API CONFERENCE

Simplifying APIs

API Whitepaper

Stay up to date with the best practices in API design, security, and development and take your API journey to the next level.

Contents

Speeding up your software release cycles

Don’t Let Developer Toil Affect the Business Value of Your Apps

Michael Coté, Staff Technologist at VMware looks into how to address developer toil to speed up software release cycles, make developers more productive, and increase the business value of your apps.

By Michael Coté

It is no surprise that the world today is driven by apps. Organisations all over the world are basing their business goals on their capacity to integrate new apps quickly, and they need to constantly reimagine how they interact with and communicate with their consumers and employees in order to fulfil changing expectations.

To do this effectively, businesses need to operate at a high level of speed and agility. But in many cases this isn’t happening. The phrase ‘move fast and break things’ could not be further from the truth when it comes to how enterprises approach innovation today. In fact, according to Forrester Consulting, 48% of executives [1] say they haven't changed their applications in a year or more.

One reason for this may be that organisations are dealing with a high amount of developer toil and technical debt. Developer toil is the repetitive, predictable, constant stream of tasks that support the creation of software and how it's run, but don't actually affect the features in the software that people use. Or in other words "any activities that don't directly create business value”.

Typically, developers spend too much time on processes that could be automated or even eliminated. Ironically, this toil builds up over time as their organisations prioritise shipping features rather than addressing these problems in their overall software process. While developers can change their software quickly at first, just like debt in real life, if this toil and tech debt are neglected, it takes over and consumes the organisation's ability to grow. This results in long, infrequent release cycles. A feature that seemed simple and once took just 15 minutes now takes weeks, even months to get in front of users.

Conducting a Developer Toil Audit

Anything you can do to speed up your software release cycle will improve the quality, resilience, and business value of your software. Continuously addressing developer toil provides a significant boost to your organisation's software capabilities. For one of VMware’s customers, addressing developer toil reduced the time taken to introduce a new feature significantly – from 400 hours to just 24 by doing a developer toil audit and addressing key factors contributing to this technical debt.

So how can organisations overcome developer toil? One way to do this is through a Developer Toil Audit. This is a systematic process for finding, valuing, and prioritising fixing waste in your software process. First, the process finds wasted time and process debt in how you build and release software. Second, the process then helps you justify delaying working on features in favour of eliminating and automating your software creation process. The result is speeding up your software release cycle. The more frequently you can change and release software, even with just small changes, the more opportunities you must learn what works and put those features in front of customers, employees, and other users.

The process of conducting a Developer Toil audit includes:

1. **Asking the right questions:** A great way to uncover developer toil is to ask about people's struggles, frustrations, and even boredom in the form of a tailored question set. We advise using questions that are specific to your organisation in addition to general questions like "how long does it take to perform a full build?". For example, highly regulated organisations should ask about tasks involving compliance. Once the survey is completed, you can do some quick analysis to locate and prioritise developer toil.
2. **Combine into usable metrics:** You now have a single metric for each type of developer toil. And by combining all the questions you create a single metric to track overall developer toil. Extra points for creating dashboards with visualisations! As with all such aggregate metrics, these are more directional than exacting - their job is to point you to problems that should be solved.
3. **Link these metrics to business value:** A simple ranking of developer toil is better than nothing. However, before deciding which developer toil to address, we recommend linking each type of developer toil to business value created by fixing the toil. Put briefly, the value you get will be related to the time saved, and, thus, the ability to ship more features in the future.

The less time developers spend on toil, the more time they can spend on tasks that directly benefit the business. Another way of looking at this is plain old productivity: developers can now do more with the same amount of time.

Another important outcome is increased staff morale. The less time developers spend on boring, repetitive work – toil – the happier they'll be. Stronger employee hiring ability and retention are, or should be, a strategic imperative for any organisation that depends on software. Morale also increases software quality: happier people make better software.

Slow Down to Speed Up

With the survey results in hand, you should have a pretty good idea of which developer toil to fix. The last step is to do the usual product management prioritising to weight fixing these items with other tasks you could do. These are strategic decisions that product managers need to make. To fix toil, you need to stop shipping features to free up time. You need to slow down the business. At the start, before product managers have been empowered to make these kinds of decisions, higher level management should be involved to calibrate how much slow down you're willing to put up with for future agility. The calibration you're doing is this: if I fix one item of toil and ship just one feature (instead of two) this release cycle, then in the next release cycle I can ship four features. Humans are not great at that kind of bird in the hand versus birds in the bush thinking so you'll need to figure out what works in your organisation.

Through our experience as consultants working with product teams in different industries, we've used Developer Toil Audits to focus and motivate product managers and developers to fix toil and speed up their release cycles and, thus, improve how their organisations build software. This directly improves the business by adding more capabilities and capacity to deliver more features, even in the short-term. Each time you fix any technical debt, you gain more capacity and capabilities to create new features and improve your software. This is how you use a modern software culture to increase business agility. Or, put more simply: less developer toil leads to better software, and better software leads to better business.

Michael Coté, Staff Technologist, VMWare, studies how large organizations get better at building software to run better and grow their business. His books Changing Mindsets, Monolithic Transformation, and The Business Bottleneck cover these topics. He's been an industry analyst at RedMonk and 451 Research, done corporate strategy and M&A, and was a programmer. He also co-hosts several podcasts, including Software Defined Talk. Cf. cote.io, and is @cote on Twitter.

Links & Literature

[1] https://www.vmware.com/cio-vantage/articles/customer-experience-starts-with-apps.html

12 Reasons Why Developers Should Consider Apache Pulsar

Streaming technologies unlock the ability to capture insights and take instant action on data that's flowing into an organization; they're key to developing applications that can respond in real-time to all kinds of events, like user actions or security threats. In other words, they're a key part of building great customer experiences and driving revenue.

By Chris Bartholomew

We built our own service DataStax Astra Streaming [1] on Apache Pulsar [2], so this is a great time to take a deeper look at the benefits of this open source technology.

What is Pulsar?

Pulsar is a cloud-native messaging and streaming platform that was open sourced by Yahoo in 2017. Since then, Pulsar has grown in popularity, from 10,000 stars and over 450 contributors on GitHub to a Slack community with more than 5,000 members, Pulsar has taken off with phenomenal usage by enterprises like Verizon Media, Yahoo! Japan, Tencent, Comcast, and Overstock.

Pulsar has become the platform of choice to manage hundreds of billions of events every day.

This project is a cloud-native, distributed, open-sourced, pub-sub messaging and streaming platform. It originated as an event broker from Yahoo! in 2015 and it was contributed to the Apache Software Foundation (ASF) as a top-level project in 2017.

Pulsar is interesting as it was designed to be a horizontally scalable distributed system running on commodity hardware that reliably streams messages without losing data. It was originally designed to support Internet-scale applications such as Yahoo! Mail and Yahoo! Finance. As it was designed for these large implementations, it was built to handle the most demanding data movement use cases out there.

So why should you consider Pulsar for your application streaming requirements? Here are 12 reasons why there is a high level of interest and momentum in Pulsar, and why every enterprise should be using it.

Reason 1: Messaging, streaming, and queuing, all-in-one

The first reason to consider Pulsar is that it can cover more than application streaming use cases. Pulsar can be compared to other leading, traditional pub-sub messaging system options like Apache Kafka [3], and to messaging queuing systems like RabbitMQ [4] or ActiveMQ [5].

Kafka is the de facto standard for streaming use cases, and for good reason: it is great at streaming and pub-

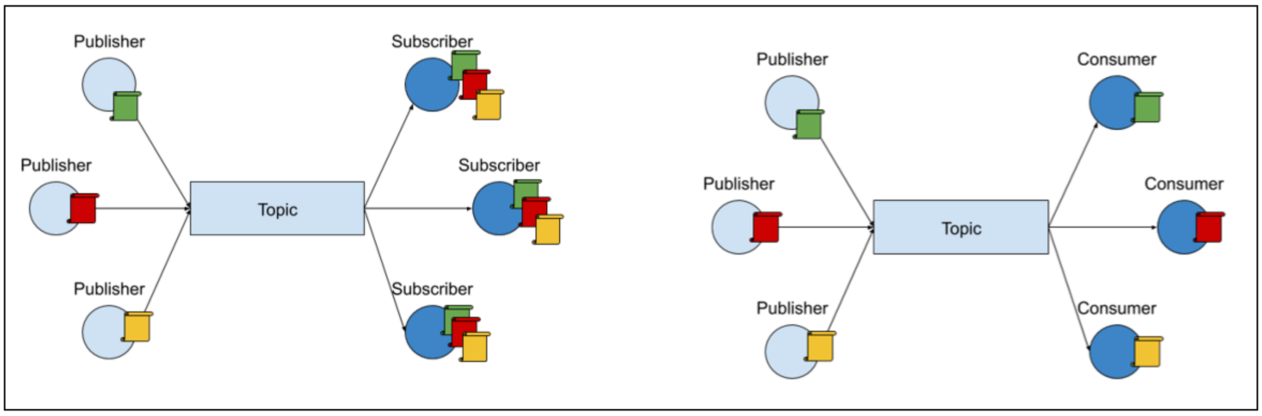


Figure 1: Streaming (left) and Queuing (right) with Apache Pulsar

sub and at delivering messages to multiple consumers. In **Figure 1**, multiple publishers are publishing to a topic, and the same message is sent to multiple consumers.

RabbitMQ or ActiveMQ are great at queuing messages and competing consumer use cases. This is where we are publishing publishers and sending messages into the topic, but only one consumer is consuming each of these messages. It's trickier to accomplish competing consumer use cases in Kafka, because it works on the partition level and you can end up with extra partitions you don't need or consumers that don't consume any messages.

Pulsar combines the best features of a traditional messaging system like RabbitMQ with those of a pub-sub system like Kafka. You get the best of both worlds in a high performance, cloud-native package. Rather than having to run multiple different systems to cover these use cases, Pulsar can cover them all from one deployment. This simplifies the infrastructure side for developers, while also making it easier to scale over time.

Reason 2: Performance

Pulsar was designed for high performance and to achieve hundreds of thousands or even millions of messages per second. A key consideration for Pulsar's original design was that it needed to have less than 10 millisecond producer latency. When you publish a message, you receive acknowledgement within 10 milliseconds consistently. Pulsar's architecture is optimized to get high throughput and low, consistent latency.

Pulsar also supports up to millions of topics. This is something that Kafka struggles with, so it can be more suitable for use cases where the volume of topics supported is high.

In a third-party, vendor-neutral analysis by Software Mill, Pulsar was rated as a high-performance messaging platform in their report issued in July 2021 [6]. It is considerably faster than traditional messaging systems and can hold its own with the pub-sub crowd.

Reason 3: Modernize legacy applications

Pulsar's flexibility makes it easy to modernize legacy applications. This will be a significant trend over the coming year where developers will need to think about their future approach. According to research by IDC [7], 86% of developers said their organizations had modernized more than 50% of their legacy applications in 2021, up from 65% in 2020. Similarly, O'Reilly research in December 2021 [8] found that about 25% of developers said that their companies planned to move all of their applications to the cloud in the coming year.

Pulsar was designed to process message queuing exchange patterns, which means it can support older enterprise applications written for RabbitMQ or Java Messaging Service (JMS) without rewriting them. If you have existing legacy JMS applications, you can do a drop-in replacement by switching your broker type in your application to Pulsar using DataStax's Starlight for JMS [9], turning your Pulsar cluster into a JMS 2.0 compliant broker.

Similarly, if you have legacy RabbitMQ applications, you can use DataStax's Starlight for RabbitMQ to turn your Pulsar cluster into a RabbitMQ-compatible broker.

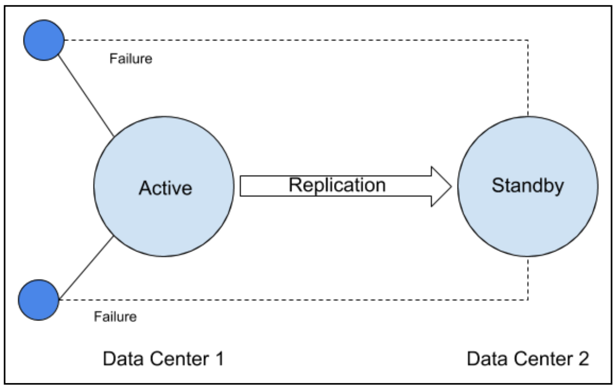


Fig 2: Geo-replication on multiple topologies

This saves costs by consolidating the brokers down into a single large horizontally scalable Pulsar cluster. You can then write new applications using event driven architectures and more modern techniques that live together on the same platform.

This approach makes it easier to migrate your old applications into the cloud, and also take advantage of Pulsar features like message retention and replay.

Reason 4: Multi-tenancy to support different teams

Once you have a high-performance, scalable messaging system in place, you will want to share it between different teams and groups within your organization. It doesn't make sense to have to replicate the system to make sure different teams don't impact each other or build a complex overlay system to simulate multi-tenancy.

Multi-tenancy is the ability for different user groups to use the same underlying resources in a fair way. Pulsar can limit the amount of resources different tenants in namespaces have access to. You can set the maximum number of producers and consumers and maximum rates on how much storage each consumer or topic can have. Unlike Kafka, you don't have to build an entire multi-tenancy overlay manually or spin up a whole cluster for a new user group.

Reason 5: Geo-replication for data recovery

Another feature that's built into Pulsar is geo-replication which you can easily manage through Pulsar CLI or REST API, without the need to install another package on top. Geo-replication (**Figure 2**) is key to recover your data during disasters or enhance the performance of your application.

Pulsar supports multiple topologies that replicate data from the active data center to the standby data center. If the active one fails, you can reconnect to the standby data center. You can also have a more complex topology where you publish a message in one data center in North America and consume it in a different data center in Europe. Basically, you can have an entire global message bus by using the geo-replication feature.

Pulsar's global configuration store also allows you to standardize the policies, namespaces and data centers, store them in a central location, and propagate the data to all data centers automatically.

Another built-in feature, replicated subscriptions, is ideal for disaster recovery scenarios. In case of failure, a consumer can restart consuming from the failure point in a different cluster.

Reason 6: Kubernetes and cloud-ready architecture

According to the SlashData Developer Economics research report for the Cloud Native Computing Foundation (CNCF) [10], Kubernetes is continuing to grow in popularity, with 57% of backend developers using containers in the last 12 months. Larger enterprises in particular are adopting Kubernetes faster, with 78% of organizations using the technology.

This means that developers have to look at how their applications can support Kubernetes and cloud-native deployments. Pulsar was built to support this approach from the start. Because Apache Pulsar uses a multiple layer approach, separating where consumers connect (brokers) from the storage layer (BookKeeper), it fits very well into cloud infrastructures, which also separate these two concerns. (**Figure 3**) Without having to expand both storage and computing at the same time, you won't be paying for compute or storage you don't need.

Apache Pulsar works naturally in Kubernetes, supporting rolling upgrades, rollbacks, and horizontal scaling. When coupled with persistent volumes backed by cloud storage with configurable performance dimensions, Pulsar is a highly durable and highly flexible messaging system that can scale from small test deployments to large production deployments with ease.

Pulsar also has a proxy component that solves some of the Kubernetes networking challenges that you can

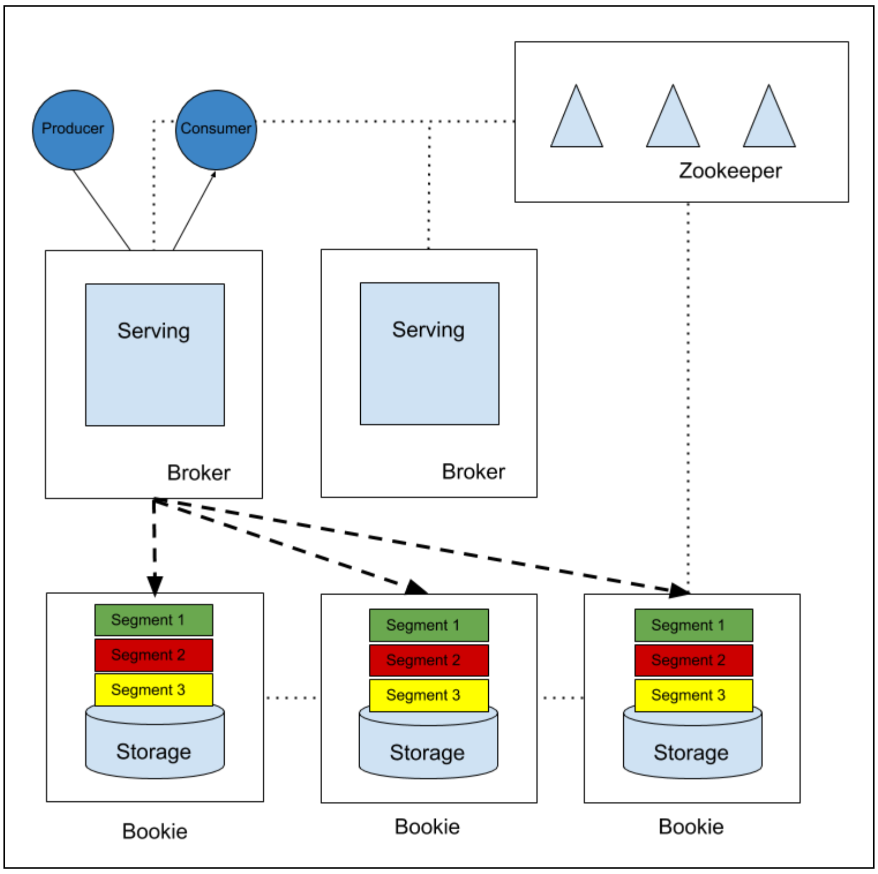


Fig. 3: Pulsar's separation of BookKeeper layer and broker layer

have with systems like Kafka. With the growing popularity of Kubernetes, Pulsar continues to develop and work with these innovations.

Reason 7: Easy scaling, up and down

For developers, scaling application infrastructure up and down is a task they would rather avoid. Typically, developers will deal with this by estimating the amount of resource their applications will want over time, and then adding a buffer to cope with any peaks in demand. This adds more expense, as well as being a management task to track. Developers will also have to manage adding resources to deal with growth over time too.

Apache Pulsar is built to make it easier for enterprises to scale their cluster up and down. In other systems, when you scale up, you have to work to redistribute and rebalance the load between brokers. Pulsar actively monitors the resources on your broker CPU memory network and automatically redistributes the load when it's overloaded. And you can scale down brokers just as easily, letting Pulsar automatically redistribute the load. What if you need more storage? Just add more BookKeeper nodes.

Whether you need to add more storage or need more throughput, Pulsar handles everything automatically with no manual partition rebalancing or long maintenance periods required. This simplifies the experience for developers, letting you concentrate on applications rather than infrastructure.

Reason 8: Support for better cost management with tiered storage

Developers are more concerned about the long term cost to run their applications in the cloud. According to research by the FinOps Foundation and CNCF [11], 68% of respondents found their spending had gone up over the past year, leading to overspends and budget pressures. Finding ways to benefit from cloud native approaches while managing budget more effectively will be a concern.

This means looking at smarter ways of managing application infrastructure over time, so that costs are controlled more effectively. A good example of this is using tiered storage. This involves shifting older data from more expensive cloud resources to less expensive options, such as transferring older messages stored on Apache BookKeeper on high-speed, high-performance SSDs and moving them into lower-cost storage options, like Amazon S3, Google Cloud and Azure Blob Storage.

Pulsar supports this automatically without requiring direct developer intervention, and it's all transparent to the client. Tiered storage is really helpful if you want to store a lot of events in Pulsar at little or no cost. Instead of spending premium dollars in storage, you can save a significant amount of money on long term data with Pulsar.

Reason 9: Fully open source approach

Pulsar is available for free under the Apache Software Foundation without needing additional vendor proprietary code or projects for functionality. This makes it easier to adopt and benefit from Pulsar, as well as helping the project to expand over time.

When projects launch as open source under the control of one vendor organization, there is a risk that licensing terms will change in the future. This can involve adding new restrictions for use, or changing the project to a closed-source license. However, because Pulsar is under the Apache Software Foundation, this organization will not change the licensing terms to be more restrictive. Pulsar is open source today, and will be open source tomorrow. More importantly, Pulsar has all its functionality covered by one license so it is one project, rather than requiring extensions to cover common use cases.

Reason 10: Pulsar functions and IO connectors for easy connections

Pulsar has a framework for lightweight stream processing. This means you can use Pulsar functions which are fully integrated with the Pulsar Command Line Interface or API to carry out additional tasks. Example use cases for this would include cleaning and enriching your data as part of the process, as well as routing events and writing functions in Java, Python and Go.

You can also get your data in and out of Pulsar using IO connectors without actually writing any code. This involves configuring the connectors and it will start sinking and sourcing the data using built-in connectors such as Elasticsearch, MySQL, Postgres and RabbitMQ.

Reason 11: Schema registry to prevent data incompatibility

When sending messages between producers and centers that are decoupled, which is always the case in a messaging or streaming platform, you want to make sure that they can agree on the format of the data. If this agreement is not in place, then those application components cannot communicate with each other.

This becomes very important if you're running tens or hundreds of microservices application components. Pulsar uses a built-in schema registry that supports Avro and JSON schemas, which allows producers and consumers to register or learn the schema of the data sent on the topic. This approach prevents data compatibility issues.

You can also enforce particular schemas on specific topics or change your schema over time to make sure that your application components communicate effectively with each other, and that they are both backwards and forwards compatible.

Reason 12: Protocol compatibility with Kafka

One of the developments around Apache Kafka is how it can connect to multiple other components using Kafka Connect, allowing it to bring in data and export it out. By supporting Kafka Connect as a protocol, other services can connect with systems that support Apache Kafka without changing anything. Kafka Connect is becoming a de facto standard for pubsub and messaging services, but it can also be used by other services.

There are multiple projects working towards protocol-level compatibility with Kafka, including DataStax's Starlight for Kafka [12] project. These projects add features like multi-tenancy, geo-replication, and Kafka to Pulsar interworking. For developers that have invested in Kafka previously, these projects will make it easier to either add Pulsar to applications where it makes sense, or to replace Kafka with Pulsar over time. This can be useful for saving on costs around application infrastructure over time. Pulsar has a lower total cost of ownership when you combine its features, such as geo-replication, tiered storage, performance and supporting multiple use cases with the same project. It's a great cost saving option for developers dealing with complex scenarios and high data volume.

For example, when Splunk replaced Kafka with Pulsar, they found that its CapEx costs for servers and storage were reduced by 1.5-2 times and its OpEx costs decreased by 2-3 times. GigaOm also reported in 2021 that Pulsar offers up to 81% lower cost in 3 years compared to Kafka [13].

Overall, Apache Pulsar supports a wide range of use cases around application streaming and messaging. The project fits with how developers want to build and deploy their applications.

Chris Bartholomew is Engineering Lead at DataStax. He is responsible for the company's work around application streaming and support for Apache Pulsar. Chris previously founded a cloud messaging company based on Pulsar and provided support for some of the largest deployments using this open source project. He is the author of the O'Reilly ebook: Apache Pulsar Versus Apache Kafka: Choosing a Messaging Platform.

Links & Literature

[1] https://www.datastax.com/products/astra-streaming

[2] https://pulsar.apache.org/

[3] https://kafka.apache.org/

[4] https://www.rabbitmq.com/

[5] https://activemq.apache.org/

[6] https://softwaremill.com/mqperf/

[7] https://www.idc.com/getdoc.jsp?containerId=prUS48058021

[8] https://www.oreilly.com/radar/the-cloud-in-2021-adoption-continues/

[9] https://www.datastax.com/blog/fast-jms-apache-pulsar

[10] https://www.cncf.io/wp-content/uploads/2021/12/Q1-2021-State-of-Cloud-Native-development-FINAL.pdf

[11] https://www.cncf.io/blog/2021/06/29/finops-for-kubernetes-insufficient-or-nonexistent-kubernetes-cost-monitoring-is-causing-overspend/

[12] https://github.com/datastax/starlight-for-kafka

[13] https://www.datastax.com/gigaom-pulsar

Principle steps

Logic-based threats and how to combat them

APIs bring tremendous value to businesses in every sector. API traffic is growing exponentially at 30% year-on-year. However, the security model for APIs is problematic. What are the issues facing CISOs trying to secure their APIs? In this article, you will learn about the current issues and how to tackle them.

By Denis Jannot, Chuck Herrin

The move away from the older, monolithic compute methodologies of the past towards the microservices-centric, API-based environments of the future is what characterises the modern enterprise. With digital transformation in everyone's sights, organisations are rapidly adopting APIs for many good reasons.

APIs bring tremendous value to businesses in every sector. They're great for promoting collaboration and partnership and allow a kind of 'mashup mentality' which has freed DevOps from more restrictive legacy architecture models. API traffic is growing exponentially at 30% year-on-year – accounting for an estimated 83% of web traffic today.

The security model for APIs, however, is problematic. In 2017, Gartner predicted APIs were going to become the number one attack vector by 2022, a prediction they had to revise in late 2021 as they saw the landscape explode far beyond expectations.

APIs – a novel attack surface

What are the issues facing CISOs trying to secure their APIs? Firstly, adoption is by far outpacing security. The rapid, iterative nature of API and endpoint development means code is routinely updated multiple times per week, often multiple times per day. This makes manual API security testing incredibly difficult.

Secondly, modern API attacks are mostly logic-based. They can't be defended by traditional rules-based systems like WAFs (Web Application Firewalls), or gateways set up to monitor traditional threats, and legacy code scanning tools lack the context to chain together the paths leveraged in logic-based attacks. Traditional defences can't protect businesses from this new attack vector.

How significant is this problem? Roughly 50% of all APIs in customer organisations are unmanaged [1]. This is a huge blind spot and a major obstacle for security teams who find the proliferation of highly distributed compute environments often outpaces the ability to secure them.

Not hacking, exploiting inherent insecurities

You can't defend what you can't see, and when you change from monolithic to microservices-based architectures, we naturally expect microservices to do more than they ever have before – become more portable and scalable. This, potentially, makes applications more vulnerable, and materially changes the attack surface exposed to the outside world. This is why it's important to look at governance tools – modern API gateways, for example, where you can enforce policies consistently across your API and mesh architectures. That's not the end of it, though, APIs still expose application logic which makes them vulnerable to attack.

Most API attacks are not the familiar injection-based kind, and many aren't hacks to begin with. A good example is the extraction of tens of millions of user records through the Facebook mobile app. In this instance, the assailant didn't crack any encryption keys, didn't hack a password, and didn't attempt any SQL injection. The API logic allowed it and the third party took advantage of it – it was an unsanctioned usage of that API, but it wasn't really a 'hack'.

This happens a lot – attackers figure out the API and uncover inherent issues in the business logic. By exploiting failures such as broken object level authorization (BOLA) and broken functional level authorisation (BFLA), they don't need to hack the APIs themselves. What we realise, therefore, is our defences must be different. Our WAFs and gateways can't (on their own) defend against logic-based threats.

Adopting a holistic view to API security

What can we do to protect APIs before attacks damage our businesses? The problem is widespread, and still relatively new. The market is moving so quickly that Gartner recently modified its reference architecture to include API security as a dedicated layer in the stack. To give a sense of how many companies are on top of this issue, only 11% have a full-blown API security program [2] in their organisations. So, it's a relatively new challenge for everyone.

There are several elements that combine to solve API security problems. Firstly, it's about gaining proper visibility of the traffic running through the stack. Starting with your traffic analysis and traffic inspection to watch what's going on within the stack is critical. It's not enough to observe traffic, though, you must go beyond just tracking inspection to really get into the code.

Code analysis provides the chance to not only do your standard SAS testing for API vulnerabilities, but also provides us with a useful lens through which to view third party APIs and endpoints being called from the code base. Wib's Code Analyzer also creates a logic flow model to understand how software components interact with each other that legacy SAST tools miss. This is particularly important in large enterprises of distributed teams working on different products; closing these blind spots and applying different lenses to the data is essential. Wib, for example, provides a lens at the bottom of the VCR compliance analyser which brings visibility to a company's compliance obligations. Managers have line of sight into which APIs serve PII, which are in scope for PCI and credit card compliance.

The third piece of the puzzle is attack simulation to identify real attacks against your APIs and endpoints, for two reasons. One, it finds current vulnerabilities and exposures you weren't aware of. Two, it helps eliminate false positives before your security team or your DevOps team starts issuing a slew of remediation requests. So, how do we get there?

Practical Steps

In a microservices-based framework, application teams want to mitigate the complex authentication and authorization headache inherent in the model. API Gateways have been through a period of evolution in recent years, particularly as the adoption of Kubernetes has hastened. API Gateways, such as Solo.io's Gloo Edge, allow companies to implement all these mechanisms at the Gateway event. In a modern API gateway, teams can secure web applications with API keys or geo tokens and can put additional defences in place to prevent API abuse and block common threats. Gateways can also be used to expose threats in legacy VMs and bare metal – particularly important for many of the core datacentre systems still relied upon by the largest companies.

Finally, before selecting an API gateway, it's worth thinking about whether adding a service mesh is part of the long-term plan. Choosing a gateway based on Envoy will make it significantly easier to adopt a service mesh (based on Envoy) – providing the capability to manage the complexities of modern, API-centric businesses. The adoption of Kubernetes and Istio in the enterprise in particular can make digital transformation simpler for a greater number of organisations.

Denis Jannot is the Director of Field Engineering at Solo.io, a company building application networking solutions for the edge and service mesh. Denis is a passionate engineer who has spent his career in technical roles working directly with customers and users in architecting and adopting technologies like Object Storage, Big Data, Containerization, Service Mesh into their infrastructure. He enjoys sharing what he learns with the community and can be found creating demos, writing blogs, and speaking at events.

Chuck Herrin is Wib's CTO, bringing with him over 18 years as the CISO of multiple global financial services firms with operations in over 100 countries. Mr. Herrin has broad and deep experience partnering with business leaders, board members, and other stakeholders to achieve their goals, enhance risk management discipline, improve transparency, and deliver transformational change.

Links & Literature

[1] https://www.gartner.com/en/documents/4009103

[2] https://www.prnewswire.com/news-releases/salt-security-state-of-api-security-report-reveals-api-attacks-increased-681-in-the-last-12-months-301493728.html

Who should have access to a resource?

Security Shifts to Identity

In the eight years since we first launched the annual research that would become the State of Application Strategy Report, we've seen the steady rise of security to the top of the app security and delivery services stack.

By Lori MacVittie

Availability as a general category, comprising technologies like load balancing and caching and CDNs, maintained the highest priority for about as long as it took for a newly launched web server to survive unmolested on the Internet in 2003. Which, for those unaware, was not very long.

Security shot to the top of the stack and has remained there, unchallenged since about 2017—until now.

This year, for the first time, we saw a non-core security service rise to the top of the "most deployed." That service is **identity**.

But it's not just that identity and access have risen to become the most deployed technologies. There is ample evidence throughout our research that points to a significant shift toward identity-based security.

Consider API security. Yes, people are deploying it. But we dug into the details and asked about specific types of protections that respondents considered valuable. We grouped them loosely into three categories:

1. **Traditional.** These protections derive largely from the web-based protections included in web application firewalls for years. Rate limiting, OWASP Top Ten, and of course encryption/decryption.
2. **Modern.** These protections have emerged in the past few years and risen as a significant source of security for APIs. This group includes payload (content) inspection like seeking out malware and malicious content and authentication/authorization. Spoiler: that's the identity part.
3. **Adaptive.** Adaptive protections are a new category, fuelled by the ability to leverage AI and machine learning to perform behavioural analysis that can differentiate between human and non-human users. These techniques tend to form the foundation for anti-fraud and bot protection services.

We asked what respondents considered the most valuable protections on this list. The results showed a high degree of security sophistication, especially among those who had implemented API protections in the past year. As with deployment of services, identity was

„At the heart of zero trust is a simple question: who should have access to a resource?"

at the top of the list of most valuable protections for APIs.

The value placed on adaptive methods is promising. That's not entirely surprising given the eager embrace of AI and machine learning to fuel security services. Given the volume of data and the impact of missing an attack, it's no surprise that the entire industry is turning to more advanced and adaptive methods of security to protect everything from infrastructure to applications to the business itself.

Both identity and behavioural analysis are important parts of a comprehensive security strategy, especially for APIs given the role they increasingly play in powering the digital economy. Inspection remains key as well, as many attacks—particularly malware and malicious content—are often easily identified by a unique signature that can be matched against the payload of an API transaction. Speed of identification is as important as confidence in identifying a possible attack, and inspection remains a quick and reliable method of identifying malicious content.

Lastly, we see identity-related technology deployment as a result of COVID-accelerated digital transformation. We asked respondents what kinds of changes were being made to their security strategies post-COVID. More than one-quarter (26%) have implemented a credential stuffing solution and 34% have implemented API security frameworks.

That first number is the relevant one to this topic, as credential stuffing [1] is all about protecting the identity (credentials) of people in a digital world. Given the incredible rise in digital options for every kind of business over the course of the pandemic, it's heartening to see at least some taking their responsibility to protect identity seriously.

This is a relatively fast-moving trend in security, and we expect it will continue to become more pervasive as organizations further expand their presence in the digital economy. The importance of APIs foretells a need to identify more accurately the 'user' of APIs, especially with the growing importance of APIs in automation, cloud-native application architectures, digital ecosystems, and, of course, IoT. Protecting APIs in a digital economy is not just a technology concern but a business one as well.

But the trend also indicates how important identity is in a digital world, and why it's only somewhat surprising to see identity-related services rise to the top of the most deployed application security and delivery technologies in 2022.

This shift toward identity revealed by our research [2] is also significant as the market embraces zero trust as a foundational approach to security. Zero trust was named by 40% of respondents as the "most exciting" trend or technology. As an architectural model, zero trust focuses on securing and protecting applications and infrastructure [3] by designing networks with secure micro-perimeters and limiting risks by restricting user privileges and access.

At the heart of zero trust is a simple question: who should have access to a resource? While there's definitely a lot more that goes into answering that question and enforcing resulting policies across core, cloud, and edge, without identity the entire approach falls apart.

Whether identity stays top of mind remains to be seen but, given that emerging trends like Web3 also place a heavy emphasis on identity as a core construct, we think it is likely that the shift in security toward identity is just beginning.

Lori MacVittie is the Principal Technical Evangelist, Office of the CTO at F5 Networks

Links & Literature

[1] https://www.f5.com/services/resources/glossary/credential-stuffing

[2] https://www.f5.com/state-of-application-strategy-report

[3] https://www.f5.com/services/resources/white-papers/why-zero-trust-matters-for-more-than-just-access

Making data easier to work with

How Data Gateways Simplify Developers' Lives

Developers want to concentrate on what they do best: building applications; and they want to reduce tasks that pull them away from development. Data gateways make it easier for developers to work with data, without getting bogged down in the details of managing schema, data migrations, and other database administration tasks.

By Jeff Carpenter

According to research by SlashData and the Cloud Native Computing Foundation, there are around 6.8 million developers working with cloud native technologies, which equates to around 41 percent of all back-end developers today. They're keen to use new approaches like containers and Kubernetes, as they can move faster and implement their applications quickly.

Data is quickly becoming more and more critical to these applications. But developers want to spend less time installing and managing databases; they prefer to have these instances managed by others. According [1] to Gartner, spending on cloud database management systems (DBMS) will be 50% of the total database market by 2022.

Why is this shift taking place? Developers want to concentrate on what they do best: building applications; and they want to reduce tasks that pull them away from development.

Another example of this shift is the trend of teams creating their own API layers (known as data gateways) that sit between application code and databases. Data gateways make it easier for developers to work with data, without getting bogged down in the details of managing schema, data migrations, and other database administration tasks.

How data gateways work in practice

To learn more about data gateways, let's consider the reasons why developers create these abstractions. One common use case is to mask changes to the underlying database over time.

For example, consider the changes that occurred when the Apache Cassandra® project changed its primary interface from Thrift to Cassandra Query Language (CQL). This change represented a major upheaval for developers that had built multiple applications or microservices on the Thrift API. Some of these teams elected to introduce abstractions that implemented the Thrift API, translated incoming Thrift messages into CQL invocations on Cassandra, and returned the results as Thrift messages. This approach preserved the previous investment in application development, at the cost of maintaining abstraction layers over time.

These abstraction layers provide a level of self-protection for developers: by using a utility or tool that meets their needs, they can avoid some of the infrastructure management problems and pitfalls that would otherwise arise. However, these layers need to be supported over time to avoid technical debt building up. They also should be evaluated alongside new approaches to avoid any sunk cost fallacies that might affect project roadmap decisions, and the opportunity cost associated with failing to adopt new features that could improve performance or reduce cost.

Generalized API data gateways are now emerging to provide a more uniform approach to abstracting database infrastructure. Rather than internally developed and managed software, external open source projects are proving easier to adopt and use over time. These projects enable a second common use case, which is potentially even more valuable than the "self-protection" approach described above: enabling rapid development of applications that support massive scale.

How Stargate's APIs simplify development

As an example, the open source project Stargate [2] makes it easier to deploy Cassandra databases and interact with data using developer-friendly APIs. Instead of being forced to invest in learning yet another database-specific query language, developers can use APIs that they are familiar with, such as GraphQL, REST, gRPC, or JSON documents.

While some developers have deep experience with open source databases and are able to work with a variety of different schemas, others will not have that same level of expertise. This can make it hard to create and adapt schemas that will promote high performance at scale. One aspect of Stargate that is especially intriguing is its flexible approach to managing schema. Stargate provides APIs that support both "schema-first" and "API-first" approaches. For example, the Stargate Document API completely abstracts the underlying CQL of the Cassandra database, providing a schemaless "API first" approach for storing JSON documents.

As another example, the Stargate GraphQL API supports two different types of schema-first development. The "CQL-first" approach allows access to existing Cassandra tables defined via CQL. This requires familiarity with how the database works. Once the database is up and running and the schema defined, the data gateway can plug into this and then serve up data through the GraphQL API.

Alternatively, when the developer team just wants their database instance up and running, the data gateway can handle this process for them – without requiring them to know the best approach to the underlying database schema in that particular situation. This "GraphQL first" approach defines a standard set of API queries based on a simple GraphQL schema provided by the developer, and then the data gateway automatically works on how best to deploy the database and create the underlying CQL schema to implement the developer's intent.

Regardless of skill and experience levels across the team, the data gateway can simplify the process of schema creation and ongoing management to ensure a consistent approach. In turn, this results in more efficient queries, improving performance for the whole application.

Data gateways: Value, scale, and speed

Businesses are using more and more data in their operations in order to meet customer expectations, so developers will continue to need to learn new skills to build data-intensive applications. Developers can meet these expectations and execute with speed by thinking more in terms of data services, as opposed to traditional databases. Using a data service via an API is easier than specifically developing for one database or another. Instead, the service should provide the information that the application needs, on-demand.

The data gateway abstracts database infrastructure; it interacts with the database to carry out requests, and then presents that information back to the app. Using an open source data gateway rather than designing new abstraction layers for each application reduces time to market and management overhead for the team.

By combining a data gateway with APIs and cloud native computing, developers can automate the majority of their data infrastructure. This can make scaling up and down in response to demand easier and faster compared to traditional approaches. With more and more applications going cloud native, data gateways are becoming an essential part of how developers achieve value and scale quickly.

DataStax Astra DB [3], a DBaaS built on Cassandra, includes the open-source data API layer Stargate.

Jeff Carpenter is a Software Engineer working on the open source Stargate project, and involved in Developer Adoption at DataStax. He leverages his background in system architecture, microservices and Apache Cassandra to help empower developers and operations engineers build distributed systems that are scalable, reliable, and secure.

Links & Literature

[1] https://www.gartner.com/doc/reprints?id=1-28F8N1L2&ct=211213&st=sb

[2] https://stargate.io/

[3] https://www.datastax.com/products/datastax-astra/apis?

The right algorithm in the right place – binary search and sorting algorithms

Scalable Programming

Java continuously introduces new, useful features. For instance, Java 8 introduced the Stream API, one of the biggest highlights of the past few years. But is aggregating data with the Stream API a panacea? In this article, I'd like to explore if there's a better alternative for certain cases from a complexity perspective.

By Ikuru Otomo

Some of you have probably used the following code in a program in order to spontaneously measure a logic's runtime:

```
long start = System.currentTimeMillis();
doSomething();
long time = System.currentTimeMillis() - start;
```

Clearly, it's easy to implement and you can quickly check the code's speed. But there are also some disadvantages. First, the measured values can contain uncertainties as they can be influenced by other processes running on the same machine. Second, you can't compare the readings with other readings taken from different environments. Declaring that one solution is faster than the other isn't helpful if they were measured on different machines with different CPUs and RAM. Third, it's difficult to estimate how the runtime could extend when working with larger amounts of data in the future. It's become much easier to filter and aggregate data since the Stream API was introduced in Java 8. The Stream API even opens up the possibility to parallelize processing [1]. But do these solutions continue to perform when you need to work with 10 or 100 times the amount of data? Is there a measurement that we can use to answer this question?

Time complexity

Time complexity is a measurement for roughly estimating the time efficiency of an algorithm. It focuses on how runtime increases as the input gets longer. For example, if you iterate a list of n elements with a *for* loop, then n and the runtime have a linear relationship. If you have multiple *for* loops nested and executed n times each, then this logic has an exponential effect on runtime.

Big O notation is a way to represent the relationship between the input length and the runtime. A linear relationship is represented by $O(n)$, $O(n^2)$ represents a quadratic relationship, where n is the input's length. If the runtime is independent of the input's length and is constant, then we write $O(1)$. **Figure 1** shows typical big O notation values for how the runtime grows as the input's length increases.

There are two important rules for representation using big O-notation:

Only the term with the highest degree is considered. For example: If the time complexity is $n + nlogn + n^2$, simply write $O(n^2)$, as the term n^2 has the strongest effect on runtime.

The coefficient is not considered. For example, the time complexity of $2n^2$, $3n^2$, and $½n^2$ is equal to $O(n^2)$.

It's important to emphasize that time complexity only focuses on scalability. Especially when n is a smaller value, one algorithm may have a longer runtime even if it has a better time complexity than others.

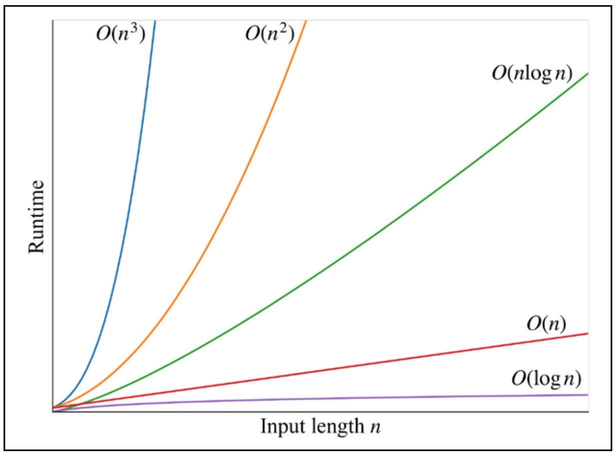


Fig. 1: The relationship between runtime and input length per time complexity

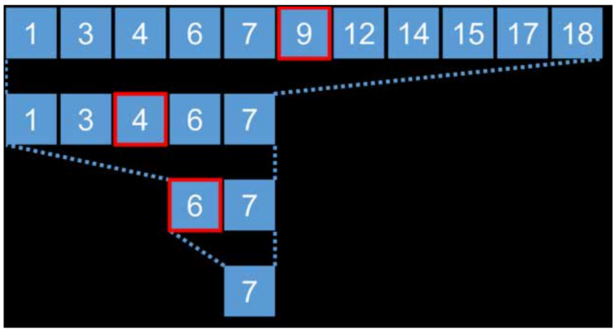


Fig. 2: Binary search sequence

Space complexity

In addition to time complexity, there's another measure for representing an algorithm's efficiency: space complexity. It looks at how memory requirements grow as the input's length increases. When you copy a list with n elements into a new list, the space complexity is *O(n)* because the need for additional memory increases linearly when you work with a larger input list. If an algorithm only needs a constant amount of memory, regardless of the input length, then the space complexity is *O(1)*.

There's often a trade-off relationship between time complexity and space complexity. Depending on the case, when comparing multiple algorithms, it's important to consider if runtime or memory is more important.

Binary search

As shown in **Figure 1**, an algorithm with time complexity *O(logn)* has better time performance than *O(n)*. Binary search is one of the algorithms with this time complexity. It's applicable when you want to search for a target value from a sorted list. In each operation, the algorithm compares if the target value is in the left or right half of the search area. For example, imagine a dictionary. You probably won't start on the first page of the dictionary to find the word you're looking for. You'll open up to a page in the middle of the book and start searching from there.

Figure 2 shows how the binary search proceeds when searching for the target value 7 in a list of eleven elements. The element marked in red represents the middle of the current operation's search area. If the number of elements in the search area is an even number, then it takes the "left" element in the middle. In each operation, you compare if the target value (7, in this case) is less than or greater than the middle. Cut the search area in half until you reach the target value. log2n is the maximum number of necessary comparison operations to find the target value with the binary search, where n is the length of the input list. Let's take *n = 8* as an example. The length of the search area starts with 8 and decreases to 4 after the first operation. After the second operation, it is divided in half again to 2 and after the third operation, there's just one value in the search area. From this example, we can conclude that the number of operations needed is at most a logarithm of 8 to the base 2 *(log28 = 3),* because 23= 8. In big O notation, we omit the base and write only *O(logn)*.

In Java, implementation of binary search is found in the *java.util.Arrays.binarySearch* [2] and *java.util.Collections.binarySearch* methods [3]. If you work with an array, you can use the methods in the *java.util.Arrays*. If you work with a list, then the methods in the class *java.util.Collections* are applicable.

Sorting algorithm

There are several kinds of sorting algorithms, each with different time complexities and space complexities. In practice, typical sorting algorithms used are Quicksort, Mergesort, and their variants. On average, the time complexity of these two methods is *O(nlogn)* [4], [5]. There are also sorting algorithms with better time complexities, but these often have limitations in the arrangement of input list or require special hardware.

The methods for sorting in Java are implemented in *java.util.Arrays.sort* [2] and *java.util.Collections.sort* [3]. Since Java 8, the List interface also provides the *sort* method [6], while the Stream API has the intermediate *sorted* operation [1]. According to Java documentation, these methods are implemented by default with Quicksort, Timsort, or Mergesort. But this can vary depending on the JDK vendor.

Task 1: Searching in a sorted list

The first task is finding the target value in an already sorted list. One potential solution is using the *contains* method of the *List* interface (Listing 1).

This solution's time complexity is *O(n)*. In the worst case, it searches the whole list until you reach the end. Another solution is to take advantage of the fact that the input list is already sorted, so the binary search is applicable (Listing 2). *Collections.binarySearch* returns an integer greater than or equal to 0 if the target value is

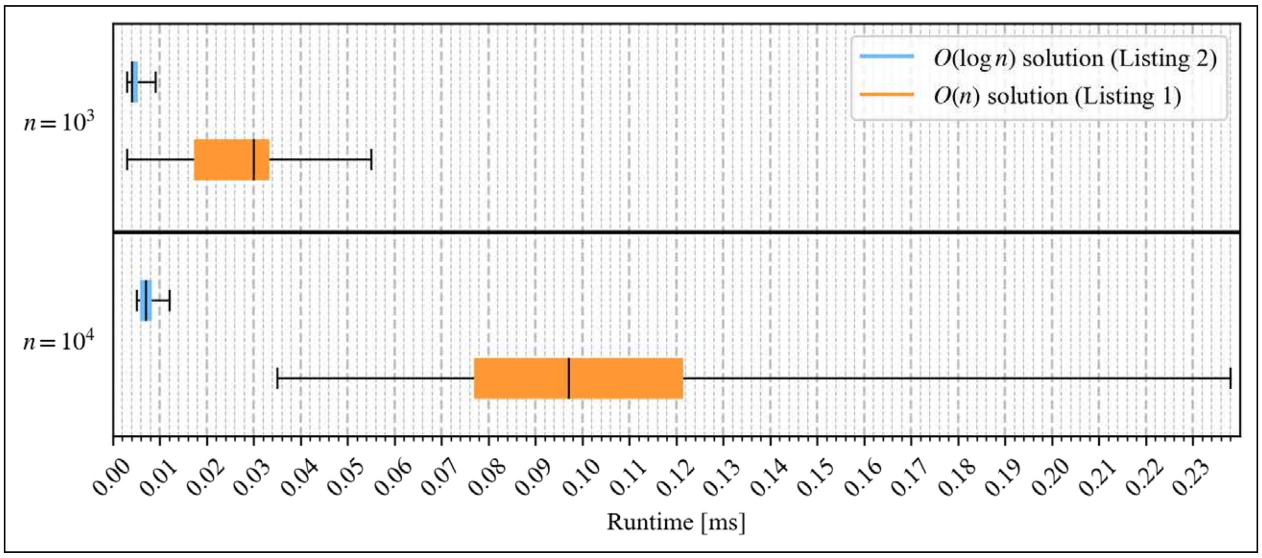


Fig. 3: Running times of the respective solutions for task 1

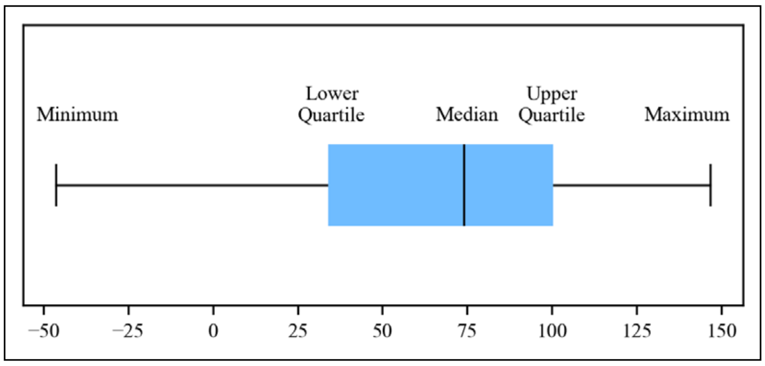


Fig. 4: Box Plot

in the list. The two solutions' space complexity is *O(1)* since they only need a consistent amount of memory to fix the result, regardless of input values.

I generated test data with 103 and 104 elements and used it to compare the runtime for the two solutions. The target values for the search are selected at regular intervals and the runtime was measured multiple times for each target value. The tests were run on a Windows 10 PC with Intel Core i7-1065G7 CPU 1.30GHz and 32 GB RAM. I used the Amazon Corretto 11.0.11 JDK and runtimes were measured with the Java Microbenchmark Harness [7].

Figure 3 shows the results for each length of input as a box plot. Each box plot contains the measurements of calls that were executed with different target values. A box plot graphically represents the distribution of measurement results and represents the median, the two quartiles (whose intervals contain the middle 50% of the data), and the two minimum and maximum values of the data (**Fig. 4**). You can see in Figure 3 that the solution's runtime in Listing 1 is more scattered than the binary search in Listing 2. This is because the runtime of the solution in Listing 1 heavily depends on where the target value is located in the list. This tendency becomes clearer when comparing results between the test cases *n = 103* and *n = 104*. Between the two cases, the worst-case runtime of Listing 1 increased significantly compared to Listing 2.

Listing 1

```
// input
List<Integer> list = List.of(12, 15, 19, 20, 21, 24);
int target = 19;

// solution
boolean answer = list.contains(target);
```

Listing 2

```
// input
List<Integer> list = List.of(12, 15, 19, 20, 21, 24);
int target = 19;

// solution
boolean answer = Collections.binarySearch(list, target) >= 0;
```

Task 2: Searching a range of values in a sorted list

The next task is counting the occurrence of values in a sorted list greater than or equal to a and less than *b (a ≤ xi < b)* where xi is the respective value in the input list. The requirements are that the input values a and b must always satisfy *a* ≤ *b* and there must not be any duplicates in the input list. An intuitive idea is using the intermediate operation *filter* in the Stream API to collect just the elements in a specific range of values, and ultimately, count the number of elements with the terminal operation *count* (Listing 3).

The time complexity of this solution is *O(n)*, because you must iterate once through the whole list, checking

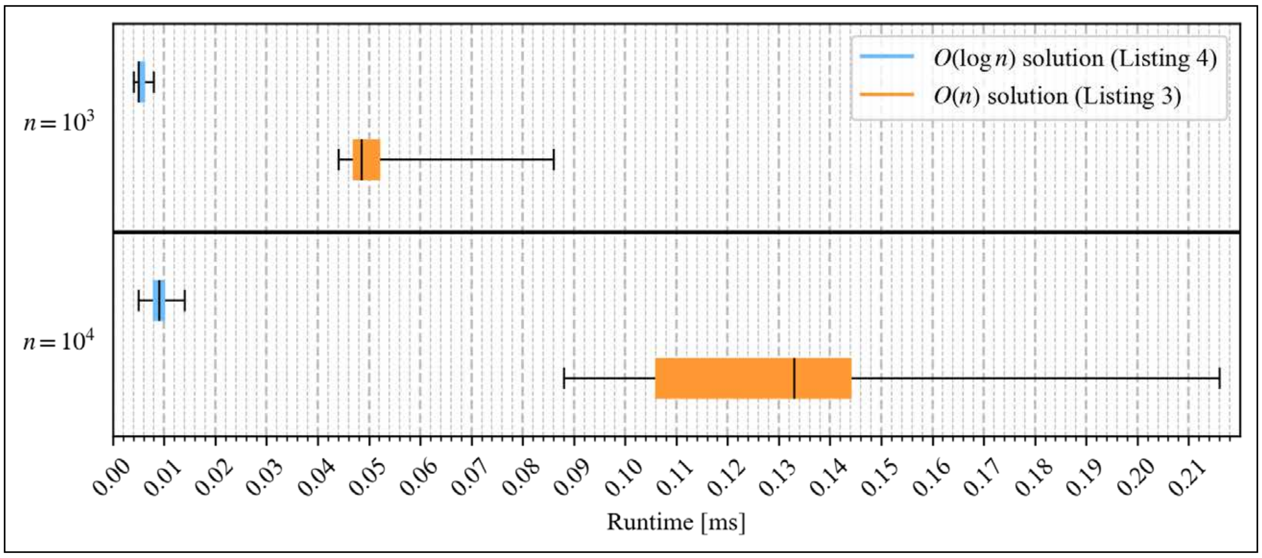


Fig. 5: Running times of the respective solution in Task 2

each list element to see if the value is in the range. But is it possible to use binary search for this task too? What if we could set the following two pieces of information:

- Position of the value *a* in the input list, if included. Otherwise, the position in the input list where you can insert the value *a*.
- Position of the value *b* in the input list, if included. Otherwise, the position in the input list where you can insert the value *b*.

The difference between the two calculated positions is the number of elements between the two thresholds. In this solution, the binary search is performed twice. But

Listing 3

```
// input
List<Integer> list = List.of(12, 15, 19, 20, 21, 24);
int a = 14, b = 19;

// solution
long answer = list.stream().mapToInt(Integer::intValue)
                        .filter(value -> a <= value && value < b).count();
```

Listing 4

```
// input
List<Integer> list = List.of(12, 15, 19, 20, 21, 24);
int a = 14, b = 19;

// solution
int lower = Collections.binarySearch(list, a);
int upper = Collections.binarySearch(list, b);
lower = lower < 0 ? ~lower : lower;
upper = upper < 0 ? ~upper : upper;
int answer = upper - lower;
```

since we don't consider the coefficient in the big O notation, the time complexity of this solution is still *O(logn)*. This is for the same reason as in Task 1: The space complexity of the two solutions is *O(1)*.

Listing 4 shows a sample implementation for this solution. Be aware that this code will not work if the input list has duplicates. As described in the documentation of *Collections.binarySearch* [3], the method does not guarantee which one will be found if the target value is included more than once in the list.

Collections.binarySearch returns an integer greater than or equal to 0 if the target value is in the list. Otherwise, it returns a negative value where *-(insertion point)-1* is. The insertion point is the position in the list where the target value should be inserted in order to keep the list sorted. In order to calculate the insertion point back from the return value *-(insertion point)-1*, you can simply use the bitwise NOT operator ~.

Just like with Task 1, **Figure 5** plots the running times of the two solutions as a box plot, measured with different lengths of input and target values. Again, it's easy to see that the solution in Listing 4 with binary search has more stable run times than the one in Listing 3.

Task 3: Find the largest value in an unsorted list

Now the task is finding the largest value in an unsorted list consisting of integers. One possible solution using the Stream API is to use *IntStream* and its terminal operation *max* [8] (Listing 5).

This solution has time complexity *O(n)* and space complexity *O(1)*. A different idea is to sort the list in descending order and return the first value in the list (Listing 6). As previously mentioned, Java provides several ways of sorting a list. To sort in descending order, you must specify a comparator in Java that compares backwards, since by default, the list is sorted in ascending order. You must also not use an immutable list, except when working with the intermediate *sorted* operation in the Stream API, be-

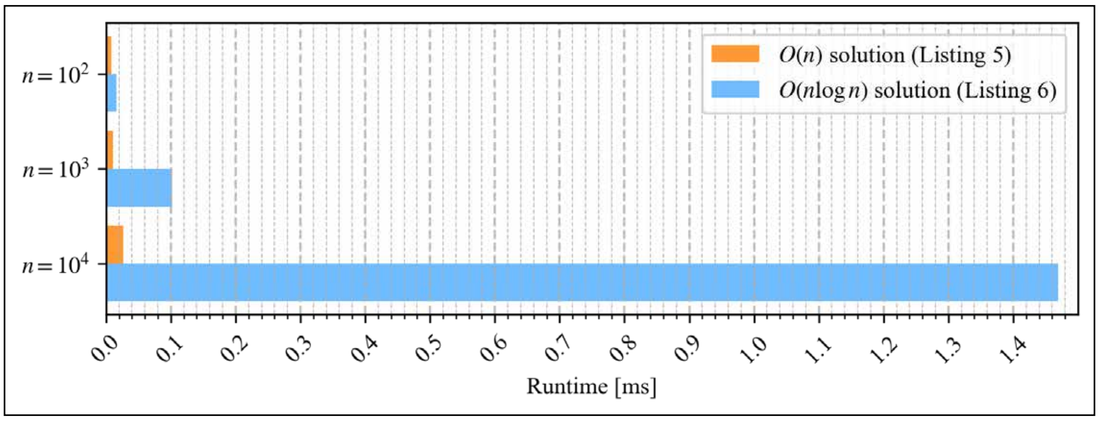


Fig. 6: Average runtimes of the respective solution for Task 3

cause the sort methods will process the list directly. For instance, the *List.of* method returns an immutable list.

This solution has time complexity *O(nlogn)*. However, the solution's space complexity depends on the method used in the implementation of the *sort* method. As previously seen in **Figure 1**, the time complexity *O(nlogn)* is worse than *O(n)*. In fact, you can see in **Figure 6** that as the length of the input list *n* increases, the solution's runtime from Listing 6 also increases dramatically with sorting – more than it did in Listing 5. However, in the next task, we will see that in certain cases, sorting the list is a good idea.

Task 4: Find the largest k elements in an unsorted list

In the last task, we saw that sorting isn't necessary if you only want to know the largest value of an unsorted list. What about needing the *k* largest values from the list? So, if *k* = *3*, then you must find the three largest values from the list (assuming that *k* is less than the input length). In this case, it's no longer enough to iterate through the input list once. But the solution with sorting will continue to work (Listing 7).

This solution can be easily optimized with a priority queue. A priority queue is implemented in Java with a binary heap [9] and is an abstract data structure that can be used to query the smallest value (or largest, depending on which comparator is specified) in the queue. Generally, the time complexity for adding and deleting values is *O(logn)*. For querying the smallest value, it is *O(1)*, where *n* is the length of the priority queue.

In our case, we add individual elements from the input list to the priority queue and delete each time the smallest value from the priority queue as soon as the queue's size is greater than *k*. Lastly, you insert individual elements from the priority queue into a list. Listing 8 shows a sample implementation of this solution. A small optimization, the priority queue is instantiated with an initial capacity of *k+1* since it can contain *k+1* elements at most. This solution's time complexity is *O(nlogk)*, since you insert n elements from the input list into the priority queue at a time, but the priority queue's size is limited to *k*. The space complexity is *O(k)* because you keep *k* elements in the priority queue temporarily so that you can eventually create the result list.

Listing 5

```
// input
List<Integer> list = List.of(23, 18, 15, 38, 8, 24);

// solution
OptionalInt answer = list.stream().mapToInt(Integer::intValue).max();
```

Listing 6

```
// input
List<Integer> list = Arrays.asList(23, 18, 15, 38, 8, 24);

// solution
list.sort(Collections.reverseOrder());
int answer = list.get(0);
```

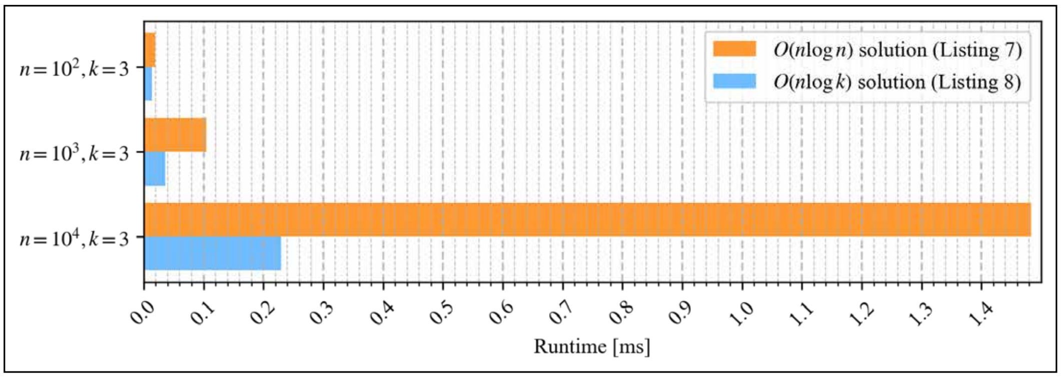


Fig. 7: Average runtimes of the corresponding solution for Task 4

Figure 7 shows the average run times of the respective solutions when measured with different lengths for input list n. The larger the difference between n and k, the larger its effect on the runtime.

Conclusion

In this article, I summarized the ideas of time and space complexities, and – in particular – I compared how time complexity affects runtime when working with a large amount of data. It's good practice to keep the two measures in mind and consider other criteria like code readability or maintainability during trade-offs. The Stream API is a very powerful tool for smaller data sets. But basically, the time complexity is O(n) if you filter or search over the entire input and don't prematurely terminate. If there's a possibility of the input growing in the future, then from the beginning you should consider if there's a better solution from the point of view of both complexities.

Listing 7

```
// input
List<Integer> list = Arrays.asList(23, 18, 15, 38, 8, 24);
int k = 3;

// solution
list.sort(Collections.reverseOrder());
List<Integer> answer = list.subList(0, k);
```

Listing 8

```
// input
List<Integer> list = List.of(23, 18, 15, 38, 8, 24);
int k = 3;

// solution
Queue<Integer> queue = new PriorityQueue<>(k+1);
for(int v : list) {
  queue.offer(v);
  if(queue.size() > k) {
    queue.poll();
  }
}
List<Integer> answer = Stream.generate(queue::poll)
  .takeWhile(Objects::nonNull).collect(Collectors.toList());
```

Ikuru Otomo graduated with a Master of Information Science and Technology from Hokkaido University and currently works as a Senior Software Developer at sidion GmbH.

Links & Literature

[1] https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/stream/Stream.html

[2] https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/Arrays.html

[3] https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/Collections.html

[4] https://www.inf.hs-flensburg.de/lang/algorithmen/sortieren/quick/quick.htm

[5] https://www.inf.hs-flensburg.de/lang/algorithmen/sortieren/merge/mergen.htm

[6] https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/List.html

[7] https://github.com/openjdk/jmh

[8] https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/stream/IntStream.html

[9] https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/PriorityQueue.html

Hitchhiking through the JavaScript jungle

Documentation – how, what, and when should I comment?

Documentation usually ranks behind testing in the most loved topics in software development. And after that, there's usually only topics like endlessly managing legacy code bases that you haven't even developed yourself.

By Sebastian Springer

I admit, I don't really like writing comments in code. I don't really think much of it. However, there are some places in your code where documentation in the form of comments is absolutely necessary. So, let's have a look at why I find a lot of comments to be nonsensical, and why it's good form to be prolific in comments elsewhere.

My code is self-explanatory, I do not need comments

The book "Clean Code" was published in 2008 and much of it is still valid today, even if Uncle Bob, his views, and public appearances are up for discussion. Among other things, "Clean Code" teaches you that you should write self-explanatory code that speaks for itself. What do comments do in the code? In many cases, they explain what the code does. Why not write the code so that it's obvious what it does? In any case, this sounds like a good idea. But it's not that simple. What constitutes understandable code is up to the individual. What seems obvious to one person may be completely incomprehensible to another. But the solution isn't to ask for comments in the code review explaining every single line. Here, it's much better to sit down together with the code and rewrite it so that it's understandable for everyone involved. Because of this, I think that pair and mob programming are excellent approaches for producing high quality code. Ultimately, the discussion about comprehensibility in the development process moves forward, to when the code is created and not just after the damage has already been done.

But back to our actual topic: comments. Basically, comments are completely superfluous for the JavaScript engine. Developers write comments either for themselves or others involved in the project. Hopefully, we agree that comments like the ones in Listing 1 have no place in our application's code.

One problem in this example is that some of the comments are obvious. You should avoid comments like this if possible, since they tend to disrupt the reading flow and merely point out facts in the code. Similarly, comments visually divide functions into multiple blocks. If you see that this approach is becoming common in your project, there's a simple solution: outsource the individual blocks to functions and give them a meaningful name. Mostly, the separator comment already gives you a good hint about potential function names. Admittedly, this isn't the case in our example.

Listing 1: Comments in code

```
function doSomething(x, y, z) {
  if (!x) {
    return false; // return false if x is falsy
  }
  // ---- first block ----
  const value = x + 10; // adds 10 to x
  ...
  // ---- second block ----
  ...
}
```

Inline comments can make sense in some places in the code, namely if they clarify the developer's intention that led to the corresponding code location. And this is only if it's not immediately obvious. Technical requirements in algorithms are good candidates for these comments, as they can be easily misinterpreted as errors in the code. Here, it's always good if the comment (or at least the commit containing the comment) contains a reference to a ticket or other further documentation.

But why do inline comments have such a bad reputation? Basically, they're unnecessary in cleanly implemented code, as the code speaks for itself. Additionally, the biggest danger of comments is that they become outdated. If the source code is updated, the comment is forgotten. It has to be done quickly. Cleanup work like documentation and tests can be done later. The comment is ignored and the test is skipped. This comment can quickly lead us down the wrong track when we read the code. It can do more harm than if it weren't there in the first place. You'd better develop the habit of instead of writing comments, ask yourself how you can write your code to make the comment redundant instead.

However, there are places where comments are mandatory, and it's always where your code is used by other people. Typical examples include the interface between server and client, shared libraries, and helper functions and classes. In other words, in all public interfaces. There are tools for each of the different agencies to make the work easier.

TypeScript – the better JavaScript?

JavaScript has a weak type system and doesn't have the capabilities to describe the signature of functions in code. Of course, with JSDoc and co., tools can help solve this problem. However, you must make sure that the types of the parameters and the documentation are synchronized. TypeScript helps solve this and many other problems. Another advantage is that TypeScript is used in an application not only for public interfaces, but for all interfaces. So they can all be described and documented at least in a very lightweight way.

Modern development environments take information from the signatures of the functions and help development by displaying information such as names and parameter types. However, the type information of parameters and return values and whether a parameter is optional are only part of a reasonable interface documentation. Therefore, at least the public interfaces of applications are often supplemented with further information. One of the best tools for this is JSDoc. Just like Javadoc, Doxygen, or phpDocumentator, the idea here is that the documentation of a function or class is handled in a comment block. JSDoc defines some standard elements for documentation, such as @param for parameters or @return for the return value.

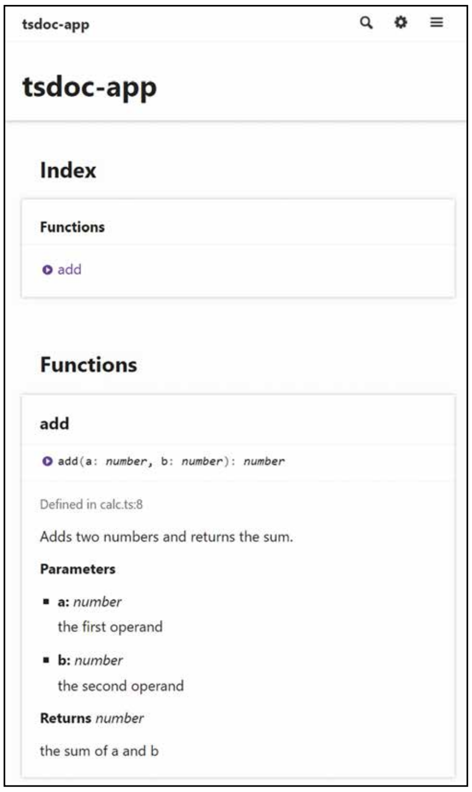
Listing 2: Function with TSDoc

```
/**
  * Adds two numbers and returns the sum.
  *
  * @param a - the first operand
  * @param b - the second operand
  * @returns the sum of a and b
  */
function add(a: number, b: number): number {
  return a + b;
}
```

TSDoc – clean documentation

A similar project exists for Type-Script called TSDoc. It follows the same schema, but is adapted to TypeScript and has some extensions. Listing 2 contains a simple example of an add function that adds two numbers.

So much for obvious documentation. However, this example is intended to show TSDoc and its tools, rather than its shining elegance. For example, these comments can be inserted into the TSDoc Playground at [1] to see the result. This website is suitable for quickly checking simple structures, but not for more extensive applications. The ESLint plug-in named *eslint-plugin-tsdoc* is much more purposeful. It reviews the project and makes



tsdoc-app

tsdoc-app

Index

Functions

add

Functions

add

```
add(a: number, b: number): number
```

Defined in calc.ts:8

Adds two numbers and returns the sum.

Parameters

- a: *number*

  the first operand

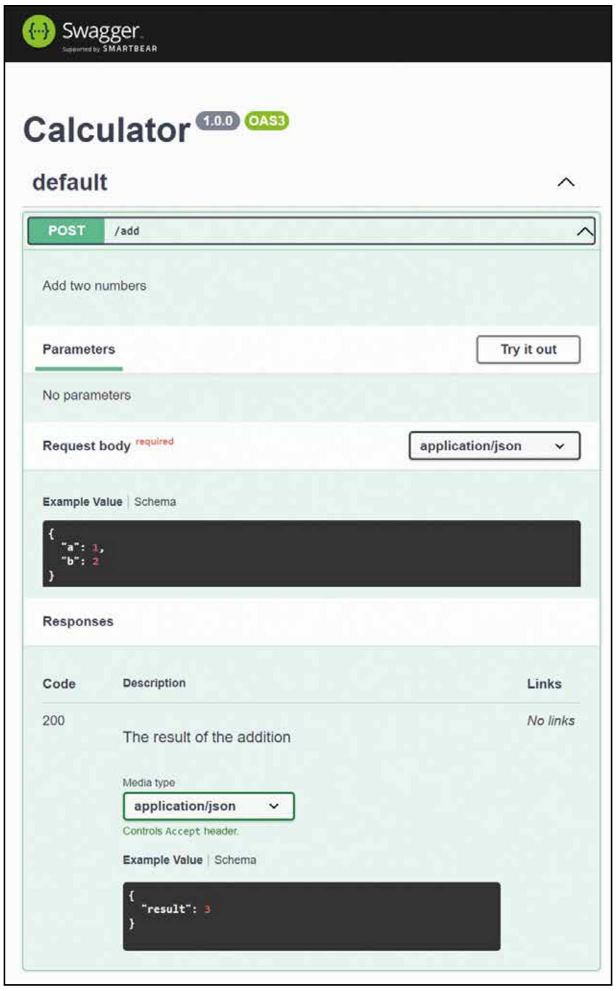
- b: *number*

  the second operand

Returns *number*

the sum of a and b

Fig. 1: Documentation display generated with TypeDoc



Swagger
Supported by SMARTBEAR

Calculator 1.0.0 OAS3

default

POST /add

Add two numbers

Parameters

Try it out

No parameters

Request body required

application/json

Example Value | Schema

```
{
  "a": 1,
  "b": 2
}
```

Responses

Code	Description	Links
200	The result of the addition	*No links*

Media type

application/json

Controls Accept header.

Example Value | Schema

```
{
  "result": 3
}
```

Fig. 2: Adding two numbers in swagger-jsdoc

sure that it follows the TSDoc specification rules. For example, the plug-in reports an error if the hyphen between the parameter name and the description is missing.

TypeDoc - we print our own manual

Now you've neatly tagged all the interfaces you want to share with other people with TSDoc comments. What's next? The ESLint plug-in slaps you on the wrist when you violate the standard. The IDE gives helpful tips when you use the documented functions, but it's still not clear or easy to read. So we need another solution. It comes in the form of TypeDoc, a generator for documentation. This tool reads your source code files and generates the appropriate documentation from the combination of TSDoc comments and TypeScript's type information. You can run TypeDoc with the command *npx typedoc *.ts*. It generates documentation for all TypeScript files in the current directory. By default, it exists in a directory called docs in the form of a series of HTML files. The entry point is the *index.html* file. You can either open the documentation locally in the browser or make the documentation accessible to others over a web server. You can see the result of our add function in TypeDoc in **Figure 1**. It looks much fancier than the comment block in the code.

Now let's turn to a more specialized area: the interface between server and client.

OpenAPI - the standard for server interfaces

If you implement server-side applications, the endpoints you expose to your clients are your interfaces. Fortunately, as far as REST APIs are concerned, there's a widely used description standard called OpenAPI. For example, if you implement your application with Express, you can use the swagger-jsdoc and swagger-ui-express packages to generate your documentation. Here, let's take another very simple interface that adds two numbers. This time, it is a web interface implemented in Express that can be used by other services and clients (**Fig. 2**).

Usually, the source code of these web services is not available for all using instances. Without documentation, using it is a wild guessing game that you don't want your users playing. Now, it's a matter of properly documenting the interface. You can see a simple example of such interface documentation in Listing 3.

The most important thing at this point is that you give your interface's users all the information they need to use it. You should try to make the *description* fields as understandable as possible with explanations. However, the most important thing is to have meaningful examples for both inputs and outputs.

In our case, we will use the *swagger-jsdoc* package. This allows us to describe the interface in terms of comments, so that the documentation can be done directly on the interface. If you change the interface, be sure to update the documentation too. Otherwise, the same applies to other comments in your application. Before you

Listing 3: API documentation in an express application

```
import { Router } from 'express';
const router = Router();

/**
 * @swagger
 * components:
 *   schemas:
 *     Addition:
 *       type: object
 *       required:
 *         - a
 *         - b
 *       properties:
 *         a:
 *           type: integer
 *           description: First number
 *         b:
 *           type: integer
 *           description: Second number
 *       example:
 *         a: 1
 *         b: 2
 */

/**
 * @swagger
 * /add:
 *  post:
 *    description: Add two numbers
 *    requestBody:
 *      required: true
 *      content:
 *        application/json:
 *          schema:
 *            $ref: '#/components/schemas/Addition'
 *    responses:
 *      "200":
 *        description: The result of the addition
 *        content:
 *          application/json:
 *            schema:
 *              type: object
 *              properties:
 *                result:
 *                  type: integer
 *              example:
 *                result: 3
 */
router.post('/add', (req, res) => {
  const result = parseInt(req.body.a, 10) + parseInt(req.body.b, 10);
  res.json({ result });
});
export default router;
```

mislead your users with false comments, leave out the comments (or the interface documentation) completely.

And what did we learn from this?

Before you start "enriching" your source code with comments in every conceivable place, try replacing unnecessary comments with clean, self-explanatory source code. If you lose the bigger picture in the code, divide it into smaller function blocks with meaningful names and be sure to name variables meaningfully. The length of a variable name doesn't matter. In your build process, these names are usually optimized anyway. This turns the code itself into documentation. When you have the chance, you should talk to others about your code and get feedback.

Once there's a chance of another person working with certain parts of your application, documentation for interfaces becomes mandatory. Make sure your documentation is short, concise and, most importantly, always up-to-date and error-free.

Yes, documenting is a pain. So try to automate it as much as possible and use tools that do most of the work for you.

Sebastian Springer is a JavaScript developer at MaibornWolff in Munich and is mainly concerned with the architecture of client- and server-side JavaScript. Sebastian is a consultant and lecturer for JavaScript and regularly shares his knowledge at national and international conferences.

Links & Literature

[1] https://tsdoc.org/play