# We Are Developers!

Summer – 2/2023

A Special Issue by
Heise Medien GmbH & Co. KG

Powered by heise Developer

Capgemini
top EMPLOYER DEUTSCHLAND GERMANY 2023
CERTIFIED EXCELLENCE IN EMPLOYEE CONDITIONS

VOM BECKENRAND ZUSCHAUEN?

ODER AUFS KARRIERE-SPRUNGBRETT STEIGEN?

Mit innovativen Lösungen treiben wir die digitale Transformation branchenübergreifend voran und arbeiten gemeinsam an einer integrativen und nachhaltigen Zukunft. Wir bieten dir ein inspirierendes Team, flexible Karriere- und Weiterentwicklungsmöglichkeiten sowie die Freiheit, mit deiner Arbeit Perspektiven für dich und andere zu schaffen.

Mehr über Capgemini und deine Einstiegsmöglichkeiten auf
**www.capgemini.de/karriere**

GET THE FUTURE YOU WANT

# EDITORIAL

## A World of Decisions

Dear reader,

Every day we make a sheer infinite number of decisions—from the tiniest and seemingly most inconsequential to large life decisions. In the tech world, no different than in other areas, the trends are measured regularly, and the results can provide some guidance. There are countless studies that claim to have found "the most popular JavaScript framework" or "the most popular programming language" of the month or the year. Though the methods are often debatable, there are certain names that tend to show up regularly. As every decision may open up a whole new world, it is up to you to choose which of the trending topics to dive into. This magazine highlights two popular candidates, but also looks at the bigger picture and proposes a new way of software delivery.

In line with the need for an ever higher performance, Astro is a new web framework designed for speed. This year, it reached its second major version and seems to have a promising future in the vast array of emerging web frameworks. The makers of the annual study "JavaScript Rising Stars" recognized this as early as January 2022, when they declared Astro one of the most notable projects of the previous year. In this issue, Timo Zander provides a tutorial for creating a blog with Astro that showcases some of the framework's defining features. While several programming languages have been around for decades—such as C or C++—, modern languages set out to address some of their shortcomings. Rust has been the winner of the annual "Stack Overflow Developer Survey" for seven years in a row. It incorporates ideas of C and C++ while combining them with a modern syntax. As Stefan Baumgartner demonstrates in his introduction for curious developers, Rust lays its focus on memory safety while maintaining a good developer experience. And lastly, Robert Hoffmann addresses the topic of platform engineering: What are the benefits of an internal developer platform and how can it increase flexibility in delivering software?

I hope you enjoy reading this issue and making decisions,

[illegible]

Maika Möbus

CONTENTS

## Young Professionals Write for Young Professionals

This journal is based on a series of articles published by Heise Online, where we provide a platform for young professionals to publish their first professional articles. The journal is released in German language in a six-monthly rhythm, and appears in English once a year. For writing, the authors receive mentoring from the Heise Developer editorial team. The series is also intended to encourage young people to develop their talent as authors; for example, to share important experiences with your peers or to present an own project you are engaged in. Or simply because you have always wanted to write a technical article, preferably in German.

**Your First Professional Article for Heise**

Contact: developer@heise.de

# > Building Fast Websites With Astro

Timo Zander

The JavaScript rendering framework Astro is one of the stars of 2022's State of JavaScript survey. Its focus on static content makes it much more than just another framework. A practical hands-on.

Ever since JavaScript applications have not only taken over the web but—thanks to Electron—also power many of the most popular desktop applications, their performance has gained a more prominent focus among developers. In an attempt to solve performance pitfalls, Astro emerged in 2022 and was able to gain attention among the sea of competitors due to its fresh ideas and new concepts. Especially when using popular front-end frameworks like React, Vue or Svelte, it becomes a challenge to implement highly responsive webpages. Visitors of JavaScript-heavy pages have had to develop a certain tolerance against sluggish behavior and long loading times. This issue is homebaked, as offloading all rendering onto the client's device removes control over the application's performance and can potentially lead to an unexpectedly high "time to interactive."

The architecture of such applications is called Single Page App (SPA), meaning that the browser effectively only loads a single HTML file, from which point the JavaScript code takes over. Even with server-side rendering, the browser often still fetches the pre-rendered parts of markup using client-side JavaScript code. Compared to more traditional Multi Page Apps (MPA) that were prominently used in the era of technologies like PHP or ASP.NET, SPAs will always come with a performance trade-off.

Many rendering frameworks aim to improve that: Popular choices like Next.js or Nuxt not only help to build a robust, large-scale app using their respective front-end frameworks. They also provide a way to pre-render pages on a server. Yet, server-side rendering features were not the core reason why those frameworks were built and rather organically developed to become one of their selling points. The JavaScript rendering framework Astro, on the other hand, is built with a dedicated focus on performance.

Astro targets content-heavy use cases. Blogs, online magazines and the like rarely need heavy client-side JavaScript and can rather purely rely on the server to do the legwork. However, web developers are used to the ease and comfort that comes with JavaScript development. Not only the availability of high-quality tools and IDEs, but also features like Hot Module Replacement can drastically speed up the developer's workflow. Therefore, going back to traditionally server-side heavy technologies may not be feasible.

## Performance Baked In

Astro aims to reduce the amount of JavaScript code that the client receives as much as possible. Therefore, all content is pre-rendered, either during runtime on a server or during the build. Especially the latter option can heavily decrease the cost. Hosting static HTML pages has also become very inexpensive and can therefore cut infrastructure costs. In an age of surging cloud bills, this can present a significant advantage.

Nonetheless, using interactive elements within Astro is possible. Its concept of so-called "islands" effectively breaks up the web page into isolated pieces of the user interface. This allows the majority of the webpage to be pre-rendered and ready to use, even though some pieces might still need the client to first evaluate some JavaScript. Astro Islands are an opt-in feature: If a developer does not actively decide that a certain piece of the website needs to be rendered on the client side, it is assumed to be static. This is supposed to reduce cognitive overload and prevent accidental performance traps, which front-end frameworks are prone to due to their Virtual DOM and re-rendering strategies that can become obscure with complex data flows.

Since Astro is framework-agnostic, its interactive islands can be implemented in any front-end JavaScript framework (or plain JavaScript). Additionally, even the pre-rendered content does not require any Astro-specific technology. Developers can implement it in popular choices like React, Vue, or

## In a Nutshell

> Applications built with popular front-end frameworks like React or Vue can suffer from long loading times due to client-side rendering.

> Designed with performance in mind, the newcomer Astro aims to solve this by using an innovative architecture, allowing for dynamic content on an otherwise static page.

> This tutorial shows how to create an Astro project from scratch and highlights new features in Astro 2.0.

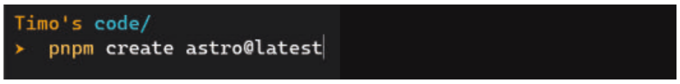
Svelte. The Astro Compiler then translates it into HTML, so that the client's browser never receives any of this technological backbone for static content.

## Setting Up a New Astro Project

Trying out the framework does not require much setup: Astro offers an online sandbox to quickly prototype a new application (all sources for this article can be found here: ix.de/zwet). For creating a new Astro project, developers can use a JavaScript package manager of their choice (Figure 1).

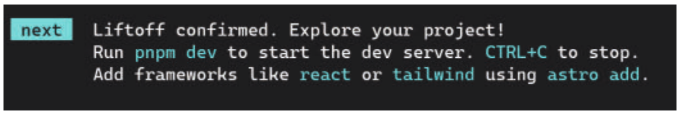
The CLI then guides them through all the steps needed for project creation: Location, the template, and whether to immediately install all dependencies. Note that TypeScript is the default for Astro, as all of the framework's features are fully typed. The shown example follows the recommended default settings. In the end, Astro reports the successful creation of the new project (Figure 2). When starting the development server, Astro displays a welcome page that provides several links to the documentation. The framework defaults to a specific directory structure: The src folder contains all source code which will be processed and compiled by Astro, while the public directory includes static files that Astro should not alter. Typically, these static files are images or a favicon (Figure 3). Within the source folder, Astro provides components, layouts, and pages. They are the building blocks of any Astro project.

Components are reminiscent of their counterparts of the same name in front-end frameworks like React. These small parts of a website, like a button or an input box, are typically reused throughout different pages. Layouts are a way to create a basic skeleton for your website: The content is usually



```
Timo's code/
›  pnpm create astro@latest
```

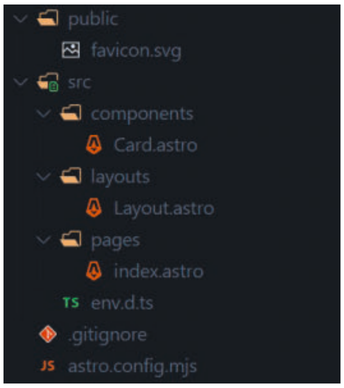
**>> Astro offers a CLI to create a new project which can be used with any JavaScript package manager (Figure 1).**



```
 next    Liftoff confirmed. Explore your project!
         Run pnpm dev to start the dev server. CTRL+C to stop.
         Add frameworks like react or tailwind using astro add.
```

**>> Astro successfully initialized the project and can now be used (Figure 2).**

wrapped in a chosen layout, which contains the basic HTML markup like the `DOCTYPE`, `head` and `body`. However, layouts are technically fully equivalent to components. Splitting them into separate folders is more of a conventional choice than a technological one. The folder titled pages is at the heart of the page: It contains the actual content of the website that can be accessed later. Therefore, the webpage has no content if this folder is empty. The routing is also based on the file structure within this directory. For example, accessing the page under /blog/hello-world will cause Astro to look for a file located at src/pages/blog/hello-world.



```
∨ public
    favicon.svg
∨ src
  ∨ components
      Card.astro
  ∨ layouts
      Layout.astro
  ∨ pages
      index.astro
    env.d.ts
  .gitignore
  astro.config.mjs
```

**>> The standard structure of a newly created Astro project (Figure 3).**

LAUTERBACH DEVELOPMENT TOOLS

Our heart beats for embedded.
Where great ideas come to life.

join us @
career.lauterbach.com

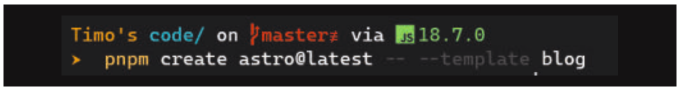
## Learning by Doing: Creating a Personal Blog

To demonstrate some of Astro's most important features, the following example will show how to create a new personal blog using Astro as its backbone. For that, Astro provides many themes which facilitate the initial setup and provide some basic styling options. This example uses Astro's official blog theme, but there are also many community-made themes that often already come with an integrated UI framework.

Starting a freshly created project using the blog theme (Figure 4), results in a very minimal blog, including navigation, a footer and some text. Looking at the contents of the pages folder gives a good overview of the template's ingredients. Specifically, the index.astro file illustrates how .astro files are structured (Listing 1). They consist of a JavaScript header and a mixed HTML and JavaScript template.

The HTML markup is mostly standard. However, developers can use JavaScript expressions similar to the JSX syntax popularized by React. For example, the variable `SITE_TITLE` is passed as a property to the custom component `BaseHead`. This component is responsible for setting all important head attributes (Listing 2). Besides setting the title and description, it also creates Open Graph tags that are used by social media websites to create a preview of the page. Astro's TypeScript integration works out of the box: The framework detects the interface titled `Props` and will use this to type the properties that the component can receive. Moreover, the JavaScript code, rather than the HTML markup, imports the CSS stylesheet. While in this simple case, both methods are equivalent, when using CSS preprocessors like Sass or frameworks like Tailwind CSS, importing stylesheets within the JavaScript code allows Astro to pre-process and compile them. When stylesheets are referenced using `link`-Tags, Astro treats them as static.

The about.md file reveals one of the major features of Astro. It consists of plain Markdown text with an extra header using the Frontmatter syntax. The header always offers the two attributes `layout` and `title`: They set the page's layout component and title, respectively. In the example, the About page will reuse the layout for blog posts. After specifying this, it becomes possible to fill out the layout component's properties. Because of Astro's TypeScript support, most IDEs will suggest the properties that are allowed. If the corresponding integration is installed, Astro also supports MDX—the Markdown superset that enables the use of JavaScript expressions and components within Markdown.



```
Timo's code/ on ᚠmaster≠ via ⬢18.7.0
› pnpm create astro@latest -- --template blog
```

**>> Creating a new Astro project can be done using a template (or theme) (Figure 4).**

## Organization Through Content Collections

The actual blog posts are located in a content collection, one of the new features that Astro 2.0 introduced in January 2023. It allows grouping structurally similar content within collections (e.g. blog posts, newsletters, or podcast episodes) to then more easily generate a static page for each. Located under src/content, each collection has its own

**Listing 1: The index.astro file**

```
---
import BaseHead from '../components/BaseHead.astro';
import Header from '../components/Header.astro';
import Footer from '../components/Footer.astro';
import { SITE_TITLE, SITE_DESCRIPTION } from '../consts';
---

<!DOCTYPE html>
<html lang="en">
    <head>
        <BaseHead title={SITE_TITLE} description={SITE_DESCRIPTION} />
    </head>
    <body>
        <Header title={SITE_TITLE} />
        <main>
                <h1>Hello, Astronaut!</h1>
        </main>
        <Footer />
    </body>
</html>
```

**Listing 2: Astro component to set header attributes**

```
---
// Import the global.css file here so that it is included on
// all pages through the use of the <BaseHead /> component.
import '../styles/global.css';

export interface Props {
    title: string;
    description: string;
    image?: string;
}

const { title, description, image = '/placeholder-social.jpg' } = Astro.props;
---
<!-- ... -->

<!-- Primary Meta Tags -->
<title>{title}</title>
<meta name="title" content={title} />
<meta name="description" content={description} />

<!-- ... -->
```

# WHAT'S NEXT?

JOIN ZF

ZF is hiring innovative and motivated individuals to develop cutting-edge solutions for the future of mobility.

Apply now for a rewarding career and a chance to make a difference.

# NextGenerationMobility
# SoftwareDefinedVehicle

see. think. act.

ZF

**Listing 3: Transforming content collections into HTML**

```
---
import { getCollection } from 'astro:content';

const posts = (await getCollection('blog')).sort(
  (a, b) => a.data.pubDate.valueOf() - b.data.pubDate.valueOf()
);
---
<ul>
  {
    posts.map((post) => (
      <li>
        <time datetime={post.data.pubDate.toISOString()}>
          {post.data.pubDate.toLocaleDateString('en-us', {
            year: 'numeric',
            month: 'short',
            day: 'numeric',
          })}
        </time>
        <a href={`/blog/${post.slug}/`}>{post.data.title}</a>
      </li>
    ))
  }
</ul>
```

directory. Additionally, subdirectories within a content collection offer the option to further structure the data. For example, subfolders can provide a way to easily deal with multi-language content. While creating such a content collection is already achieved by simply creating a new directory, the src/content/config.ts file is highly recommended to enable some of the optional features, such as type safety and validation. Both are powered by Zod, the TypeScript schema validation library that has grown to become one of the most popular libraries according to the current State of JavaScript survey (all links are available here: ix.de/zwet). In this article's example, all blog posts are contained within one such content collection. The `defineCollection()` function allows developers to create as many different collections as they choose. Exporting the collections as a key-value pair tells Astro in which directory to look for the specified types. In the given example, the blog posts all come with a mandatory title and description. The publication date shows the expressive syntax of Zod: This parameter must be specified as a JavaScript date or as a string, which will then be transformed into a date. Similarly, the date of the last update, `updatedDate`, is an optional string that will be parsed into a date. Due to Zod's rich features, the control over the content collection's type covers nearly all edge cases.

The user then writes the actual entries of a content collection as Markdown or MDX files. Their headers specify the required properties in the typical Frontmatter syntax, while their bodies can contain any type of desired content. What is special about this feature is that the entries will not be automatically displayed or generated into pages. Instead, Astro provides a rich API to retrieve a collection's content and transform it into various output options, like HTML lists, grid tiles, slideshow entries, and more. Therefore, content collections could also contain email templates or social media posts that should not become a static HTML page but are instead used via another channel, such as an API. In this example, the blog posts should be listed under the Blog menu point and lead to generated HTML pages where the content can be read (Listing 3). Using one of Astro's APIs for content querying, `getCollection()`, all blog posts are listed. Note that this code is fully executed during compile time: The final webpage will only contain static links to pre-generated, plain HTML files. Again, thanks to TypeScript, the IDE even provides auto-completion for the existing collection names. The resulting array can be used like any other JavaScript array and does not carry any inherent special features. It is merely an array of plain objects.

## File-based Routing: Balance of Simplicity and Versatility

Astro supports dynamic routing, meaning that any blog post is accessible via its slug in the URL, such as /blog/my-first-post. This type of readable URLs is regarded to be more favorable for search engines and users alike. Since routing is based on the file system structure, Astro makes use of

**Listing 4: Demonstration of dynamic routing**

```
Nested route is matched:
/src/pages/articles/[…slug].astro -> /articles/this/is/a/post ✓

Route is not matched:
/src/pages/articles/[slug].astro -> /articles/this/is/a/post X
```

**Listing 5: Example of a getStaticPaths() implementation**

```
---
export async function getStaticPaths() {
    const posts = await getCollection('blog');
    return posts.map((post) => ({
           params: { slug: post.slug },
           props: post,
    }));
}
type Props = CollectionEntry<'blog'>;

const post = Astro.props;
const { Content } = await post.render();
---

<BlogPost {...post.data}>
    <h1>{post.data.title}</h1>
    <Content />
</BlogPost>
```

## DEVELOPING CLOUD SOLUTIONS OF THE FUTURE IN AGILE WORKSTREAMS.

## THIS IS WHY WE'RE AT DATEV.

We're working together to implement secure cloud solutions and innovative apps. As a cloud developer at DATEV, you can look forward to a wide range of tasks in an agile culture of innovation. Learn more about job openings and exciting projects at one of Europe's leading IT service providers.

**DATEV.COM/CAREERS**

DATEV

Shaping the future – together.

specifically formatted filenames, consisting of square-bracket-wrapped sections. They can either catch exactly one subsection of the URL or be nested, essentially catching the whole tail of the URL. Without rest parameters, the dynamic routing only captures individual blocks separated by slashes (Listing 4).

The very idea of dynamic routing essentially goes against Astro's core principle: generation at build time. Therefore, all dynamic routes must export a `getStaticPaths()` function that returns an array of objects with a `params` property each. Those `params` then contain the term from the filename's square bracket blocks. For example, a file [authorName].astro must have its static path function return an array, including objects like `{params: {authorName: 'Timo Zander'}}`. This is then used to give a name to the generated HTML file and its corresponding link. Also, there may be multiple blocks within a given filename. For example, a file under src/pages/[lang]/[author]/[...slug].astro will look up a certain author's post in a given language with the respective slug (or title). Therefore, in the blog example, the src/pages/blog/[...slug].astro file needs to query the content collection and return all posts with their respective slugs in its `getStaticPaths()` function (Listing 5).

The shown file queries all blog posts and returns their slugs in its JavaScript head. The HTML body merely renders the blog posts' contents. The values of the `params` key within `getStaticPaths()` are crucial for generating the route and HTML file. Using the `props` property, one can pass data that is needed for the eventual rendering. This data can be arbitrarily typed, including objects or arrays. It is then possible to access this data in the body of the page. Thus, Astro executes the `getStaticPaths()` function and then, while generating each individual file, provides the received props as context to the file. In this example, the whole post is passed to be then rendered in the corresponding `BlogPost` layout. The `render()` function is a special feature of content collections. It transforms the content into HTML which is returned as a `Content` component, which can then be used in the Markup.

## Fetching Dynamic Content Using APIs

Astro cannot just load and pre-generate file-based data. Instead, all the query APIs work equivalently with dynamically fetched content. To illustrate that, an example will use

the *heise Developer* RSS feed to load and list the five latest articles on the front page. This requires a change in the src/pages/index.astro file (Listing 6). Since JavaScript's built-in capabilities to parse XML files are limited to the browser environment, the rss-parser package will come into play. Alternatively, other DOM parsers like jsdom would also work. The benefit of the RSS-specific package is that it nicely converts the data to typed objects, making it easier to work with them. It is important to remember that the JavaScript code is fully executed during build time. Adding a `console.log()` statement will not show up in the browser's console but instead in the terminal that runs the dev command. Moreover, the framework allows for top-level `awaits`. This means that the global context is executed as an async function, enabling the usage of the `await` keyword. Again, this is not affecting the user but only the build tool, meaning that the Astro build could be slowed down if high-latency resources are fetched. Listing the fetched links as an unordered list is done using the JSX syntax. Due to the type safety of the RSS library, parsing the publication date requires an extra check: If the `pubDate` field is undefined, the time will not be rendered. In practice, data might be fetched from a CMS, an API or a combination of many data sources. Therefore, Astro proves flexible to adapt to any given environment. The documentation provides a guide for the most common Content Management System (CMS) software options and how to integrate them with Astro. There are specific integration packages available for some that ease consuming the API of the CMS.

## Generating Endpoints From Scratch

File-based routing not only supports dynamic HTML files, but also so-called endpoints. An endpoint can be any resource accessible from a web server, such as JSON files, XML sitemaps, images or assets of any kind. Adding the .js or .ts file ending to the desired name of the generated output will create an endpoint. For example, to generate an asset called sitemap.xml, the file src/pages/sitemap.xml.ts will be executed. Such files must export a `get()` function that returns an object containing a `body` attribute that will be the file's content.

Furthermore, there are many optional attributes such as file encoding. The interface `APIRoute` gives an overview of all implemented properties.

**Listing 6: Modified index.astro file, loading an RSS feed**

```
---
import Parser from "rss-parser";

const parser = new Parser();
const feed = await parser.parseURL('https://www.heise.de/developer/rss/news.
rdf');

const feedItemsToShow = feed.items.slice(0, 5);
---
<!-- ... -->
<h3>Recommended reads by {feed.title}</h3>
<ul>
{
    feedItemsToShow.map((item) => (
    <li>
           <a href={item.link} target="_blank">
           {item.title}
           </a>

           {item.pubDate && (
           <time datetime={item.pubDate}>
                      {new Date(item.pubDate).toLocaleDateString("en-us")}
           </time>
           )}
    </li>
    ))
}
</ul>
<!-- ... -->
```

As an example, one can add the logo of Astro to the footer of the blog. Since this is a png file, a new file like astro-logo.png.ts will then fetch the logo from the official Astro press kit (Listing 7). Referencing this in the Footer component does not require any special syntax (Listing 8). The possibilities that this feature offers are plentiful. Assets could be fetched immediately from cloud storage instead of being manually downloaded and bundled during the build. APIs could be queried for their current versions during compile time to generate a JSON file with a version overview. Therefore, Astro can cover many responsibilities traditionally fulfilled by build systems.

## The Island Architecture: Embedding Client-side JavaScript

Some features simply require JavaScript instead of static HTML only. Astro's island architecture allows achieving this without losing much performance. The given blog requires client-side scripts to allow the user to switch between light and dark modes. Since dark mode styling is a deep topic in itself, the example will only implement a rudimentary version of a well-styled dark mode.

For one thing, the CSS code needs to implement a specific behavior for the dark mode. In this case, a `dark` class is added to the body whenever the dark mode is needed. To allow the user to switch between the two modes, a client-side component is placed inside the footer (Listing 9).

The JavaScript code first ensures that the `dark` class is correctly initialized. Specifically, if the user's browser has dark mode set as the standard option, it should respect that.

**Listing 7: Dynamically generating an image asset with the file astro-logo.png.ts**

```
export async function get() {
    const response = await fetch("https://astro.build/assets/press/full-
logo-light.png");
    const buffer = Buffer.from(await response.arrayBuffer());
    return {
           body: buffer,
           encoding: 'binary',
    };
}
```

**Listing 8: Referencing a dynamically generated image as if it were a static file**

```
<footer>
  Powered by <img
    src="/astro-logo.png"
    alt="Astro"
  />
</footer>
<style>
footer img {
  display: inline;
  height: 20px;
  vertical-align: middle;
}
</style>
```

**Listing 9: Dark mode toggle component to switch between light and dark mode**

```
<button class="dark-mode-switch">Dark</button>

<script>
  const darkModeSwitch = document.querySelector(".dark-mode-switch")!!;

  document.body.classList.toggle(
    "dark",
    window.matchMedia &&
      window.matchMedia("(prefers-color-scheme: dark)").matches
  );

  function updateDarkModeSwitch() {
    if (document.body.classList.contains("dark")) {
      darkModeSwitch.textContent = "Light mode";
    } else {
      darkModeSwitch.textContent = "Dark mode";
    }
  }

  if (darkModeSwitch) {
    darkModeSwitch.addEventListener("click", function () {
      document.body.classList.toggle("dark");
      updateDarkModeSwitch();
    });
  }

  updateDarkModeSwitch();
</script>
```

The remaining code makes sure that every click on the button either adds or removes the `dark` class and updates the button's text.

In practice, client-side components can be more complicated, leveraging the power of UI frameworks or embedding pre-existing components from the ecosystems of React, Vue or Svelte.

## Continuous Deployment Is Key

Astro's focus on build-time generation makes it a prime candidate for automated deployments. When external data is fetched, like the RSS feed before, it is crucial to regularly re-build and re-deploy the application. In static mode, the build output only consists of HTML, CSS, and optionally some JavaScript files. Webpage deployment then only requires uploading the build output onto any web server. Therefore, doing this frequently does not come with any notable cost or overhead, especially not when a CI tool takes care of the task. The official documentation offers an overview of possible hosting options, some of which come with Astro-specific integration to facilitate the setup. In principle, any web hosting offering is compatible with Astro (at least in static mode), as there are no specific requirements.

With server-side rendering, deploying requires an additional adapter that provides a runtime for the rendering engine, which typically can be any Node.js or Deno server. The realm of server-rendered Astro and its specifics, though, would go beyond the scope of this article.

## Embracing Change

At first glance, Astro might look like yet another JavaScript framework. When taking a closer look, it seems like an overly hyped time travel to the 2010s when having static web pages was the norm. What those judgements miss, however, is how web development has changed. The surge of JavaScript-heavy apps has come with an improvement in developer experience and created an according ecosystem. Astro does not try to fight this movement but embraces it: Due to its frictionless integration with whatever frontend framework, CSS preprocessor or backend a developer might already be familiar with, it offers a low barrier to building performant web pages without introducing numerous abstraction-heavy concepts as other frameworks do. Therefore, it is worth a look for anyone with the intent to (re-)build a content-heavy website that could use an extra performance boost. (mai)

**Sources**

All sources for this article: ix.de/zwet

**Timo Zander**

studied Applied Mathematics and Computer Science and works as a software developer. He is interested in open source, the JavaScript universe and emerging technologies.

WIBU SYSTEMS
CodeMeter – Eine Symphonie von Software-Monetarisierungs-Tools
Komponieren Sie Ihren eigenen Code
Orchestrieren Sie Ihre Lizenzstrategie
Stimmen Sie Ihren IP-Schutz genau ab
Verbreiten Sie Ihr gestaltetes Werk
Klingt einfach, oder? Und das ist es auch mit CodeMeter
CM Embedded
CM Dongle
CM LC
CM Cloud
Starten Sie jetzt und fordern Sie Ihr CodeMeter SDK an
wibu.com/de/sdk
+49 721 931720
sales@wibu.com
www.wibu.com
SECURITY
LICENSING
PERFECTION IN PROTECTION

# > Rust for Curious Developers

**Stefan Baumgartner**

The programming language Rust has been on the rise for several years. This article dives into the reasons, demonstrates Rust's unique capabilities and puts them into perspective.

The Rust programming language has been the most beloved in the annual Stack Overflow developer survey for seven years in a row. Not only do people love it, but it is also gaining traction in the industry, with large companies like Microsoft, AWS, and Google betting heavily on the unique properties of Rust. What makes it so popular among programmers worldwide?

## A Modern Programming Language

Rust was created by Graydon Hoare at Mozilla in the early 2010s as part of their experimental browser platform Servo. The underlying goal of Rust was to allow developers to create new browser features without having to deal with all the baggage and memory allocation problems that C and C++ carried. Similar to the original idea of the programming language Go, Rust was intended to create fast software that was fun to write. It draws influences from many other programming languages. While the syntax is very C-like, there are traces of functional programming languages as well, especially OCaml, in which the original Rust compiler was written. This makes Rust a modern language that has the upper hand over other programming languages in several ways.

For example, Listing 1 shows a Rust function that calculates the score of a word in a game of Scrabble. It is as readable as equivalent Python code but contains convenience features like expressions. They enable having a `match` statement just next to the addition of two numbers. The `match` statement or expression itself allows for pattern matching across a set of valid values. In this case, it goes through all possible characters, and the `match` keyword requires a developer to take care of all possible values. Since some sets can include a plethora of values, a default case with the underscore sign can catch all remaining values. It indicates that in all other variations, the return value will be 0.

Understanding why companies in Big Tech are using Rust requires a closer look at three crucial features:

- Memory safety without garbage collection
- Zero-cost abstractions
- Fearless concurrency

## Memory Safety Without Garbage Collection

For decades, the most widely used programming languages were divided into two camps: On the one hand, developers could write software in programming languages with managed memory allocations, such as Python, Java, or JavaScript. On the other hand, developers could choose manual memory management like in C or C++, which allowed for highly performant software because there was no need for a runtime to deal with garbage collection.

Rust follows a different strategy: It relies on compile-time memory allocations based on a strict ruleset that developers must follow. This is more effort to learn, but creates 100 percent memory-safe applications. Scenarios like use after free, double free, or buffer overreads and overwrites create software vulnerabilities that are easy to exploit. Both Micrsoft and Google found out that in Windows and the Chrome project, respectively, 70 percent of all severe security bugs were in fact memory issues (all sources for this article are available here: ix.de/z64k). As of December 2022, 20 percent of all native code of Android is written in Rust, and to this date, there have been zero memory safety issues in Android's Rust code.

Rust achieves memory safety through its “ownership and borrowing” system. Its fundamental principle is that there can be only one owner of data. When a value is assigned to a

### In a Nutshell

> Rust provides a unique, modern syntax for elegant code that includes traits to provide abstractions without overhead.

> It introduces the ownership memory model to ensure optimized memory management at compile time.

> Examples show how it uses both zero cost abstractions and ownership to prevent data races.

variable, this variable becomes the owner. When the owner goes out of scope, memory is freed. Ownership can be transferred, though. The example in Listing 2 shows a typical scenario that would not pose a problem in a programming language like Java or JavaScript but causes compile errors in Rust. The moment the variable `other_numbers` is assigned to the value of `numbers`, the variable `numbers` no longer has a value. `other_numbers` owns the data, and the compiler will not allow developers to use the original binding anymore. The good thing: The Rust compiler clearly informs developers about what has happened and usually also provides mitigations. Oftentimes, granting multiple access to the same data is necessary. Therefore, Rust introduces the concept of borrowing data: Developers can create shared references that point to the same data, while the original binding remains the owner (Listing 3). The ampersand (&) is part of the type. It shows developers exactly what the software expects, for example, when they ask for shared references in function signatures. The Rust compiler then checks whether the bindings that own the data live long enough and do not go out of scope before the references do. If a reference is used after the owner is dropped, Rust will not let the code compile.

**Listing 1: Calculating a Scrabble score**

```
pub fn score(word: &str) -> u64 {
    let mut score = 0;
    for ch in word.to_lowercase().chars() {
        score = score + match ch {
            'a' | 'e' | 'i' | 'o' | 'u' | 'l' | 'n' | 'r' | 's' | 't' => 1,
            'd' | 'g' => 2,
            'b' | 'c' | 'm' | 'p' => 3,
            'f' | 'h' | 'v' | 'w' | 'y' => 4,
            'k' => 5,
            'j' | 'x' => 8,
            'q' | 'z' => 10,
            _ => 0
        }
    }
    score
}
```

**Listing 2: One value, two variables pointing to it. This example will not compile.**

```
fn main() {
    let numbers = vec![1, 2, 3, 4];
    let other_numbers = numbers;
    println!("{:?}", numbers);
//                   ^^^^^^^
// Error: borrow of moved value: `numbers`
}
```

**Listing 3: An example of data borrowing**

```
fn main() {
    let numbers = vec![1, 2, 3, 4];
    let other_numbers = &numbers;
    println!("{:?}", numbers);
}
```

Shaping the future.

We live in a changing world. Discover new horizons with us at the German Aerospace Center (DLR), where 11,000 bright minds combine their knowledge, talent and research to shape the future together. We're working on solutions to address the greatest challenges facing our society and we're looking for support by bright minds.

Sven Torstrick-von der Lieth is one of us. As a materials engineer at the DLR Institute of Lightweight Systems he and his team carry out groundbreaking research in aircraft construction.

Would you like to shape the future with us? Discover our 200 open positions in computer science and IT: **DLR.de/jobs/IT**

DLR

© Fotos: DLR/Ingo Boelter

# Stay up to date with the developer world

Get the latest developer news delivered to your inbox every week

Dev Digest

Hello fellow developer,

Welcome to the newest issue of Dev Digest, curated by WeAreDevelopers.
This week's topics include: Ubuntu updates, third party Rust support in Chrome an
an AI that lets you talk to anyone!

Maybe your new year's resolution is to learn something new and have fun with yo
team? How about bringing them to Berlin this summer.
Get tickets for WeAreDevelopers World Congress

Top developer news

- CircleCI incident report for January 4, 2023 security incident
- How Docker 2.0 went from $11M to $135M in 2 years
- Ubuntu 22.04 LTS servers and phased apt updates
- 30 years of Brainf***
- Chromium will support third-party Rust libraries
- This ChatGPT rival lets you talk to anyone

Reading list

- 5 Tech recruitment trends shaping talent acquisition in 2023
- Tips for developers to build an efficient remote workflow
- Software engineer career: things you should know
- How to escape tutorial hell
- ChatGPT the good, the bad, the ugly

More articles

Coffee with Developers

**Join 100k+ developers in the loop**

**Handpicked content**

**From developers, for developers**

**Sign up for free at:**

**wearedevelopers.com/devdigest**

WeAreDevelopers

**Listing 4: Mutable access to numbers to change the data**

```
fn append(vec: &mut Vec<u32>) {
    vec.push(5);
}

fn main() {
    let mut numbers = vec![1, 2, 3, 4];
    append(&mut numbers);
    println!("{:?}", numbers);
}
```

To mutate data, developers must declare a binding as mutable. Additionally, they must create mutable references if other parts of the code should change the owner's data (Listing 4). There can be multiple shared references, but only one mutable reference. Also, if the mutable reference is used, all shared references become invalid. This ensures that no pointer points to data that might become invalid, for example, if the vector shrinks or if the program needs to reallocate.

The most important aspect of ownership and borrowing is that what is happening in the code becomes obvious. The types show developers what to expect, and the compiler understands how memory must be allocated. No matter how a Rust program is structured, the ownership and borrowing rules provide substantial information about the intricacies of the bindings. The chance of running into memory problems due to a lack of knowledge about the codebase is eliminated.

## Abstraction Without Overhead

One of Rust's mantras is to allow abstractions without overhead. Rust achieves this goal by introducing traits. Traits are similar to interfaces in how they define and abstract shared behavior across types. Implementing certain traits on types makes them compatible with the broader ecosystem. Listing 5 shows an implementation for Fibonacci numbers in the

**Listing 5: Creating Fibonacci numbers by implementing the Iterator trait and using checked_adds to ensure values can still be created**

```
struct Fibonacci {
    curr: u128,
    next: u128,
}

impl Iterator for Fibonacci {
    type Item = u128;

    fn next(&mut self) -> Option<Self::Item> {
        let new_next = self.curr.checked_add(self.next)?;
        self.curr = self.next;
        self.next = new_next;
        Some(self.curr)
    }
}
```

128-bit wide unsigned integer type. The `struct` stores a `current` and `next` number, but the calculation of Fibonacci numbers is implemented using an `Iterator` trait. Developers must define the associated type `Item` to explain which results to expect and then calculate the new number with every call to `next`.

The `next` method yields an `Option`, which is an enum containing two possible values. It can either have some value or no value (None). The result is one of the two variants, and Rust requires developers to take care of both. Either through the `match` operation or through built-in mechanisms like the iterator, where `None` is a sign to stop iterating. This makes iterators compatible with `for` loops. A `for` loop calls the `next` method until the method yields `None`. Developers may just say, "iterate over this collection." The method `checked_add` prevents overflow. It also returns an `Option`, and this is where the question mark operator comes in. It allows developers to make their code much more concise, as the "bad" case—`None`—will be returned immediately, and the "good" case—`Some`—will be unpacked. This

**Fight cancer with Rust**

Join the Medical Faculty Mannheim, University Heidelberg, as a Software Developer!

**MIT HERZ DABEI!**

The division of Complex Data Processing in Medical Informatics designs, develops and operates national and international IT infrastructures for the secure exchange of patient data. We support physicians and scientists in the transfer of novel therapies into real-world applications.

We develop open-source in Rust, Java, Python, TypeScript (Angular) and Clojure.
Application via **www.umm.de/karriere**

Contact: Prof. Dr. Martin Lablans (martin.lablans@medma.uni-heidelberg.de)

UMM UNIVERSITÄTSMEDIZIN MANNHEIM

Medizinische Fakultät Mannheim der Universität Heidelberg

Universitätsklinikum Mannheim

**July 27 – 28, 2023**

**WeAreDevelopers World Congress (Berlin)
with heise Developer Area**

**wearedevelopers.com/world-congress**

heise Developer Area
featuring selected talks from heise Developer conferences

**Book your ticket now!**

Organizers iX MAGAZIN FÜR PROFESSIONELLE IT heise Developer

Sponsors adesso bdr. SECURE CODE WARRIOR WIBU SYSTEMS

**Listing 6: Using an atomic reference counter (or Arc) allows multiple access to the same data.**

```
let counter = Arc::new(Mutex::new(0));
let mut handles = vec![];

for _ in 0..5 {
    let counter = Arc::clone(&counter);
    let handle = thread::spawn(move || {
        let mut num = counter.lock().unwrap();
        *num += 1;
    });
    handles.push(handle);
}
```

operation is known as "bubbling up" an `Option`. The Rust compiler removes all abstractions and creates the most optimized code possible. If possible in a given scenario, it will execute the Fibonacci iterator at compile time and simply write the result into the binary.

## Fearless Concurrency

Rust's ownership and borrowing system, as well as the abstractions in the form of traits and types, prevent encountering data races. Preconditions for data races are two or more pointers that access the same data simultaneously, but with one of the pointers having write access. If a program has no mechanism for synchronizing access to data, it is likely to show undefined behavior. In concurrent scenarios, multiple threads often access the same data. Listing 6 increments the same memory across multiple threads. A mutex ensures exclusive access to this value. Ownership requires developers to make sure that every thread can own a reference to the same memory. By adding an atomic reference counter, or `Arc`, multiple references can point to the same data while allowing for thread-safe access. The `move` keyword gives ownership of all required variables to the thread closure.

## Bottom Line

Those were just three aspects of Rust that excite developers around the world. There is much more to love, though. For example, Cargo and crates.io provide excellent tooling and a rich ecosystem of third-party libraries, similar to what npm offers in the JavaScript world, and the Rust syntax is optimized for ergonomics. If developers go back to other programming languages, they will miss the effect that a simple drop of a semicolon has on their programming. And, last but not least, there is a community that cares for inclusion, empowerment, and enablement. (mai)

**Sources**

Links for this article are available here: ix.de/z64k

**Stefan Baumgartner**

is a software architect and developer based in Austria. He helps organizations with Rust and TypeScript under oida.dev. Stefan is the author of "TypeScript in 50 Lessons" (Smashing Magazine, 2020) and "The TypeScript Cookbook" (O'Reilly, 2023). In his spare time, he organizes ScriptConf and Rust Linz. Stefan enjoys Italian food, Belgian beer, and British vinyl records.

# > Running With a Decentralized Developer Platform

**Robert Hoffmann**

An internal developer platform can simplify software delivery for development teams. In the shape of a decentralized developer platform, it can provide them with a high degree of ownership over managing their infrastructure, thereby increasing flexibility and agility.

Organizations rely on high-performing developer teams to react quickly to changing market conditions and deliver new products and features to consumers faster. Key characteristics of such teams are that they are small, self-sufficient, and can perform most or all of their work independently from other teams, as well as make most or all decisions on their own. Their flexibility and development speed, though, must be balanced against the need to build well-architected services. This includes being secure, compliant with company policies, and following best practices for operational excellence and reliability.

## In a Nutshell

> The increasing complexity of cloud-native tech stacks is leading to high cognitive load for developers.

> Internal developer platforms provide tools, services, and artifacts to reduce complexity for developers and make it easier for them to build well-architected applications.

> The decentralized developer platform is a particular variant where mature developer teams consume curated patterns for deploying and operating applications in their own environments.

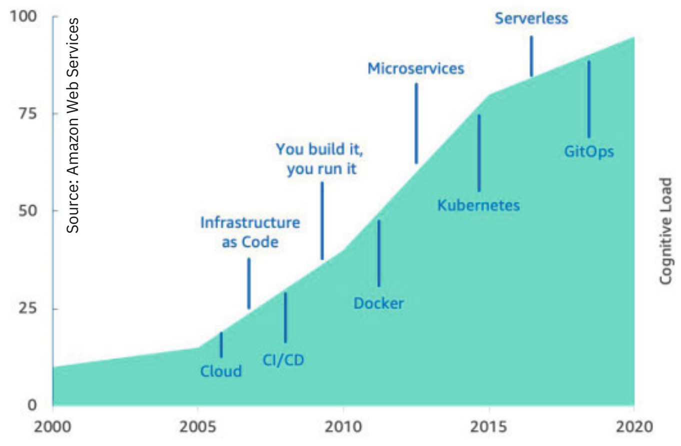
Developers commonly experience the adherence to company requirements as a slowdown of their development activities, while the demand for non-functional requirements rises. At the same time, cognitive load for developers has increased as they need to deal with complex technology stacks to build and run cloud-native applications. To bridge the gap, companies can employ platform engineering, which is the discipline of simplifying software delivery for product teams through developer enablement and self-service, in the form of an internal developer platform (IDP), hereafter referred to as "developer platform." This article explains how platform engineering evolved from the DevOps movement and what types of developer platforms exist. Going further, it introduces the decentralized developer platform as an advanced type that favors organizational scalability, self-service and ownership by the involved developer teams. Finally, it provides an overview of the guiding principles and required technical capabilities of a decentralized developer platform, which can act as a guide for its implementation.

## The Origin of Developer Platforms

Companies have been building internal developer platforms for years, even since before the term was coined, and the concept is known throughout the IT industry. For example, Amazon Web Services (AWS) has been specifically discussing it in the scope of building platforms in the cloud since 2019, aptly calling it cloud platform engineering (all links for this article: ix.de/z66z). AWS describes it as codifying differences between stock AWS service configurations and enterprise standards, packaged and continuously improved as self-service deployable products available to internal customers.

The origins of platform engineering can be traced back to the DevOps methodology. The "you build it, you run it" paradigm formulated by AWS CTO Werner Vogels in 2006 urged developers to get into contact with the day-to-day operations of their software and, ultimately, their customers. This led to a movement which puts more responsibility into the hands of developers to own the lifecycle of their applications—from writing the code to deployment in production. At the same time, the complexity of applications and technologies to run them increased as well: For example, development teams started to break monoliths down into microservices, running them on distributed compute platforms like Kubernetes, with the infrastructure and deployments defined as code. In addition, cloud services and software as a service (SaaS) were adopted to reduce undifferentiated heavy lifting and improve the user experience through reduced latency, global availability, and additional features.

**>> Cognitive load for developers has increased over the years as new technologies and methodologies have been invented and subsequently adopted in organizations (Figure 1).**



Adopting these technologies and building cloud-native applications helped teams to reduce the time to release software, and address the growing expectations of users. As a tradeoff, the cognitive load for integrating and running applications increased (Figure 1). Platform engineering is reducing the cognitive load by providing tools, services and artifacts so that developers can focus on product delivery, while reducing lead time and improving the quality of their applications. The exact definition of quality depends on the context, but a common way to frame it is the Well-Architected Framework by AWS, which provides guidance for designing and operating secure and efficient applications through six pillars (Figure 2).

A highly automated self-service developer platform is beneficial to both the platform teams that create and maintain the platform as well as to the users of the platform, the developers. Platform teams spend less time with onboarding activities and tickets, and have more time to focus on the developer experience of using the platform. Developers can request resources and services on demand and experience reduced cognitive load, which helps them focus on product delivery.

## Moving From Centralized To Decentralized Operational Models

An organization offers one or multiple operational models for building and running applications, and this model informs the

**Become our next #pioneer**

Driven by our mission „Detect and Protect“, HENSOLDT is developing intelligent and integrated technologies for the best protection of our customers in the air, on the sea, on land as well as in space and cyberspace.

**Apply now and make the world safer together with us!**

Find jobs in the field of Electrical Engineering and Information Technology

- **SAP Application Specialist**
- **Software Architect & Developer**
- **Systems Engineer & Safety Engineer**

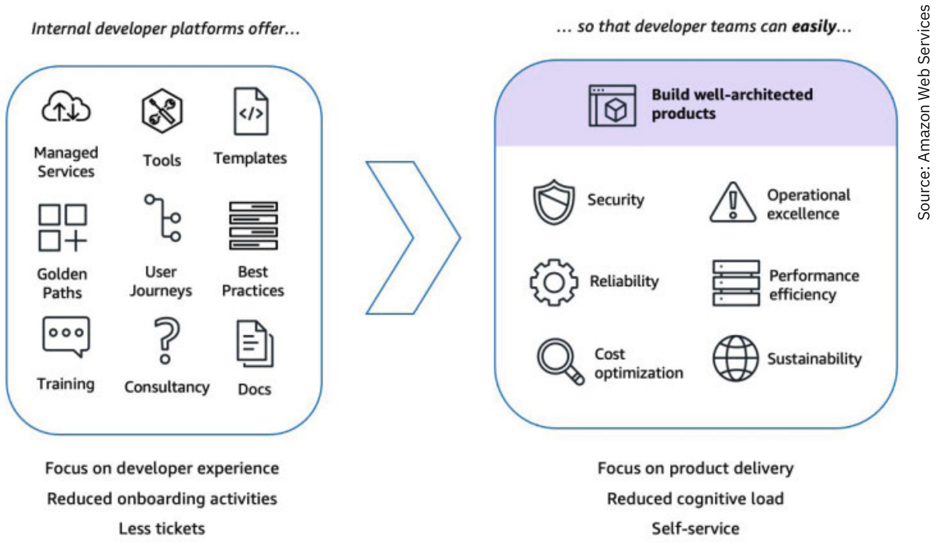
in Taufkirchen, Ulm, Oberkochen and Immenstaad

hensoldt.net/career

HENSOLDT
*Detect and Protect.*

properties of the developer platform(s). A key property that defines an operational model is the allocation of the shared responsibility between developer and platform teams:

- In a centralized provisioning model, the responsibility for architecting, deploying, and managing infrastructure falls primarily on a centralized platform engineering team. The request model is often based on tickets that are sent to the platform team, and the developers wait for the provisioning of resources on their behalf.
- The platform-enabled golden path model is an approach that allows for developers to have some form of customization while maintaining consistency by following a set of standards. In this model, platform engineers clearly lay out “preferred” standards with sane defaults, guard rails, and good practices based on common architectures that development teams can use as-is—the “golden path.” In order to grant developers more freedom and self-service, platform teams offer services with a narrow interface that allow self-service configuration and application deployment. Typically, this manifests as one or many managed container clusters, where the infrastructure is owned and maintained by the platform teams, and developer teams can use the cluster’s API to run containers.
- In the embedded DevOps model, more responsibilities shift to the developer teams, as shared or dedicated DevOps engineers become team members. The DevOps engineers adhere to central platform standards and implement them as infrastructure owned by the individual teams, adding customizations if necessary.



**>> Internal developer platforms offer various types of artifacts and services that help developer teams to build well-architected applications with reduced cognitive load (Figure 2).**

## Decentralized DevOps

Going even further, a decentralized DevOps model gives development teams full ownership and responsibility for defining and managing their infrastructure. In large companies, the number of developers can quickly outweigh the number of platform engineers and DevOps engineers, so teams need to work as self-sufficiently as possible. Organizations adopt decentralized DevOps as an advanced operational model to realize the “you build it, you run it” mindset. The model offers great agility and flexibility to developer teams, but also requires a cultural shift in the organization because these teams now own the entire stack. Practicing decentralized DevOps is a question of preference and maturity. As a prerequisite, developer and platform teams need to agree on supporting this model and shifting responsibility and privileges to developers. This is only possible with a high maturity level of culture and systems. In practice, teams will choose an operational model from a set of options that are available in their organization.

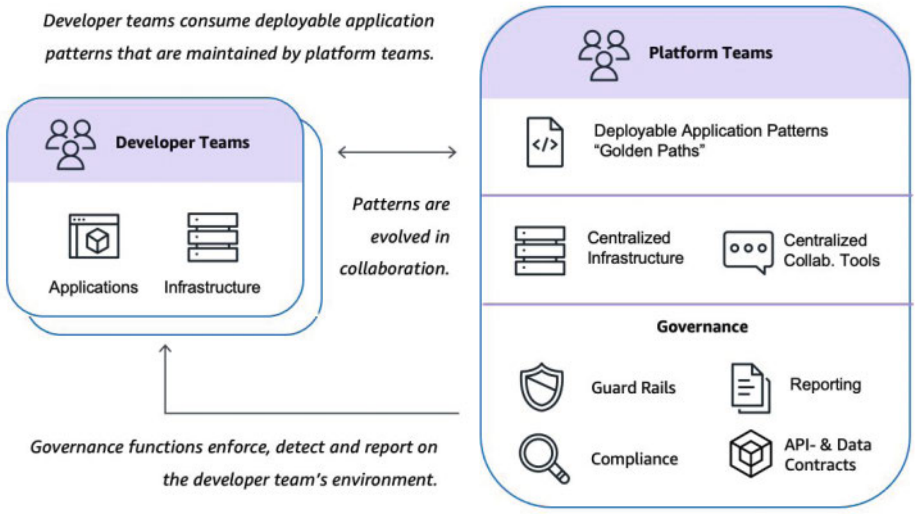
In the decentralized DevOps model, platform teams continue to be responsible for implementing centralized governance in the form of guard rails and boundaries. This reduces the chance that developer teams get lost in their newly found freedom. These centralized governance functions validate configurations, enforce compliance, and detect security vulnerabilities. In a strict definition of decentralized DevOps, the influence of platform teams ends with implementing governance functions, and developers have full autonomy for everything else.

## Decentralized Developer Platform

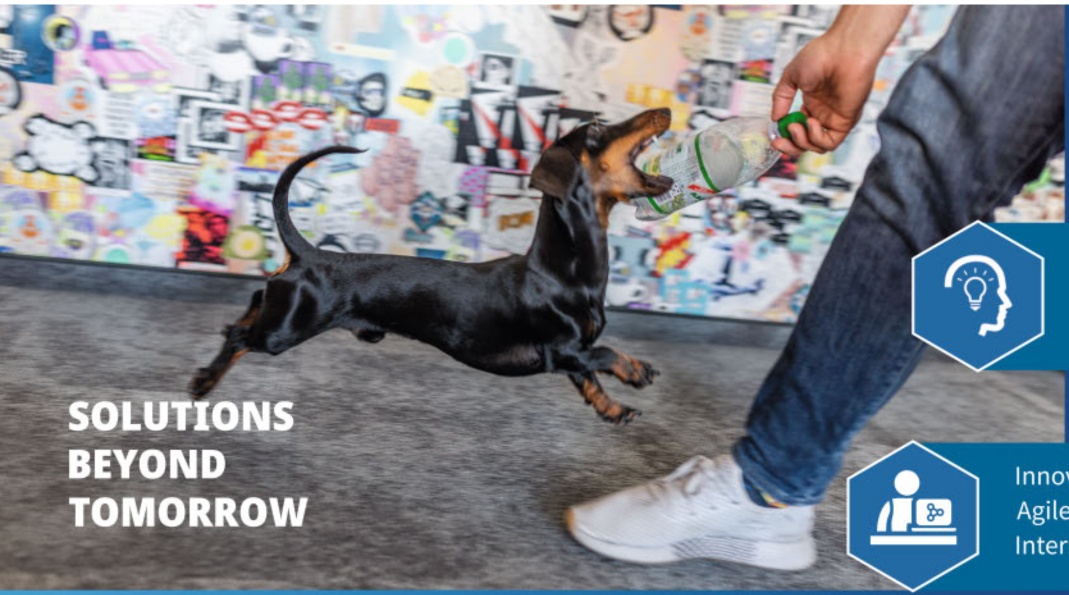
In practice, organizations might find that full developer autonomy is too extreme, and that there are tangible benefits of collaborating more closely with platform teams (Figure 3). Concretely, developer teams would still retain most of their autonomy, but benefit from platform teams acting as cultivators of tech stack consistency, commonly used configuration patterns, best practices, overall developer experience and governance functions. At a minimum, governance includes security and compliance, while more mature organizations additionally implement API and data contract enforcement to simplify data exchange between different teams.

One way this work would manifest is in a repository of deployable application patterns, which are configuration templates that developer teams use to deploy services with sane defaults and built-in best practices. These patterns are a form of golden path, with the distinction that they are treated like internal open-source code. This means that developer teams can inspect, configure, and even arbitrarily customize ("fork") them. Platform teams would be the custodians for these patterns, ensuring their maintenance and improvement. This is the core idea of the decentralized developer platform, which slightly modifies the operational model of decentralized DevOps to shift some influence back to the platform teams. As the name implies, it decentralizes a major part of the responsibility, ownership and corresponding infrastructure,

Source: Amazon Web Services


Developer teams consume deployable application patterns that are maintained by platform teams.
Developer Teams
Applications
Infrastructure
Patterns are evolved in collaboration.
Platform Teams
Deployable Application Patterns "Golden Paths"
Centralized Infrastructure
Centralized Collab. Tools
Governance
Guard Rails
Reporting
Compliance
API- & Data Contracts
Governance functions enforce, detect and report on the developer team's environment.

**>> Developer teams consume deployable application patterns in their own, dedicated environments. Platform teams act as the custodians of these patterns (Figure 3).**



KRONES .digital

New work environment
Mobile and flexible working
Strong team collaboration

Innovative technologies
Agile mindset
International setup

One Community.

One step ahead of the future.

We are an international team of technology enthusiasts, cloud pioneers and data specialists with different skills, talents and personalities - and we are looking for people who are just as colourful as we are.
Who recognize opportunities and implement ideas.
Who won't settle for the status quo, but always think one step ahead.

**Convinced?**

Then become part of our community:

< (Senior) Cloud Software Developer >
< Scrum Master / Agile Coach >
< Cyber Security Engineer >
< Senior Data Engineer >
< ...and much more! >

moving it to the developer teams. Typically, this means that there is a physically or logically separated environment for each team in which their applications and infrastructure (services) run.

Decentralized infrastructure empowers developer teams to work efficiently, delivering customer-facing features with increased agility. By running applications and infrastructure independently, teams can make changes and upgrades on their own timeline, decoupling their roadmap planning from others. This approach minimizes outages and scaling issues by separating workloads, for example into multiple Kubernetes clusters.

Platform teams play a crucial role in a decentralized developer platform. They maintain necessary centralized services, implement governance functions for security and compliance, and collaborate with developers to build and maintain deployable application patterns. These patterns serve as abstractions, reducing cognitive load for developers and incorporating company policies and integrations. Shifting from operating services to composing managed services benefits platform teams, enabling them to respond quickly to new features and gather user feedback. For developer teams, the provided patterns offer flexibility and the ability to contribute changes back to the platform teams.

Collaboration between developer and platform teams to improve and scale the library of deployable application patterns can be fostered by adopting InnerSource (further information is available here: ix.de/z66z). This practice mirrors open source methodologies and promotes collaboration in developing proprietary software.

## Guiding Principles for a Decentralized Developer Platform

Having introduced the concept of a decentralized developer platform, the following shows a condensed North Star vision of how to produce and consume such a platform. A decentralized developer platform is developed and made available like an InnerSource product:

- Its offerings (sub-products) are versioned pieces of software that platform teams evolve with the user community.

**Contact**

**iX | heise Developer**
Postfach 61 04 07, 30604 Hannover; Karl-Wiechert-Allee 10, 30625 Hannover, Germany
**Phone:** 0511 5352-387, fax: 0511 5352-361, **e-mail:** post@ix.de

**Publishers:** Ansgar Heise

**Editor-in-Chief:** Dr. Oliver Diedrich (responsible for the text section)

**Concept and Editorial Leadership:** Maika Möbus (mai@ix.de) -589

**Editor:** Rainald Menge-Sonnentag (rme@ix.de) -884

**Authors of this Issue:**
Stefan Baumgartner, Robert Hoffmann, Timo Zander

**DTP Production:**
Lisa Hemmerling, Heise Medienwerk, Rostock

**Proofreading and Final Correction:**
Maika Möbus

**Cover Image:**
© Freepik; Montage: Lisa Hemmerling

**Publishing House:**
Heise Medien GmbH & Co. KG,
Postfach 61 04 07, 30604 Hannover; Karl-Wiechert-Allee 10, 30625 Hannover, Germany;
phone: 0511 5352-0, fax: 0511 5352-129

**Managing Directors:**
Ansgar Heise, Beate Gerold

**Members of the Executive Board:**
Jörg Mühle, Falko Ossmann

**Advertising Director (responsible for the advertising section):**
Michael Hanke (-167),
e-mail: michael.hanke@heise.de,
www.heise.de/mediadaten/ix

**Head of Sales and Marketing:**
André Lux (-299)

**Printed by:**
Dierichs Druck + Media GmbH & Co. KG,
Frankfurter Straße 168, 34121 Kassel, Germany

## Advertisers

| | | |
|---|---|---|
| Capgemini Deutschland GmbH | München | 2 |
| Cariad SE | Wolfsburg | 27 |
| DATEV eG | Nürnberg | 9 |
| DLR Deutsches Zentrum für Luft- und Raumfahrt e.V. | Weßling | 15 |
| HENSOLDT Sensors GmbH | Ulm | 21 |
| Lauterbach GmbH | Höhenkirchen-Siegertsbrunn | 5 |
| Red Hat Limited | IRL-Cork | 19 |
| Rosen Technology and Research Center GmbH | Lingen | 11 |
| Software Quality Lab GmbH. | A-Linz | 28 |
| Syskron GmbH | Wackersdorf | 23 |
| Universitätsklinikum Mannheim GmbH | Mannheim | 17 |
| WIBU-SYSTEMS AG | Karlsruhe | 13 |
| ZF Group | Friedrichshafen | 7 |

The page numbers printed here are not binding. Editorial reasons may make changes necessary.

- They are accessible because they are well documented, come with tutorials, community events, open roadmaps, and bug reporting mechanisms.
- Users choose what products to provision in their environment, and the products will not change unless they upgrade or customize it.

Products mainly consist of deployable application patterns for common developer journeys, also referred to as golden paths:

- Deployable application patterns turn into resources (infrastructure, services, workflows) in the developer team's dedicated environments. Most resources are dedicated instances, so deployment is distributed, and centralization is kept at a minimum, mostly for governance functions.
- Resources are often implemented with ready-to-use SaaS and cloud services. The value-add comes through sane defaults and integration with the organization's IT environment (e.g. single sign-on).
- A product is a transparent abstraction in that it will not lock away its implementation, because every abstraction will leak at some point. Products explain how they integrate resources.
- Products can be consumed in a self-service manner without requiring any human intervention.

Thus, the guiding principles of a decentralized developer platform are:

- Versioned
- Decentralized
- User-centered
- Customizable
- Transparent
- Self-service

## Implementing a Decentralized Developer Platform

Implementing a decentralized developer platform requires a number of key technical capabilities (Figure 4) to enable two broad categories of use cases: collaboration (including publishing services and patterns) and consumption. Collaboration involves developers and platform engineers working together to extend and improve application patterns. The InnerSource Commons community provides knowledge for these use cases. This section focuses on consumption use cases, which include provisioning infrastructure for building, testing, deploying, and running services. Developers request one or more managed environments, which are the logical or physical containers to host their resources.

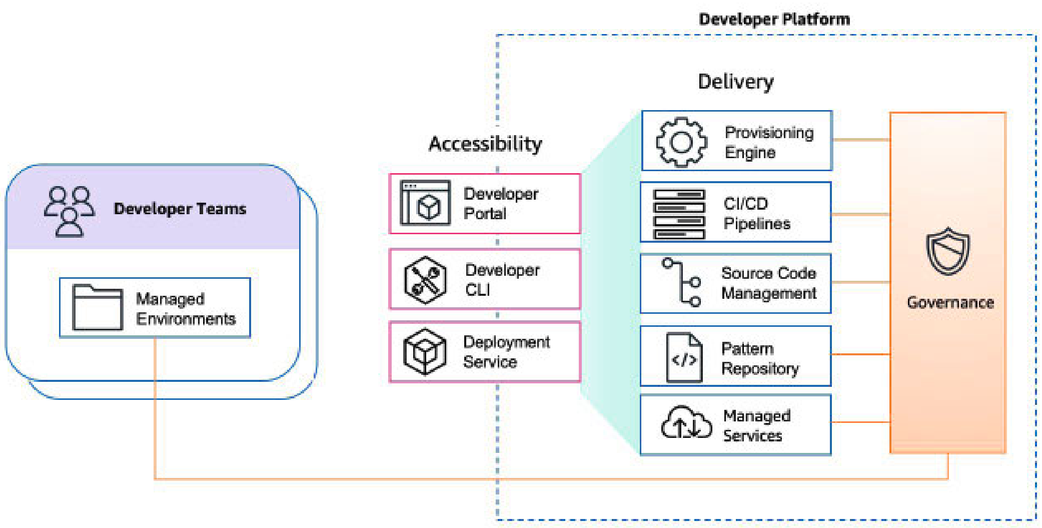
Accessibility capabilities are optional but simplify the exploration, deployment, and interaction with deployable application patterns and underlying services. Examples include developer portals and CLIs that provide search, deployment automation, and documentation access. These interfaces can connect directly with delivery capabilities or be part of a deployment service. A deployment service, also known as an application delivery service or platform orchestrator, simplifies the developer experience of using deployable application patterns. First, a developer can choose a pattern and provide some additional configuration. The deployment service then orchestrates the underlying systems like code repositories, CI/CD pipelines and provisioning engines to deploy the pattern.

The accessibility functions are backed by delivery capabilities to enable self-service provisioning. In order to provision resources, developers apply GitOps, which is defining the infrastructure as code (IaC) and maintaining it in a Git repository. Then, provisioning engines turn the desired state into the deployment and composition of services. This process can be triggered by a CI/CD pipeline. Finally, governance functions ensure compliance and security. They scan environments for vulnerabilities and validate desired states against policies. Preventive and detective guard rails, such as role-based access control and privileged systems, protect against policy circumvention or disabling.

## Key Decisions for Mapping Capabilities to Implementations

For each of the capabilities outlined above, it is possible to adopt one or more tools and services. The range to pick from depends on some key decisions:

1. How are managed services acquired? A common example is choosing a public cloud provider to have a broad set of services available.
2. What is the standard provisioning engine? Typical choices are Terraform, the native engine of the public cloud provider, e.g. CloudFormation for Amazon Web Services (AWS), or the Kubernetes control plane.
3. The previous decisions typically provide information about what kind of governance solutions are employed. For example, cloud services tend to come with a native set of preventive and detective controls, while Kubernetes-based environments are secured with policy engines like the Open Policy Agent.



Source: Amazon Web Services

**>> The technical capabilities of the decentralized developer platform include essential delivery capabilities and—optional, yet highly effective in improving the developer experience—accessibility capabilities (Figure 4).**

4. Next, it is possible to select a deployment service that supports the technology choices made in the previous steps. For example, if managed services are provided by AWS, a natural choice could be its native deployment service, AWS Proton. If the provisioning engine is Kubernetes, a common approach is using the open-source extension Crossplane, which can manage the lifecycle of resources like managed services.
5. The choice for a developer portal and CLI might be predetermined by the selection of the deployment service. Alternatively, some companies opt for not offering such convenience tools, but instead adopt open-source products like Backstage (a platform for building developer portals) or build their own. Typically, these in-house tools provide scaffolding for multiple systems like code repositories and CI/CD pipelines.

## Aligning Developer Platforms With Operational Models

Internal developer platforms help simplify software delivery for product teams through developer enablement and self-service. The allocation of responsibilities between developer and platform teams depends on the chosen operational model. If the organization is practicing decentralized DevOps, the decentralized developer platform is a sensible choice to support that operational model. It lets developer teams consume deployable application patterns, which are versioned and customizable software artifacts. Developer teams turn the patterns into infrastructure running in dedicated environments that they own. Platform teams maintain the patterns but treat them like InnerSource products. Thus, developer teams can build infrastructure and applications independently, while still benefitting from codified, consistent, and maintained best practices and company policies. Realizing a decentralized developer platform requires suitable technologies, for example leveraging the AWS cloud for managed services and Kubernetes as the provisioning engine. Building a proof-of-concept implementation of these capabilities can be a first step towards supporting a decentralized developer platform. Moreover, some of the tools and services can also power more centralized developer platforms—if the chosen solutions feature a sophisticated access management, they enable distributing responsibilities between developer and platform teams in multiple variations. This allows organizations to pick one or more operational models that best fit their particular culture. (mai)

### Sources

Links to the topics discussed in this article are available here: ix.de/z66z

**Robert Hoffmann**

is a Senior Solutions Architect at AWS. Previously, he worked for top smart device and telecommunication brands, pioneering cloud-native applications during the early days of Docker and Kubernetes. At AWS, he is supporting some of the world's largest retail brands on their cloud journey. Robert is passionate about observability, infrastructure as code and developer productivity. You can find him discussing these topics at conferences and on Twitter (@robhoffmax).

**We are united.**
To find the ones who dare to move the future.

**#youmakeitwork**

**Apply now and become part of the team.**

CARIAD

software Quality lab Academy

International Requirements Engineering Board · ISTQB International Software Testing Qualifications Board · iSAQB

Competence for Engineers

## Requirements Engineering

- **IREB Certified Professional for Requirements Engineering Foundation Level (IREB CPRE-FL)**
- **IREB Certified Professional for Requirements Engineering Foundation Level (English, IREB CPRE-FL)**
- IREB Certified Professional for Requirements Engineering Advanced Level: Requirements Elicitation (IREB CPRE-AL: Requirements Elicitation)
- IREB Certified Professional for Requirements Engineering Advanced Level: Requirements Management (IREB CPRE-AL: Requirements Management)
- IREB Certified Professional for Requirements Engineering Advanced Level: Requirements Modeling (IREB CPRE-AL: Requirements Modeling)

## Architecture & Modeling

- **iSAQB Certified Professional for Software Architecture - Foundation Level (iSAQB CPSA-F)**
- **iSAQB Certified Professional for Software Architecture - Foundation Level (English, iSAQB CPSA-F)**
- iSAQB Certified Professional for Software Architecture - Advanced Level: Soft Skills für Software Architekten (iSAQB CPSA-A SOFT)
- iSAQB Certified Professional for Software Architecture Level - Advanced Level: Domain-Driven Design (iSAQB CPSA-A DDD)
- Clean Code

## Agile Methods

- ISTQB Certified Tester - Foundation Level Extension: Agile Tester (CTFL-AT)
- IREB CPRE RE@Agile Primer
- IREB CPRE-AL RE@Agile

## Testing & Quality Assurance

- **ISTQB Certified Tester - Foundation Level (ISTQB CTFL)**
- **ISTQB Certified Tester - Foundation Level (English, ISTQB CTFL)**
- ISTQB Certified Tester - Advanced Level: Test Analyst (ISTQB CTAL-TA)
- ISTQB Certified Tester - Advanced Level: Technical Test Analyst (ISTQB CTAL-TTA)
- ISTQB Certified Tester - Advanced Level: Test Manager (ISTQB CTAL-TM)
- ISTQB Certified Tester - Advanced Level: Test Manager (English, ISTQB CTAL-TM)

## Test Automation

- ISTQB Certified Tester Test Automation Engineer (Deutsch, ISTQB CT-TAE)
- ISTQB Certified Tester Test Automation Engineer (English, ISTQB CT-TAE)

## User Experience

- UXQB Certified Professional for Usability and User Experience - Foundation Level (UXQB CPUX-F)

**QA-Trainings with more than 2,000 participants annually!**

Check your course dates and availability as public trainings, exclusive inhouse trainings or e-learnings:

› **academy.software-quality-lab.com**