to your device, so the development process should be familiar.

Microsoft also supplies standard IoT Edge modules for Azure Functions, SQL databases, Stream Analytics and more. Hence, like in the cloud, build on these existing modules to avoid reinventing the wheel.

Conclusion

Getting a new IoT project off the ground can be tricky. Starting with small prototypes and proof of concepts is an excellent way of testing the waters. Chances are, you already have what it takes to get started on the edge. There have never been more options for software developers to get involved without much embedded development experience, be it microcontrollers or edge computing devices. And when you start by building on the same secure, reliable and high-performing cloud components that support millions of devices worldwide, you can focus on what makes your project unique. And when the time comes to scale up your platform, you won't ever have to run your platform from your attic. </>

Matthijs van der Veer
Azure IoT Specialist

xpirit.com/matthijs

Azure container Apps: The future of Microservices in Azure?

Looking at the current state of software development, we can conclude a few things:
1. Containers are here to stay. Over the years, containerized workloads have become more and more popular, and we see most mature software companies benefit using containers from the cloud to the edge.
2. The DevOps movement is still growing and growing; the mantra "You build it, you run it" really works for building better software. DevOps teams must take into account the whole picture of building applications, from features to costs, from application monitoring and underlying infrastructure instead of only being responsible for building features for their applications.

Authors Geert van der Cruijsen and Bas van de Sande

Combining these two trends in the market explains why technologies such as Serverless became popular. Development teams must focus on everything related to building functional, resilient, and robust applications while taking costs into account. Serverless helps in reducing the amount of moving parts you must manage as a development team.

Kubernetes is another technology that took the world by storm over the past several years. Containerized workloads are popular, and Kubernetes gives you a vast number of options to deploy and run these workloads, either in the cloud or the edge, with flexibility between all clouds and self-hosted options.

Kubernetes also offers great tools for autoscaling, recovery of failing containers, zero downtime deployments, and controlling the network within the applications with service meshes. Because of that, all cloud providers have invested heavily to create ways to run Kubernetes on their clouds. That is why Kubernetes is becoming the standard infrastructure for modern cloud native applications.

There are also some downsides to Kubernetes. Managing Kubernetes itself is quite complex and although the public cloud providers are all investing in making running applications easier and easier, Kubernetes itself is still far from a PaaS or serverless service that needs little to no configuration for production workloads. In a world where we want to have T-shaped development teams that can build and run their applications, also having those teams know everything about Kubernetes can be quite a burden.

Microsoft acknowledged this and realized most companies do not need all the features Kubernetes has to offer. Their aim when building Azure Container Apps was to create an opinionated way of deploying containerized workloads to Azure that brings several features that Kubernetes could

provide without having to manage a cluster: autoscaling, zero downtime deployments and traffic shaping with control over ingress.

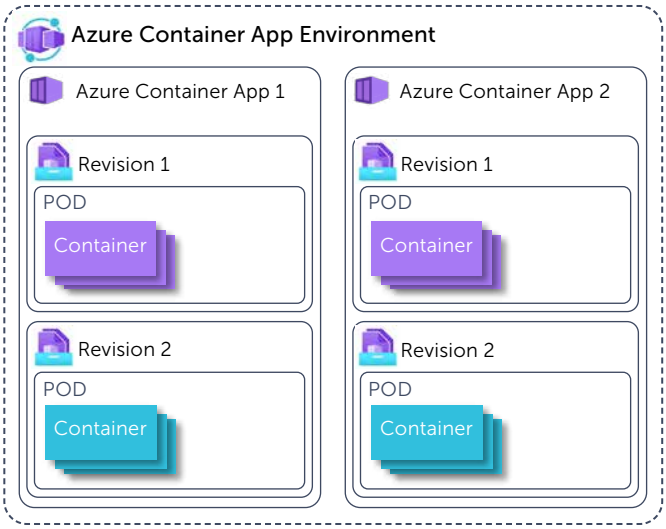
Introducing Azure Container Apps (ACA)

As mentioned, Microsoft aimed for creating an optimized way of deploying and running containerized workloads when building Azure Container Apps. Their focus was to build a solution that makes it easier for development teams to build Microservice architecture-based applications and deploy those to Azure. The idea being, giving development teams the features they really want from Kubernetes without having to deal with Kubernetes itself.

Azure Container Apps behind the scenes is still based on Kubernetes but as a developer you should not care about it. It has been set up for you, so it feels (and costs) like a serverless way of deploying containerized workloads to Azure, focusing on Cloud native applications and Microservice architectures.

What features does Azure Container Apps have to offer?

What are the features that development teams want when building and hosting microservices? ACA offers a way to deploy and scale a set of containers that make up an application while making sure all components can communicate with each other, scale based on load, are accessible from the outside, and can be deployed without downtime.



Looking at the components that define an Azure Container Apps solution, you always start with an "Azure Container Apps Environment". The Environment is a secure boundary around several "Container Apps" and makes it possible for these different container apps to communicate, much like an App Service Environment when using Azure App Services.

Within the Azure Container App Environment, you can create Azure Container Apps. Each app represents a single deployable unit which can contain one or more related containers. You could compare an Azure Container App to a deployment in Kubernetes. For each app, you can create a number of "revisions". Revisions are a way to deploy multiple versions of an app where you have the option to send the traffic to certain versions. Between revisions the ACA can be composed totally different: think of using different images, having additional containers etc.

These features are the basic concepts to run API's and frontends, but ACA also has features to host workers or background processes that are part of the microservice application. ACA has Kubernetes Event-driven Autoscaling (KEDA) built in. KEDA can scale background workers based on scaling rules , such as number of requests or the number of messages in a queue. These rules can be set up for each Container App individually, allowing them to scale based on their own needs.

Deploying an application to Azure Container apps

Azure Container Apps are built upon Kubernetes technologies, technologies which are hidden beneath the surface while deploying a new Container App. To get a better understanding of the technologies involved and the heavy lifting that is done, here is what actually happens when a Container App is deployed.

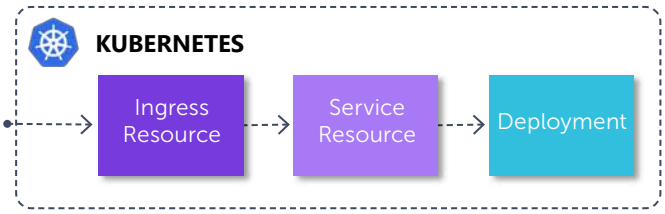
Containers can be deployed in Kubernetes in multiple ways. One way to rollout containers is by using deployments. A Kubernetes deployment can be defined as a yaml declaration describing which containers, storage volumes, and ports should be created, as well as the number of replicas. An example of a Kubernetes deployment which deploys three instances of the Nginx web server is shown below:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
   app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Each time the definition of the yaml changes and is reapplied on the cluster, a new revision is made and the running deployment is updated gradually to the new revision. One of the advantages is the revision history is stored within Kubernetes, allowing the administrator to roll back the deployment to a previous revision.

When an Azure Containerized App is deployed to Azure, the app will be packaged as a Kubernetes deployment, leveraging the benefits of a Kubernetes deployment. Each update to the ACA will result in a new revision that can be rolled back if needed.

For the ACA to allow ingress, a Service and an Ingress resource are created as well in the underlaying Kubernetes cluster.



The Service resource is a static endpoint inside the cluster and a mapping for Kubernetes to tie containers to specific ports. This is done using the key/value pair in the selector. In the example this is "app: nginx".

```
kind: Service
apiVersion: v1
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
  - port: 80
```

The Ingress resource then describes how the incoming traffic to the cluster is routed to the correct containers. In the example, when incoming http traffic on port 80 is detected, the traffic is forwarded to the service with the service name "nginx-service". The nginx-service will then route the traffic to all pods or deployments with the selector "ap: nginx".

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: nginx-ingress
  annotations:    ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - http:
    paths:
    - path: /
      backend:
        serviceName: nginx-service
        servicePort: 80
```

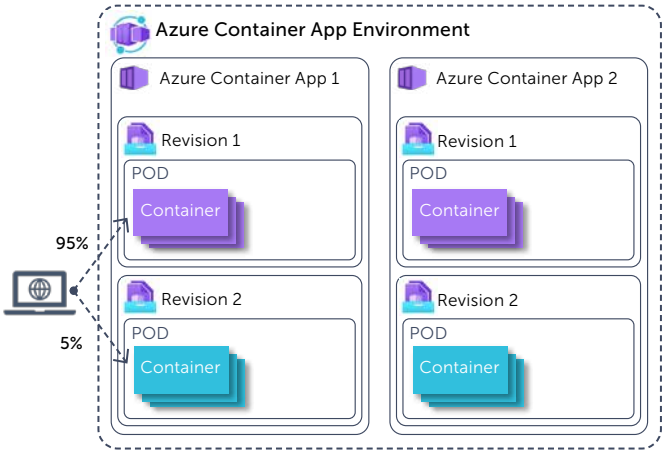
The equivalent of all this heavy lifting is done behind the screens when the following Bicep containerApp resource is deployed to Azure.

```
resource containerApp 'Microsoft.Web/
containerapps@2021-03-01' = {
     name: nginx-example
     kind: 'containerapps'
     location: 'west-europe'
     properties: {
     kubeEnvironmentId:  'xxxxxxxx'
     configuration: {
        secrets: secrets
        registries: []
        ingress: {
          'external':true
          'targetPort':80
        }
     }
     template: {
       containers: [
         {
           'name':'nginx'
           'image':'nginx:1.14.2'
           'command':[]
           'resources':{
              'cpu':'.25'
              'memory':'.5Gi'
            }
          }
       ]
     }
}
```

The template section in the Bicep example describes the containers to be deployed (`name: nginx`). The configuration section describes the equivalent of the Kubernetes service (`targetport: 80`) and the Kubernetes ingress (`external: true`).

Traffic splitting between revisions

Revisions in Container Apps allow us to split traffic between revisions to roll out new functionality gradually to our users.



Traffic splitting is done by adding traffic rules, in which the different revisions of the Container App get a different weight (note: the sum of the weights must equal 100).

```
{
  ...
  "configuration": {
    "activeRevisionsMode": "multiple"
    "ingress": {
      "traffic": [
        {
          "revisionName": <NAME_OF_REVISION_1>,
          "weight": 95
        },
        {
          "revisionName": <NAME_OF_REVISION_2>,
          "weight": 5
        }
      ]
    }
  }
}
```

In order to use traffic splitting, the `activeRevisionsMode` of the ContainerApp should be set to "`multiple`". If this mode is set to "`single`", a new revision would cause other revisions to be deactivated automatically.

Background workers in Azure Container Apps

Azure Container Apps bring the possibility to deploy "background" applications on Azure. These are applications that do not expose public endpoints and which can run forever.

By using standard Kubernetes Event-driven Autoscaling (KEDA) technologies, Azure Container Apps can scale up and down based on the number of events needing to be processed. The maximum number of replicas is set at 25 replicas. In most cases, Azure Container Apps can scale back to 0 replicas when they are idle.

Many KEDA scalers are available for a wide range of technologies (such as AWS, GCP, Azure, Redis, etc). Per Container App, the scaling metrics can be specified based on a number of rules that are different for each technology.

In the example below, the container app will scale up gradually to a maximum of 5 Replicas when the number of concurrent Http Requests is 100.

```
{
  ...
  "resources": {
    ...
    "properties": {
      ...
      "template": {
        ...
        "scale": {
          "minReplicas": 0,
          "maxReplicas": 5,
          "rules": [{
            "name": "http-rule",
            "http": {
              "metadata": {
                  "concurrentRequests": "100"
              }
            }
          }]
        }
      }
    }
  }
}
```

When you start working with Container Apps and KEDA triggers, documentation on the trigger specification can be found on the KEDA website (https://keda.sh).

The KEDA documentation shows code examples in YAML, while the Container Apps ARM template is in JSON. As you transform examples from KEDA for your needs, make sure to switch property names from kebab casing (everything in lowercase, with dashes between words) to camel casing (everything lowercase, all words after the first word start with uppercase).

Microservices using Dapr in Azure Container Apps

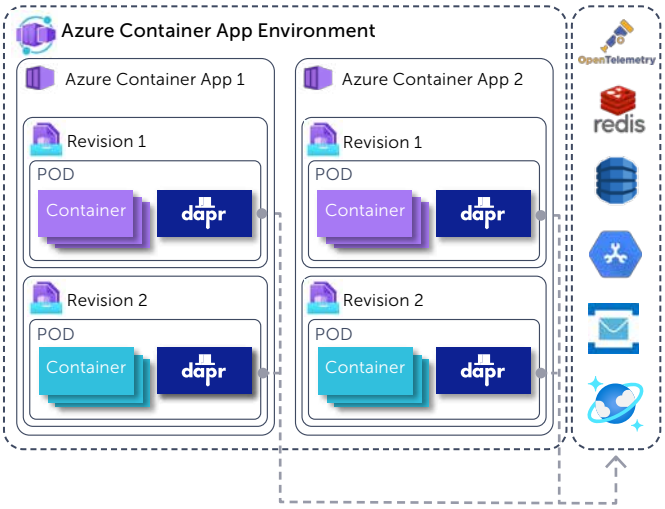
To make different components in the microservice landscape work together, ACA offers Dapr support out-of-the-box, by setting a configuration value to true. From there, on each app can use all the features of Dapr such as service location, pub/sub messaging, or distributed tracing.

Dapr is an open source project started a few years ago in the CTO office of Azure and in 2021 was donated to the CNCF Foundation and is now a CNCF Incubation project[1]. Dapr stands for "Distributed APplication Runtime" and helps developers focus more on their applications instead of knowing everything about the network, storage, monitoring tools, etc.

Dapr creates an abstraction for developers, so they only need one set of APIs to call other services, store state, or send messages. We wrote an article about Dapr in XPRT Magazine #10[2] and there are loads of information on the Dapr.io website.

Dapr works through a sidecar architecture. Azure Container apps make it possible by just checking a box to enable these sidecar containers to your Azure Container app without needing to setup anything yourself. By enabling this feature for your app, you can immediately use all the features Dapr provides.

Adding this feature again proves that Azure Container Apps chose an opinionated way of building containerized Microservice applications, although the use of Dapr is completely optional.



Azure Container Apps compared to other Container hosting options in Azure

With all the computing solutions Microsoft Azure is offering, it can be hard to choose the right one. When should you choose Functions, Web Apps, Container Instances, Kubernetes, or Container Apps? In this section we will help you make the choice.

[1] https://landscape.cncf.io/serverless?selected=dapr
[2] https://pages.xpirit.com/magazine10

Azure Container Instances

Azure container Instances are the simplest way to run a container in Azure. This is great for running certain processes that are not a web application or background worker because Azure Web Apps and Azure Functions offer more functionality in those scenarios. For other applications that are just a single container, Azure Container Instances is a great fit. Another downside of Azure container Instances is they do not have the ability to scale down to 0 instances, so you always have a certain cost.

Azure Function Apps

Function apps are serverless applications that run based on triggers such as http requests, timers or messages in a queue. Azure Functions need the least amount of configuration of infrastructure so you can focus on business logic. For smaller applications or processing jobs this is the perfect solution. The major downside of functions can be its "cold starts", where processing a first request after being idle can take a while. Because of that, we would not recommend it for APIs or hosting user facing web applications.

Azure Web Apps

Azure Web apps are the go-to solution for basic web applications or API's. As a PaaS solution, configuration is quite simple. There is no built-in support for multi region, so that must be managed by you outside of Azure Web Apps. Azure Web Apps also supports running containers and can be a great combination for front ends combined with workers as Function apps or container instances.

Azure Container Apps

ACA can be seen as a "Kubernetes To Go" solution, in which the developer can use a large amount of the power of Kubernetes without the hassle of maintaining a cluster. However, not all Kubernetes functionality is available for the user. An ideal scenario would be the containerization of microservices or any background process with fluctuating load peaks by using KEDA auto scalers. Having a full application in Azure Container apps in the future could combine the best of both worlds comparing it to the features that Kubernetes brings versus the simplicity of the combination of Azure Web Apps & Azure Functions.

Azure Kubernetes Service

AKS is the Kubernetes PAAS offering by Microsoft, in which Microsoft maintains the underlying cluster technologies and virtual machines. This does not mean it does not need maintenance. User management, ingress & egress routing, networking, security, and resource allocations are all things that have to be taken into account by you when using AKS. The learning curve can be steep to start working with Kubernetes, but it does offer a stack that can run almost any application in any technology stack. Combining that, along with scaling and self-healing / auto recovery services Kubernetes provides, makes this is a great option for organizations that want to host business critical applications.

Are Azure Container Apps the future of microservices on Azure?

Azure container Apps (ACA) is currently in public preview. It was announced at Ignite near the end of 2021 and still has some time to go before it will become Generally available (GA).

Since we thrive by the "you build it, you run it" mantra and love building microservice architectures, we're often in a love/hate relationship with Kubernetes. It has so many good features, but they come at a price of added complexity that development teams often can not grasp. Therefore, we love the concept of Azure Container Apps which brings a lot of these features without the complexity. However, Azure Container apps in its current state does have some flaws. We hope these would get solved soon and some of them are already on Microsoft's roadmap.

Some major improvements we would like to see?

Managed Identities

Managed Identities are the way to connect running services to other Azure resources such as databases or queues. At this moment Managed Identities are not supported yet but Microsoft announced this will be available before GA.

Investigating running containers

At this moment the only way to inspect running containers in ACA is through the logging in Log analytics. Microsoft already announced that they are working on a way to improve this. We have the hope this will make investigating issues during development will be a lot easier if that is possible.

Advanced traffic shaping

The current revisions within ACA will allow you to shape traffic to each revision. The only way to do this is by setting a certain % of the traffic towards it. We think this is a nice idea but in practice almost nobody uses it this way. It would be a lot better if we could shape the traffic in more advanced ways like sending traffic with certain http request headers to 1 revision or the other or other options all defined in the SMI-Spec[3].

Regional failovers

Azure Container apps currently have no options for regional failovers when there is an outage. One of the benefits of Kubernetes is you can have a cluster expand over multiple regions and it can handle failure of the compute in a zone or region. You could deploy the same ACA in 2 regions and put an Azure Frontdoor in front of them to direct the traffic, but it would be nice to have this built into the service especially in countries that focus on 1 region. ACA does automatically deploy to multiple availability zones for high availability so that's a good start.

Limited hardware configuration options

When creating Container Apps, you can allocate CPU and memory resources to them. Currently there are only options ranging from 0.25 CPU cores and 0.5Gi memory to 2 CPU and 4Gi memory. We think this should be more flexible for apps that are not heavy on the CPU but do need more memory or the other way around.

The Future of hosting microservices in Azure?

Azure Container Apps is still in preview. There are a lot of improvements already underway. Time will tell. We are enthusiastic about the movement to a more serverless way of running a Kubernetes-like environment for our microservices. Azure Container Apps is a big step into the right direction. As of this writing, there are just a few features missing that would prevent us from using this in production, such as the lack managed identity. We do believe, if these would be added, this could become a dominant platform for hosting containerized workloads, especially for microservice based applications. </>

Geert van der Cruijsen
Trainer, Digital Kickstarter, Enabler for companies to embrace DevOps, Cloud & improve their engineering culture

xpirit.com/geert

Bas van de Sande
Azure Coding Architect, Consultant, Integrator

xpirit.com/bas

[3] https://github.com/servicemeshinterface/smi-spec/blob/main/apis/traffic-split/v1alpha4/traffic-split.md

Beacons create safe routes for Maersk's voyages through the cloud

Maersk is the world's largest container shipping company, with end-to-end services spanning the entire supply chain. The company's logistics web uses a robust infrastructure that makes use of a cloud platform. Adequate protection against today's regular malware and virus attacks is of crucial importance for Maersk's business continuity. To steer a safe course in combining agile ways of working and cloud-based development with compliance with Maersk's security and quality standards, Maersk collaborated with Xpirit, and introduced the Unified Delivery Model (UDM). The model uses beacons to signal teams as well as stakeholders of the extent in which they are on course, compliant with standards, and secure against any threats. The beacons even provide functionality for automatic reconciliation, thus repairing IT components from any malfunction. The beacons proved their value in December 2021 when a Log4j vulnerability was signaled in time and any affected component and server could be returned to business as usual, within a minimum of time.

Author Bruno Amaro

Comprehensive cloud infrastructure

Maersk's full-service portfolio goes well beyond container shipping, and comprises the complete supply chain, from factory to warehouse and from farm to refrigerator. A robust infrastructure is the backbone of the company's intricate logistics web, with IT components that are being developed by teams in different locations. The teams use a cloud-based environment, which was introduced in 2018. Bruno Amaro, head of Cloud Compliance for Maersk, describes the journey into the cloud: "Naturally, compliance with our security and quality standards was, and is, of key importance for our business continuity and quality of service. This required extremely thorough analysis while we were designing, developing and implementing the cloud infrastructure, which is why we involved Xpirit's experienced consultants. They provided great help in getting the infrastructure implemented, but their special value lies in their thorough analysis. They constantly challenge you in your ideating process and tell you when you're about to deviate from your mindset and intended course. Their typically Dutch directness proved to be a great help and by not leaving a stone unturned."

Compliance with security and quality standards monitored by beacons

Bruno continues: "Naturally, the approach of teams working in all corners of the world working in an agile, DevOps-based manner required constant and through monitoring. Not only in terms of compliance with safety and security, but also operational efficiency. This is why we introduced the Unified Delivery Model (UDM), a framework that allows teams to be responsible for their own value-chain via a self-service portal. A key component of the model consists of beacons that signal teams of the extent to which they are on course, compliant with standards, and secure against any threats. The beacons light up in green to indicate that you're on the right course without any issues, while they light up in red to indicate any type of risk or non-compliance. For instance, when Microsoft releases a new security patch, the beacons automatically turn red, alerting the teams that an action is required to get back to compliance. Stakeholders also see the beacons, so in addition to keeping the development teams on track, they serve as an efficient communication channel."

Erick Segaar, one of Xpirit's team of consultants who were involved in the project for years: "One of the great qualities of beacons is that they don't block anything, unlike many other compliance measures. They signal possible issues in time and ahead of actual problems and notify you when you need to adjust your approach. What's more, the beacons offer a self-healing mechanism, thus reconciling and remedying issues without any human interaction. While this feature in itself was not super complex to implement, the challenge was to leave the responsibility with the teams without limiting their autonomy."

Bruno adds on a light note: "While the beacons constantly alerted each of our teams, I was caught off guard when I visited the Xpirit team in their office in Hilversum. The day I arrived we did some great team-building, went go-karting, lots of drinks and dinner in the evening. However, that night in my hotel room I got so sick, I saw all colors of the rainbow, and I believe I said yes to everything during the meetings on the second day, not seeing any red beacon."

Immediate and timely measures against log4j vulnerability

One remarkable result of using the beacons was our extremely effective response to a log4j vulnerability that occurred in December 2021. Bruno: "The beacons signaled us well in time of the risk that certain servers were affected, and this allowed us to take the required measures well in time, thus preventing any impact on our business. In short, a wonderful confirmation of the security measures we took on our cloud journey, as well as the valuable cooperation with Xpirit." </>

Bruno Amaro
Senior Engineering Manager - Cloud Engineering Maersk Technology

Stop wrestling with ARM Templates, work on your Biceps

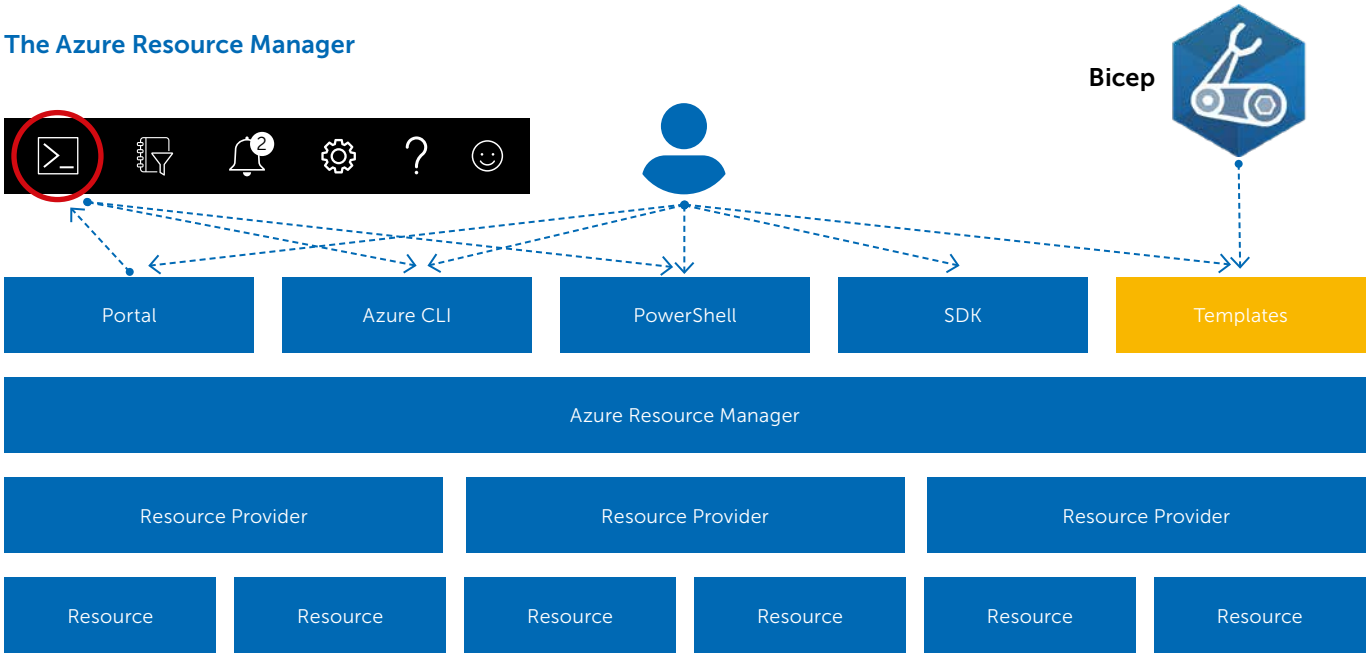
Creating resources in the Azure cloud can be done in many ways. If you've ever used Azure, you most certainly made a resource using the Portal experience at portal.azure.com. Besides this portal, you can also use PowerShell or the Azure CLI. When you want to manage your infrastructure from an application, you can work with an SDK and, for example, create a resource using C#. Finally, there are also options to manage resources using templates. ARM templates have been around for quite a while, and now you have a new option: Bicep! Bicep aims to make managing your infrastructure in a declarative way much easier than it was with ARM templates. ARM templates are written in JSON and are therefor harder to write, read, and much larger in size. It is also quite hard to break up ARM templates into multiple modules to create maintainable and reusable templates. Bicep is a domain-specific language (DSL) aiming to solve all of these problems for us!

Author Erwin Staal

All the different options you have to manage infrastructure in Azure have one thing in common: they use the Azure Resource Manager underneath. The below diagram shows the various options in relation to the Azure Resource Manager.

The first row in the diagram shows you the various options we just mentioned; the portal, Azure CLI, PowerShell, SDKs, and templates. On the second row, you see the Azure Resource Manager. It's the service in Azure that allows you to deploy and manage resources. The actual work of creating resources

The Azure Resource Manager



is delegated to a resource provider. There is, for example, a resource provider for everything around virtual machines and another one for all things related to Web Apps.

Except for the portal, all these options allow you to create and manage your infrastructure using Infrastructure as Code practices in a descriptive model. As with source code for your applications, you get the same benefits as versioning, auditability, traceability, and repeatability by storing it in source control and deploying it using a deployment pipeline.

Creating your first resource

Now that you know a bit about where to place Bicep in the Azure-provisioning landscape, let's dive in by creating a simple resource using Bicep. Before you can start, you need to install a few tools:

- Install Visual Studio Code (https://code.visualstudio.com/download)
- Install the Bicep extension for Visual Studio: 'ms-azuretools.vscode-bicep' (https://marketplace.visualstudio.com/items?itemName=ms-azuretools.vscode-bicep)
- Install the Azure CLI (https://docs.microsoft.com/en-us/cli/azure/install-azure-cli)

For example, we will create a storage account in Azure using Bicep. Open Visual Studio Code and create a new file called "storageAccount.bicep". Within that file, start typing 'stor'. You will see the extension will immediately begin helping you write the templates by presenting you with a few snippets.

```
stor
    res-storage                          Storage Account
    res-app-security-group
    res-cosmos-sql-container
```

Hit Enter and the snippet will be inserted. It looks like the following example:

```
resource storageaccount 'Microsoft.Storage/
storageAccounts@2021-02-01' = {
  name: 'name'
  location: location
  kind: 'StorageV2'
  sku: {
    name: 'Premium_LRS'
  }
}
```

The snippet starts with the keyword resource, indicating you want to create a resource. Next is the deployment's name, followed by the resource type and its version. Within the curly braces, you find the details of the resource like its name, location, and SKU.

While you haven't touched your mouse or keyboard yet, you will see that the editor selected the deployment name, allowing you to change that value. When you hit the tab key, you will automatically move to each property you can edit.

Again, a nice benefit of the extension. Notice how the extension also lists available options for the properties with a fixed set of options, like the 'kind' on the storage account. That saves you from having to look them up and make typing errors.

```
resource storageaccount 'Microsoft.Storage/
storageAccounts@2021-02-01' = {
  name: 'mystorageaccount'
  location: resourceGroup().location
  kind:   'StorageV2'
  sku: {  'Storage'
    name  'BlobStorage'
  }       'BlockBlobStorage'
}         'FileStorage'
```

When you've edited the properties, your storage resource looks like this example:

```
resource stg 'Microsoft.Storage/storageAccounts@2021-04-01' = {
  name: 'mystorageaccount'
  location: resourceGroup().location
  kind: 'StorageV2'
  sku: {
    name: 'Standard_LRS'
  }
}
```

The name of the storage account is currently hardcoded. That is not ideal since you want to use this template for multiple environments, such as test and production. The Microsoft naming convention recommends making the name reflect that. In Bicep, you can use a parameter to provide values, like the environment, at runtime.

Define a parameter as follows:

```
param env string = 'test'
```

You start with the keyword 'param' followed by its name. Next, you define its type, in this example, a string. Other options are an integer, bool, array, or object. Optionally, you can set a default value like 'test' in the example above. The same can be done for the location property of the storage account, or you can use a function like in the previous example to get the location from the resource group in which it lives.

In addition to parameters, we can use variables for values that you want to reuse across your templates but are not provided at runtime. Creating a variable that holds the name of the storage account looks like this:

```
var storageAccountName = 'stordemo${env}'
```

Notice how you can use string interpolation to combine 'stordemo' with the 'env' parameter into the variable. The result of using both parameters and a variable is shown below:

```
param env string = 'test'
param location string = 'westeurope'

var storageAccountName = 'stordemo${env}'
```

```
resource stg 'Microsoft.Storage/storageAccounts@2021-04-01' = {
  name: storageAccountName
  location: location
  kind: 'StorageV2'
  sku: {
    name: 'Standard_LRS'
  }
}
```

Another interesting feature that you can add is the use of the output keyword. That allows you to, for example, return the URL of the blob endpoint on the storage account. Defining an output looks like this:

```
output blobEndpoint string = stg.properties.
primaryEndpoints.blob0
```

Defining an output is similar to defining a parameter. The output keyword is used, and it's given a name: 'blobEndpoint'. You specify its type and then provide a value. Notice how you can use the deployment's name and the dot notation to get the properties of a resource.

Deployment

Now that you have written the first resource, let's deploy to Azure. The funny thing is that Azure itself doesn't know Bicep at all. Azure understands good old ARM templates, so your Bicep template will be transpiled into an ARM template and deployed to Azure. You can do that transpilation yourself, but the Azure CLI also supports deploying a bicep file directly and will do the transpilation for you. Let's do the transpilation to see the result using the Azure CLI. Run:

```
az bicep build -f storageAccount.bicep
```

The output is the following ARM template:

```
{
  "$schema": "https://schema.management.azure.com/
schemas/2019-04-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "metadata": {
    "_generator": {
      "name": "bicep",
      "version": "0.4.1008.15138",
      "templateHash": "7691361711088743744"
    }
  },
  "parameters": {
    "env": {
      "type": "string",
      "defaultValue": "tst"
    },
    "location": {
      "type": "string",
      "defaultValue": "westeurope"
    }
  },
  "functions": [],
  "variables": {
    "storageAccountName": "[format('stordemo{0}',
    parameters('env'))]"

  },
  "resources": [
    {
      "type": "Microsoft.Storage/storageAccounts",
      "apiVersion": "2021-04-01",
      "name": "[variables('storageAccountName')]",
      "location": "[parameters('location')]",
      "kind": "StorageV2",
      "sku": {
        "name": "Standard_LRS"
      }
    }
  ],
  "outputs": {
    "blobEndpoint": {
      "type": "string",
      "value": "[reference(resourceId('Microsoft.
              Storage/storageAccounts',
              variables('storageAccountName'))).
              primaryEndpoints.blob]"
    }
  }
}
```

You immediately see that the ARM template is almost three times as large as the bicep equivalent! You will probably also agree that the bicep version is much more readable than the JSON in this ARM template. Those are just a few of the benefits of using Bicep over ARM tempaltes.

We will use the Azure CLI to do the deployment. You first need to login and select the correct subscription using the following commands:

```
az login

az account set -s <subscription id or name>
```

As every resource in Azure lives in a resource group, you first need to create that. Later, we will see how to create one with Bicep. For now, use the Azure CLI:

```
az group create -l westeurope -n rg-bicepdemo-test
```

Now that your resource group is ready, you can deploy the template using the command below:

```
az deployment group create --resource-group rg-bicepdemo-
test --template-file storageAccount.bicep
```

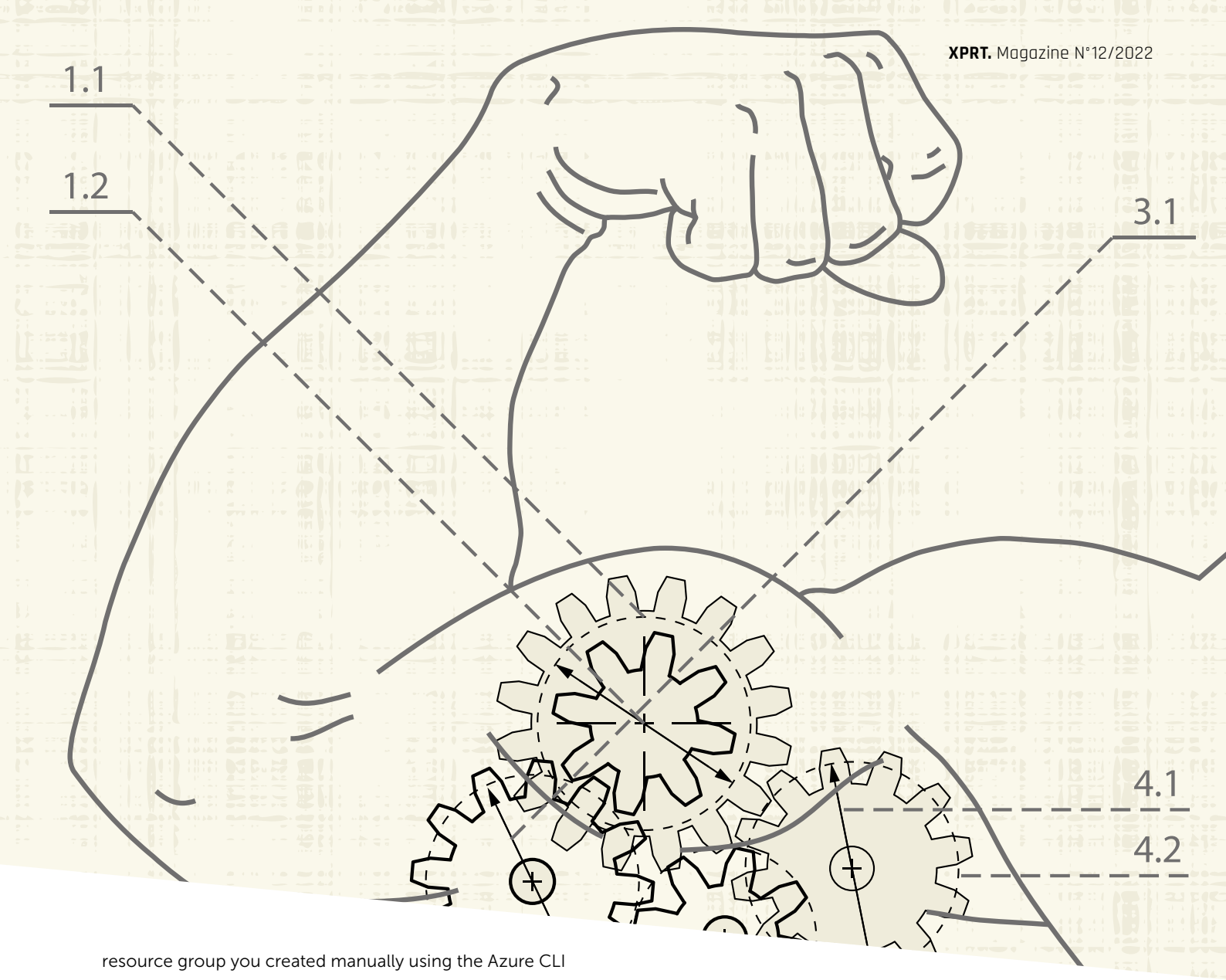
When you do not provide values for the parameters, the defaults in the template will be used. The following command shows how to provide a parameter while deploying the template. Passing parameters allows you to reuse the template and target multiple environments.

```
az deployment group create --resource-group
rg-bicepdemo-test \
    --template-file storageAccount.bicep  \
    --parameters '{ \"env\": { \"value\": \"prod\" } }'
```

Now open the Azure portal and verify the storage account has been created.

Modularize your Bicep template

If you continue to add resources to the file we just created, it will get bigger and bigger. Eventually, it will become harder to read and maintain, and the template will get harder and harder to reuse. Luckily, Bicep has the concept of modules. Modules allow you to break up your template into smaller, reusable parts. The template you've just created is an excellent example of what can be in a module. Now let's create the

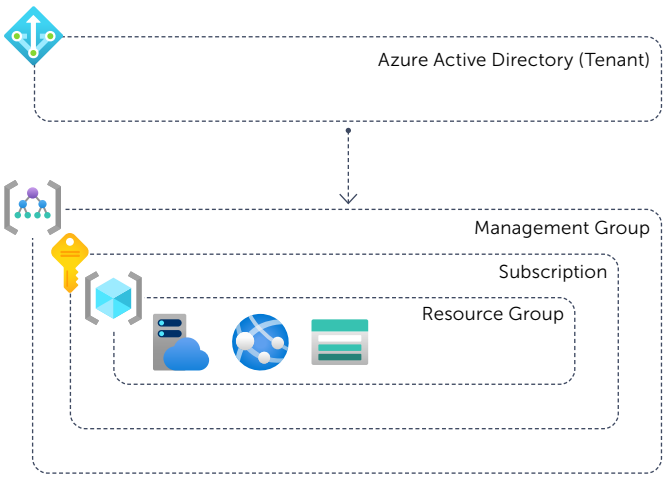


resource group you created manually using the Azure CLI using Bicep and see how we can use the storage account template as a module. Start by creating a new file called "main.bicep", and add the following snippet to the "main.bicep" to create the resource group:

```
param env string = 'test'
param location string = 'westeurope'

resource stg 'Microsoft.Storage/storageAccounts@2021-04-01' = {
  name: 'rg-bicepdemo-$(env)'
  location: location
}
```

The above snippet creates a Resource Group of which the deployment is called 'rg'. Now you've done that, you will see that VS Code will show an error on the above new resource. By default, a bicep template is deployed at the scope of a resource group. As you may know, in Azure, there are different levels at which we can deploy resources. We call these the deployment scopes. At the root, you have your Azure Tenant. That can contain one or more Management Groups. These can contain one or more subscriptions, and each subscription can contain one or more resource groups. Finally, the resource groups contain the actual resources like Virtual Machines, a Web App or Storage Account. This hierarchy allows you to group and manage your resources in a structured way and is shown in the image below.



You get the above error since you cannot create a resource group within a resource group; the deployment scope is wrong. A resource group needs to be deployed at the subscription scope, so you need to add the following line to the top of the "main.bicep":

```
targetScope = 'subscription'
```

The error should disappear. To use the storage account file as a module, you use the 'module' keyword instead of the 'resource' keyword. You give it a name like you do when using the 'resource' keyword. Instead of specifying a type, you now reference the just created module using its path. Below the resource group, start typing 'module stg <space>', and VS Code should show you all available modules:

```
module stg
          storageAccount.bicep
```

Select the storageAccount.bicep. Type '= <space>' and then select the 'required-properties' option in the drop-down.

```
module stg 'storageAccount.bicep' =
                for
                for-filtered
                for-indexed
                if
                required-properties        Required properties
                {}
```

The generated snippet looks like this:

```
module stg 'storageAccount.bicep' = {
  scope:
  name:
}
```

On the first line in that module, you find 'scope'. That is where you define to what scope this module should be deployed. Remember that the "main.bicep" template targets the subscription scope, but a storage account can only be deployed within a resource group. This scope property allows you to set it. You simply do that by using the name of the resource group you declared earlier like so:

```
module stg 'storageAccount.bicep' = {
  scope: rg
  name: 'storage'
}
```

Remember that the storage account also has two parameters. They were not added when you created the module using the 'required-properties' since they have a default value. You can pass a value to them by specifying params on the module like so:

```
module stg 'storageAccount.bicep' = {
  scope: rg
  name: 'storage'
  params: {
    env: 'prod'
  }
}
```

Deploying this template is slightly different from when you deployed storageAccount.bicep previously Since we now target the subscription scope, you need to specify that in the command. The command now becomes:

```
az deployment sub create --template-file main.bicep
-l westeurope
```

Notice that instead of `group` you now use `sub` to indicate the different deployment scope. When you run the command, it should succeed, and the result in Azure should be the same since the resource group and storage account already exist.

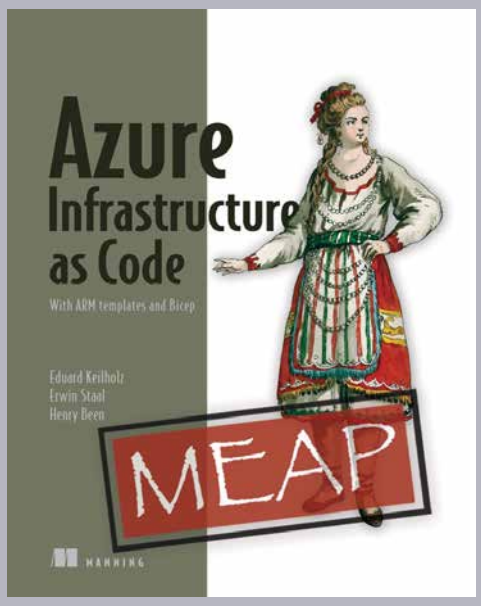
In this article, you've learned how easy it is to get started with Bicep to create and deploy your first resource. You've also learned how to create a module to craft small, reusable, and maintainable Bicep templates. If you want to know more about sharing these modules within your organization, then make sure to find the article on 'Shift left using blessed templates with Bicep' by Erick Segaar elsewhere in this magazine. </>

Erwin Staal
Azure Architect

xpirit.com/erwin

Want to know more about Infrastructure as Code on Azure?

Erwin, together with two friends, wrote a book on it! It discusses ARM templates and, of course, Bicep. It shows how to deploy these templates using Azure DevOps or GitHub Actions, talks about sharing templates across the organization, how to govern your Azure environment using Azure Policy, and much more. Find out more and buy the book at https://www.manning.com/books/azure-infrastructure-as-code



Shift left using Bicep

Blessed templates: It is a good practice to offer some building blocks of blessed templates for the infrastructure within many organizations. A Cloud Competence Center of Excellence commonly provides these templates. By the nature of control, the focus is primarily on the Security, Architecture, and Governance part of things, compared to Engineering enablement and Operational Excellence. The organization can run the strategy "comply or explain[1]" by providing blessed templates, this means that it is easy to comply by using the blessed templates and no questions are asked. When you do need to deviate from the blessed templates you need to explain and go through a review board. The templates are blessed in the form that the involved parties have already approved for use. Many teams will be able to run their solutions based upon these templates.

Author Erick Segaar

Managing changes to the blessed templates can be a bit challenging. How do we patch it efficiently on all resources during a security vulnerability? What if a product team has moved on to the next project and do not actively support the previous project? How would teams know if they need to change or re-deploy their infrastructure? The effectiveness stands or falls by the convenience for the teams to comply. How easy is it to reach teams that use your template, and how easy is it to change the template and re-deploy.

This article will explain why you can keep using your blessed templates or easily convert them to bicep files and gain their benefits. For more information about the general use of Bicep, you should read the article "Stop wrestling with ARM Templates" written by Erwin Staal in this same magazine.

Using the modular improvements introduced in Bicep v0.4.1008 to support the Bicep registry, you can improve your support of blessed templates to your consumers and have compile-time validation to support complete CI/CD scenarios for your IAC (Infrastructure As Code). Let's see how this impacts the ease of use and blessed templates lifecycle.

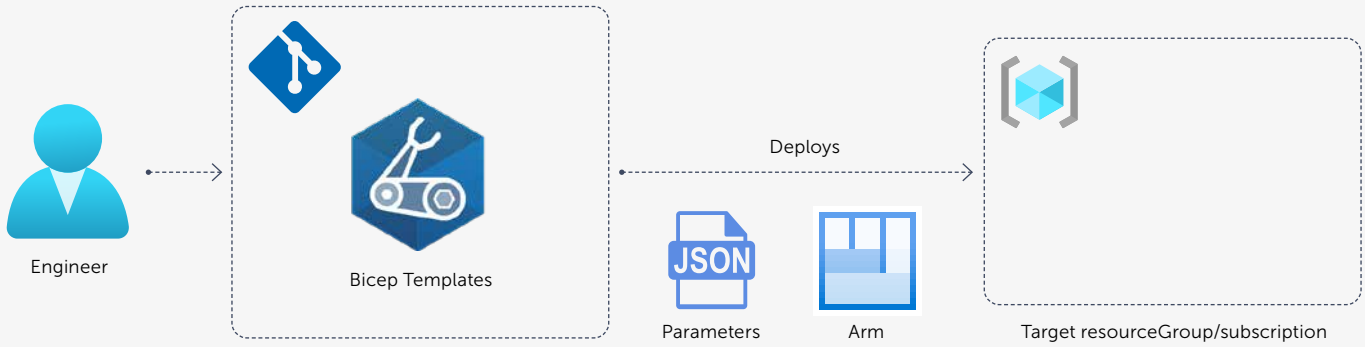


Figure 1. Basic Bicep deployment

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

Basic Bicep deployment

To understand the value of using a Bicep registry for your templates, we first need to understand how things work without templates, as shown in the diagram below.

A relatively standard CI/CD pipeline for infrastructure written in Bicep, where you don't use templates:

1. An engineer makes a change to a Bicep file in Git
2. When pushing the change, a pipeline will be automatically triggered
3. The first step in the pipeline is to transpile the Bicep template into an ARM template and store that as an immutable artifact for later use
4. Deploy the artifact, with environment-specific parameters, to an Azure -ResourceGroup, -Subscription, -Tenant, or -Management group

The deployment is nothing more than running an az CLI command using the "deployment group" arguments with the transpiled Bicep template file passed with the --template-file switch, as shown in the following Powershell.

```
[CmdletBinding()]
param
(
  [Parameter(Madatory=$true)]
  [string] $environmentCode,

  [Parameter(Madatory=$true)]
  [string] $resourceGroupPrefix,

  [Parameter(Madatory=$true)]
  [string] $templatefile

1 reference
function DeployEnvironment {
    [CmdletBinding()]
    param(
        [string] $environmentCode,
        [string] $resourceGroupPrefix,
        [string] $templateFile
    )

    $resourceGroup = "$resourceGroupPrefix-
    $environmentCode"

    az group create `
        --location westeurope `
        --resource-group $resourceGroup
    if ($LASTEXITCODE) {
      throw "Unable to create resourcegroup
      [$resourceGroup] in westeurope"
    }

    az deployment group create `
        --template-file $templateFile `
        --resource-group $resourceGroup `
        --parameters env="$environmentCode" `
        --output tsv
    if ($LASTEXITCODE) {
      throw "Unable to deploy bicep to resourcegroup
      [$resourceGroup]"
    }
}

DeployEnvironment -environmentCode $environmentCode
-resourceGroupPrefix $resourceGroupPrefix
-templateFile $templateFile
```

Figure 2. Deploy Bicep to a resourcegroup

Using templates

The use of templates helps you re-use definitions you already created. When deploying a web service, you always want to deploy application insights with a log-analytics workspace. Using templates this is done in multiple ways, for example, by referencing:

- a (local) folder in the same project
- a storage account in Azure
- a template-spec resource in Azure
- a module in the Bicep registry

Template-spec

Before the possibility to push Bicep templates to a Bicep registry existed, the preferred way of sharing your templates was by publishing the ARM template to a Template-Spec-Resource. Below you can see such a template-spec resource in Azure.

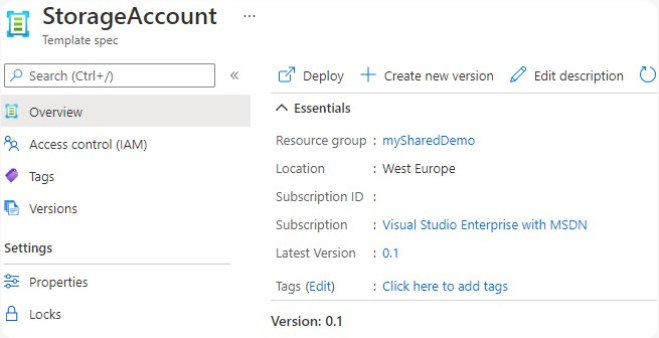


Figure 3. Template-spec resource example

Notice how this can leverage the RBAC (a consumer needs only read-access), versioning, and even release documentation when drilling down to a specific version.
A template-spec is published using the CLI like this:

```
az ts create `
    -g $resourceGroupName `
    --name $templateSpecName `
    -v 0.1 `
    -l westeurope `
    --template-file iac/templates/StorageAccount.js `
    --display-name StorageAccount `
    --description 'Blessed template for StorageAccount' `
    --version-description "Simplified blessed template"
```

Figure 4. Publish template-spec

Another Bicep file can then use this Template-Spec-Resource by using "Microsoft.Resources/deployments" type with a "templateLink" property referencing the Template-Spec-Resource with a specific version as shown below:

```
resource templatespec 'Microsoft.Resources/
deployments@2021-01-01' = {
  name: 'blessed-sa'
  properties:{
    templateLink:{
      id : '/subscriptions/ .../resourceGroups/myShareDemo/
      providers/Microsoft.Resources/templateSpecs/
      StorageAccount/versions/0.1'
    }
```

Figure 5. Using a template link

The template link describes a full URI, including the desired resource's name and version.

Compared to our Basic Bicep Pipeline, using the "Template Specs" helps you already to achieve a blessed template structure, having semantic versioning and release documentation together with your template definition. Template specs are already altering the CI/CD flow using the blessed template during deployment.

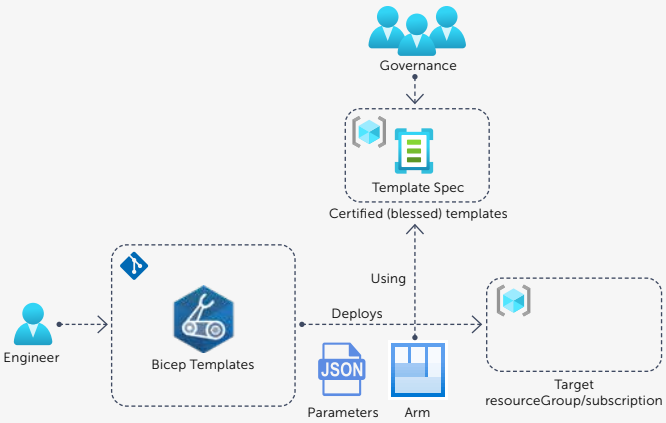


Figure 6. Deploy Bicep using template specs

The maintenance, approval, and availability are taken care of by a separate team, having the ability to publish new templates and versions. The consumers only have read privileges on the template spec and can use it during the deployment of their resources.

The use of the template-spec as a URI reference, as shown in figure 3, is a bit clumsy. Due to not having IntelliSense over the Template-spec, you have to know the names of the parameters to pass them. The same applies to the output of the template spec. You also need to know the name of the parameter to retrieve it. We can improve on this using a module instead of a resource. A module can refer to a local file or a template spec. To use a template spec, use the following format:

```
module <symbolic-name>'ts/<alias>:
<template-spec-name>:<version>' = {
```

When looking at the template below, you can see no direct reference to a subscription anymore:

```
module templatespec2 'ts/
BlessesTemplates:StorageAccount:0.1' = {
  name: 'blesses-sa'
  params:{
    name: saName
    container: [
     'mycontainer'
    ]
  }
}
```

Figure 7. Using template-spec module

The reference is abstracted away into a configuration file, making your definitions more readable and easier to maintain.

```
{
  "moduleAliases": {
    "ts": {
      "BlessedTemplates": {
        "subscription": "...",
        "resourceGroup": "myShareDemo"
      }
    }
  }
}
```

Figure 8. Configure bicepconfig.json for template-spec

You place the configuration file, called called "bicepconfig.json", in the root of your project. Here you can define the "ts" "Template Spec". The alias in this example is 'Blessed-Templates'. The other benefit of using the module approach is Bicep will recognize the link, start downloading the definition to your local user's folder, and provide IntelliSense during your development. It, therefore, becomes much easier to use parameters or the outputs of the resource.

There are a few shortcomings of using "Template Specs":

› While using template specs, you reference it by a link. This causes validation to happen at deployment time, instead of build time, which is a bit late

- The content of the template-spec is not known client-side. Looking at the ARM template, you will only see the reference to the template-spec instead of seeing the nested resources of the spec. In our example of the web services, you would only see a reference to a web service template-spec and not be aware that an application insight and a log analytics workspace would also be deployed.

Bicep registry

Using the modular improvements introduced in Bicep v0.4.1008, we can now use a Bicep module registry. This improvement enables us to publish Bicep modules to an Azure Container Registry, as shown in the following command.

```
az acr login --name mybicepsharedregistry.azurecr.io
bicep publish StorageAccount.bicep --target
br:mybicepsharedregistry.azurecr.io/bicep/modules/
storage:0.1
```

First, you need to log in to the Azure Container Registry and publish the Bicep file. Next, add the configuration to the bicepconfig.json and reference the module as you did for the template spec. This time, you use the "br" keyword. This keyword helps Bicep to understand it can retrieve the modules from the bicep registry and thus enabling IntelliSense and compile-time validation.

```
{
  "moduleAliases": {
    "br": {
      "StorageModules": {
        "registry": "mybicepsharedregistry.azurecr.io",
        "modulePath": "bicep/modules"
      }
    },
```

Figure 9. Configure bicepconfig.json for bicep-registry

```
module templatespec3 'br/StorageModules:storage:0.1' = {
  name: 'blessed-sa'
  params:{
    name: saName
    containers: [
      'mycontainer'
    ]
  }
}
```

Figure 10. Using bicep registry module

Another significant benefit, compared to the template-spec, is that when the Bicep file is transpiled into an ARM template, you will get nested templates instead of a link to the template. A nested template explains what resources will be modified. In contrast, a template link only references a template spec that will be accessed during deployment, making it harder to understand what is happening while reviewing an artifact for deployment approval. As an example, the first screenshot below shows the use of a template spec, the second one the use of a module in the Bicep registry.

```
  "resources": [
    {
      "type": "Microsoft.Resources/deployments",
      "name": "blessed-sa",
      "properties": {
        "templateLink": {
          "id": "/subscriptions/ ... /resourceGroups/
          myShareDemo/providers/Microsoft.Resources/
          templateSpecs/StorageAccount/versions/0.1"
        }
      }
    }
  ]
```

Figure 11. Template-spec in an ARM template, showing only resource/deployment

```
  "resources": [
    {
      "type": "Microsoft.Resources/deployments",
      "name": "blessed-sa",
      "properties": {
        "template": {
          "resources": [
            {
              "type": "Microsoft.Storage/storageAccounts",
            },
            {
              "type": "Microsoft.Storage/storageAccounts/
              blobServices",
              "dependsOn": [
                "[resourceId('Microsoft.Storage/
                storageAccounts', parameters('name'))]"
              ]
            }
            {
              "type": "Microsoft.Storage/storageAccounts/
              blobServices/containers",
              "dependsOn": [
                "[resourceId('Microsoft.Storage/
                storageAccounts', parameters('name'))]"
              ]
            }
          ],
          "outputs": {
            "blobserviceId": {
              "type": "string",
              "value": "[resourceId('Microsoft.
              Storage/storageAccounts/blobServices',
              parameters('name'), 'default')]"
            }
          }
        }
      }
    }
```

Figure 12. Bicep-registry usage in an ARM template, showing resource/deployment with all child resources used

Shift left

Using either the template-spec or bicep registry will gain the Shift Left[2] capability of compile-time validation instead of deploy time validation as shown below.

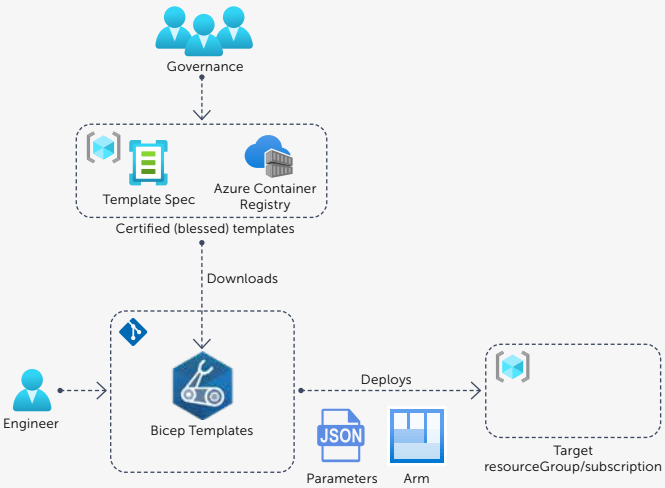


Figure 13. Compile-time validation using container registry

Because Bicep will download the ARM templates and the Bicep files to the local user's folder, it will validate during compile-time. Compile-time validation will help you fail your pipeline in the build step before creating the immutable artifact you want to deploy to your environments.

Renovate-bot dependency automation

One of the questions we asked at the beginning of this article was how we manage changes to our blessed templates and enable our consumers to detect changes that they need to deploy quickly. We can use a dependency manager like Renovate-bot to detect new versions using semantically versioned Template-specs or Bicep registries.

Implementing the Renovate-bot will enable the following flow:

1. Renovate-bot will scan the organization's repositories for out of date dependencies

[2] The capability to find defects earlier in your development lifecycle

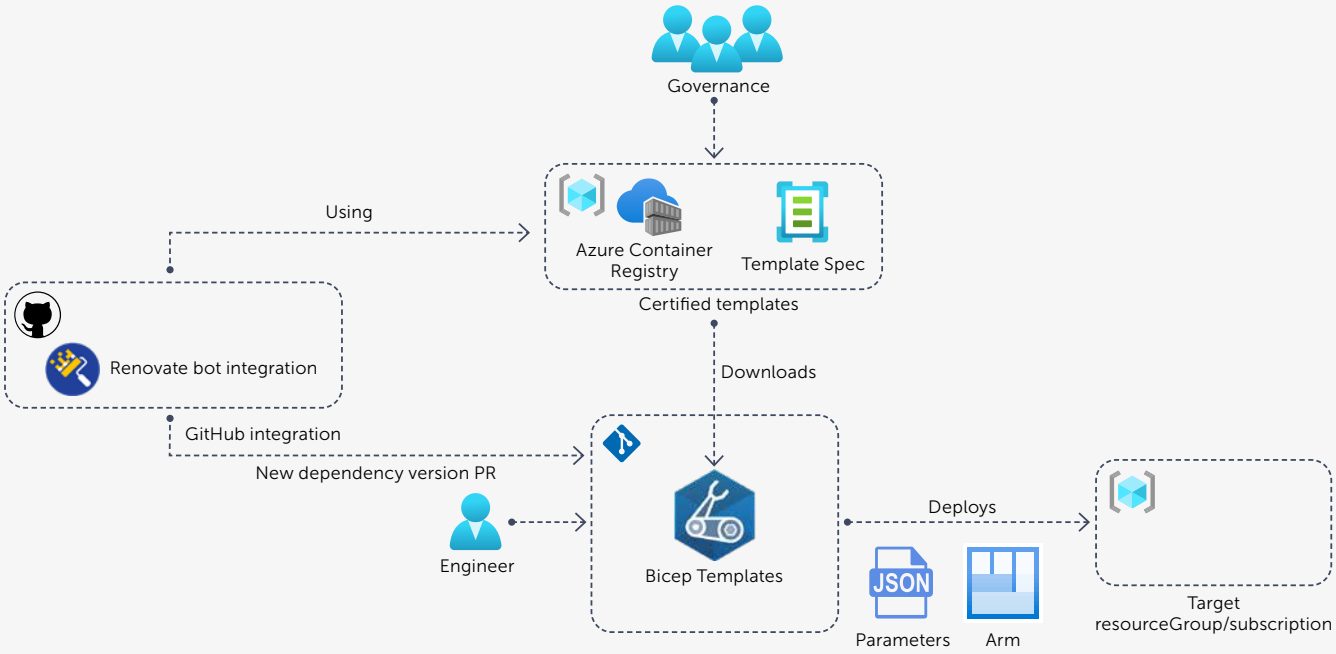


Figure 14. Renovate-bot dependency manager

2. Renovate-bot submits a Pull Request into the repositories that use the blessed templates, enabling your consumers to approve or auto-approve the Pull Request and stay secure and compliant.
3. Approval will automatically trigger your CI/CD pipeline and roll out the new templates to their environment.

Configure Renovate-bot

To make use of the Renovate-bot, follow the website[3] guidelines. Renovate-bot can be integrated with industry-standard CI/CD tooling and runs on a hosted or on-premise environment. The easiest way to enable this is by installing it as a service into your GitHub account.

As a team managing the blessed templates, you want Renovate-bot to pick up on published changes. To pick up those changes you can use an example as shown below.

```
- shell: pwsh
  id: publish-version
  run: |
   $manifest = az acr repository show-manifests `
        --name mybicepsharedregistry `
        --repository bicep/modules/storage `
        --top 1 `
        --orderby time_desc | ConvertFrom-Json
    $version = [version]$manifest.tags[-1]
    $newversion = "$($version.Major).$($version.Minor+1).0"

    az acr login --name mybicepsharedregistry.azurecr.io
    --expose-token
    bicep publish .\src\storageaccount.bicep `
        --target br:mybicepsharedregistry.azurecr.io/
        bicep/modules/storage:$newversion
    echo "::set-output name=version::$newversion"
```

```
- name: Create tag
  uses: actions/github-script@v3
  with:
    github-token: ${{ github.token }}
    script: |
      github.git.createRef({
        owner: context.repo.owner,
        repo: context.repo.repo,
        ref: "refs/tags/${{ steps.publish-version.outputs.
        version }}",
        sha: context.sha
      })
```

Figure 15. Upgrade version, tag, and publish

To always have a valid version, this script does the following:

1. Get the current version from the latest Bicep registry manifest
2. Increment the minor version to create a new, unused version
3. Publish the Bicep with the incremented version to the registry
4. Create a GitHub tag on the current SHA[4] used to run the build

Now that we've tagged the release in GitHub, we can use the Renovate Managers[5] to configure an override for bicep files. Renovate managers are like package managers. These managers know, for a specific resource (such as docker, dotnet, golang, etc.), how to determine the latest published version and compare it to the version used in the repository. Because there is no dedicated manager for Bicep, we need to configure our own using the generic regex manager.

[3] https://www.whitesourcesoftware.com/free-developer-tools/renovate/
[4] "SHA" stands for Simple Hashing Algorithm. The checksum is the result of combining all the changes in the commit and feeding them to an algorithm that generates these 40-character strings. A checksum uniquely identifies a commit.
[5] https://docs.renovatebot.com/modules/manager/

```
"regexManagers": [
   {
    "fileMatch": ["\\S+.bicep$"],
    "matchStrings": ["'br:mybicepsharedregistry\\
    .azurecr\\.io\\/bicep\\/module\\/storage:
    (?<currentValue>\\S*)'"],
    "datasourceTemplate": "github-tags",
    "depNameTemplate": "ErickSegaar/blessed-bicep-deploy"
  }
]
```

Figure 16. Renovate regex-manager configuration

To configure the use of the regex manager, we change the renovate.json and add a "regexManagers":

1. Configuration to match all *.bicep files. This configuration will limit this manager's configuration to only search in the .bicep files and not any other files you have set up.
2. Define your "matchStrings" regular expression. This configuration will search for the semantic versioning in the "br:****" annotation. A regular expression group 'currentValue' will contain the found version.
3. Configure the "datasourceTemplate" so the regexmanager can compare the 'currentValue' to GitHub-tags. GitHub-tags is a known data source[6] for renovate-bot.
4. Configure the GitHub repository to search for the tag in "depNameTemplate".

The result is an automatic Pull Request whenever a new blessed template publishes in a foreign repository.

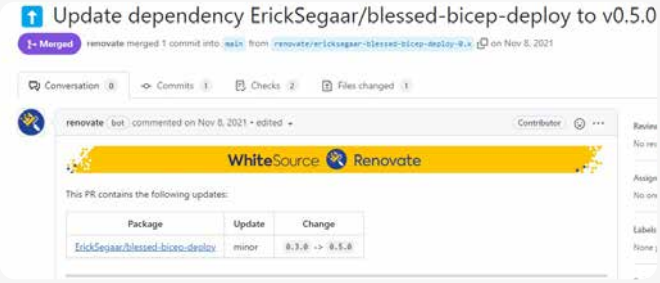


Figure 17. Renovate's Pull Request

Erick Segaar
Consultant

xpirit.com/erick

Conclusion

Many organizations already have blessed templates, and their success depends on the ease of use. The blessed templates should be beneficial for the organization to make sure that all consumers work in a secure and compliant manner. Consumers should rely on the service offered by the blessed templates to update their dependencies to maintain compliance automatically. Using this provided service should not be hard. It should be a golden path to take, enabling engineering capability instead of restricting it. </>

Take away

- You can keep using the existing approved blessed templates with Bicep
- You can make use of IntelliSense by using bicep-configuration
- Using bicep registries enables you to have compile-time validation (which is actually transpile time)
- Use Container Registry over Template Specs to have a more explicit transpiled ARM template
- Support all repositories with automated dependency updates on your blessed templates using Pull Request created by renovate-bot

[6] https://docs.renovatebot.com/modules/datasource/

Never stop learning – Thoughts after four years with our epic team

On New Year's Eve, I raised a glass (the last one for January) on four fantastic years at Xpirit. On a regular basis, I validate what I'm doing, both personal and professional. There's definitely going to be a five-year celebration. My job gets a solid 10 out of 10. Actually, the only situation that could make me hesitate would be the F1 Red Bull team offering me the same job. It's been an amazing journey, and since one of our values is "Sharing Knowledge", I feel the urge to finally contribute to our magazine some of the learnings that surprised me the most: my imposter syndrome, another perspective on IT for many organizations, and our BHAG (Big Hairy Audacious Goal – a goal you'll set to give direction, but probably won't ever actually make – 'Built to Last' from Jim Collins).

Author Immanuel Kranendonk

You could wonder why it took so long, especially since I'm the one helping new colleagues to overcome their imposter syndrome. Every new colleague is a bit overwhelmed by the Xpirit crew and what we've achieved so far. If you add a culture in which we continuously help each other by focusing on what we can still improve, that sometimes can become a bit scary. The suggestions come from a good heart, and everyone can make a mistake, but still. 'Out of our Xpirit-sight' I started coaching computer science students from a university on Agile and Growth Mindset. I even gave three guest presentations at the university of Utrecht, but never dared to do a dry run for the group. Asking for Xpirit-feedback during a dry-run is also one of the things I try to motivate new colleagues for.

Arjan and Rob, especially, helped me (without knowing) to take this step. Arjan kept asking me for an article for the magazine and was persistent. Rob motivated me by what he did: a gazillion blogs and presentations since he joined Xpirit, a month after I did. He described overcoming his fear in a recent blog post. Rob describes his journey in overcoming his imposter syndrome and he's doing a great job. In a few years, he has grown to become a respected source of information for many people on DevOps and GitHub. He gives real meaning to "Sharing Knowledge". And together with Rob, there are many colleagues that share our knowledge. This time, I'm finally joining them in our awesome magazine.

I guess the imposter syndrome naturally comes when you join a great team, and you don't have a big ego. It is uncomfortable to break the barriers of your own imprints, but sharing knowledge is incredibly motivating and gives an enormous amount of positive energy. And, you will probably learn something new in return. An open mindset helps to be receptive to the learning part. The open mindset should also help you ask for some help if you get stuck. Really, it's ok that you don't know everything.

Another thing that gave an inspiring epiphany was reading 'Seat at the Table' by Mark Schwartz. The book describes the C-level perspective on IT.

"Big Hairy Audacious Goal – a goal you'll set to give direction, but probably won't ever actually make – 'Built to Last'"

– Jim Collins

For most domains, for example Marketing, a budget should lead to a maximum result. Adding budget should hopefully provide more results. During many conversations with customers, I experienced what the book describes: a C-level looks at IT in the same way. We purchase solutions that should lead to a certain result or service. This is true, but only half of it. It is true, IT-solutions are most often purchased or built to achieve a certain goal. But as soon as it's in production, it must be maintained. Mandatory maintenance to mitigate new risks. In general, if you don't maintain software, it will get "bit rot": vulnerabilities or hacks that will jeopardize the stability and reliability of the solution. From a risk point of view, IT-budget should be addressed as budget to prevent catastrophes. You can spend a limitless budget on IT, but the question is the other way around: what risk are you willing to take, and how much budget is needed to mitigate the risks you're not willing to take. This is one of the reasons why IT budget may feel unpredictable at times; you never know what new threats arise.

The last topic I would like to address is our main theme for the upcoming period: Creating Engineering Cultures for our Customer. Here is my two cents on why and how C-Level should embrace this.

IT keeps innovating at incredible pace and most companies are highly dependent on IT. By now, you could argue most companies are mainly IT-driven. Changes used to take years and it was fine to take years to adjust. However, the current IT landscape changes daily, with regular updates in the cloud, and newly added open-source possibilities, but also with new vulnerabilities and threats. To keep up, your IT colleagues are like the true Avengers, continuously saving the world and your company from destruction. Yes, by spending time and money, but there is not only a downside to this; there is a great opportunity if your company wants to become a great IT employer. How? Here are some ingredients:

People First: everything starts with your colleagues. Give them trust and autonomy; they know what they are doing. If you take good care of them, they will take care of you, taking into account what's important for your company. Put people at the beginning of everything you do. Happy colleagues will lead to a happy business and happy customers.

To make sure your IT colleagues know what they are doing: facilitate continuous learning. IT won't stop evolving, and your people must stay on par with what's going on.

Attending conferences and meeting peers will help them do a better job. Yes, education and seminars cost money, but the learnings are priceless compared with what may happen if your company has a security breach.

As C-level you should understand and truly adopt an Agile mindset. Agility is mandatory if you want to keep up with the ever-changing IT world, but Agile is not an IT-trick (nor is it the solution to everything). C-level has to blend in: for instance, by celebrating successes but also showing an open mindset when mistakes are made. Learn and move forward. (don't create a procedure to prevent the same mistake.). Agile is a company way of working, not an IT one.

These are some of the aspects for empowering your IT-crew and creating and truly adopting an engineering culture that will help your company to attract and keep top talent. Follow Xpirit and my colleagues on LinkedIn to find out more about the topic.

Inspired by a lot of inspiring conversations and some recommended reading: Seat at the table / Time to Think / Getting Naked. </>

Immanuel Kranendonk
Chief Operational Officer

xpirit.com/immanuel

Xpirit as an IT Beehive

Xpirit is not your standard company. We do things our own way, which is sometimes different from what other companies do. It's not always easy to explain our mission and values, because what you see on the surface is not necessarily the same as what happens inside. In coming up with ways to tell our story, we found a lot of similarities between Xpirit and a beehive. Let us tell you our story as if we are a bee colony and find out how well the analogy can convey our view on work and our people.

Author Alex Thissen

It is all about the colony

Xpirit is a group that consists of wonderful people that believe in the adventure of running a people first company. They joined and started their journey within Xpirit. Some of them joined 7 years ago, others more recently, yet all hearing the same, unchanged story. We think our people define who we are. There is no Xpirit without the people. Nor would there be if we did not form the group as we have over the past years. The future of Xpirit is determined by the wellbeing of the group, which in turn is dependent on the participation and contributions of every individual.

The same interdependency of a group to its members and vice versa can be found in a bee colony. The colony is formed by all the bees that are part of it. An individual bee cannot survive without the group, and the colony cannot exist without the bees.

Different roles and all equally valuable

Each bee plays a unique role in the colony by the work they do. There are various types of bees in a colony, such as worker bees, drones, and queens. Most of the bees are worker bees. These take on different tasks during their lives, ranging from foraging for nectar and creating honey, building the hive, taking care of the queen and young drones, and nurturing the eggs and newly born bees. In turn, the drones make sure the colony stays healthy and grows, and the queens run the beehive and its colony by taking care of the bees and raising them. It is a symbiotic ecosystem inside the colony, where each bee, regardless of type or tasks, plays a crucial part and makes an essential contribution. Even though there are different types of bees, every single bee is equally important, even the queen. They all need each other to survive and flourish.

"We want to help you unleash your potential and "inner bee" by becoming part of our colony."

You can see the same at Xpirit. Each person chooses the role they want to play and performs tasks accordingly. Some choose to work more inside of the beehive and are less visible to the outside world. They work in teams or pairs for our customers, but also build inside of Xpirit. They show the direction of Xpirit by sharing their experience and knowledge and organizing social events and off-sites to grow as a group. They act as a voice in internal initiatives and help shape the future and direction of Xpirit.

Others find joy in being more public and share knowledge in meetups, conferences and with management of our customers. These people are more in the public eye, and therefore more visible to the outside world. However, not everyone does presales, is a speaker or a Microsoft MVP, nor is that expected of anyone. Not every bee goes outside foraging. Some are building the honey rates, creating honey or ventilating the hive to make a nice atmosphere and help the hive run smoothly. And again, all people and roles are equally important.

Hivemind

Bee colonies appear to act as a single organism, made up of all the individual lives and actions of the bees. They seem to have a shared mind, composed of all the unique minds and thoughts of individual bees. The shared mind, also known as hivemind, is about the mission and purpose of the colony. It implies a form of internal communication that is both spoken and unspoken to keep everyone in sync and aware of what is happening. It allows the colony to act in a purposeful manner, instead of performing uncoherent individual actions.

Xpirit also has a shared mindset. Everyone knows the mission of learning and growing together, and the values that help guide us. They have the same vision of building high-quality software using the Microsoft platform and helping customers achieve success by applying these ideas and insights. Our people have a combined body of knowledge and experience. Sharing that knowledge is in our genes. This way, everyone can benefit from one another and tap into more knowledge than any single person could ever manage to attain. Asking one person how modern software development is done, will give you the same consistent answer. It might be different, but is part of the same, bigger picture. Some are software engineers writing code, while others consult with management on processes and organizing teams to align with business requirements and architecture. Still others maintain stable and robust solutions in the cloud. Even though the answer might be specific to a certain phase or aspect of software development, the puzzle pieces of the answers you receive will fit together into a perfect picture.

Busy bees all together

With all the energy floating around, we are a bunch of busy bees. There are lots of individual actions that lead to many accomplishments. We organize meetups, bootcamps, write a magazine, spend evenings together in a leisurely fashion, brainstorm ideas, and make plans. We reflect on how we are doing and reorganize accordingly, do activities related to marketing and company or personal branding. Some of it is publicly visible, and some internal. Everyone is encouraged to venture and explore by themselves and do what they feel is good and beneficial for the group of Xpirit. No rules, just guidance and the freedom to make autonomous decisions: it keeps the energy flowing. And we have lots of energy to keep us all going and inspiring each other.

Interacting with our environment

Bees are well known for their contribution to the environment. They live in a permanent symbiosis with their surroundings, collecting nectar to survive and pollinating the flowers in the process. Young bees rarely venture out of the hive. It is at a later stage that they are harvesting nectar and discovering the landscape in search of new flowers and sources of food and materials. That is how we also bring experience to the outside world and share knowledge to cross-pollinate between companies, organizations, and communities. In the process, we are collecting newfound knowledge, ideas, and insights to share both internally and with the rest of the world. Some choose to do this in a more private setting with customers, others prefer presenting or blogging about it, or by coaching and advising within Xpirit, our hive.

More than meets the eye

Some aspects of a bee colony are plain to see, such as the harvesting bees on the outside, going out into the world, all very exposed and visible. But there is much more happening on the inside of the beehive. Working at Xpirit does not mean that you must be a very senior or experienced engineer, be a public speaker, or have a high rating on StackOverflow. Xpirit exists because everyone plays a role that is equally valuable, and many of those roles cannot be seen from the outside. There is plenty of opportunity to grow. We all share the same mindset, share our knowledge, and thrive together, because of the balanced ecosystem that we form together. Some of this has to be experienced in person. You are invited to come take a look at our beehive, the Xpirit headquarters, in Hilversum. Come talk to us so we can talk more about who we are and how we work. If you consider yourself a bee without many flying hours or a specific role, don't worry. We welcome everyone who wants to learn and grow. </>

Alex Thissen
Architecture and coding

xpirit.com/alex

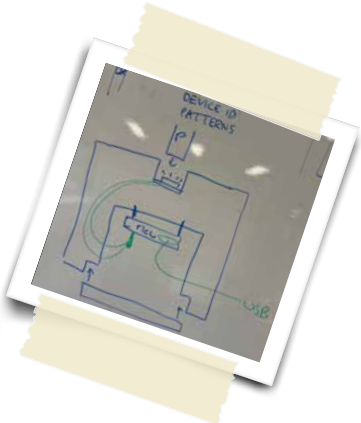
The epic story of Blinky

How an idea born at the lunch table turns into a handcrafted connected device, delivered on every colleague's doorstep, in only 3.5 weeks. The story of Blinky started just like many more typical Xpirit stories do. On Friday, December 3, while having a nice lunch with a bunch of Xpiriters, someone spoke the magical words: "we should do something creative for an end-of-year gift. Something crazy." Now, as you may know, we use a mantra at Xpirit: Do Epic Shit. It's what we do. And we love that so much, we have created a logo and t-shirts of it. How we came up with that mantra is a whole other story by the way. Very much worth reading, but let's focus on the story of Blinky.

Author Maarten Blok

While trying to evolve 'something crazy' into something achievable, we came up with the idea to create a Christmas version of our Do Epic Shit t-shirt with actual lights on it. That idea quickly evolved to a big button at the office, so if someone needed help, they could press the button and the lights on the t-shirts would blink. After discussing the possible dangers of electrocution, we decided that it might be best to do something crazy but safer.

However, we did like the idea of a button at the office that we could use to call for help and also the use of our Do Epic Shit mantra. And so, someone said: *"maybe we could create something, like a bat-signal, in the form of our Do Epic Shit logo!"* And there it was, the conception of Blinky. But what would it look like? What should the 'thing' do? And could we manufacture and deliver it before Christmas? The next few minutes, the wildest ideas flew over the table. We decided to think about it and involve our creative genius, Olaf Walther, from Studio OOM.



The first sketches

On December 6, the WhatsApp Group 'Epic Shit 2021' was created. We came up with the idea of creating a Christmas package with a Christmas edition of our t-shirt, but without electronics, and creating the Do Epic Shit logo in plexiglass with LED lighting. That would be our bat signal. On Friday the 10th, we brainstormed with Olaf on how to create all of this, and we came up with the sketches.

> "Maybe we could create something, like a bat-signal, in the form of our Do Epic Shit logo!"

Then the idea evolved to creating an XKea DIY kit with Swedish instructions. But, for the sake of quality, we decided to assemble the device ourselves at the office. We did, however, like the 'XKEA DIY' thing, so we thought of a Swedish name for the device and an XKea instruction manual.

The box in which the device would be delivered would also have to be epic, of course. So, what would the design of the box look like? Again, we discussed many ideas, from retro branding to Orwell-style and a superhero theme. The last one made the link to the bat signal quite nicely, so we decided to go with that.

Then the materials we would need to assemble the device. We required 70 MCUs, 1.5 meters Douglas Wood beams, 2m² 10 mm plexiglas, 10 meters Led, 70 customized boxes. And remember, we wanted to deliver the package before Christmas. We're talking December 13 now. Next, we asked ourselves what kind of bat signal we would create. The first was flashing violently for 10 seconds. Then we came up with the idea to send "epic shit" in morse-code as an easter egg.

On December 14, we all gave an update in the WhatsApp group about the project's status and what was left to do.

We were still in the process of digitalizing the drawings and creating the device code. Luckily, most of the materials were received, but not everything. At this point, we had two questions left: what kind of slogan would we engrave on the foot of the device, and how would we deliver the packages to our colleagues?

The machine that goes BLINK

Again, as things often go around here, many slogans were dropped in the chat. Some excellent examples: "Allo Allo Nighthawk, this is London Calling!" Or: "In case of epic emergency, this sign will flash!" But the next day, one of our brilliant Xpiriters came up with the absolute best: "The machine that goes blink." If you don't know why this is the best slogan, watch this Monty Python video: https://www.youtube.com/watch?v=wd9NQxIeAAc. After some discussions, we decided to go with this one. But we also wanted to name the device to use in our day-to-day conversations. We quickly came up with Blinky. And finally, on December 16, we had a name, a slogan, and a design for the box.

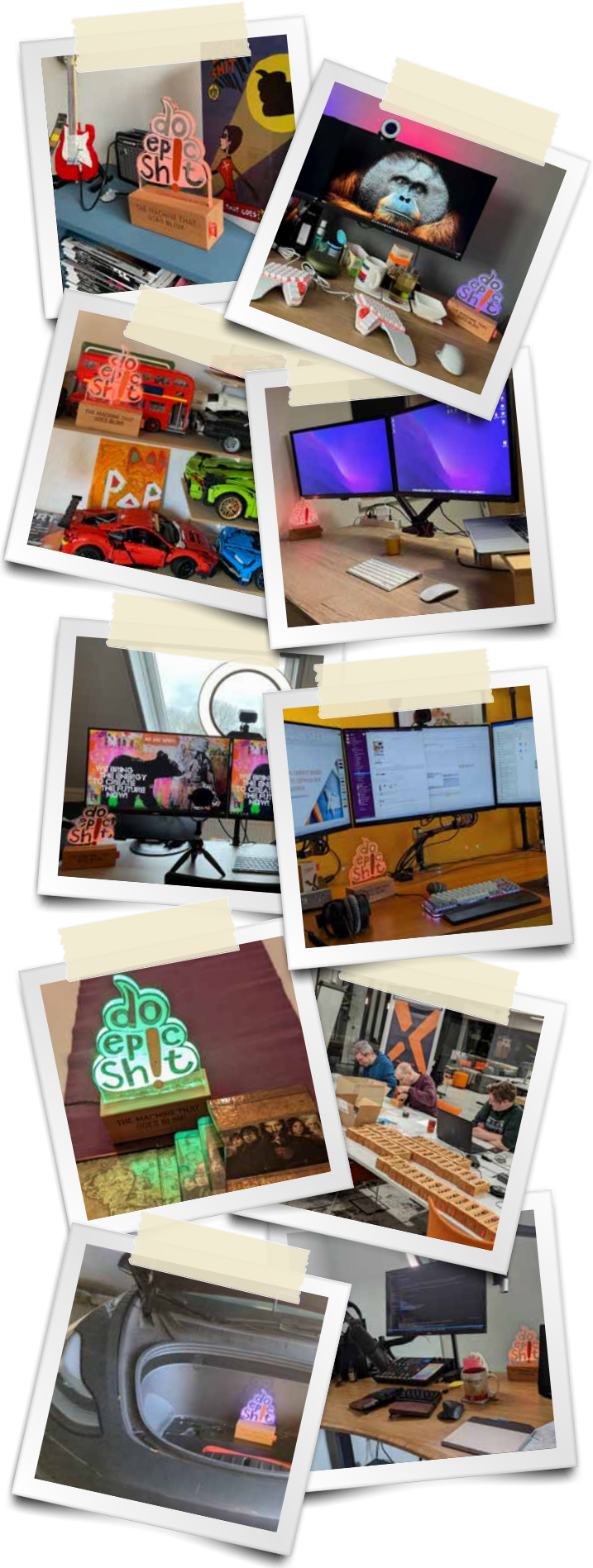
On December 17, we still had some challenges. We had not yet decided on how to deliver the packages. In-person delivery would be more personal, but it also would be quite a challenge logistically. On the other hand, how reliable would the courier services be around Christmas? in addition, all the materials had still not been delivered. We also didn't have a date to assemble all the devices, and finally, we had to write instructions so everyone could connect Blinky to their wifi. There was also some good news: although the morse code didn't work yet, the connectivity and colors per device did work.

Powered by log4j

Being a cloud consultancy company, we have some colleagues who are very keen on security. So, Marcel dropped a remark: "If we want to connect the devices to the personal wifi of our colleagues, some of them will for sure have questions about safety. How are we going to handle this?" That was a good question. We thought about a disclaimer, but then Matthijs came up with a genius idea: "Updates will be deployed remotely via log4j" (if you don't know about the log4j hack, read this blog post by Jesse Houwing https://jessehouwing.net/azure-devops-patch-for-log4j-vulnerability/). We all agreed this was genius, so that's how "powered by log4j" ended up on the box.

Assembling 70 Blinkies

Although we still didn't have the cables, on December 19, we agreed we would assemble the devices on December 22. There was a lot of work to be done before we could start building. During the night of Sunday, December 19 to Monday, December 20, around 3:14, Olaf showed us what his workbench looked like after many hours of milling (Thanks, Olaf!). But the job was almost done. On December 20, we decided to deliver the devices to our colleagues' doorstep by courier.



Then, on December 22, it was D-day. We wanted to assemble the devices at the office. Therefore, we had to wait until the evening so the office would be empty because we wanted to keep it a surprise. After some delicious pizza, we started assembling. We divided the tasks and started working. As you can imagine, the joy was great when the first Blinky morse signals blinked.

Around 11:30 PM, every device was assembled, and we carefully cleaned up the evidence.

Delivery and sending messages in morse

On December 27, the first Blinkies were delivered. After the first delivery, the Xpirit WhatsApp group exploded.
The colleagues who received it were very excited but didn't post any spoilers, other than that an epic gift was on the way. Every time a Blinky was received, there was an enthusiastic reaction in the group. And so, Chris changed the subject of the WhatsApp group to has *your package already been delivered*. After sending the first message (which was pretty simple, just 'Blinky') through morse-code, the group exploded again. Everyone was as busy figuring out how to decode the message. Most of the Xpiriters figured it out pretty quickly. During the next week, we sent a message each day like, 'take me places', 'feed me', 'I'm tired', and 'why am I here'. This resulted in some nice pictures that were dropped in the group.

The last scheduled message was sent on New Year's Eve: Happy new year!

That was the story of Blinky. In 3.5 weeks, an idea, born at the lunch table, evolved from a lit t-shirt to a connected device, all handcrafted by Xpiriters and, of course, Olaf. It's typical of Xpirit; that's how things often go. We start with an idea and take it a few levels higher. We want to thank every Xpiriter who has contributed to Blinky and Olaf Walther from Studio OOM, and we are looking forward to doing more Epic Shit together. </>

Maarten Blok
Marketing Manager

xpirit.com/maarten

Xpirit

Transforming your business will not work without the right knowledge

- **Certified Microsoft Azure Fundamentals** (AZ-900)
  Foundation – 2 days
- **Certified Microsoft Azure Administrator** (AZ-104)
  Professional · 4 days
- **Certified Microsoft Azure Developer** (AZ-204)
  Professional · 5 days
- **Certified Microsoft DevOps Engineer Expert** (AZ-400)
  Expert · 5 days

xpirit.com/training

The value of your development toolchain

The value and costs of a healthy development toolchain for maximized developer productivity. One of our key principles at Xpirit is "people first". This means people always come first in everything we do; it doesn't imply money and shareholders are equally important or that we have to find some kind of equilibrium. It means people always come first! And this applies not only to our employees – it also includes the people that work for our clients.

Author Michael Kaufmann

Because of our people-centric culture, we deeply care about developer productivity. In more than twenty years of experience working with developers, I've seen the impact it can have if they feel productive vs. unproductive. I've seen people leaving companies because they did not feel productive, and I have seen teams thrive after implementing DevOps tooling and an engineering culture. I've seen the joy that people felt when they felt productive after a long time of unproductiveness. It was Satya Nadella, the CEO of Microsoft, who once said he would always choose developer productivity over features for end users, as developer productivity benefits everyone and increases the feature delivery long term. This shows the importance of developer productivity to companies like Microsoft.

"If any engineer has to choose between working on a feature or working on developer productivity, always choose developer productivity."
– Satya Nadella

Figure 1. Developer velocity and the war of talent

The benefits of improved developer productivity

The question is: What will happen if your developers are not productive and have to work with old technologies and processes? The good developers will most likely leave. This will leave you with the ones that don't care or can't find another job easily. This will further reduce your entire productivity.

In the *war of talent*, when an engineer can find a new job very easily, a productive environment is crucial and a very important factor for people that are driven by intrinsic motivation – the talents you are looking for.

This wheel can spin in two directions: having a productive environment with a high developer velocity and satisfaction will help you attract and retain talent. This will accelerate your developer velocity and productivity and attract other good engineers. If your environment is unproductive, however, good people will leave, and the velocity will decrease further and keep other good engineers from joining your organization.

The increased productivity and increased quality in staff leads to better and more reliable software that gets shipped faster. This leads to faster feedback loops, ensuring you build what your customer wants. In the end, you achieve an increased customer satisfaction, which further helps you attract new talent.

In April 2020 McKinsey published their research study about the Developer Velocity Index (DVI) (*Srivastava S. & Trehan K. & Wagle D. & Wang J., 2020*). It's a study taken among 440 large organizations from 12 industries that considers 46 drivers across 13 capabilities. The study shows that the companies in the top quartile of the DVI

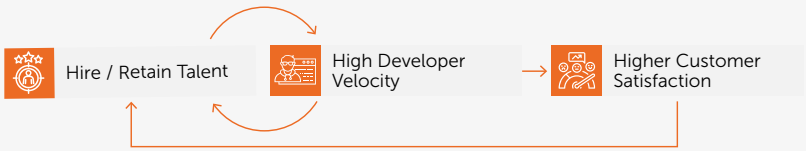


Figure 2. Acceleration and customer satisfaction

outperform other companies in their market by four to five times. Not only on overall business performance - companies in the top quartile score between 40% and 60% higher in:

- Innovation
- Customer satisfaction
- Brand perception
- Talent management

The findings align with the results from the DORA State of DevOps Report – but take them one step further by adding the business outcomes. The 2019 State of DevOps Report shows that elite performers, compared against the low performers, have:

- **Faster Value Delivery:** they have a 106 times faster lead time from commit to deploy.
- **Advanced Stability and Quality:** they recover 2,604 times faster from incidents and have a 7 times lower change failure rate.
- **Higher Throughput:** they perform 208 times more frequent code deployments.

High performance companies not only excel in throughput and stability, they also are more innovative, have a higher customer satisfaction, and a greater business performance.

With this research in mind and the priority companies like Microsoft and Google give developer productivity, it is obvious developer productivity should be a top priority for all companies that rely on software development.

The influence of the development toolchain on productivity

There are multiple factors that influence developer productivity. The most important ones, besides people, are:

- Culture
- Processes
- Toolchain

Having an engineering culture of trust and experimentation in which people are allowed to experiment and commit errors has a strong influence on developer productivity. But influencing or changing the culture is difficult. You can't just write some values on a PowerPoint slide and be done. The corporate culture is the result of the system, and you must change the entire system to change the culture. Nurturing a good culture should be an ongoing task, but it is not going to achieve results in the short term. This leaves processes and the toolchain. They go hand in hand. You can have the best tooling, but if your processes are slow and heavy, your developer productivity will be low. You can have great processes and be fully committed to DevOps principles, but if you don't have a toolchain to support the process, the productivity will still be low. Processes and the toolchain are the determining factors to increase the developer productivity in the short term.

Figure 3 - Benefits of improved developer productivity

Governance and processes

The problem with the development toolchain is its volatility. I've seen IT departments install DevOps tooling (such as build environments) and expect it to be like a mail system: Install once and just run it for years with few changes besides patches. But the amount of requests of developers to install and maintain new tools overwhelmed them. This normally leads to one of two scenarios:

- **The developer "wild west":** Developers are allowed to do everything. They have admin rights if necessary and are responsible for the tooling themselves. "The problem with this approach is often, that there is no alignment between the development teams. Each team uses the tools it wants and there is no maintained standard. This makes onboarding difficult and the allocation of teams to products inflexible: you cannot have teams work on other products or have developers switch teams without a longer onboarding period.
- **The IT castle:** The IT department provides a bare minimum of tooling and is the gate keeper to production. Requests from developers are mostly ignored or declined. This approach normally leads to shadow IT and build servers under the desks of developers.

This is, of course, an exaggeration and oversimplification. But I'm sure some readers will recognize their company in one of the two extremes.

A good development toolchain must have the right amount of governance. Give the teams the freedom to experiment – but have a common standard that is documented and provide training and guidance on that standard.

It is ok if one team wants to use Angular even if the current standard is React. There might be good reasons for it. But the decision should be explicit. Maintaining two UI frameworks and providing guidance when to use which one increases complexity and is therefore expensive.

A good governance process should look like this:

- Developers can request resources to experiment with new tooling at any time. In the cloud era this should be self-service and on demand.
- If the new tool is promising, there must be alignment with the other development teams and the team that provides the shared resources. Should the tool be added to the standard toolchain? Should it be supported? Should the tool be rejected, or allowed as a specific exception? In the end it will be a decision that affects all teams!
- If the tool is added to the standard toolchain, the documentation must be updated. There should be training for the other teams, as well as updated onboarding training.

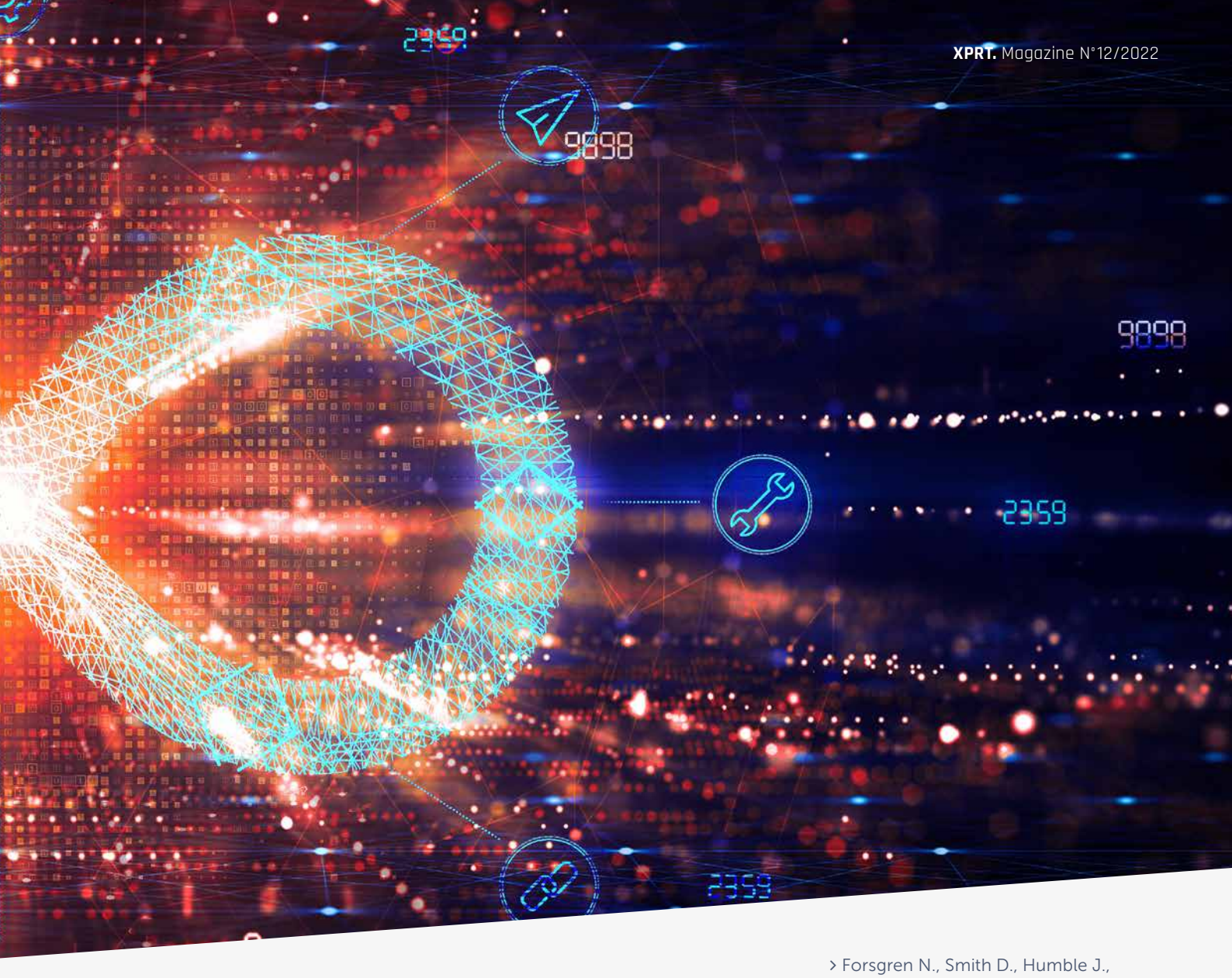
Figure 4 - Example governance process for toolchain

In-sourcing or outsourcing

Due to the high impact on developer velocity, and therefore business outcome, the first reaction normally is trying to in-source the toolchain and manage the entire process on your own. And I think that's valid, if you have the resources and knowledge to do it.

But – as I mentioned before – a development toolchain is not another "mail system" that you install and run. If you don't have the resources to manage it in a good way, it's better to have someone do it for you that is specialized in that area.

In the end, a good development toolchain will impact the performance of your business – but it will not impact your customers directly. If you change the company that provides the service, your customers probably wouldn't realize it.



That's why we offer the complete development toolchain as a managed service. We call it the Managed DevStack. We can host it in a regional data center to allow complete data residency. The offering includes the complete development stack – including build environments – and the governance process to manage changes. I think this is a good option if you don't have the resources or experience to host it yourself.

If you prefer to host your toolchain yourself, we can help in setting up the process, documentation, and trainings.

Conclusion

The impact of a good developer toolchain is enormous. It has a direct influence on developer productivity and is responsible for increases or decreases in your development velocity. It has a massive impact on innovation, customer satisfaction, brand perception, and talent acquisition.

But building and maintaining a healthy toolchain is not easy. It takes a lot of effort, experience, and a fine-tuned governance process. It's better to seek help – or utilize a good managed service – than provide a bad toolchain on your own. </>

Furter readings

- Srivastava S., Trehan K., Wagle D. & Wang J. (April 2020). *Developer Velocity: How software excellence fuels business performance*
- Forsgren N., Smith D., Humble J., Frazelle J. (2019). *DORA State of DevOps Report*
- Brown A., Stahnke M. & Kersten N. (2020). *2020 State of DevOps Report*
- Forsgren N., Humble, J., & Kim, G. (2018). *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations* (1st ed.) [E-book]. IT Revolution Press.

Michael Kaufmann
CEO / Managing Director

xpirit.com/michael

Customizing Codespaces

You've probably had this situation at least once on your career: you join a new team and it takes you at least 10 days to finally get the build to succeed on your local machine, the tests to pass, the application to launch without issues, and for the debugger to work. There's a document somewhere or in the projects wiki with a lot of steps, and the last person who walked through it did so 9 months ago. Situations like this cost precious time and are a big source of frustration.

Author Jesse Houwing

Some companies solve this by having you work on a Virtual Machine, either locally on Hyper-V or remotely in a datacentre or the cloud. This solves quite a few problems, but the cost is often prohibitive, and I've personally never liked having to work inside a remote desktop, often on a machine that was shared with others, while my own desktop has twice the power.

GitHub Codespaces provides a solution for many of these issues.

What is Codespaces

For most people Codespaces can be described as Visual Studio Code in the browser. Yet, it's much more. It's a cloud-based container platform for developers to run their complete development environment.

When you launch Codespaces from an enabled repository or organization, by default it launches a version of Visual Studio Code with all the latest developer tools pre-installed for just about every popular programming language. And developers can write their code, run their tests, and even run and debug their application, inside the browser!

And while Codespaces runs in the cloud, your editor "runs" inside of your browser or inside of a local instance of Visual Studio Code.

Code Spaces are hosted in Azure and Visual Studio Code uses Remote Containers[1] to connect. All changes made to the Codespace's filesystem are automatically captured.

Even when your Codespace is paused, it will resume right where you left off.

Interesting use-cases

In the past few months, we have used Codespaces to deliver online interactive workshops where participants could get started with new technology and tools they had never used before without installing anything to their local laptops. This greatly simplified the preparations for the workshop and completely took away the need for pre-provisioned workstations.

We've configured Codespaces for internal projects so that all it takes for a developer to contribute to the project is to start the Codespace and wait a few seconds for the Codespace to start. From this point forward, they can change the code, run the tests, and run a local instance without having to configure anything locally and without any interference with any of their ongoing projects.

I'm personally considering adding a Codespace configuration to most of my open-source projects to make it much easier for people to contribute.

We plan to leverage Codespaces for the upcoming Global DevOps Bootcamp[2] so that every participant has access to a fast and pre-configured IDE in the cloud regardless of their own hardware and circumstances, hopefully enabling many more people to participate in the event.

[1] Developing inside a Container using Visual Studio Code Remote Development
[2] https://globaldevopsbootcamp.com/

Getting Started

To start using Codespaces, you don't need to know how to create your own image. There is a large list of starter containers available, and the default container has tools for just about every popular programming language pre-installed. Just click the "New Codespace" button in your repository to open the repository in a new instance of Visual Studio Code inside your browser.

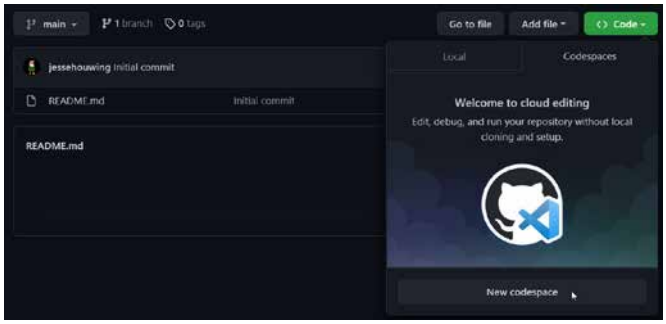


Figure 1. Creating a new Codespace

While the default image is convenient, it's also a bit big and probably has many tools installed you're unlikely to ever use. To pick one of the other available images, choose the "Add Development Container Configuration Files..." from the command palette.

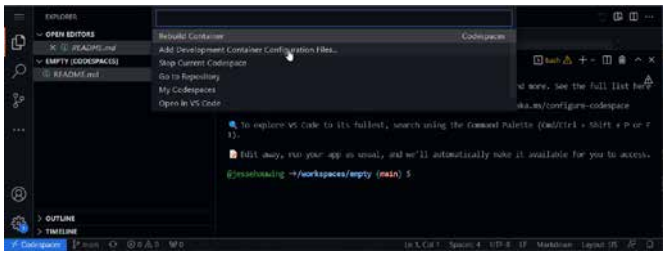


Figure 2. Add Development Container Configuration Files...

And choose the container image you want to use. When in doubt, pick the "GitHub Codespaces (Default)". A complete overview of all the images and what's installed on them can be found on GitHub[3].

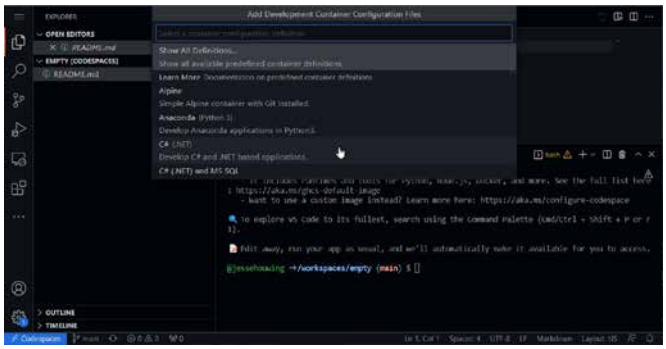
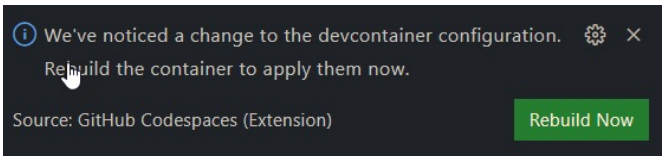


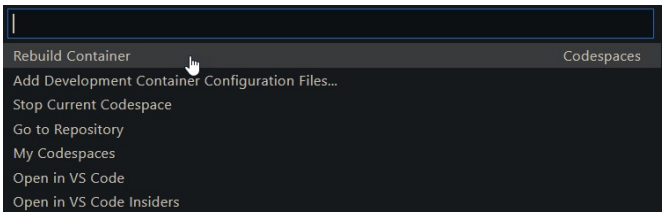
Figure 3. Choose the container image matching your environment

Visual Studio Code will add several files to your repository and then prompts you to rebuild the Codespace:

Note: If you've missed the prompt, you can always manually trigger a rebuild from the command palette (**Ctrl+Shift+P**).



This can also be useful when you want to make multiple changes and then rebuild the Codespace.



You'll see a new folder in your repository containing these new files: `.devcontainer/devcontainer.json` and `.devcontainer/DockerFile`. These files are used to store most of the settings of your Codespace.

Anatomy of a Codespace

The configuration of your Codespace is stored in several places. You've already seen the first two in the `.devcontainer` folder. But there are more. Let's go over them to see what they are:

.devcontainer/devcontainer.json

The `devcontainer.json` is the main configuration file for your Codespace. It contains environment variables, extensions, docker volume mounts and a few other settings. It also points to the container image used to run your development container. The default points to the DockerFile in the same directory, but you can also reference any image from a docker repository of your choice.

The `devcontainer.json` can also be used to run one of more commands after visual studio code has launched, at this point your git repository contents will also be available.

.devcontainer/DockerFile

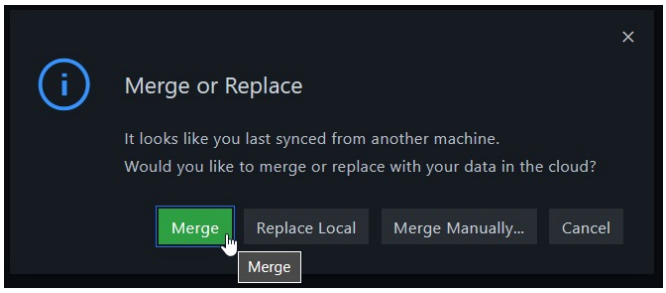
The `DockerFile` is used to select the base image and to optionally install additional tools into your container. By default, it's a simple pointer to the image you selected when you had Visual Studio Code add the Development Container Configuration Files to your repository.

Your GitHub profile

Additional settings, such as themes, keyboard bindings, snippets and globally installed extensions can be synced with your GitHub profile into your Codespace by turning on Settings Sync[4].

[3] vscode-dev-containers/containers at main · microsoft/vscode-dev-containers · GitHub
[4] https://code.visualstudio.com/docs/editor/settings-sync

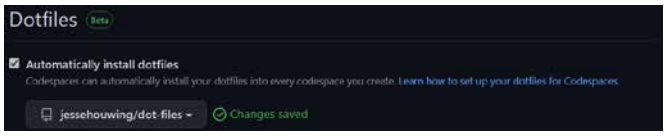
Codespaces will ask what settings to synchronize and will ask what to do in case there are conflicting settings:



Some customizations, like keyboard bindings, can only be configured through Settings Sync or through Visual Studio Code extensions.

Your dotfiles repository

In your personal GitHub settings, you can configure a repository containing your Linux dotfiles[5]. These can be used to configure your default shell, your preferred editor, and many other settings of your Linux user profile.



Codespaces Secrets

You may need to access other resources from your Codespace, such as a GitHub Container Registry, Cloud resources etc. To prevent accidentally committing these secrets to your repository it's recommended to **not store these credentials on the filesystem**. Instead, store them in Codespaces Secrets. When a Codespace starts, there secrets are made available as environment variables.

Secrets can be stored on multiple levels:
› Repository (most specific)
› User Settings
› Organization Settings (least specific)

The most specific level will be used by your Codespace.
Note: Whenever a secret is updated, you must rebuild your Codespace for these changes to take effect. Unfortunately, there is no indication this is required from inside your Codespace.

Note: You can't store secrets with a key that stats with GITHUB_. Which is unfortunate since some tools expect that. In that case you'll need to copy the value from a different name to the reserved name after the Codespace has started.

There are special secrets to allow access to private docker repositories[6]. These must be named:
› ***_CONTAINER_REGISTRY_PASSWORD
› ***_CONTAINER_REGISTRY_SERVER
› ***_CONTAINER_REGISTRY_USERNAME

Where *** is a custom label to identify the container registry.

Common scenario's

The most common reason to need to customize your own Codespace, is probably the need to install additional tools that are required for your development process or changing the set of installed extensions.Every time you make changes, you can immediately test them by rebuilding your Codespace. When you are satisfied with your changes, commit your changes to the repository to persist them and to share them with the world.

Installing additional tools

While the default Codespace container has many things installed, you may need to add something extra to it. Either a custom-built tool, or something that requires a license to run.

You can add these by editing the `DockerFile` in the `.devcontainer` folder.

```
FROM mcr.microsoft.com/vscode/devcontainers/
universal:1-focal

USER root
RUN apt-get update

USER Codespace
RUN az extension add --name azure-devops
```

You can run commands at the container lever (`USER root`) or at the user level (`USER Codespace`).

Adding extensions

The list of extensions to install is stored in the .devcontainer.json. You can manually add extensions to the list and then rebuild the your Codespace.

```
38    "extensions": [
39        "GitHub.vscode-pull-request-github",
40        "MS-vsliveshare.vsliveshare",
41        "ms-vscode.azure-account",
42        "ms-vscode.azurecli",
43        "ms-azure-devops.azure-pipelines",
44        "ms-azuretools.vscode-docker",
45        "ms-vscode.powershell",
46        "redhat.vscode-yaml",
47        "cschleiden.vscode-github-actions",
48        "github.remotehub"
49    ],
```

Figure 4. Manually add an extension to the devcontainer.json

[5] https://docs.github.com/en/codespaces/customizing-your-codespace/personalizing-codespaces-for-your-account#dotfiles
[6] https://docs.github.com/en/codespaces/codespaces-reference/allowing-your-codespace-to-access-a-private-image-registry

But there is an easier way to achieve this. When you're inside your Codespace, you can add the extension from the Extensions Marketplace:

Figure 5. Add an extension through the Extension Marketplace

Find the extension you need, then add it to the `.devcontainer.json` from the ⚙ cogwheel menu.

Caching containers inside the Codespace

One of the great advantages of Codespaces is that you can get started on a project quickly with the click of a button. Once the Codespace has started, you can pull additional images, so they're cached locally:

```
{
    "postCreateCommand": "docker pull ghcr.io/
    jessehouwing/mycustom-cli:latest && ... "
}
```

To pull the image from a private repository add the previously mentioned `***_CONTAINER_REGISTRY` secrets.

Storing the Codespace container in GitHub Container Registry

It may not be desirable to build your container from scratch each time it's started up and you may not want to store the container in a publicly accessible location. In that case you can store your container in GitHub Container Registry and grant access to Codespaces.

First build and tag your container image:

```
> docker build .
[+] Building 0.2s (6/6) FINISHED
 => [internal] load build definition from Dockerfile  0.1s
 => => transferring dockerfile: 162B                  0.0s
 => [internal] load .dockerignore                     0.1s
 => => transferring context: 2B                       0.0s
 => [internal] load metadata for mcr.microsoft.com/
    vscode/devcontainers/universal:1-linux            0.0s
 => [1/2] FROM mcr.microsoft.com/vscode/devcontainers/
    universal:1-linux                                 0.0s
 => CACHED [2/2] RUN az extension add --name
    azure-devops0.0s
 => exporting to image                                0.1s
 => => exporting layers                               0.0s
 => => writing image sha256:aa1d12f58610a60d4ee53b
      7dfc06b2b5a9581f5e26de19931deb61c3b66b120f      0.0s
```

Tag and publish the image to GitHub Container Registry:

```
> docker tag aa1d12f58610a60d4ee53b7dfc06b2b5a9581f5e
  26de19931deb61c3b66b120f ghcr.io/jessehouwing/
  Codespaces-demo:latest
> docker push ghcr.io/jessehouwing/Codespaces-demo:latest
The push refers to repository [ghcr.io/jessehouwing/
Codespaces-demo]
......
latest: digest: sha256:d928fbe90f267882d4d4de4194015e
ef06f5c88a045f3d9d4334aae0ea104612 size: 4538
```

Then navigate to the package settings for the container image you just pushed and grant access to GitHub Codespaces:

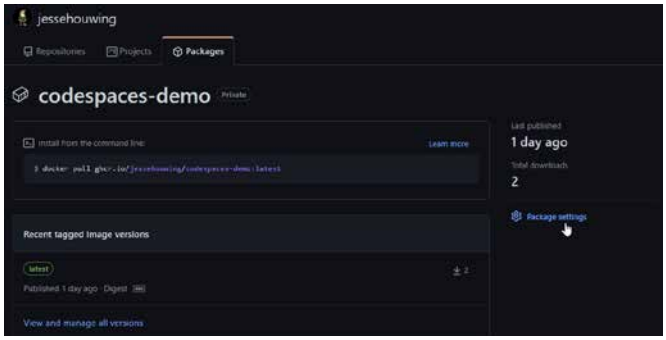


Figure 6 Find the newly published Codespace container and open the Package Settings

Grant the repository you want to launch this Codespace image from access to this package:

Figure 7. Manage Codespace access to add the repository

Now update the `DockerFile` in the repository to use this image:

```
FROM ghcr.io/jessehouwing/Codespaces-demo:latest
```

And rebuild your Codespace.

Beyond extending the base image

Your requirements for the Codespace image may go beyond the standard images, maybe you need a different Linux distro, standard libraries, a specific kernel version etc. In that case you can also build a Codespace from scratch. A nice getting started point could be to take the Codespaces default container[7] and either re-use the elements you need or use them as inspiration for your own image.

To start customizing the image copy the contents of the `.devcontainer` folder of one of the standard images and replace the `DockerFile` with the `base.DockerFile`. You'll find all the scripts used to install the different toolsets in the `library-scripts` folder.

Either commit the `.devcontainer` folder and its contents directly to your repository or build the container and publish it to a container registry as described above.

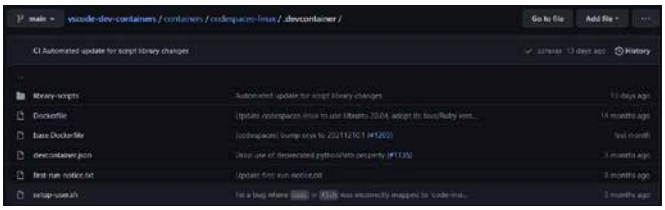


Figure 8. Take the full contents of a Codespace image to customize it ever further

Summarizing

Codespaces enables people teams worldwide to contribute to GitHub. It drastically reduces the time needed for anyone to open a project and contribute their changes.

Even when the standard options won't fulfill your needs, it's easy to extend and change what is installed and updates can be rolled out to your team effortlessly. </>

Jesse Houwing
Trainer, coach, tinkerer

xpirit.com/jesse

[7] https://github.com/microsoft/vscode-dev-containers/tree/main/containers/codespaces-linux/.devcontainer

Preparing for a security assessment

Your team has been working on a web application for several sprints. They are now preparing for the first official release to the production environment, where the end-users will start registering accounts and interacting with their data. The user stories are implemented, tested, and ready to be deployed. There's just one last hurdle to overcome: the security assessment.

Author Wesley Cabus

A team of security experts will review your web application from top to bottom, trying to abuse the application to gain access to data they're not supposed to see or - even worse - to the filesystem where the web application is being hosted. But don't panic! Let's walk through some steps your team can – and should – take to prepare for the security assessment.

Gather information about your system

The first step is to know everything about your web application, the actors, and the infrastructure it's using or interfacing with. This means you'll probably need to involve additional people from various divisions to answer all the questions: the development team, infrastructure, DevOps, architects, and someone from the security reviewers.

Here are some questions to get you started:

- Which frameworks are we using in our code (.NET, Java, React, Xamarin, ...) and which dependencies do we have (NuGet, Maven, NPM, ...)? Are we using the latest versions where possible? Do we know why some versions have to stay behind? Are there alternatives?
- How are our applications being configured at runtime? In the case of static files, are these accessible from the outside world? Are secrets being protected during the deployment process or are they visible at any time?
- Which services are we interacting with, both internally and externally? How are their interactions protected? Think about credentials, certificates, firewall rules, explicitly allowed IP address ranges, ...
- Where are our web application components being hosted and how is the environment configured? Are we using a virtual network, are web servers configured to use the latest TLS version or not? Do we have additional protection in place around our components, like a firewall and/or API management layer?
- If your application has multiple roles/rights per user or action, do you have a clear overview per resource, action, and user type or role in which actions should be allowed or rejected?

Create a threat model

After gathering a lot of information, it's time to visualize the system and highlight interactions between all components. There are several threat modeling tools available to assist you in this process. One of the oldest tools is the Microsoft Threat Modeling Tool[1], which focuses on the Microsoft technology stack and is freely available as a Windows client at.

Whichever threat modeling tool you end up using, they all have in common making it easier to raise potential vulnerabilities or risks for every interaction in the model. For example, if you add an interaction between a web server and a SQL database, one of the identified risks is: "An adversary can gain unauthorized access to the SQL database due to weak network security configuration." In Azure, this could mean that you've configured the SQL database to allow access from all networks or even from all Azure

[1] https://www.microsoft.com/en-us/securityengineering/sdl/threatmodeling

services. That's a risk as this option also includes Azure services managed by other Azure customers.

In the following screenshot, you can see an example threat model for a web application which communicates directly with a SQL server instance. Azure Traffic Manager is placed in front of the Azure App Service to redirect traffic coming from a user's browser to multiple instances of the App Service. For the sake of brevity, these multiple instances have been left out of the diagram. Between every component or actor in the diagram, request and response flows have been added to show how the components interact with each other:

The threat model report, as shown above, highlights the request between the TodoItems MVC app service and the TodoItemsDb SQL database and has identified a potential risk:
An adversary can gain unauthorized access to Azure SQL database due to weak account policy.

Every identified risk should be sized and discussed by the assessment team to decide on the next steps. When sizing a risk, think about the impact an exploited risk has, for example, using the DREAD model:

- Damage: how bad would the exploit be?
- Reproducibility: how easy is it to reproduce the exploit?

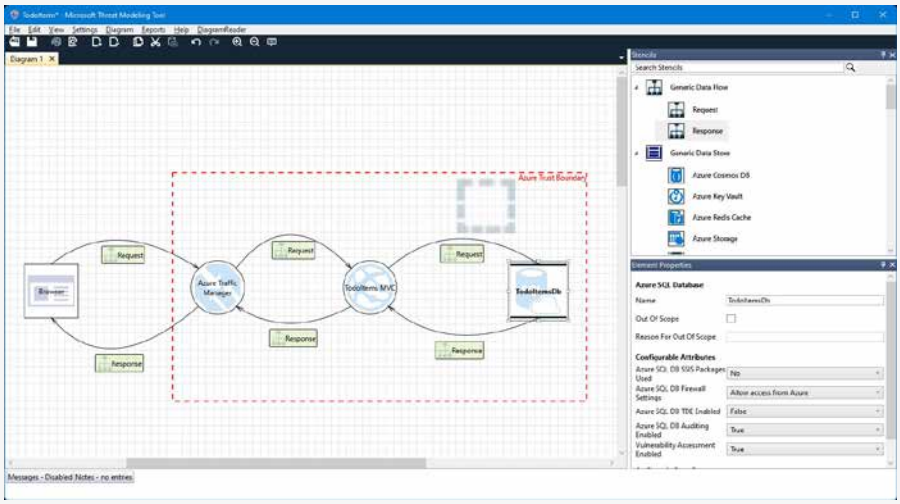
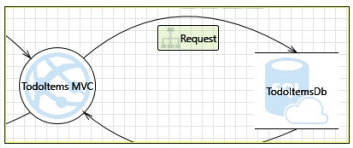


Figure 1. Designing a threat model

After designing the threat model, you can generate a report using Microsoft's Threat Modeling Tool. This report will analyze all drawn interactions and, depending on the configured attributes, will create a list of potential risks per interaction or component, as shown in the screenshot below.

- Exploitability: how much work is it to execute the exploit, to make it work?
- Affected users: how many people would be impacted?
- Discoverability: how easy is it to discover the threat?

Interaction: Request



1. An adversary can gain unauthorized access to Azure SQL database due to weak account policy [State: Not Started] [Priority: High]

Category:	Elevation of Privileges
Description:	Due to poorly configured account policies, adversary can launch brute force attacks on TodoItemsDb
Justification:	<no mitigation provided>
Possible Mitigation(s):	When possible use Azure Active Directory Authentication for connecting to SQL Database. Refer: th10c">https://aka.ms/tmt-th10c
SDL Phase:	Implementation

Figure 2. Example of an identified risk

The last letter of DREAD, discoverability, needs to be treated with caution however: it's easy to dismiss a risk as being difficult to discover, for example, when configuration files containing secrets can't be discovered because the web server doesn't list directory contents. But when an attacker knows your application uses a specific framework which expects a file called "appsecrets.xml" to be present at the path "/config" and they can still access that file, that could potentially turn a low risk into a very big issue.

You have gathered information, built a model, and sized the risks. But before letting the security reviewers examine your new web application, why not go the extra mile and check if there aren't some points you can improve already?

Hardening the application

In our example diagram, we have a SQL Server and Azure App Service running within an Azure trust boundary network: all services within this boundary are freely able to communicate with each other. This does not however limit access to our specific resource group, but also allows any other Azure-hosted service to attempt and access the resources from our application.

To prevent unwanted access, we can configure a virtual network in Azure. In this virtual network, we add the App Service and SQL database, and only allow Azure Traffic Manager to access the App Service from outside the virtual network. This ensures that only the Azure App Service instances can access the SQL database.

HTTPS all the things!

Let's start with the most basic rule that everyone should be applying to their web applications from day one: use HTTPS everywhere, even during development. If it means creating a self-signed local certificate for your local development environment, so be it, but use HTTPS from the get-go.

This includes containerized applications as well: please refrain from only protecting the external endpoints and then switching to HTTP traffic inside

the container network! If you focus only on the outside layer being protected, then the chances are high that this methodology will make its way into the production environment. And if someone with malicious intent manages to get their own container deployed into your network, then it's game over. But to be honest, if that would happen, there are probably other systems that need your immediate attention as well.

Add or remove HTTP headers

Every web application should respond with the correct set of HTTP headers.

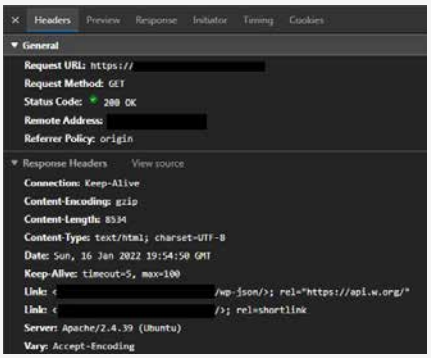


Figure 3. Example HTTP request, showing the response headers

However, most web servers and application frameworks add some information in these HTTP headers that immediately divulge what server or framework is running on that system. Combining this data could be enough for someone to launch a specific attack just by looking up some CVE (Common Vulnerability and Exposure).

When visiting a website, this was the response my browser received, which tells me three important things:

- The web server is running Apache version 2.4.39, as the Server HTTP header tells us. At this moment, the latest release is 2.4.52, and the most recent update includes two fixes for CVE's.
- The web server runs Ubuntu Linux, also divulged by the same Server HTTP header. While we don't get a version number, the combination of Apache and Ubuntu could lead us to specific vulnerabilities.
- The website uses WordPress. This is a bit trickier to find out, but if you look at the first Link header closely, you'll see wp-json and a reference[2]. While this or any other header doesn't give us the version number, that's easily located in the page's source. In this case, the website is using WordPress version 5.2.3, while 5.8.3 is the current latest release.

With only two HTTP headers, which seem completely harmless, and a bit of digging into the returned web page, we've received enough information to start searching for potential ways to gain control over this website or even the server. If this web server had been configured not to send the Server HTTP header, my attack vector would have been reduced significantly.

[2] https://api.w.org

Some application frameworks tend to add their own HTTP header, X-Powered-By, to announce that a framework is being used, often including a version number as well, for example:

```
X-Powered-By: PHP/5.4.0
```

This information is a goldmine for hackers! Do yourself a favor and remove this header from every HTTP response as well.

Other HTTP headers, however, are worth adding to your application to make your web applications more secure:

- **Strict-Transport-Security:** this header tells the browser that your website will always support HTTPS. You can also preload this information by announcing your website[3]. This allows browsers to immediately redirect visitors to HTTPS, even if they've never visited your website before.
- **Content-Security-Policy:** in short, this header is used to specify which sources are allowed or denied to be used by the web application. Think about CSS styles, fonts, JavaScript, but also AJAX requests, hosting framed content or being hosted as a frame, etc.
  However, setting up a good CSP[4] (Content-Security-Policy) header[5] is a daunting task and will most likely also require rewriting parts of the application if it's heavily using JavaScript. Whatever you do, do not take the easy route and specify "unsafe-inline" for the script-src directive because that would open up your web application to any JavaScript to run.

Is your cookie jar secure?

Server-side web applications use cookies to store if a visitor is authenticated, to keep track of their shopping cart, language preference and many other things. But are your cookies properly secured?

Cookies should be marked **HttpOnly** and Secure, and should have the correct SameSite policy applied to them:

- **HttpOnly** prevents cookies from being used in JavaScript.
  A language preference cookie doesn't need HttpOnly to be set, so that a script could properly format a date value using the correct locale. But JavaScript should never have access to authentication cookie values, so these cookies should definitely be marked as HttpOnly.
- **Secure** indicates that the cookie will only be sent over HTTPS connections.
- **SameSite**[6] defines or limits how your cookies can flow. The most strict setting limits cookies to only be sent when the user is interacting within the website, while the least secure setting will allow cookies to also be sent when clicking on a link from within an email, for example.

And, just as with the **Server** and **X-Powered-By** HTTP headers, some cookies names include a clear indication of the framework being used by the web application. For example, when using cookie-based authentication in ASP.NET Core, you'll see a cookie appearing with the name ".AspNetCore.Cookies". It's very easy to reconfigure these cookies and rename them, which is a quick win to make it less obvious to potential attackers what framework your web application is built with.

And many more...

There are still a lot of application hardening steps you can take depending on your architecture and the functionality of the web application:

- Parsing XML can potentially load external data during the parsing process;
- Uploaded files could contain viruses, or uploads could attempt to overwrite data from other users or the system;
- Each data store type has a form of injection attacks, so make sure you're correctly parameterizing your queries;
- Is your containerized application using a well-known base image or one you found somewhere randomly? Are you applying patches yourself? Then keep them up-to-date as well.

Conclusion

By gathering knowledge about your system and creating a threat model, it can become clear where to focus your efforts as a team to make your web application more secure, even before the actual security assessment starts. And if you go the extra mile and already implement some additional fixes, then the security team will most likely be happily surprised and challenged to test even more thoroughly.

In any case, you're going to be better prepared to face the security assessment, and together, you'll be able to deliver a more secure solution. </>

Wesley Cabus
Coding Architect

xpirit.com/wesley

[3] https://hstspreload.org
[4] https://content-security-policy.com/
[5] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy
[6] https://web.dev/samesite-cookies-explained/

Embrace chaos to achieve stability

Imagine this. You have built a website to sell your company's products. After a few months of hard labor, the application finally goes live. Of course, the application has been thoroughly tested. It all started with Unit Tests. First on the local machine and after the engineers filed a Pull Request a whole series of checks were executed. Each quality gate passed successfully and a fully automated pipeline successfully deployed the application on a cloud environment. But you went that extra mile. Performing load tests, security tests, pen tests and smoke tests. And finally, to make sure you have the least downtime possible when the sh*t hits the fan, you created and implemented failover scenarios and disaster recovery plans. Now you are all set. Let the sales begin!

Author René van Osnabrugge

And then... everything goes black. The datacenter is down, and the failover you carefully set up does not work as expected. And, after a few hours of stress, when the data-center has recovered, the way back does not go well.

This scenario is not something that only exists in a fantasy world. It is a real scenario. Things happen and you need to be prepared. And the truth is, you cannot prepare for everything. When you operate a business in the cloud (but also in your own datacenters) you need to embrace the fact that things can go wrong. The question is, how well can you deal with it.

Chaos Engineering

When Netflix moved to the cloud in 2011, they wanted to address the fact they lacked sufficient resiliency tests in production. To make sure they were prepared for unexpected failures in production, they created a tool called Chaos Monkey. This tool caused outages and breakdowns on random servers. By testing these "unexpected" scenarios they could validate and learn if their infrastructure could deal with, and recover from, failure in an elegant manner. Without meaning to, Netflix introduced a whole new practice. Chaos Engineering.

Breaking servers was one way to test this, but quickly other scenarios became relevant. Slow networks, unreliable messaging, corrupt data etc. Not much later, other tech companies, especially those running large scale and complex landscapes in the cloud, also adopted similar practices. This practice, where the mindset shifts from expecting stable production systems to expecting chaos in production, is called Chaos Engineering.

Chaos Engineering is a concept that uses hypotheses and experiments to validate the expected behavior of complex systems. This way you can grow confidence in the reliability and resilience of these systems.

Why Chaos Engineering?

Chaos Engineering lets you compare what you think will happen to what actually happens in your systems. You literally "break stuff " to learn how to build more resilient systems. Therefore you can look at Chaos Engineering as a test practice. But there are important differences. First of all Chaos Engineering, when done right, is also performed on production systems. Secondly, with Chaos Engineering you don't really test for failure. You test beforehand, and by conducting Chaos experiments you try to prove the assumptions you made in your test scenarios and architecture are actually valid and working.

With the rising complexity of our infrastructure, due to s oftware architectures like microservices, but also the "connected" systems we build nowadays, the traditional QA approach is not sufficient anymore. There is simply too much that can go wrong, and with the dynamic nature of the software and infrastructure stack this can be different every day. With Chaos Engineering it all starts with a hypothesis. And based on the hypothesis, you define and conduct experiments to prove that your hypotheses is correct.

Here is an example hypothesis, "When the external payment provider I use is unavailable, my customers get the option to pay afterwards and continue their checkout process".

The 5 principles of Chaos engineering

To get started with chaos engineering you can use the following simple plan. I will explain these steps in detail in the rest of this article.

Before we get started you should understand that Chaos Engineering is not something you can do on a rainy Sunday afternoon. Chaos Engineering needs careful planning and impact analysis. You need to understand what happens if your hypothesis is wrong. You also need to understand "the blast radius". In other words, what breaks if things do not work out as you planned. And linked to that, are there people available during the execution of the experiment, so they can jump in when things go not as planned?

The website [Principles of Chaos Engineering][1] describes 5 principles you should consider when doing Chaos Engineering:

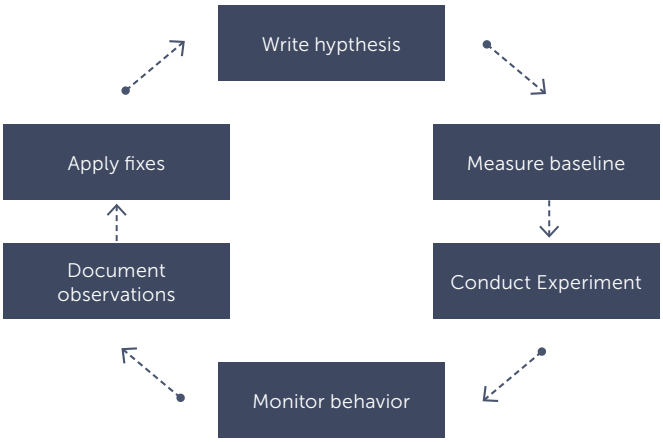
1. **Build a hypothesis around steady state behavior**
   This means you should focus on what is visible for the customer. Not the internal working of a system or things you can only influence when you know the inner workings. Focus on the steady state and the metrics that belong to a steady state.
2. **Vary real world events**
   Prioritize events based on expected frequency. Consider everything that can influence the system steady state. For example, disk failure, servers dying, or network outages.
3. **Run experiments in production**
   Simulation and sampling is great, but running on real world data and metrics is better. Try to run on production whenever possible. Of course, this requires careful planning and involvement of people. Usually this is done on so-called "game days", where people are ready for the "game". When you start with Chaos Engineering, it might be a better idea to validate your hypotheses on non-production systems. Start there, to get an idea what to expect and what you should measure. Production introduces an extra level of complexity and control because you need to make sure your users are not impacted.
4. **Automate experiments to run continuously**
   As with almost everything in DevOps, automation is key. Running experiments and gathering metrics is time intensive and hard work. Make sure you automate experiments so you can run them repeatedly.
5. **Minimize blast radius**
   Experimenting in production has the potential to cause unnecessary customer pain. So be mindful of that. Make sure there is room in your error budget or prepare for some issues. There must be an allowance for negative impact but keep the fallout of experiments minimal.

How does it work?

Chaos engineering involves going through a number of steps. These steps are followed for each new experiment. As I described before, it is important to plan this carefully. Because many of the chaos experiments are executed on production systems, you can easily break things that have customer impact. Often companies choose to organize so-called Game days. On these days people know that chaos experiments will be executed and can be on standby or be extra careful to monitor the systems for strange behavior.

[1] PRINCIPLES OF CHAOS ENGINEERING - Principles of chaos engineering

When running chaos experiments you can follow this structure:



Write a hypothesis
With chaos engineering, it starts with a hypothesis. This is important! It is not a test. For example, the hypothesis "The payment service should respond" is not a valid hypothesis. This is something you should already have tackled in your test suite. Chaos Engineering is about making sure your application becomes more resilient. You should already be quite certain your system can deal with unknown situations and your hypothesis should build on that. For example. "When the payment service goes down, we offer our customers an alternative way of payment". Think about the user. How can the user continue its journey with the least impact . A good example that Netflix uses when the login functionality stops working, is they offer services for free, without logging in. That way, users can still utilize the service.

Measure baseline behavior
Before you run any experiment, you should be aware of the baseline behavior. How does your system normally respond? In other words, can you recognize anomalies? You should have a good idea of the baseline because otherwise you may draw the wrong conclusions. For example, if you run an experiment to prove your response times will stay the same as "usual", you should know what usual is. Maybe this varies throughout the day due to traffic on your site. If you run an experiment in that timeframe, you might see strange things that are caused by factors other than your experiment.

When you think about creating the baseline, you should think of metrics and user metrics that are important to look at in the light of the experiment and hypothesis you are working on. Not everything is relevant at the same time.

Conduct experiment
When you created the hypothesis and baseline, you can start running an experiment. Running an experiment is causing the behavior that could disproof your hypothesis. Slowing down traffic, bringing a service down, shutting down or killing containers etc. There are several tools that can help you in running Chaos experiments. Many of them are targeted at virtual machines or a Kubernetes cluster and cause havoc on the infrastructure layer. Of course, you can also write your own scripts or tools to help you with your experiments.

Some examples of tools you can use are:
> Gremlin[2]
> Chaos Toolkit[3]
> Chaos Mesh[4]
> Azure Chaos Studio[5]

Monitor the resulting behavior
When you conduct the experiment, it is time to look at the metrics again. What do you see? Do you see the expected behavior of your system? Is the hypothesis valid? When you see the system does not behave as expected, try to gather as much information as possible why this is the case. Also, make sure you keep the blast radius and real user impact in focus.

Document the process and observations
After the experiment is complete, you have either proved or disproved your hypothesis. Make sure you document the process you executed, especially when you found that your hypotheses failed. Make sure you document your learnings. Consider performing a blameless learning review to find out what happened and document the learning review for future use.

Identify fixes and apply them
When you find your hypotheses did not work, apply the necessary fixes and automate the experiment. Make sure you can run the experiment multiple times, maybe even on a schedule. Systems change, and environments change, and you need to validate your hypotheses over and over again.

How can I get started with Chaos Engineering?

Getting started with Chaos Engineering is something you can do any time, as long as you take the user impact and blast radius into account. A common way to introduce chaos is to deliberately inject faults that cause system components to fail. The goal of Chaos Engineering is to create a more resilient and reliable application. With Chaos Engineering practices, you need to test and validate your application is indeed more resilient. Architectural patterns like circuit breakers, failover, and retry can help to make your application more robust. Then, after you have built your application, you need to observe, monitor, respond to, and improve your system's reliability under adverse circumstances. For example, taking dependencies offline (stopping API apps, shutting down VMs, etc.), restricting access (enabling firewall rules, changing connection strings, etc.), or forcing failover (database level, Front Door, etc.), is a good way to validate that the application can handle faults gracefully.

[2] https://www.gremlin.com/
[3] https://chaostoolkit.org
[4] https://chaos-mesh.org/
[5] https://azure.microsoft.com/en-us/services/chaos-studio/

It is important to start small. Start by defining a hypothesis and a very small experiment and go through the different steps that I described above. To define your first hypothesis, you should look at things you expect to go right but that you never actually look at. A good source of inspiration is a keynote of Adrian Cockroft[6]. In this keynote, he explains some basic things that go wrong. For your convenience, I have listed a number of these categories and things that can go wrong:

Infrastructure Failures	
Device Failures	Disk, power supply, cabling, circuit board, firmware
CPU failures	Cache corruption, Logic bugs
Datacenter failures	Power, Connectivity, cooling, fire, flood, wind, earthquake
Internet Failures	DNS, ISP, internet routes
Software stack Failures	
Time Bombs	Counter wrap round, memory leak
Date bombs	Leap year, leap second, epoch
End of unix time	
Expiration	Certificates timing out
Revocation	License or account shut down by supplier
Exploit	Security failures e.g. Heartbleed
Language bugs	Compiler, interpreter
Runtime bugs	JVM, Docker, Linux, Hypervisor
Protocol problems	Latency dependent or poor error recovery
Application Failures	
Time bombs (in application code)	Counter wrap around, memory leak
Date bombs (on application code)	Leap year, leap second, epoch, Y2K
Content bomb	Data dependent failures
Configuration	Wrong config or bad syntax
Versioning	Incompatible versions
Cascading failures	Error handling bugs
Cascading overload	Excessive logging, lock contention, hysteresis
Retry storms	Too many retries, work amplification, bad timeout strategy
Operations failures	
Poor capacity planning	
Inadequate incident management	
Failure to initiate incident	
Unable to access monitoring dashboards	
Insufficient observability of systems	
Incorrect corrective actions	

Summary

Chaos Engineering is fairly new to many people. Although it has existed for several years, it is not yet embraced by the broad audience. That is a shame because chaos engineering can really help build more resilient systems. By defining hypotheses and conducting experiments to prove your hypotheses you can test your system to deal with unexpected situations. There are many small experiments you can execute on your system, so getting started should be very simple. However, always take the potential user impact and blast radius into account and carefully plan your game day. </>

René van Osnabrugge
ALM, DevOps, Continuous Delivery, Initiator and Inspirator

xpirit.com/rene

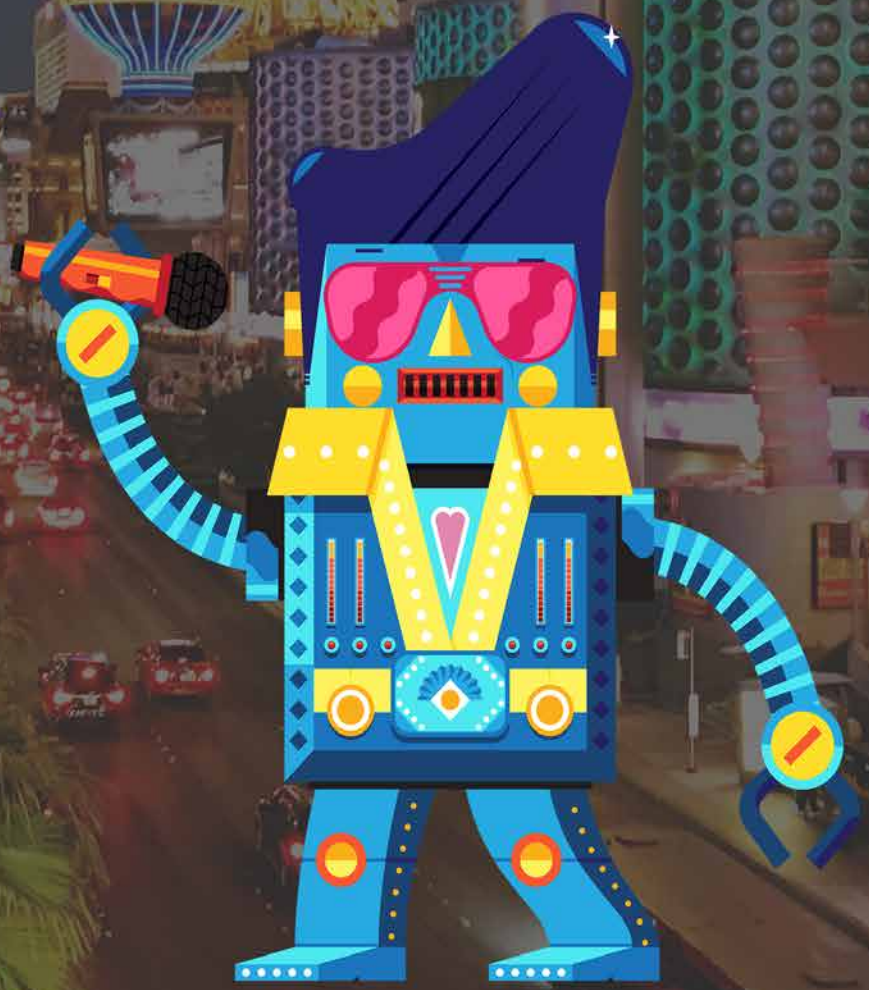
[6] https://www.youtube.com/watch?v=cefJd2v037U

TECHORAMA

DEEP KNOWLEDGE IT CONFERENCE

23 - 25 MAY 2022
ANTWERP | BELGIUM

10 - 13 OCT 2022
EDE | NETHERLANDS



TICKETS:
www.techorama.be | www.techorama.nl

www.xpirit.com

Together we drive change.

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