THE FUTURE OF APIS

API Whitepaper

Learn the latest trends in API security, design, and management while exploring where the worlds of API, DevOps, and JavaScript overlap.

API Conference
@apiconf

@api_conference
#APICON

apiconference.net

Contents

Different ways of specifying contracts

API Contract Definitions

When running one or multiple services, it is essential that they have reliable service contracts [1] defining their exposed APIs. Those contracts mostly consist of declarative interface definitions, which strongly define and type the API exposed by the respective service. As such, it is crucial that the code making up the service exactly implements the interface and therefore fulfills its side of the contract. Regressions need to be detected and changes reflected in a well-communicated update to the contract. Here, we want to look at different ways of specifying contracts for what is one of the most common protocols for exposing service APIs: HTTP.

by Lena Fuhrimann

HTTP works great as a means of communication for microservices because it is open, reliable, programming language-agnostic, and works great over the wire. All these features are crucial to modern services, as they allow engineers to change the underlying technologies (e.g., change the back-end code from Python to Go) without it affecting the contract. Therefore, the API's consumers don't even need to know about the implementing technology and the providing team can take independent decisions respectively.

Service contracts usually contain the following four components:

- Available endpoints and operations on each endpoint
- Operation parameters input and output for each operation
- Authentication methods
- Contact information, license, terms of use, and other information

Specification and implementation

When working with services and their respective contracts, one has to maintain both the specification and the implementation. Ideally, these should always be in sync, as the best documentation is useless if it does not accurately reflect the reality of the API implementation.

Manual specification

The easiest way of creating a contract is to manually write it, and then write the respective code that should implement the contract. This is quite tedious and error-prone, as you have to basically write everything twice. When you change your implementation, you have to think about also changing the documentation and contract in the exact same way and vice versa. A way better approach is to either pick a technology that is contract-based and incorporates the interface specification in the exposed API or to at least automate either the generation of the contract from the implementation or the other way around.

Automated generation

There are two basic approaches to keeping the contract and the implementation in sync in an automated way. The first one is to write the code first and have the contract generated from that (Implementation First). The second approach is to write the contract and have the respective implementation code generated from that (Contract First).

Using either the contract first or implementation first approach guarantee that there is a single source of truth and that the other part is always in sync. As such, both are viable approaches. However, in general, it is preferred to write the contract first and generate implementation code from it. The reason being that when you

```yaml
paths:
  /pokemon/{id}:
    get:
      summary: Returns a pokemon
      responses:
        "200": # status code
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Pokemon"

components:
  schemas:
    Pokemon:
      type: object
      properties:
        id:
          type: string
        name:
          type: string
      required:
        - id
        - name
```

Fig. 1: OpenAPI HTTP endpoint definition

```
type Query {
  # Returns a pokemon
  pokemon(id: ID!): Pokemon
}

type Pokemon {
  id: ID!
  name: String!
}
```

Fig. 2: GraphQL schema

begin implementing your service, ideally, the contract has already been defined and communicated with potential consumers of your API to allow them to work independently of your implementation. Having a human- and machine-readable contract checked into your source code repository allows you to track changes to that contract over time and additionally serves as documentation for what the implementation code does (or at least what it should do).

Technologies

Here, we'll look at three different technologies that allow you to write a clearly defined and declarative contract for your services: OpenAPI, GraphQL, and gRPC. These all have their advantages and disadvantages, which will be laid out and discussed. Obviously, there are many more technologies which allow declaring contracts, but the ones presented here are three very popular ones which are easy to use and have great communities around them. They will be illustrated along the simple example of an API where one can query Pokémon by their ID.

OpenAPI

OpenAPI (formerly known as Swagger) is a very widespread way of specifying REST and other HTTP APIs. It is easy to write because the specification is just a JSON or a YAML file which defines what your API looks like by following a clearly defined specification.

An HTTP endpoint definition in OpenAPI might look as follows (**Fig. 1**):

OpenAPI [2] itself doesn't come with any tools to generate the specification from your implementation or vice versa. However, because it is such a popular format, there are many tools that allow you to parse your implementation code (and possibly additional annotations) and generate a valid OpenAPI specification from it. A great example of such a tool is springdoc-openapi which takes Java classes with their properties, methods, and annotations and automatically generates an OpenAPI specification from those. There are also tools to do it the other way around. These take an existing OpenAPI spec and generate boilerplate code from it for a compliant implementation. A popular example of such a tool is oapi-codegen which creates Go code from a valid specification.

Obviously, OpenAPI not being directly integrated into the implementation frameworks has a great disadvantage: It does not enforce (e.g., at compile time) that your implementation actually perfectly fulfills the specified contract. However, you can achieve a similar outcome by adding a check for your code's compliance to your automation pipeline, which prevents releases that diverge from their contract in an unwanted manner.

At this point, it is noteworthy, that REST applications can include so-called HATEOAS links. These are URLs included in the response body to a request, which lead to further endpoints providing actions for an element. If a client automatically follows those links, contracts can rely on that and therefore drop some of the actual URLs and paths from their specification. However, not too many applications in the wild reliably implement HATEOAS [3] links, and they have their caveats and shortcomings.

```
service PokemonService {
  // Returns a pokemon
  rpc GetPokemon (GetPokemonRequest) returns (Pokemon) {}
}

message GetPokemonRequest {
  string id = 1;
}

message Pokemon {
  string id = 1;
  string name = 2;
}
```

Fig. 3: Protocol buffer interface

GraphQL

GraphQL [4] calls itself "a query language for your API." The technology is about defining a schema which strongly types your endpoint methods and the objects they expect and return.

A simple GraphQL schema might look as follows (**Fig. 2**):

It is not only much more concise than the above OpenAPI specification, but it also has great advantages because it is part of the GraphQL specification. Almost every GraphQL endpoint exposes its schema automatically, which is a direct product of the endpoints it actually exposes. This allows clients to query the contract directly from the endpoint and therefore know that it is always up to date. Tests can be run against that exposed schema, which would detect breaking changes automatically and potentially prevent releasing such. These conventions of how the endpoint exposes its documentation allow us to use comprehensive client frameworks such as apollo-client.

With GraphQL, there are also frameworks that allow writing a schema first and generating the respective boilerplate code from it. A popular tool for doing so is gqlgen in Go.

gRPC

Another popular technology for declaring contracts is gRPC [5]. It is based on Protocol Buffers [6], which is a way of specifying how to serialize structured data. The interface of a protocol buffer is defined in a file that might look like this (**Fig. 3**):

One big difference between protocol buffers and the other technologies mentioned is that the data exchanged is in binary format rather than plain text. This makes them very performant but also harder to debug, which makes having a clearly defined schema and API crucial. A compiler of such a Protocol Buffer file is built into the toolchain and lets you generate boilerplate code from the specification and enforce compliance with the defined contract.

Conclusion

There are many ways of writing contracts for your service APIs. A good contract has the following characteristics:

- It is human-readable
- It is machine-readable
- It is declarative and comprehensive
- It is tracked via version control
- It is programming language-agnostic
- It enforces that the implementation fulfills the contract
- Breaking changes to the contract are detected and properly communicated to potential consumers

This makes the above technologies excellent choices, and all of them are a great step up from simply writing your contract somewhere in a wiki.

Lena Fuhrimann is an energetic software engineer and architect. She founded the company bespinian in 2019 with Mathis Kretz and has since worked with many customers and interesting technologies. Her primary areas of interest include security, serverless technologies, public clouds, and infrastructure as code. She has, however, worked extensively with Kubernetes and its ecosystem, and has deployed numerous applications to those platforms using automation and GitOps. She uses Arch.

Links & References

[1] https://cloud.google.com/appengine/docs/legacy/standard/java/designing-microservice-api

[2] https://www.openapis.org/

[3] https://en.wikipedia.org/wiki/HATEOAS

[4] https://graphql.org/

[5] https://grpc.io/

[6] https://github.com/protocolbuffers/protobuf

Interview with Jeff Williams, Chief Technology Officer at Contrast Security

"Most organisations are not prepared for the scale of security breaches to come"

The API ecosystem is evolving rapidly, allowing for faster innovation but also exposing many businesses to security risks. We talked about APIs and API-related vulnerabilities with Jeff Williams, the co-founder of and Chief Technology Officer at Contrast Security and a founding member and major contributor to OWASP, a nonprofit foundation dedicated to improving software security.

by Jeff Williams

devmio: Why are we seeing such a fast, widespread adoption of APIs and an increase in API traffic?
Jeff Williams: APIs have grown so rapidly because they allow enterprises to innovate and interconnect more rapidly. When browsers became able to get data from APIs using Ajax, it kicked off an unstoppable market shift that is still playing out. Today, almost all websites use JavaScript in the browser to call APIs that populate the pages you see with data.

devmio: A report from Gartner predicts that APIs will potentially cause the biggest security vulnerability in history. Do you think that is an overestimation, or are people simply not prepared for the scale of security breaches to come?
Jeff Williams: This is accurate. Most organisations are not prepared for the scale of security breaches to come because they include APIs in their regular security scans of software, relying on legacy web application security (AppSec) testing tools to scan lines of code for known vulnerabilities. However, traditional security tools don't work on APIs: They were designed for web apps, not to test the security of an API. This leads to a false sense of security, and pride before a breach.

devmio: How do most API attacks occur? What is the most common weak point?
Jeff Williams: APIs are not only the connective tissue that holds together the different parts of a piece of software, they are also often exposed directly to the internet and are easy for attackers to target. Further, APIs often have direct access to sensitive data in backend systems. This makes successful exploits more serious, as there aren't multiple layers of code between attackers and sensitive data.

devmio: What is API sprawl, and what problems can this cause?
Jeff Williams: APIs are relatively small compared to traditional web apps. So, you need a lot of them. Pretty soon you have version control problems, rogue APIs being stood up, several different API platforms… and you have a sprawling mess. This leads to difficulty ensuring that all of your APIs are getting the right security attention.

devmio: Where should teams begin when creating a security-focused API strategy? What should they focus on?
Jeff Williams: Ensure they deploy a modern, integrated API security platform that manages what traditional API or application security can't do: namely, to secure APIs from the inside out.

- **API inventory:** You can't secure what you don't know. You need an inventory process.
- **API security testing:** You've got to write secure code, and that means finding unknown vulnerabilities in APIs, microservices and functions. After all, the OWASP Top 10 vulnerabilities are just as applicable with APIs as they are in traditional web apps.
- **Components:** You have to secure your supply chain, including finding known vulnerabilities in active third-party libraries, frameworks and services.
- **API protection**: In order to protect production, you've got to identify probes and attacks on both known and unknown vulnerabilities and prevent exploits.
- **API access:** Strong authentication and authorization on functions at the API level as well as at the data layer are crucial.

devmio: What are the key steps in properly validating an API and ensuring proper user identity verification?
Jeff Williams: Authentication is straightforward. You should definitely use a product instead of implementing yourself. There are many subtle and tricky ways that you can implement authentication. Just like encryption, your mantra should be "don't build it yourself".

devmio: How can threat modeling help teams improve their API security?
Jeff Williams: Threat modeling can help identify architectural weaknesses in API deployments, including APIs that aren't protected by encryption, authentication, and authorization.

devmio: What tools would you recommend including in every team's security stack?
Jeff Williams: Teams should look to deploy the following three tools:

- **Interactive application security testing (IAST):** Uses instrumentation to continuously monitor and analyze APIs from within as they run in development and test environments. This approach yields real-time analysis as software is being developed and tested. This makes them ideal for Agile, DevOps, and DevSecOps environments, as they enable IT to find and fix security flaws early in the SDLC when they are easiest and cheapest to remediate. IAST provides teams with the full context of what's going on inside the code of an API, enabling them to see API traffic, code, configuration, framework, libraries, backend connections, and much more. Using this context enables users to detect the behavior of vulnerable code and report detailed findings back to developers for remediation.
- **Software Composition Analysis (SCA):** Enables businesses to protect their software supply chain by identifying real threats from third-party components across the entire SDLC — from code through test and on through production. SCA uses instrumentation to identify vulnerable libraries and how APIs use them. With this context, developers receive actionable remediation guidance to help them fix and protect against API attacks.
- **Runtime application self-protection (RASP):** RASP provides two key API security capabilities: First, it creates visibility into exactly who is attacking you, what attack vectors they are using on your APIs, and which of your APIs is being targeted. Second, RASP prevents most of the major classes of vulnerabilities from being exploited, including both zero days and custom code flaws. RASP uses instrumentation to add lightweight security sensors to your API code and platforms. These sensors can directly measure the security-relevant behavior of your APIs and detect malicious events. Working from inside APIs themselves, RASP security is able to detect, block and mitigate attacks immediately, protecting as they run in real time by analyzing both their behavior and context.

Thank you for taking the time to share your expertise with our readers!

Jeff Williams is the Co-Founder and Chief Technology Officer of Contrast Security, the industry's most modern and comprehensive Application Security Platform, focusing on fully automated application security at DevOps scale and speed. He is also a founding member and major contributor to OWASP, where he served as Global Chairman for eight years and created the OWASP Top 10, OWASP Enterprise Security API, OWASP Application Security Verification Standard, XSS Prevention Cheat Sheet, and numerous other widely used free and open projects.

API as spacecraft

What the James Webb Space Telescope Can Teach Us About Engineering APIs

Have you seen some of the images of deep space taken by the James Webb telescope? As much as these images make me marvel at the universe, they are also a testament to the capability of today's space engineering, which is capable of designing, building, and operating a telescope on a spacecraft far from Earth that delivers these images.

by Matthias Biehl

Think about it, once such a spacecraft has been launched and is out in space, it needs to work flawlessly! If something unexpected happens, you cannot just order a service technician to check what's wrong. Once launched, spacecraft are simply out of reach, travelling far away and at high velocity, which makes it impossible to change any aspect of their design. And with this in the back of their mind, space engineers do everything to get them prepared for launch day. They know they have just one chance to get it right.

And with APIs? At first sight, the engineering of APIs is much different from the engineering of a spacecraft. After an API has been launched, you could easily change it. The code is right there, the gateway configuration is at your fingertips, and pushing out a change is a matter of seconds.

But should you? And I am not talking about fixing bugs – of course, you should fix them – I am talking about design changes. Unlike with spacecraft, all the artefacts you need for a change seem to be available and within reach – so why wait?

When an API gets published, it starts to get used by API consumers. It just means that the API consumers write application code that calls the API. And in this code, they "bake in" a reasonable assumption about the API: that it will stay exactly as it was at development time. This assumption is completely reasonable because it allows making an API call in a simple manner. But this assumption also makes applications very sensitive to changes in the API specification. You could

say that most applications are inflexible to API changes and that even the slightest change will break them. By and large, applications rely on unchanged APIs. And today, with APIs being used in business-critical applications, many businesses depend on the used APIs to run in exactly the same way as they did the day before.

As a side note, the HATEOAS [1] principle would not require you to make the above assumption, because it introduces a level of indirection and dynamic discovery of the API endpoint. But it adds to the complexity for the API consumer and is thus not widely used in practice. So HATEOAS is excluded from this article.

When you have a number of API consumers, some of them depend on one set of data fields, and others depend on another set of data fields. For each aspect of your API, you can find an API consumer that depends on it. And as Hyrum's Law tells us, this dependency is not limited to the documented features in your API contract (i.e., your Open API specification) but pertains to any observable behaviour of the API. With a sufficiently large group of API consumers, you cannot easily change the design without breaking some code and creating problems for at least one of the API consumers.

Even though we could readily change APIs – we should not do it. Once an API is published, API consumers depend on it, and API changes will break their application. Or as Amazon CTO Werner Vogels phrased it in his best practices for APIs: "APIs are forever." They cannot be changed.

Be agile before launch day and conservative afterwards.

It helps to take a space engineering mindset and think of an API as a spacecraft. Before you launch, you can change the design, iterate on it and work on it in an agile, iterative fashion, where feedback is readily incorporated into its design. The same holds for an API before publication when it is in the initial design and prototyping phase – maybe even in the implementation phase. But launch day changes everything. After launch, the spacecraft is physically out of reach for design changes. And for APIs, you should think about your deployment to production and publication of the API in the same way. But since all artefacts are physically within reach, you need to set up some rules to prevent design changes.

And if you (or an important API consumer of yours) wanted to change the API anyway? Well, you would need to handle it in exactly the same way as you would change a spacecraft: You would go to the work shed and build a new version of the spacecraft – with the new, improved design – and you would launch it into space. Sounds costly? It would not only be costly for a spacecraft, but also for an API. When you introduce a new version of an API, while the old version remains as is, the effort for running and maintaining the API doubles.

As for the design of a spacecraft, you need to get the design of your API right the first time! It will be out there forever. Designing an API is not "rocket science," but the mindset of space engineering will help you to create dependable APIs that your API consumers can rely on.

Matthias Biehl is an API Strategist at Software AG [2]. He empowers customers to discover their opportunities for innovation with APIs & ecosystems, define actionable digital strategies and execute API initiatives. Based on his experience in leading large-scale API initiatives in both business and technology roles in banking, insurance, media, government, and telco, he shares best practices and provides strategic guidance. Matthias is the author of several books on APIs, [3] runs the API-University [4], and regularly speaks at technology conferences.

Links & References

[1] https://api-university.com/blog/rest-apis-with-hateoas/

[2] https://blog.softwareag.com/author/matthias_biehl/

[3] https://api-university.com/books/

[4] https://api-university.com/

Proactively looking at data services and APIs together

Preventing Data Infrastructure Sprawl – What Developers Can Do

For developers, building applications is exciting - who doesn't want to create the next generation app that customers love? However, the way we build applications today in the cloud leads to potential problems around data, finds Ovais Tariq, CEO at Tigris Data.

by Ovais Tariq

As we get more data from our applications, we have to organize this, and it leads to more infrastructure. To deal with the problem of data infrastructure sprawl, we have to understand why this sprawl takes place, and then be proactive in how we approach the issue. By looking at data services and APIs together, we can improve how we support data over time.

Microservices and data

While traditional applications would use a single database that would act as its data store, modern applications are designed based on connecting multiple microservices. Using microservices running in software containers offers more flexibility and freedom in how to build an application, but this compartmentalized approach does require more database instances to capture all the data that is created.

Rather than a couple of large databases that hold all the data involved, each application might have thirty or forty database instances to capture and store data from each microservice over time. This is exacerbated by the need to use different data models and functionality such as search, indexing and event streaming. This results in deploying multiple different database technologies which add to the fabric of your data infrastructure and the complexity grows.

However, this data infrastructure will then need to be managed over time as well. Each database technology used needs to be deployed, configured, secured,

monitored, and maintained as infrastructure components. This can take developers away from the work that they love doing, and put their focus on operational tasks instead. This slows down the innovation process.

To diagnose this as a problem can be difficult. When you start small, developers will often pick a database that they are most comfortable with, and that they can get running quickly. This will normally lead to them using the same database for multiple use cases, and where some of them are not good fits. For example, developers will typically start with a database like MySQL for OLTP workloads, then try to apply this for other workloads like full text search and for analytics. But because existing database technologies are not flexible enough to continue to support diverse workloads as the application scales or the needs of the application evolves, they end up going to a complicated data architecture with multiple different database technologies.

Cloud service providers can offer services that can step in to reduce some of the management overhead. These providers have proven time and time again that the platform approach is popular. Picking a cloud provider and locking into their tech stack can give you some reduction in operational costs and effort. But all the cloud providers are doing is providing you "as a service" instances of the popular database technologies. This does not solve the data infrastructure sprawl problem.

Fixing infrastructure sprawl starts with developers

Solving this problem is about managing data more efficiently as a basic principle, and then treating this as a product in its own right. This means looking at how data gets used with APIs.

From a developer perspective, interactions around data can be very simple - they want to use the standard set of actions CREATE, READ, UPDATE and DELETE, termed CRUD. Alongside these actions, developers may have to set up streaming or search services to meet user demands within an application. Putting these behind APIs can make the process easier for developers.

However, having those data services accessed through APIs rather than deploying as multiple databases is not an effective solution to the problem on its own. If the whole system is not cohesively built, then you still have to learn these different APIs. It shifts some of the infrastructure sprawl burden, but it doesn't solve the management overhead.

Using APIs alone also doesn't take away the fact that you need to connect all these systems together. To solve this effectively, you need to think about consolidating your platforms and APIs at the same time, so that you can serve all the different use cases related to data that your application developers will have over time.

This "universal API" approach has to take a platform approach into account in order to be effective. Rather than building applications with dozens of infrastructure components exposed to the developers, instead developers should be able to access these diverse functions through a single common interface. Instead of having to worry about data flowing between disparate systems, data should be available across these different functions automatically.

Working with more data in interesting ways is essential to how developers deliver what businesses want. However, this has to be considered in the longer term, so that the sheer volume of data, services and requirements does not overwhelm your team with infrastructure sprawl.

Ovais is the CEO of Tigris Data, where he leads the team building the world's first truly open source developer data platform. Prior to Tigris Data, Ovais led data and storage engineering teams in solving some of the toughest problems around developer productivity around data, including work at Uber, Khoros and Percona.

A gateway to flexibility: Stargate and Apache Cassandra

Freedom of Choice with Apache Cassandra and Stargate

Stargate APIs unlock Apache Cassandra® data for developers. In the challenge of balancing application performance with flexibility in application development, Stargate and Cassandra deliver great freedom of choice. Now developers are in control of the performance-productivity balance, choosing from a range of Stargate Web APIs and Drivers for Cassandra. And with the new version of Stargate v2, operators get even greater flexibility in how to operate Stargate by deploying and scaling APIs independently.

by Mark Stone

Engineering organizations constantly face choices between how performant the applications they develop will be, and how frictionless the development process will be for software engineers. Too much emphasis on raw performance saddles developers with a complex, constrained development process. Too much emphasis on ease in development (and thus rapid time to market) can yield an application that simply isn't performant enough to go to production. There is no right answer to this tradeoff that works for all applications and all developers, so giving organizations options is essential. A data API gateway like Stargate [1], paired with a NoSQL database like Apache Cassandra [2] is a great way to ensure an optimal balance for your projects.

With Cassandra, developers get a database with limitless scale, fast writes, and, with the right data model — fast reads — all of which are ideal for real-time applications. With Stargate, enhanced now in version 2, developers get the flexibility to choose APIs and SDKs that fit their performance requirements and fit the idiom in which they are comfortable expressing data interactions in their applications:

- CQL API, for driver-managed queries
- CQL over gRPC API, for queries via Stargate's gRPC client libraries
- GraphQL (Schema First) for GraphQL queries against an existing Cassandra data structure
- GraphQL (GraphQL First), for creating and interacting with a schema entirely from within GraphQL
- REST API, for language-independent CQL queries over HTTP
- Document API, for JSON-based data structures that don't require a pre-existing schema

Let's look at how each of these APIs deliver an ideal combination of performance and flexibility tradeoff, and how Stargate v2 improves the control organizations have over these tradeoffs.

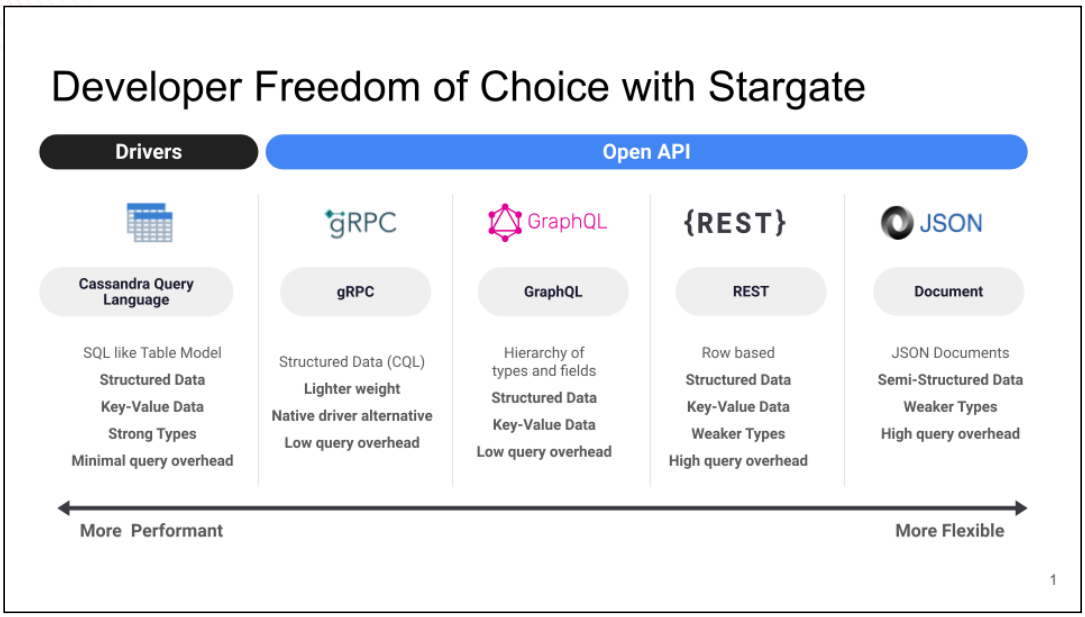

Figure 1: Developer Freedom of Choice with Stargate

CQL API and gRPC API

Making CQL queries from a gRPC client library via the gRPC API and querying from a driver via the CQL API offer the best raw performance. With Stargate V2's high-performance gRPC implementation, performance now has demonstrated parity with native drivers. This is due to Stargate v2 exploiting every aspect of the gRPC protocol: unary, client-side streaming, server-side streaming, bidirectional calls, and Google's improved serialization for gRPC.

One can, of course, make driver calls directly against Cassandra without the need for Stargate. By introducing Stargate, we separate coordinator nodes from storage nodes. This enables better performance by allowing us to tune coordinator nodes for compute-heavy workloads and storage nodes for storage-heavy workloads.

Drivers introduce some management complexity into application development (load balancing, retries, TLS termination, etc.), and not all open-source drivers are equally well supported. If you are considering drivers, you may want to first:

- Check if a supported driver is available for your language and language version
- Consider the friction of management complexity
- Consider the potential for downtime in the event of needing to mirror network configuration updates in the driver

In these situations, Stargate's gRPC libraries will be a better choice. Network management is handed off to the Stargate gRPC API, where it is handled automatically. This is a more cloud-native way to architect an application environment. Stargate currently offers gRPC libraries for Java, Go, Node.js, and Rust, and a new library in a language not yet covered can be added with relatively little effort. The Stargate community welcomes and will actively support new gRPC clients, so it's a great time to get involved [3].

GraphQL API

GraphQL is an HTTP-based API similar in many ways to the REST API. However, GraphQL offers more targeted key-value querying than one can easily do in REST, thus avoiding REST's over-fetching or under-fetching problem.

Stargate's GraphQL API is really two APIs:

- Schema first, well suited for querying against existing Cassandra data that already has a defined CQL schema.
- GraphQL first, in which no pre-existing schema is presumed, and GraphQL itself will be used to create the schema. This approach can be very effective for rapid prototyping and early development, particularly when combined with the GraphQL Playground.

So, while APIs based on HTTP 1.0 are inherently slower than native driver calls over CQL, or gRPC calls over HTTP 2.0, GraphQL offers a lot of flexibility in terms of how to structure queries, and easy exploration via GraphQL Playground. The efficiency of GraphQL's queries also makes it more performant than the REST API or Document API.

REST API

In some situations, neither native drivers nor gRPC will offer support for your language, or language version. It may also be that raw performance is not the key consideration in your application environment. In this case, the flexibility of the REST API works well. CQL queries are delivered via HTTP, meaning you can write those queries in any language you choose. You'll still see some performance benefit from querying structured data

against a known structure (in contrast to schema-less data like JSON documents).

Document API

The popularity of JavaScript (including Node.js and its siblings like TypeScript) has made JSON the most pervasive data structure in modern application development. Many application developers would prefer not to think about the underlying database at all, and instead simply think of their data in terms of JSON.

For this broad set of use cases, Stargate's Document API can be a great solution. The API does not assume or require a pre-existing schema, and instead relies on a process called "document shredding" [4] to turn JSON documents into Cassandra tables, and then rebuild those tables into JSON when queried.

Naturally, working in a schemeless idiom in this manner introduces overhead, which impacts performance. Nonetheless, the freedom and flexibility of building JSON on the fly can be a powerful accelerator of developer productivity. The Document API can be a great choice for rapid prototyping and early development for precisely this reason. In application environments where the performance requirements are not too stringent, the Document API may offer the fastest time to market for application development.

An ounce of prevention often beats a pound of cure. When your JSON is well- or even semi-structured, consider using Stargate's Document API support for JSON schema [5]. JSON schema reduces hard-to-debug-at-runtime errors by validating data as well as making documentation, annotation, and automated testing easier.

Improving Flexibility with Stargate v2

Most organizations focus on a single language for development, and a single API for data interaction. Consequently, Stargate V1's monolithic architecture introduced some inefficiencies. To deploy any of Stargate's APIs, you had to deploy all of them; to scale any of Stargate's APIs, you had to scale all of them.

In this respect, Stargate v2's pluggable, modular architecture is a game changer. Now each API is deployed as its own independent service. Your organization doesn't need to deploy APIs you won't be using. Even if you use multiple APIs, almost certainly you won't use them all equally. So the flexibility to scale APIs independently is a key boost to operational efficiency.

This modularity also makes Stargate more extensible. Is there another API your organization needs? Maybe you need a more purpose-built API rather than a general-purpose API. Perhaps there's an entirely different database other than Cassandra that you'd like to plug in, or another database alongside Cassandra with which you'd like to federate?

Because Stargate is open source, you have this freedom of choice. Because Stargate v2 provides modular services, exercising this freedom is now a practical reality. With Stargate, Apache Cassandra can now be used for a wider range of use cases, by many more types of developers.

Stargate deploys anywhere — Docker, K8ssandra, bare metal/VM, and soon you'll be able to use it from the Amazon Web Services [6] and Google Cloud marketplaces [7]. You are also welcome to skip installation and play around with it on the DataStax Astra DB [8] free plan.

Mark Stone is a technology veteran with many years of experience in product management, program management, and people management. Always working as part of the connective tissue between business stakeholders and technical stakeholders, Mark loves championing the developer experience in technology platforms and helping organizations meet developers where they are. With a rich background in both agile and open source, Mark firmly believes in the power of collaboration and bottom-up innovation.

Links & References

[1] https://stargate.io/

[2] https://cassandra.apache.org/_/index.html

[3] https://github.com/stargate/stargate/blob/v1/CONTRIBUTING.md

[4] https://stargate.io/2020/10/19/the-stargate-cassandra-documents-api.html

[5] https://stargate.io/docs/latest/quickstart/qs-document.html#add-json-schema-to-a-collection

[6] https://aws.amazon.com/marketplace/seller-profile?id=6c121bc6-9f22-421b-9957-ac944e83c141&ref=dtl_B095YKJVKY

[7] https://console.cloud.google.com/marketplace/browse?q=datastax

[8] https://auth.cloud.datastax.com/auth/realms/CloudUsers/protocol/openid-connect/auth?client_id=auth-proxy&redirect_uri=https%3A%2F%2Fgatekeeper.auth.cloud.datastax.com%2Fcallback&response_type=code&scope=openid+profile+email&state=7EsXXz3yu5QMZQ2iCyRjKVHFutc%3D

Digital essentials

The Role of APIs in Digital Government Context

Connecting information systems, applications, and registers, exchanging data, and sharing services are essential requirements for any digital service. Government and the public sector are no exception. The ability to exchange data and share services between government entities and authorities is a must-have requirement when public services are digitised. Also, the need is not limited to data exchange capabilities between government entities since the ability to exchange data and share services between public and private sectors is evenly essential.

by Petteri Kivimäki

Digital transformation is converting or substituting analogue processes with their digital counterparts. One of the goals of digitising public services is to reduce administrative burden and provide citizens with streamlined digital processes spanning multiple administration sectors. From a citizen's point of view, it means no more filling in paper forms and visiting different government offices. The required information is exchanged in the background automatically between the concerned authorities without further involvement of the citizen. This results in a single, streamlined online process that hides the underlying complexity from the citizen. Besides, it significantly reduces manual work required from different authorities and enables the development of new services. Getting there is impossible without sharing data and services between authorities and the public and private sectors.

Different shades of interoperability

The ability of information systems to exchange and utilise information is known as interoperability. Unlike what it may first sound like, interoperability is not only about technology and technical connectivity. On the contrary, interoperability consists of different layers that also include technology. The European Interoperability Framework (EIF) [1] defines four layers of interoperability:

Instead of reinventing the wheel and building everything from scratch, it is possible to use off-the-shelf, battle-proven solutions that have already been successfully used in multiple implementations.

- **Legal** – aligned legislation
- **Organisational** – coordinated processes
- **Semantical** – precise meaning of exchanged information
- **Technical** – connecting information systems and services

All four layers are equally important when building digital services and processes. In addition, challenges in one layer are often reflected in other layers. Therefore, it is essential to be aware of all the layers and not neglect them.

Data exchange scenarios

When it comes to a public sector organisation exchanging information, three top-level data exchange scenarios can be recognised:

- **Internal** – data exchange within an organisation
- **National** – data exchange on a national level
- **Cross-border** – international data exchange

The same rules, laws, and regulations don't apply to national and cross-border data exchange, which is why they are two separate scenarios instead of a single "external" scenario. Cross-border data exchange between authorities usually requires both state-level agreements and data exchange agreements between the data exchange parties.

The common factor between the scenarios is that all three require certain technical base elements, including but not limited to connectivity, secure communication protocols, interfaces, and integration services. The more standardised these elements are, the less work is required to build new connections between information systems and services. Instead, if there is no commonly agreed solution to connect information systems and manage the connections securely, the result is probably a jungle of point-to-point connections. It means agreeing on the connection details and then building the connections whenever a new connection is needed – and doing so repeatedly.

However, even if the technical base elements in all the scenarios are the same, they are usually implemented using different technical solutions and technologies. Implementing a standardised connectivity layer within an organisation is generally based on other technology than a standardised connectivity layer with external parties.

The good news is that there are already technical solutions and building blocks available that can be used for secure data exchange in different scenarios. Instead of reinventing the wheel and building everything from scratch, it is possible to use off-the-shelf, battle-proven solutions that have already been successfully used in multiple implementations. For example, eDelivery [2] is the building block of the European Commission for cross-border data exchange between the EU Member States. At the same time, X-Road® [3] is open-source software and ecosystem solution that provides unified and secure data exchange on a national level.

The once-only principle

The once-only principle (TOOP) [4] is a digital government concept initiated by the European Union (EU), whose aim is that citizens, organisations, and companies provide certain information to authorities and administrations only once. The data is then reused by sharing it between the authorities that have a right to access it. In this way, the information is collected and stored only once. In practice, if specific information is already collected and stored by one authority, another authority

that needs the same information should query it from the owning authority instead of asking it again from the citizen.

The basic idea behind TOOP sounds simple but implementing it in practice is more complicated. First, it requires accessibility and interoperability of base registers and other related information systems and services. Implementing TOOP is impossible without APIs – they are needed to enable data exchange between the authorities. In addition to APIs, successful implementation requires a secure data exchange solution, unified data models, and semantic interoperability across different information systems and applications. Otherwise, utilising the data is challenging. Besides technical questions, there are also legal and administrative issues that must be considered.

Cross-border data exchange

Technically, cross-border data exchange should not differ from data exchange on a national level. APIs enable data exchange across borders, just like within a single country. However, in practice, there are probably more differences in the APIs between authorities of two countries than between two authorities of the same country because some sort of guidelines is likely to exist nationally. Generally, API guidelines and best practices are global and utilise various internet standards. However, the challenge is that many commonly used guidelines and practices are not official standards, leading to differences in implementation between authorities and countries. This does not prevent the data exchange, but the implementation requires more effort.

When it comes to the bigger picture, APIs alone are not enough for the implementation of successful cross-border data exchange. Like TOOP, it also requires secure data exchange solutions, compatible data models, and semantic interoperability. Also, legal and administrative questions play a significant role – often, their part is even greater than technical questions. There may be legal barriers, and in many cases, agreements and contracts are required at two levels – between the countries whose authorities exchange data and between the parties that implement the data exchange in practice.

Not a silver bullet

APIs play an essential role in digital government services. Without them, many of today's and tomorrow's digital services would not be possible or would require a considerable amount of work in the form of custom integrations. Therefore, APIs are a key enabler in digital transformation. Still, other vital elements are also needed in addition to APIs, such as secure data exchange solutions, unified data models, and semantic interoperability. These areas can and should be considered when APIs are designed and implemented. Interoperability on a broader scale requires standards, common guidelines, practices, and collaboration across administration sectors, borders, and public and private sectors.

Interoperability is not just about technology – it includes legal, organisational, and semantical layers as well. The other layers are equally important and require collaboration across administration sectors and borders. Otherwise, there is a risk that innovations cannot be utilised, or they can be used only partly because of legal or administrative restrictions.

All in all, APIs are one of the key enablers in the digital government context. Still, they are not a silver bullet alone enough to resolve all the interoperability challenges. Collaboration in all areas of interoperability and the use of open standards, frameworks, and open-source solutions are the key to success.

Petteri Kivimäki is the CTO of the Nordic Institute for Interoperability Solutions, a non-profit association dedicated to the development and strategic management of X-Road® and other cross-border components for digital government infrastructure. He was the technical lead of the X-Road implementation project in Finland, as well as the coordinator of the joint open-source development of the X-Road solution between Finland and Estonia. Petteri holds a Bachelor of Science in Software Engineering from the Metropolia University of Applied Sciences, Finland, and he is a certified cloud and technology architect.

Links & References

[1] https://joinup.ec.europa.eu/collection/nifo-national-interoperability-framework-observatory/european-interoperability-framework-detail

[2] https://ec.europa.eu/digital-building-blocks/wikis/display/DIGITAL/eDelivery

[3] https://x-road.global/

[4] https://toop.eu/once-only

IT depends...

API Gateway or Just a Service Mesh Tool?

Large software systems usually do not exist alone and often have many partner systems calling its APIs. The number of partner systems can quickly reach double digits. The smaller the services are cut, which currently tends to happen in projects, the higher the number of partner systems that must be called. An extensive communication network is thus established: a so-called service mesh.

by Michael Hofmann

The topic of APIs is even more important today than it was in the past, because the mistakes of earlier times are not wanted or allowed to be made again today. Fortunately, direct access to the database of another service is no longer on the list of project managers or software architects. Instead, more and more interfaces are emerging as APIs according to Richardson's REST Maturity Model. At the same time, there is a growing desire for coordinated, controlled, and managed access to APIs. Which is not very surprising considering the increasing number of APIs.

The consideration of using an API gateway to manage these interfaces in an initial response is perfectly understandable. But on closer inspection, the question arises as to which functions of the API gateway should be used. Often, the requirements for managing and operating APIs are less than the feature set that API gateways provide. On the other hand, to manage the service mesh, one should think about using a suitable service mesh tool. From a certain size of the service mesh or a certain complexity of the communication behavior of the services, there is no way around it. Istio would be a representative of the service mesh tools that specialize in operation on a cloud platform.

The trend towards operating services in the cloud has also been taken up by API gateway manufacturers. They are currently changing their systems away from monolithic gateways to so-called micro-gateways, which are more in line with cloud philosophy.

Thus, the question arises whether one wants to fulfill the requirements for the operation and management of the APIs with a more or less classic API gateway or whether it is better to switch to a service mesh tool. Or is a mix of these two tools the better solution approach?

Functional scope of "classic" API gateways

This is not the place for a product comparison of different API gateways, but all common representatives of this guild, such as Google Cloud Apigee, Red Hat 3scale, MuleSoft, Kong or WSO2, generally offer the following functionalities.

API applications: This refers to the logical grouping of different APIs into a common application that can then be administered and operated as a whole.

Rate Limiting and Throttling: This defines how many API calls may be made within a certain time interval. If this limit is exceeded, the call is rejected with an error message. In most cases, an HTTP status code 429 ("Too Many Requests") is sent to the caller. The limit is of a technical nature in order not to overload the called

system. Throttling occurs when no more requests are allowed within the specified time interval after the limit has been exceeded. Only after this interval has elapsed further requests are allowed again.

API Quota: This functionality is focusing the commercial aspect of API access. As a rule, the call limit is agreed upon here for a longer period than is the case with rate limiting. For example, one specifies that the API may only be called a thousand times per month. In addition, this limit is set separately for each consumer and then often also leads to the possibility of separate billing of call costs per consumer. Some API gateways allow defining these quotas on API applications.

Load balancing with failover: Since every request to an API is routed through the API gateway, they are also able to offer more or less extensive load balancing with integrated failover. It should be analyzed in detail whether the API gateway is able to react dynamically to changes in the runtime availability of the services or whether it only enables a static configuration of the existing API endpoints.

Access Control: Today, no productive system can do without security and access control. It goes without saying that API gateways also offer this. Different security settings can be defined for different API applications in order to regulate the access options of the various user groups to the APIs. A connection to common OAuth or OpenID Connect servers is now state of the art. Setting or manipulating the HTTP header is also part of the gateway's basic equipment.

Logging: Every request that is controlled by the gateways must of course also be logged. Logging with a higher or lower level of detail is possible in every API gateway.

Management GUI: For managing the APIs, the common API gateways all offer a graphical user interface. However, a technical interface for automated configuration of the APIs is not available in each of the API gateways. Deployment of new or modified APIs, triggered by an event in the CI/CD pipeline, is therefore not always easy to implement.

Monitoring with analytics: Some of the gateways offer extensions that can be used to further evaluate the collected call data. The spectrum of applications ranges from simple monitoring to complex analyses of the use of the APIs. Some business models of API operators can no longer do without such analytics results.

Developer portal: The API gateways offer so-called developer portals to initiate external API partners or to simplify API client development. There, it is often possible to access released APIs for testing one's own client via self-registration. These test environments are then operated in a sandbox. Some APIs also offer SDKs for the common programming languages to simplify entry into API development.

API gateway functionality in a service mesh tool

In the following, the functionality of the API gateways will be compared on the basis of the service mesh tool Istio. For an explanation of Istio, please refer to the article "Don't be afraid of the service mesh" (p. 26).

A few remarks beforehand: Istio only covers the technical aspects of the service mesh. For the commercially oriented functionalities, such as billing, Istio currently has no corresponding capabilities. Istio also does not have analytics capabilities. Anyone who wants to analyze the calls to their APIs by customers even more precisely in order to draw more conclusions about their behavior will, unfortunately, come up empty with Istio. This is where API gateways offer add-on components that better handle the analytics space. Istio offers extension points for the technical aspect of rate limiting and throttling. API Quota functionality that goes beyond pure rate limiting, such as limiting API calls for a specific partner in a period of one month, is possible with the same extension points for rate limiting. These extension

Listing 1

```
apiVersion: "networking.istio.io/v1alpha3"
kind: Gateway
metadata:
  name: mesh-gateway
spec:
  selector:
    istio: ingressgateway # use Istio default gateway implementation
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    # this gateway is for requests coming from all other hosts
    - "*"
```

points have to be connected with additional services, installed and managed separately. Nevertheless, Istio offers functionalities that are quite comparable to those of API gateways.

Ingress Gateway and Virtual Service

An adequate way to logically group APIs into applications can be done with Istio through various Istio Ingress Gateways in combination with Istio's Virtual Service. Ingress Gateways control the entry into the service mesh in Istio. They can be defined with different routings in the Virtual Services. From a technical point of view, these routings are thus managed together.

The following Istio rules define an ingress gateway on port 80 for HTTP accesses and connects it to a Virtual Service that redirects to the myservice service in version v1 for request URLs with the /status or /delay prefix (Listings 1 and 2). This combination of Istio rules is arbitrarily extensible and thus covers all requirements that allow common control of API access.

Istio's Rate Limits

A prerequisite for the dynamic limitation of requests to a service by Istio is the activation of an additional backend and some configurations to connect to this backend. The first approach of Istio to offer rate limits was a so-called Policy Enforcement rule. This rule was declared deprecated with Istio 1.6. Starting from Istio 1.9 there are two new alternatives to define rate limits: EnvoyFilter and WebAssembly. Both alternatives enhance the Envoy proxy and delegate the service requests to a rate limiting backend to check for allowing this request. A reference implementation for this backend service, written in Go with a Redis backend, exists. The open source community maintaining the Envoy proxy is also responsible for the Envoy RateLimit service. A Redis server is needed to manage the quota values inside the service mesh.

Envoy checks every HTTP request inside the service mesh against all settings of the HTTP Filter. This HTTP filter can be enhanced with a special Istio Rule EnvoyFilter. In this rule all necessary settings must be defined to connect and delegate the checks to a special rate limiting backend service. Based on the response of this service, Envoy proxy decides what to do with the request. When a predefined limit is reached, the Envoy proxy interrupts the access with an HTTP status code 429 ("Too Many Requests").

WebAssembly on the other hand is a sandboxing technology to also enhance the Envoy proxy. A WebAssembly plugin can be developed in several programming languages and gets executed in a special WebAssembly Runtime embedded in the Envoy proxy. It is planned to have a growing ecosystem of WebAssemblies. Only the creativity of a programmer limits this open programming model and rate limiting can be one reason to use this upcoming technology.

Inside the Envoy RateLimit service, the evaluation window in which the analysis takes place can be selected as fixed or rolling. With a fixed evaluation window, the limits apply to the period from, for example, 9:00 to 18:00. The rolling window refers to the period of the last 10 minutes, for example. To establish more complex rate limits, multiple limits can be defined, which are then evaluated and monitored in the specified order. It is also possible to distinguish whether the request comes

Listing 2

```
apiVersion: "networking.istio.io/v1alpha3"
kind: VirtualService
metadata:
  name: httpbin
spec:
  hosts:
  # this VS is for requests coming from all other hosts
  - "*"
  gateways:
  # and is bound to the following istio ingress gateway
  - mesh-gateway
  http:
  - match:
    - uri:
        prefix: /status
    - uri:
        prefix: /delay
    route:
    - destination:
        port:
          number: 8000
        host: myservice
        # this subset must be defined in a DR
        subset: v1
```

from a logged-in user or not by validating an existing JSON web token (JWT). Other HTTP request headers can also be evaluated. In addition, special limits can be established if the request is from a specific IP address. These are just some of the possibilities Envoy RateLimit service offers.

Istio has clearly focused on the technical limitation of the requests. This also makes it possible to avoid denial-of-service attacks (DoS).

A so-called API quota, as with an API gateway, which is basically used for a billing model, can also be implemented with Istio's rate limit extensions, but the API gateways offer these possibilities out of the box. On the other hand the flexibility of Istio's rate limit is determined by the possibilities of the rate limiting backend.

Load balancing and resilience

Istio, which is built on Kubernetes (other platforms are also possible), works closely with Kubernetes when it comes to load balancing. The runtime information of the available Kubernetes pods that are accessed via a Kubernetes service is also available to the Envoy proxy. This enables the Envoy proxy to establish a client-side load balancing. Istio's Control Plane regularly informs itself about the currently available pods of a service and forwards this information to the sidecar. The sidecar can then intelligently distribute the load among the available pods. Together with the resilience rules (timeout, retry, circuit breaker, and bulkhead), which are also evaluated in the sidecar, problems in the calls can be compensated. Thus Istio has capabilities that go far beyond those of an API gateway.

Security

Istio provides its own security module (Citadel), which takes care of certificates and mutual TLS. In addition, Istio can be provided with Role-based Access Control (RBAC) settings and an integration with JWT based authentication is possible out of the box. Furthermore, the evaluation or manipulation of request header values has been possible in Istio for a long time.

Starting with mTLS, as one aspect of the wide range of security, it is very easy to define how traffic will be encrypted or not. By defining an Istio rule PeerAuthentication mTLS can be defined for the whole mesh, only for a Kubernetes namespace and even only for some services. To establish a migration path, mTLS can be defined in different modes: PERMISSIVE or STRICT. A PERMISSIVE connection can be either plaintext or mTLS tunnel. STRICT forces a mTLS connection. Istio manages all necessary SSL certificates out of the box and frees the admin from this annoying activity. The following listing shows an Istio rule which forces mTLS tunnels on all workloads in namespace foo:

To enable access control on workloads in the service mesh two different Istio rules must be defined. The first rule, a RequestAuthentication defines what request authentication methods are supported. It validates the JWT in the authorization header and checks whether it was issued by the corresponding OpenId Connect server. In listing 4 all requests to httpbin workload in namespace foo will be authenticated with credentials derived from a JWT issued by issuer-foo coming from OpenId Connect server example.com:

The second rule, an AuthorizationPolicy, enables all access controls based on the JWT claim values validated by the previous rule. The scope of this rule also ranges from a complete service mesh to a single workload. Multiple rules of this type will be evaluated in a predefined

Listing 3

```
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: default
  namespace: foo
spec:
  mtls:
    mode: STRICT
```

Listing 4

```
apiVersion: security.istio.io/v1beta1
kind: RequestAuthentication
metadata:
  name: httpbin
  namespace: foo
spec:
  selector:
    matchLabels:
      app: httpbin
  jwtRules:
  - issuer: "issuer-foo"
    jwksUri: https://example.com/.
well-known/jwks.json
```

Listing 5

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
 name: httpbin
 namespace: foo
spec:
 action: ALLOW
 rules:
 - from:
   - source:
       namespaces: ["test"]
   to:
   - operation:
       paths: ["/data"]
   when:
   - key: request.auth.claims[iss]
     values: ["https://accounts.google.com"]
```

Listing 6

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
 name: httpbin
 namespace: foo
spec:
 action: DENY
 rules:
 - from:
   - source:
       namespaces: ["dev"]
   to:
   - operation:
       methods: ["POST"]
```

order and if one rule matches, the request will be forwarded to the service. Listing 5 defines an AuthorizationPolicy which allows requests from namespace test to HTTP Endpoint /data in namespace foo only if the issuer of the JWT is https://accounts.google.com:

An AuthorizationPolicy can also be defined as a DENY-Rule. Listing 6 shows a rule to stop all HTTP POST requests from namespace dev to namespace foo:

The previous security listings show only a small range of Istio's security capabilities. Thus, also in terms of security requirements, the functional scope of Istio is comparable to that of an API gateway, if Istio does not even surpass the possibilities of API gateways here.

Logging and tracing in the service mesh

Without sufficient logging, no reasonable operation of a service mesh is possible, as this usually requires a high number of services. For this, Istio relies on the possibilities offered by Kubernetes or the Docker containers. Istio's homepage describes how logging can be set up with Fluentd. Creating a so-called EFK logging stack (Elasticsearch, Fluentd, Kibana) is thus very easy. Istio's own components also use this logging stack.

With the possibility of distributed tracing based on Jaeger or other components following the OpenTracing standard, Istio offers a functionality that is naturally not included in API gateways.

Kiali as a management GUI for Istio

Meanwhile, a GUI also exists in Istio to view your Istio rules and other information important for managing the service mesh. Kiali already provides help to get an overview of the service mesh. Since Istio was first started with a set of rules based on YAML files, it can be configured very well with scripts. The execution of the scripts can be integrated into an existing CI/CD pipeline. This gives Istio a plus point, as scripting in API gateways is not as prominent everywhere.

Developer portal

Istio does not offer the possibility to generate an SDK for the client developer. Self-registration must also be done with other systems. Only when it comes to sandboxing Istio is as good or bad as the API gateways. Usually, the biggest effort of sandboxing is to establish the appropriate test or simulation environment. Once that is accomplished, managing the sandboxes is only a much smaller effort. Deleting, restoring, and assigning the sandbox can be done very easily thanks to Kubernetes and Istio's scripting capabilities.

Conclusion

At the end of this article, let's return to the beginning and the question: How do you want to provide other systems with coordinated access to your APIs? In order to find a way out of the typical consultant answer "it depends", one should consider exactly which functions of an API gateway are desired. Are the possibilities of Istio or the other service mesh tools sufficient – especially under the aspect that from a certain number of services onwards a service mesh tool can no longer be dispensed with? The consequence of this would be to operate a pure API gateway as an additional component.

Because of the trend toward self-responsibility in projects and the associated self-responsible operation of all components, project managers should consider carefully whether additional systems are necessary to achieve the project requirements. However, these considerations should not be turned into the opposite by implementing missing functionality oneself at great expense.

For smaller infrastructures where it is not yet necessary to use a service mesh tool, it may make sense to use an API gateway. The same applies to infrastructures that have to get by without Kubernetes. It is often sufficient to start with a small gateway solution and only later switch to the full range of functions of an API gateway. With growing service landscapes, it will probably not be possible to do without the further advantages of Istio. But then the question arises again whether you can manage with Istio alone and whether you can or want to do without the functions of the API gateway.

It depends!

Michael Hofmann is a freelance consultant, coach, speaker, and author. He has extensive project experience in software architecture, Java Enterprise, and DevOps in both German and international environments.

Links & References

[1] https://martinfowler.com/articles/richardsonMaturityModel.html

Intro to Node.js part 2

Developing Web APIs with Node

One of the most common uses of Node.js is the development of web APIs. Numerous modules from the community are available for this, covering a whole range of aspects, such as routing, validation, and CORS.

by Golo Roden

The first part of this series introduced Node.js as a server-side runtime environment for JavaScript and showed how to write a simple web server. In addition, the package management npm was introduced, which allows us to easily install modules written by the community into our own application. So, we already know some of the basics, but the developed application still lacks meaningful functionality.

This will change in this part of the series: The application, which so far only launches a rudimentary web server, is supposed to provide an API that can be used to manage a task list. First, it is necessary to make some technical preliminary considerations, because we must define what exactly the application is supposed to do. For example, the following functions are possible:

- It must be possible to write down a new task. In the simplest form, this task consists of only a title, which must not be empty.
- It must also be possible to call up a list of all tasks that still need to be done, in order to see what still needs doing.
- Last but not least, it must be possible to check off a completed task so that it is removed from the todo list.

These three functions are essential, without them a task list cannot be used meaningfully. All other functions, such as renaming a task or undoing the check-off of a task, are optional. Of course, it would make sense to implement them in order to make the application as user-friendly and convenient as possible – but they are not really necessary. The three functions mentioned above represent the scope of a Minimum Viable Product (MVP), so to speak.

Another restriction should be specified right at the beginning: The task list shall deliberately not have user management in order to keep the example manageable. This means that there will be neither authentication nor authorization, and it will not be possible to manage multiple task lists for different people. This would

Listing 1

```
'use strict';


const getApp = require('./lib/getApp');
const http = require('http');
const { processenv } = require('processenv');


const port = processenv('PORT', 3000);


const server = http.createServer(getApp());


server.listen(port);
```

be essential to use the application in production, but it is beyond the scope of this article and ultimately offers little learning for Node.js.

Current state

The current state of the application we wrote in the first part includes two code files: app. js, which starts the actual server, and lib/getApp.js, which contains the functionality to respond to requests from the outside. In the app.js file, we already used the npm module processenv [1] to be able to set the port to a value other than the default 3000 via an environment variable (Listing 1).

The good news is that at this point, nothing will change in this file. This is because there is already a separation of content in the app.js and getApp.js files: The first file takes care of the HTTP server itself, while the second contains the actual logic of the application. In this part of the article series, only the application logic will be adapted and extended, so the app.js file can remain as it is.

However, the situation is different in the getApp.js file, where we will leave no stone unturned. But, one thing at a time. First, the package.json file must be modified so that the name of the application is more meaningful. For example, instead of my-http-server, the application could be called tasklist:

```
{
  "name": "tasklist",
  "version": "0.0.1",
  "dependencies": {
    "processenv": "3.0.2"
  }
}
```

The file and directory structure of the application still looks the same as in the first part:

```
/
  lib/
    getApp.js
  node_modules/
  app.js
  package.json
  package-lock.json
```


REST? No thanks!

Now it's a matter of incorporating routing. As usual with APIs, this is done via different paths in the URLs. In addition, you can fall back on the different HTTP verbs such as GET and POST to map different actions. A common pattern is the so-called REST approach, which specifies that so-called resources are defined via the URL and the HTTP verbs define the actions on these resources. The usual mapping according to REST is as follows:

- POST creates a new resource, and corresponds to a Create.
- GET retrieves a resource, and represents the classic Read.
- PUT updates a resource, and corresponds to an Update.
- DELETE finally deletes a resource, and corresponds to a Delete.

As you can see, these four HTTP verbs can be easily mapped to four actions of the so-called CRUD pattern, which in turn corresponds to the common approach of how to access data in (relational) databases. This is one of the most important reasons for the success of REST: It is simple and builds on the already familiar logic of databases. Nevertheless, there are some reasons against using this transfer of CRUD to the API level. The most weighty of these is that the verbs do not conform to the technical language: Users do not talk about creating or updating a task.

Instead, they think in terms of technical processes: They want to make a note of a task or check off a task as completed. This is where a business and a technical view collide. It is obvious that a mapping between these views must take place at some point – but the code of an application should tend to be structured in a domain-oriented rather than a technical way [2]. After all, the application is written to solve a domain-oriented problem, and technology is merely the means to an end. Seen in this light, CRUD is also an antipattern [3].

An alternative approach is provided by the CQRS pattern, which is based on commands and queries [4]. A command is an action that changes the state of the application and reflects a user's intention. A command is usually in the imperative, since it is a request to the application to do something. In the context of the task list, there are two actions that change the state of the list, noting and checking off a task. If we formulate these actions in the imperative and translate them into English, we get phrases such as "Note a todo.", "Tick off a todo."

Analogously, you can formulate a query, i.e. a query that doesn't change the state of the application, but returns it. This is the difference between a command and a query: A command writes to the application, so to speak, while a query reads from the application. The CQRS pattern states that every interaction with an application should be either a command or a query – but never both at the same time. In particular, this means that Commands should not return the current state of the task list, but that a separate Query is needed for that: For example: "Get pending todos."

If we abandon the idea that an API must always be structured according to REST and prefer the much simpler pattern of separating writing and reading, the question arises as to how the URLs should be structured and which HTTP verbs should be used. In fact, the answer to this question is surprisingly simple: The URLs are formulated exactly as mentioned above, POST for commands, and GET for queries are used as HTTP verbs – that's it. This results in the following routes:

- POST /note-todo
- POST /tick-off-todo
- GET /pending-todos

Listing 2

```
'use strict';

const express = require('express');

const getApp = function () {
  const app = express();

  app.post('/note-todo', (req, res) =&amp;amp;amp;gt; {
    // ...
  });

  app.post('/tick-off-todo', (req, res) =&amp;amp;amp;gt; {
    // ...
  });

  app.get('/pending-todos', (req, res) =&amp;amp;amp;gt; {
    // ...
  });

  return app;
};

module.exports = getApp;
```

The beauty of this approach is that it is much more self-explanatory than REST. POST /tick-off-todo is much more technical than a PUT /todo. Here, it is clear that an update is executed, but which functional purpose this update has is unclear. When there are different reasons for initiating a (technical) update, the semantically stronger approach gains a lot in comprehensibility and traceability.

Define routes

Now it is necessary to define the appropriate routes. However, this is not done with Node.js's on-board tools. Instead, we can use the npm module Express [5]:

```
$ npm install express
$ npm install express
```

The module can now be loaded and used within the getApp.js file. First, an express application has to be defined, for which only the express function has to be called. Then, the get and post functions can be used to define routes, specifying the desired path name and a callback – similar to the one used in the standard Node.js server (Listing 2).

With this, the basic framework for the routes is already built. The individual routes can, of course, also be swapped out into independent files, but for the time being, focus should be on implementing functionality. The next step is to implement a task list, which is initially designed as a pure in-memory solution. However, since it will be backed by a database in a future part of this series, it will be designed from the outset to be seamlessly extensible later. Essentially, this means that all functions to access the task list will be created asynchronously, since accesses to databases in Node.js are usually asynchronous. For the same reason, an asynchronous initialize function is also created, which may seem unnecessary at this stage, but will later be used to establish the database connection.

Defining the todo list

The easiest way to do this is to use a class called Todos, to which corresponding methods are attached. Again, these methods should be named functionally and not technically, i.e. their names should be based on the names of the routes of the API. The class is placed in a new file in the lib directory, resulting in lib/Todos.js as the file name. For each task that is noted, an ID should also be generated, and the time of creation should be noted. While accessing the current time is not a problem, generating an ID requires recourse to an external module such as uuid, which can also be installed via npm:

```
$ npm install uuid
```

Last but not least, it is advisable to get into the habit from the very beginning of providing every .js file with

Listing 3

```
'use strict';


const { v4 } = require('uuid');


class Todos {
  constructor () {
    this.items = [];
  }


  async initialize () {
    // Intentionally left blank.
  }


  async noteTodo ({ title }) {
    const id = v4();
    const timestamp = Date.now();


    const todo = {
      id,
      timestamp,
      title
    };


    this.items.push(todo);
  }


  async tickOffTodo ({ id }) {
    const todoToTickOff = this.items.find(item =&amp;amp;amp;gt; item.
id === id);


    if (!todoToTickOff) {
      throw new Error('Todo not found.');
    }


    this.items = this.items.filter(item =&amp;amp;amp;gt; item.id !== id);
  }


  async getPendingTodos () {
    return this.items;
  }
}


module.exports = Todos;
```

strict mode, a special JavaScript execution mode in which some dangerous language constructs are not allowed, for example, the use of global variables. To enable the mode, you need to insert the appropriate string at the beginning of a file as a kind of statement. This makes the full contents of the app.js file look like the one shown in Listing 1.

It is striking in the implementation that the functions representing a command actually contain no return,

Listing 4

```
'use strict';


const express = require('express');
const Todos = require('./Todos');


const getApp = async function () {
  const todos = new Todos();
  await todos.initialize();


  const app = express();


  app.post('/note-todo', async (req, res) =&amp;amp;amp;gt; {
    const title = // ...


    await todos.noteTodo({ title });
  });


  app.post('/tick-off-todo', async (req, res) =&amp;amp;amp;gt; {
    const id = // ...


    await todos.tickOffTodo({ id });
  });


  app.get('/pending-todos', async (req, res) =&amp;amp;amp;gt; {
    const pendingTodos = await todos.getPendingTodos();


    // ...
  });


  return app;
};


module.exports = getApp;
```

while the function representing a query consists of only a single return. The separation between writing and reading has become very clear.

Now the file getApp.js can be extended accordingly, so that an instance of the task list is created there and the routes are adapted in such a way that they call the appropriate functions. To prepare the code for later, the initialize function should be called now. However, since this is marked as async, the getApp function must call it with the await keyword, and therefore, must also be marked as asynchronous (Listing 4).

Before the application can be executed, three things have to be done:

- First, the title and id parameters must be determined from the request body.
- Second, the query route must return the read tasks to the client as a JSON array.
- Finally, the app.js file must be modified so that the getApp function is called asynchronously there.

Input and output with JSON

Fortunately, all three tasks are easy to accomplish. For the first task, it is first necessary to determine what a request from the client looks like, i.e. what form it takes.

Listing 5

```
app.post('/note-todo', async (req, res) =&amp;amp;gt; {
  const { title } = req.body;


  await todos.noteTodo({ title });
});


app.post('/tick-off-todo', async (req, res) =&amp;amp;gt; {
  const { id } = req.body;


  await todos.tickOffTodo({ id });
});
```

Listing 6

```
app.post('/tick-off-todo', async (req, res) =&amp;amp;gt; {
  const { id } = req.body;


  try {
    await todos.tickOffTodo({ id });
  } catch {
    res.status(404).end();
  }
});
```

In practice, it has proven useful to send the payload as part of a JSON object in the request body. For the server, this means that it must read this object from the request body and parse it. A suitable module called body-parser [6] is available in the community for this purpose and can be easily installed using npm:

```
$ npm install body-parser
```

It should be noted that the version number must always consist of three parts and follow the concept of semantic versioning [6]. In addition, however, dependencies can also be stored in this file, whereby required third-party modules are explicitly added. This makes it much easier to restore a certain state later or to get an overview of which third-party modules an application depends on. To install a module, call npm as follows:

```
$ npm install processenv
```

It can then be loaded with require:

```
const bodyParser = require('body-parser');
```

Since the parser will be available for several routes, it is implemented as so-called middleware. In the context of Express, middleware is a type of plug-in that provides functionality for all routes and therefore only needs to be registered once instead of individually for each route. This is done in Express via the app.use function. Therefore, it is important to insert the following line directly after creating the Express application: app.use(bodyParser.json());

Now the property body of the req object can be accessed within the routes, which was not available before. Provided a valid JSON object was submitted, this property now contains that very object. This allows the

two command routes to be extended, as shown in Listing 5.

Listing 7

```
'use strict';


const bodyParser = require('body-parser');
const express = require('express');
const Todos = require('./Todos');


const getApp = async function () {
  const todos = new Todos();
  await todos.initialize();


  const app = express();
  app.use(bodyParser.json());


  app.post('/note-todo', async (req, res) =&amp;amp;gt; {
    const { title } = req.body;


    await todos.noteTodo({ title });
    res.status(200).end();
  });


  app.post('/tick-off-todo', async (req, res) =&amp;amp;gt; {
    const { id } = req.body;


    try {
      await todos.tickOffTodo({ id });
      res.status(200).end();
    } catch {
      res.status(404).end();
    }
  });


  app.get('/pending-todos', async (req, res) =&amp;amp;gt; {
    const pendingTodos = await todos.getPendingTodos();


    res.json(pendingTodos);
  });


  return app;
};


module.exports = getApp;
```

When implementing the tick-off-todo route, it is noticeable that error handling is still missing: If the task to be ticked off is not found, the tickOffTodo function of the Todos class raises an exception – but this is not caught at the moment. So it is still necessary to provide the corresponding call with a try/catch and to return a corresponding HTTP status code in case of an error. In this case, the error code 404, which stands for an element not found (Listing 6), is a good choice.

And finally, in addition to the node_modules directory, npm has also created a file called package-lock.json. It is actually used to lock version numbers despite the roof being specified. However, it has its quirks, so if npm behaves strangely, it's often a good idea to delete this file and the node_modules directory and run npm install again from scratch. Once a module has been installed via npm, it can be loaded in the same way as a module built into Node.js. In that case, Node.js recognizes that it is not a built-in module and loads the appropriate code from the node_modules directory:

```
app.get('/pending-todos', async (req, res) =&amp;amp;gt; {
  const pendingTodos = await todos.getPendingTodos();


  res.json(pendingTodos);
});
```

Now, if you start the server by entering node app.js and try to call some routes, you will notice that some of the routes work as desired – but others do not, because they never end. This is where an effect comes into play that is very unusual at first: Node.js is inherently designed to stream data, so an HTTP connection is not automati-

Listing 8

```
const noteTodoSchema = new Value({
  type: 'object',
  properties: {
    title: { type: 'string', minLength: 1 }
  },
  required: [ 'title' ],
  additionalProperties: false
});


const tickOffTodoSchema = new Value({
  type: 'object',
  properties: {
    id: { type: 'string', format: 'uuid' }
  },
  required: [ 'id' ],
  additionalProperties: false
});
```

cally closed when a route has been processed. Instead, it has to be done explicitly, as in the case of the 404 error. The json function already does this natively, but the two command routes still lack closing the connection successfully. To indicate that the operation was successful, it is a good idea to send the HTTP status code 200. The getApp.js file now looks like Listing 7.

Validate the inputs

What is still missing is a validation of the inputs: At the moment, it is quite possible to call one of the command routes without passing the required parameters in the request body. In practice, it has proven useful to validate JSON objects by using a JSON schema. A JSON schema represents a description of the valid structure of a JSON object. In order to be able to use JSON schemas, a module is again required, for example, validate-value [7] which can be installed via npm:

```
$ npm install validate-value
```

Now the module can be loaded in the getApp.js file:

```
const { Value } = require('validate-value');
```

Listing 9

```
app.post('/note-todo', async (req, res) =&amp;amp;gt; {
  if (!noteTodoSchema.isValid(req.body)) {
    return res.status(400).end();
  }

  const { title } = req.body;

  await todos.noteTodo({ title });
  res.status(200).end();
});

app.post('/tick-off-todo', async (req, res) =&amp;amp;gt; {
  if (!tickOffTodoSchema.isValid(req.body)) {
    return res.status(400).end();
  }

  const { id } = req.body;

  try {
    await todos.tickOffTodo({ id });
    res.status(200).end();
  } catch {
    res.status(404).end();
  }
});
```

The next step is to create two schemas. Since these are always the same, it is advisable not to do this inside the routes, but outside them, so that the code does not have to be executed over and over again, ultimately ending up with the same result each time (Listing 8).

Within the two command routes, the only thing left to do is to validate the received data using the respective schema, and in case of an error, return an appropriate HTTP status code, for example, a 400 error (Listing 9).

CORS and testing

With this the API is almost finished, only a little bit of small stuff is missing. For example, it would be handy to be able to configure CORS – that is, from which clients the server can be accessed. In practice, this topic is a bit more complex than described below, but for development purposes, it is often sufficient to allow access from everywhere. The best way to do this is to use the npm module cors [8], which must first be installed via npm:

```
$ npm install cors
```

It must then be loaded, which is again done in the getApp.js file:

```
const cors = require('cors');
```

Finally, it must be integrated into the express application in the same way as body-parser, because this module is also middleware. Whether this call is made before or after the body-parser does not really matter – but since access should be denied before the request body is processed, it makes sense to include cors as the first middleware:

```
// ...
const app = express();
app.use(cors());
app.use(bodyParser.json());
// ...
```

Now, in order to test the API, a client is still missing. Developing this right now would be too time-consuming, so you can fall back on a tool that is extremely practical for testing HTTP APIs and that is usually pre-installed on macOS and Linux, namely, curl. On Windows, it is also available, at least in the Windows Subsystem for Linux (WSL). First, you can try to retrieve the (initially empty) list of all tasks:

```
$ curl http://localhost:3000/pending-todos
[]
```

In the next step, you can now add a task. Make sure that you not only send the required data, but also set the Content-Type header to the correct value – otherwise the body-parser will not be active:

```
$ curl \
  -X POST \
  -H 'content-type:application/json' \
  -d '{"title":"Develop a Client"}' \
  http://localhost:3000/note-todo
```

If you retrieve the tasks again, you will get a list with one entry (in fact, the list would be output unformatted in a single line, but for the sake of better readability it is shown formatted in the following):

```
$ curl http://localhost:3000/pending-todos
[
  {
    "id": "dadd519b-71ec-4d18-8011-acf021e14365",
    "timestamp": 1601817586633,
    "title": "Develop a Client"
  }
]
```

If you try to check off a task that does not exist, you will notice that this has no effect on the list of all tasks. However, if you use the *-i* parameter of curl to also output the HTTP headers, you will see that you get the value 404 as the HTTP status code:

```
$ curl \
  -i \
  -X POST \
  -H 'content-type:application/json' \
  -d '{"id":"43445c25-c116-41ef-9075-7ef0783585cb"}' \
  http://localhost:3000/tick-off-todo
```

The same applies if you do not pass a UUID as a parameter (or specify an empty title in the previous example). However, in these cases, you get the HTTP status code 400. Last but not least, you can now try to actually check off the noted task by passing the correct ID:

```
$ curl \
  -X POST \
  -H 'content-type:application/json' \
  -d '{"id":"dadd519b-71ec-4d18-8011-acf021e14365"}' \
  http://localhost:3000/tick-off-todo
```

If you retrieve the list of all unfinished tasks again, you will get an empty list
– as desired:

```
$ curl http://localhost:3000/pending-todos
[]
```

Outlook

This concludes the second part of this series on Node.js. Of course, there is much more to discover in the context of Node.js and Express for writing Web APIs. Another article could be dedicated to the topics of authentication and authorization alone. But now we have a foundation to build upon.

The biggest shortcoming of the application at the moment is that it is not possible to ensure code quality and the code has already become relatively confusing. There is a lack of structure, binding specifications regarding the code style, and automated tests. These topics will be dealt with in the third part of the series – before further functionality can be added.

The author's company, the native web GmbH, offers a free video course on Node. js [9] with close to 30 hours of playtime. Episodes 4 and 5 of this video course deal with topics covered in this article, such as developing web APIs, using Express, and using middleware. Therefore, this course is recommended for anyone interested in more details.

Golo Roden is founder and CTO of the native web GmbH. He advises companies on technologies and architectures in the web and cloud environment, including TypeScript, Node.js, React, CQRS, event sourcing and Domain-Driven Design (DDD).

Web: www.thenativeweb.io

Links & References

[1] https://www.npmjs.com/package/processenv
[2] https://www.youtube.com/watch?v=YmzVCSUZzj0
[3] https://www.youtube.com/watch?v=frUNFrP7C9w
[4] https://www.youtube.com/watch?v=k0f3eeiNwRA
[5] https://www.npmjs.com/package/express
[6] https://www.npmjs.com/package/body-parser
[7] https://www.npmjs.com/package/validate-value
[8] https://www.npmjs.com/package/cors
[9] https://www.thenativeweb.io/learning/techlounge-nodejs