Traefik API Gateway for Microservices

With Java and Python Microservices Deployed in Kubernetes

Rahul Sharma
Akshay Mathur

Apress®

Traefik API Gateway for Microservices

With Java and Python Microservices Deployed in Kubernetes

Rahul Sharma
Akshay Mathur

Apress®

Traefik API Gateway for Microservices

Rahul Sharma
Patpargunj, Delhi, India

Akshay Mathur
Gurgaon, Haryana, India

ISBN-13 (pbk): 978-1-4842-6375-4
ISBN-13 (electronic): 978-1-4842-6376-1
https://doi.org/10.1007/978-1-4842-6376-1

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Divya Modi
Development Editor: Laura Berendson
Coordinating Editor: Divya Modi

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, Suite 4600, New York, NY 10004-1562, USA. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-6375-4. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

To our families, for all the personal time spent on this book

Table of Contents

About the Authors

Rahul Sharma is a seasoned Java developer with over 15 years of industry experience. In his career, he has worked with companies of various sizes, from enterprises to start-ups. During this time, he has developed and managed microservices on the cloud (AWS/GCE/DigitalOcean) using open source software. He is an open-source enthusiast and shares his experience at local meetups. He has co-authored *Java Unit Testing with JUnit 5* (Apress, 2017) and *Getting Started with Istio Service Mesh* (Apress, 2019).

Akshay Mathur is a software engineer with 15 years of experience, mostly in Java and web technologies. Most of his career has been spent building B2B platforms for enterprises, dealing with concerns like scalability, configurability, multitenancy, and cloud engineering. He has hands-on experience implementing and operating microservices and Kubernetes in these ecosystems. Currently, he enjoys public speaking and blogging on new cloud-native technologies (especially plain Kubernetes) and effective engineering culture.

About the Technical Reviewer

Brijesh is currently working as a lead consultant. He has more than ten years of experience in software development and providing IT solutions to clients for their on-premise or cloud-based applications, spanning from monoliths to microservice-based architecture.

Acknowledgments

This book would not have been possible without the support of many people. I would like to take this opportunity and express my gratitude to each of them.

I would like to thank Divya Modi for believing in the project and making it work. She has been instrumental in starting the project. Moreover, during the project, her editorial support provided a constant push throughout the process. It would have been difficult to deliver the project without your support.

I would like to thank Celestin Suresh John for providing me this wonderful opportunity. Your guidance made sure that we got the correct path outlined from the start.

I would like to thank Brijesh Pant and Laura C. Berendson for sharing valuable feedback. Your advice has helped to deliver the ideas in a better manner.

I would also like to thank my co-author Akshay Mathur for his knowledge and support. Your experience and willingness have made this a successful project. The brainstorming sessions we had helped to express the ideas.

I wish to thank my parents, my loving and supportive wife, Swati, and my son, Rudra. They are a constant source of encouragement and inspiration. Thanks for providing the time and listening to my gibberish when things were not working according to the plan.

Lastly, I would like to thank my friends, who have been my source of knowledge. The discussion we had often helped me to deliberate on various topics. Often our debates have provided the testbed for evaluations.

—Rahul Sharma

I'd like to express my gratitude to my co-author Rahul Sharma for bringing me on board this project. While we had discussed the possibility, the actual opportunity still came suddenly, and I'd like to thank him for his guidance through the process. It was as fun and nerve-wracking as I'd envisioned, and he had my back the whole way as I channeled my inner Douglas Adams and (to borrow from a great man) enjoyed the sound of deadlines whooshing by.

Divya Modi was a constant source of support and encouragement. She gently helped us stay on track and was a picture of calm and confidence, helping us get to completion. And still entertained multiple last-minute requests as I kept tweaking the title.

To our reviewers, Brijesh Pant and Laura C. Berendson, thank you for all the constructive feedback in making this book better.

I'd also like to thank my ever-patient family, especially my mother, my wife, Neha, and my daughter, Inara, who kept wondering when I would actually be *done* and free to talk to them, as I kept fiddling till the last minute and repeating, "Almost there!".

I'm grateful to the two mentors who pulled me up to the level of authoring a book—Aditya Kalia and Shekhar Gulati.

Finally, our gratitude to the Traefik team for releasing an excellent product to the community. We hope this book helps in any small way to drive further adoption.

—Akshay Mathur

Introduction

Microservice architecture has brought dynamism to the application ecosystem. New services are built and deployed while older ones are deprecated and removed from the enterprise application estate. But front-end load balancers haven't been able to adapt to the components in the enterprise architecture. Most current load balancers have a static configuration. They require configuration updates as the application landscape changes. Thus, there are operational complexities when working with microservices. These are a few of the challenges of getting a microservices-based solution to work. The dynamic nature of the ecosystem requires dynamic tools that can autoconfigure themselves.

Traefik bases its foundations on the dynamic nature of the Microservice architecture. It has built first-class support for service discovery, telemetry, and resiliency. It is a modern HTTP reverse proxy and load balancer that eases microservices deployment. Its integration has been great, with many existing tools.

The book covers Traefik setup, basic workings, and integration with microservices. It is intended for developers, project managers, and DevOps personnel interested in solutions for their operational challenges.
The book is not specific to any programming language, even though all the examples use Java or Python.

CHAPTER 1

Introduction to Traefik

Over the last couple of years, microservices have become a mainstream architecture paradigm for enterprise application development. They have replaced the monolithic architecture of application development, which was mainstream for the past couple of decades. Monolithic applications are developed in a modular architecture. This means that discrete logic components, called *modules*, are created to segregate components based on their responsibility. Even though an application consisted of discrete components, they were packaged and deployed as a single executable. Overall, the application has very tight coupling. Changes to each of these modules can't be released independently. You are required to release a complete application each time.

A monolithic architecture is well suited when you are building an application with unknowns. In such cases, you often need quick prototyping for every feature. Monolithic architecture helps in this case, as the application has a unified code base. The architecture offers the following benefits.

- Simple to develop.
- Simple to test. For example, you can implement end-to-end testing by launching the application and testing the UI with Selenium.
- Simple to deploy. You only have to copy the packaged application to a server.

R. Sharma and A. Mathur, *Traefik API Gateway for Microservices*,
https://doi.org/10.1007/978-1-4842-6376-1_1

- Simple to scale horizontally by running multiple copies behind a load balancer.

In summary, you can deliver the complete application quickly in these early stages. But as the application grows organically, the gains erode. In the later stages, the application becomes harder to maintain and operate. Most of the subcomponents get more responsibility and become large subsystems. Each of these subsystems needs a team of developers for its maintenance. As a result, the complete application is usually maintained by multiple development teams. But the application has high coupling, so development teams are interdependent while making new features available. Due to a single binary, the organization faces the following set of issues.

- **Quarterly releases**: Application features take more time to release. Most of the time, an application feature needs to be handled across various subsystems. Each team can do their development, but deployment requires the entire set of components. Thus, teams can seldom work independently. Releases are often a big coordinated effort across different teams, which can be done only a couple of times per period.
- **Deprecated technology**: Often, when you work with technology, you must upgrade it periodically. The upgrades make sure all vulnerabilities are covered. Application libraries often require frequent upgrades as they add new features as well. But upgrading the libraries in a monolith is difficult. A team can try to use the latest version, but often needs to make sure that the upgrade does not break other subsystems. In certain situations, an upgrade can even lead to a complete rewrite of subsystems, which is a very risky undertaking for the business.

- **Steep learning curve**: Monolithic applications often have a large code base. But the individual developers are often working on a very small subset of the codebase. At first glance, the lines of code create a psychological bottleneck for developers. Moreover, since the application is tightly coupled, developers usually need to know how others invoke the code. Thus, the overall onboarding time for a new developer is large. Even the experienced developers find it hard to make changes to modules that have not been maintained well. This creates a knowledge gap that widens over time.
- **Application scaling**: Typically, a monolithic application can only be scaled vertically. It is possible to scale the application horizontally, but you need to determine how each subsystem maintains its internal state. In any case, the application requires resources for all subsystems. Resources can't be selectively provided to subsystems under load. Thus, it is an all-or-nothing scenario with a monolithic application. This is often a costly affair.

When faced with challenges, organizations look for alternative architectures to address these issues.

Microservice Architecture

Microservice architecture is an alternative to the monolithic architecture (see Figure 1-1). It converts the single application to a distributed system with the following characteristics.

- **Services**: Microservices are developed as services that can work independently and provide a set of business capabilities. A service may depend on other services to perform the required functionality. Independent teams can develop each of these services. The teams are free to select and upgrade the technology they need for their service. An organization often delegates full responsibility for the services to their respective teams. The teams must ensure that their respective service runs as per the agreed availability and meets the agreed quality metrics.
- **Business context**: A service is often created around a business domain. This makes sure that it is not too fine-grained or too big. A service needs to answer first if it is the owner of the said business function or the consumer of the function. A function owner must maintain all the corresponding function data. If it needs some more supporting function, it may consume the same from another service. Thus determining business context boundaries helps keep a check on the service dependencies. Microservices aim to build a system with loose coupling and high cohesion attributes. Aggregating all logically related functionality makes the service an independent product.
- **Application governance**: In enterprise systems, governance plays an important role. You rarely want to make systems that are difficult to run. Due to this, a governance group keeps check on the technologies used by developers so that the operations team can still run the system. But microservice architecture provides the complete ownership to the respective

teams. The ownership is not limited to development. It also delegates service operations. Due to this, most organizations must adopt DevOps practices. These practices enable the development teams to operate and govern a service efficiently.

- **Automation**: Automation plays an important role in microservices. It applies to all forms like infrastructure automation, test automation, and release automation. Teams need to operate efficiently. They need to test more often and release quickly. This is only possible if they rely more on machines and less on manual intervention. Post-development manual testing is a major bottleneck. Thus, teams often automate their testing in numerous ways like API testing, smoke testing, nightly tests, and so forth. They often perform exploratory testing manually to validate the build. Release and infrastructure preparation is often automated by using DevOps practices.

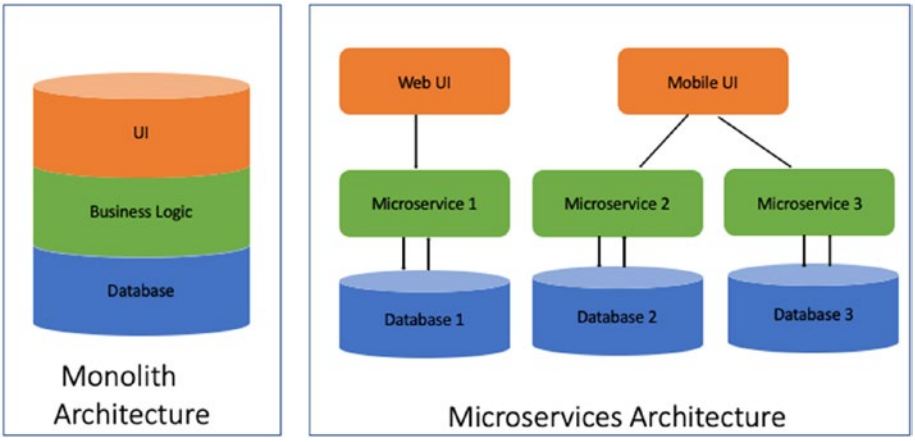

Figure 1-1. *Monolith vs. microservices*

In summary, a monolith has a centralized operating model. This means that all code resides in one place; everyone uses the same library, releases happen simultaneously, and so forth. But on the other end, microservices is a completely decentralized approach. Teams are empowered to make the best decisions with complete ownership. Adopting such an architecture not only asks for a change in software design, but it also asks for a change in organizational interaction. Organizations reap the following benefits of such application design.

Agility

This is one of the biggest driving factors for an organization adopting the microservices architecture. Organizations become more adaptive, and they can respond more quickly to changing business needs. The loose coupling offered by the architecture allows accelerated development. Small, loosely coupled services can be built, modified, and tested individually before deploying them in production. The model dictates small independent development teams working within their defined boundaries. These teams are responsible for maintaining high levels of software quality and service availability.

Innovation

The microservice architecture promotes independent small development teams supporting each service. Each team has ownership within their service boundary. They are not only responsible for development but also for operating the service. The teams thus adopt a lot of automation and tools to help them deliver these goals. These high-level goals drive the engineering culture within the organization.

Moreover, development teams are usually well aware of the shortcomings of their services. Such teams can address these issues using their autonomous decision-making capability. They can fix the issues and improve service quality frequently. Here again, teams are fully empowered to select appropriate tools and frameworks for their purpose. It ultimately leads to the improved technical quality of the overall product.

Resilience

Fault isolation is the act of limiting the impact of a failure to a limited subsystem/component. This principle allows a subsystem to fail as long as it does not impact the complete application. The distributed nature of microservice architecture offers fault isolation, a principal requirement to build resilient systems. Any service which is experiencing failures can be handled independently. Developers can fix issues and deploy new versions while the rest of the application continues to function independently.

Resilience, or *fault tolerance*, is often defined as the application's ability to function properly in the event of a failure of some parts. Distributed systems like microservices are based on various tenets like circuit breaking, throttling to handle fault propagation. This is an important aspect; if done right, it offers the benefits of a resilient system. But if this is left unhandled, it leads to frequent downtime due to failures cascading. Resilience also improves business agility as developers can release new services without worrying about system outages.

Scalability

Scalability is defined as the capability of a system to handle the growth of work. In a monolith, it is easy to quantify the system scalability. In a monolithic system, as the load increases, not all subsystems get proportionally increased traffic. It is often the case that some parts of the

system get more traffic than others. Thus, the overall system performance is determined by a subset of the services. It is easier to scale a monolithic system by adding more hardware. But at times, this can also be difficult as different modules may have conflicting resource requirements. Overall an overgrown monolith underutilizes the hardware. It often exhibits degraded system performance.

The decoupling offered by microservices enables the organization to understand the traffic that each microservice is serving. The *divide and conquer* principle helps in improving the overall system performance. Developers can adopt appropriate task parallelization or clustering techniques for each service to improve the system throughput. They can adopt appropriate programming languages and frameworks, fine-tuned with the best possible configuration. Lastly, hardware can be allocated by looking into service demand rather than scaling the entire ecosystem.

Maintainability

Technical debt is a major issue with monolithic systems. Overgrown monoliths often have parts that are not well understood by the complete team. Addressing technical debt in a monolith is difficult as people often fear of breaking any of the working features. There have been cases where unwanted dead code was made alive by addressing technical debt on a particular monolith.

Microservice architecture helps to mitigate the problem by following the principle of divide and conquer. The benefits can be correlated with an object-oriented application design where the system is broken into objects. Each object has a defined contract and thus leads to improved maintenance of the overall system. Developers can unit test each of the objects being refactored to validate the correctness. Similarly, microservices created around a business context have a defined contract. These loosely coupled services can be refactored and tested individually.

Developers can address the technical debt of the service while validating the service contract. Adopting microservices is often referred to as a monolith's *technical debt payment*.

You have looked at the advantages of Microservice architecture. But the architecture also brings a lot of challenges. Some challenges are due to the distributed nature of the systems, while others are caused by diversity in the application landscape. Services can be implemented in different technologies and scaled differently. There can be multiple versions of the same service serving different needs. Teams should strategize to overcome these challenges during application design and not as an afterthought. Application deployment is one such important aspect. Monoliths have been deployed on a three-tier model. But the same model does not work well with microservices. The next section discusses the changes required in the deployment model.

n-Tier Deployment

n-tier deployment is a design implementation where web applications are segregated into application presentation, application processing, and data management functions. These functions are served by independent components known as *tiers*. The application tiers allow segregation of duties. All communication is linear across the tiers. Each tier is managed by its own software subsystem. The *n*-tier deployment offers the benefit of improved scalability of the application. Monolithic applications are usually deployed as three-tiers (see Figure 1-2) applications.

- **Presentation tier**: This tier is responsible for serving all static content of the application. It is usually managed by using web servers like Apache, Nginx, and IIS. These web servers not only serve applications static UI components but also handle dynamic content by routing requests to the application tier. Web servers

are optimized to handle many requests for static data. Thus, under load, they perform well. Some of these servers also provide different load balancing mechanisms. These mechanisms can support multiple nodes of the application tier.

- **Application tier**: This tier is responsible for providing all processing functions. It contains the business processing logic to deliver the core capabilities of an application. The development team is responsible for building this in a suitable technology stack like Java, Python, and .NET. This tier is capable of serving a user request and generating an appropriate dynamic response. It receives requests from the presentation tier. To serve the request, the application tier may need additional data to interact with the data tier.
- **Data tier**: This tier provides capabilities of data storage and data retrieval. These data management functions are outside the scope of the application. Thus, an application uses a database to fulfill these needs. The data tier provides data manipulation functions using an API. The application tier invokes this API.

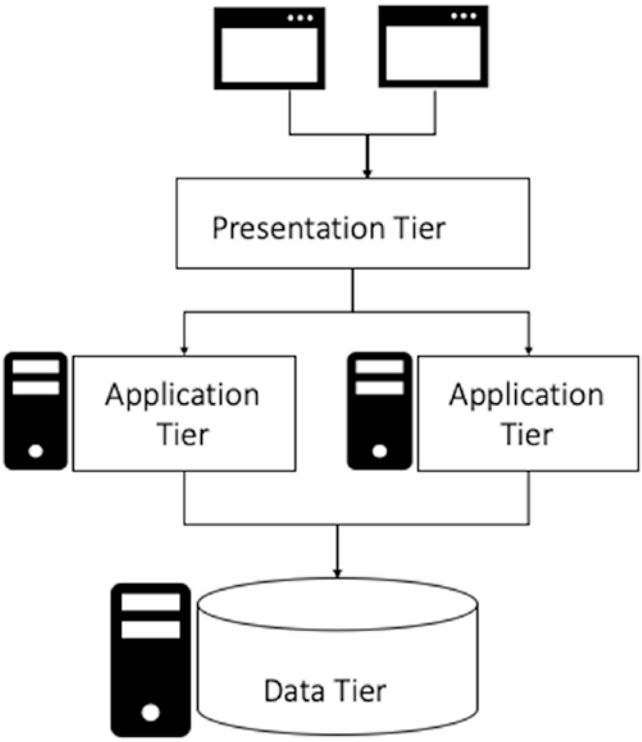

Figure 1-2. *Three-tier*

There are many benefits to using a three-layer architecture, including scalability, performance, and availability. You can deploy the tiers on different machines and can use the available resources in an optimized manner. The application tier delivers most of the processing capability. Thus, it needs more resources. On the other hand, the web servers serve static content and do not need many resources. This deployment model improves application availability by having different replication strategies for each tier.

Four-Tier Deployment

The three-tier deployment works in line with monolith applications. The monolith is usually the application tier. But with microservices, the monolith is converted into several services. Thus the three-tier deployment model is not good enough to handle microservice architecture. It needs the following four-tier deployment model (see Figure 1-3).

- **Content delivery tier**: This tier is responsible for delivering the content to the end user. A client can use an application in a web browser or on a mobile app. It often asks for making different user interfaces targeted across different platforms. The content delivery tier is responsible for ensuring that the application UI is working well on these different platforms. This tier also abstracts the services tier and allows developers to quickly develop new services for the changing business needs.
- **Gateway tier**: This tier has two roles.
  - Dynamically discover the deployed services and correlate them with the user request
  - Route requests to services and send responses

  For each request, the gateway layer receives data from all the underlying services and sends back a single aggregated response. It has to handle different scenarios like role-based access, delayed responses, and error responses. These behaviors make it easier for the service tier. The service tier can focus only on the business requirements.
- **Services tier**: This tier is responsible for providing all business capabilities. The services tier is designed for a microservices approach. This tier provides data to its clients without concern for how it is consumed. The

clients can be other services or application UI. Each of the services can be scaled based on their requests load pattern. The clients have the responsibility to determine the new instances. All of this enables a pluggable approach to the application ecosystem. New services can be built by consuming existing services. They can be readily integrated into the enterprise landscape.

- **Data tier**: This tier provides capabilities of data storage and data retrieval. Data management capabilities are still beyond the application scope. But each service has an exclusive data management infrastructure. It can be DBMS like MySQL or a document store like Mongo.

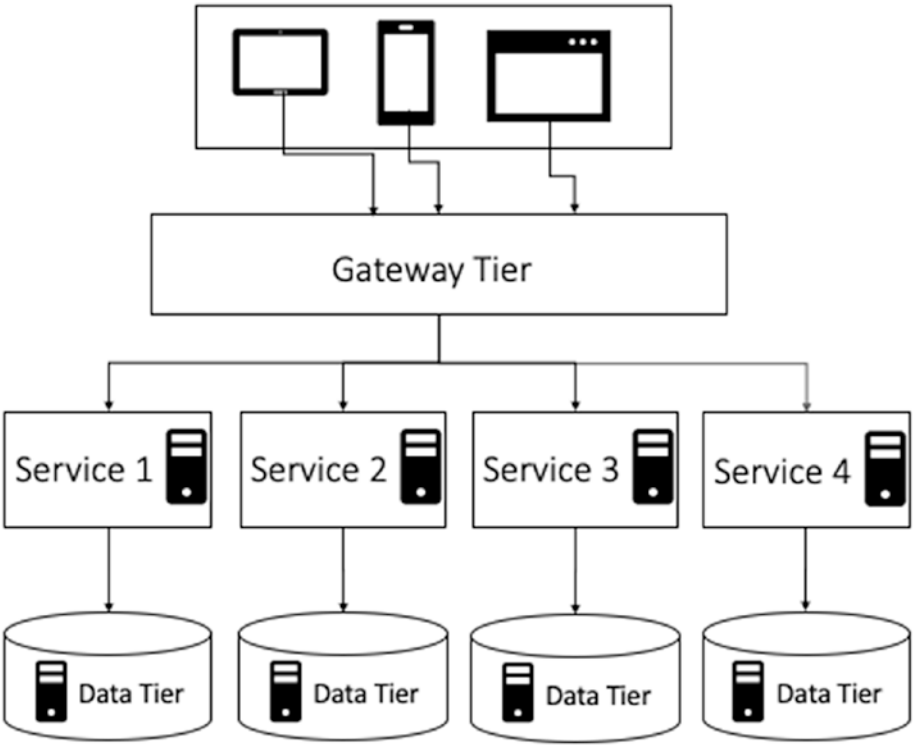

Figure 1-3. *Four-tier*

The four-tier architecture (see Figure 1-3) was pioneered by early microservices adopters like Netflix, Amazon, and Twitter. At the center of the paradigm, the gateway tier is responsible for binding together the complete solution. The gateway needs a solution that can link the remaining tiers together so all of them can communicate, scale, and deliver. In the three-tier architecture, the presentation tier had webservers that can be adopted for the gateway tier. But first, you should determine the characteristics required to be a gateway tier solution.

Gateway Characteristics

A gateway is the point of entry for all user traffic. It is often responsible for delegating the requests to different services, collate their responses, and send it back to the user. Under microservice architecture, the gateway must work with the dynamic nature of the architecture. The following sections discuss the different characteristics of the gateway component.

Application Layer Protocols

The OSI networking model handles traffic at Layer 4 and Layer 7. Layer 4 offers only low-level connection details. Traffic management at this layer can only be performed using a protocol (TCP/UDP) and port details. On the other hand, Layer 7 operates at the application layer. It can perform traffic routing based on the actual content of each message. HTTP is one of the most widely used application protocols. You can inspect HTTP headers and body to perform service routing.

Layer 7 load balancing enables the load balancer to make smarter load-balancing decisions. It can apply various optimizations like compressions, connection reuse, and so forth. You can also configure buffering to offload slow connections from the upstream servers to improve overall throughput. Lastly, you can apply encryption to secure our communication.