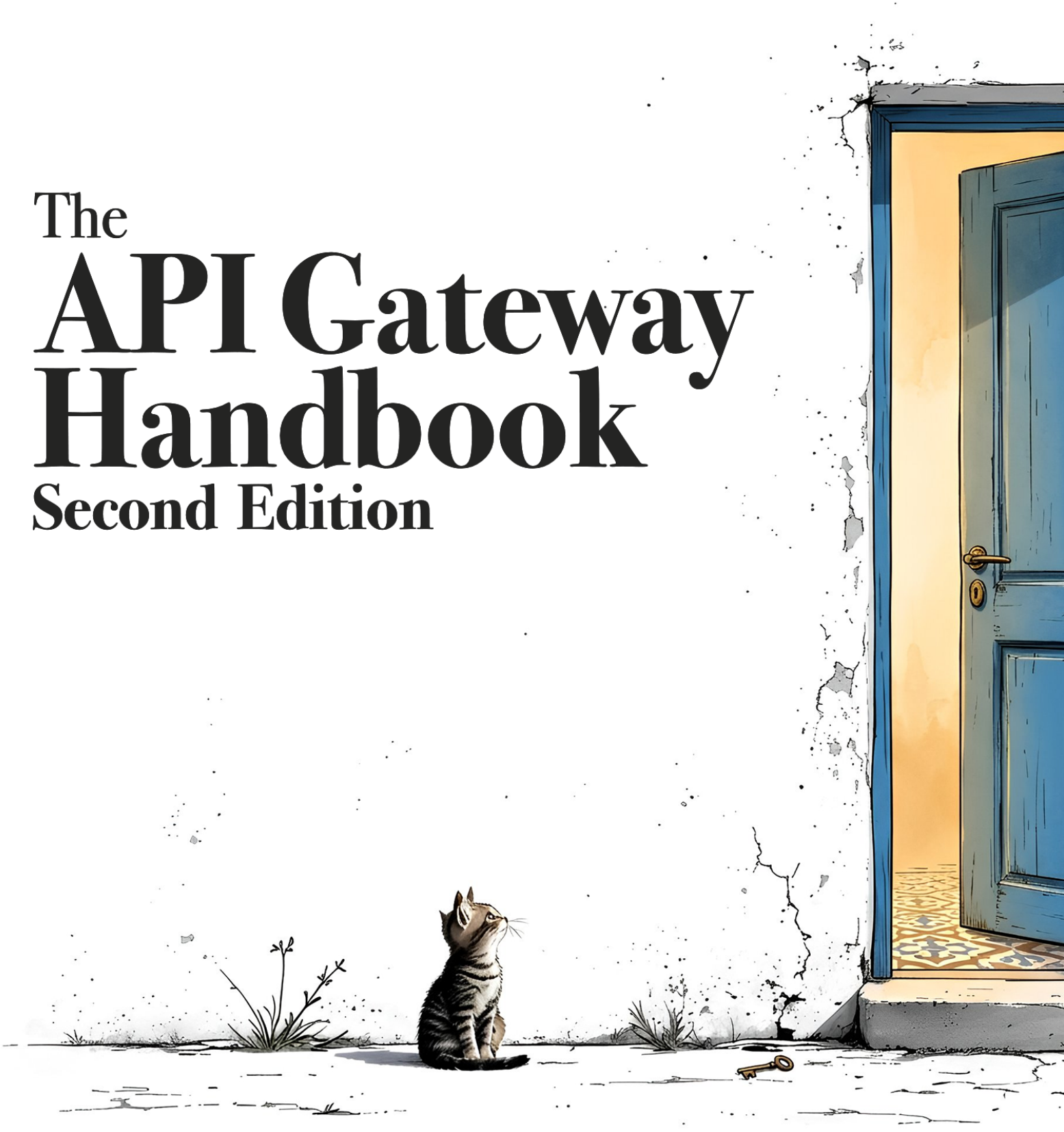
predic8 press

The API Gateway Handbook

Second Edition



Thomas Bayer & Tobias Polley

The API Gateway Handbook

The API Handbook Second Edition

by Thomas Bayer
bayer@predic8.de
https://www.linkedin.com/in/thomasub/

and Tobias Polley
polley@predic8.de
https://www.linkedin.com/in/tobias-polley/

Release 2026/3/16

Published by predic8 GmbH, Koblenzer Strasse 65, 53173 Bonn, Germany

Download the newest version as PDF

The latest edition of the API Gateway Handbook is available here:
https://membrane-api.io/api-gateway-ebook.html

License

Disclaimer

Table of Contents

Part II

0 Preface

APIs enable isolated applications to communicate with each other. It doesn't matter whether the applications live inside the same organization, in the cloud, or on the other side of the planet. APIs have become the universal language of application systems. Even artificial intelligence relies on them as a bridge to move beyond the data center and interact with the real-world.

Interfaces existed long before today's HTTP- and JSON-based APIs. But those earlier approaches were hard to understand and required experienced specialists to implement. Modern APIs changed that completely. They are designed to be simple, so simple that even high school students can use them in their projects. This simplicity has fueled widespread adoption and made APIs the backbone of digital communication.

Gateways connect frontend apps to backends, partners to platforms, and services to each other. Yet as systems grow, the challenges grow with them: Security, observability, and lifecycle management become progressively more difficult. And this is where API Gateways prove their value.

0.1 About This Book

What exactly does an API Gateway do and how can you use it effectively? This book answers those questions and gives you a solid understanding of API Gateways and the problems they solve. It covers architectural patterns, deployment models, key features, and advanced topics like Zero Trust and APIOps. Whether you are securing public APIs, managing internal traffic, or scaling your API ecosystem, gateways play a central role.

This is a practical guide: starting with HTTP basics and proxy fundamentals, then moving into how gateways work, how to deploy and configure them, and how to solve real-world API challenges such as routing, security, integration, and operations.

The book is written for architects, developers, and platform teams working with APIs in any context.

Part I lays the foundation with general patterns, principles, and best practices in a vendor-neutral way.

Part II focuses on concrete solutions to real-world problems. The examples use the Membrane Open Source API Gateway for illustration, but the patterns and techniques apply to other gateways as well.

The goal is for this book to serve both as a guide and as a toolbox for working effectively with API Gateways.

0.2 About the Second Edition

Almost a year after the first edition appeared, this second edition was released. It incorporates feedback from readers, including error corrections, questions, and ideas that led to entirely new sections.

As a result, the content has grown by more than 50 pages and now exceeds 320 pages. A major change compared to the previous version is the update of the second part to the new **YAML based configuration** language used in the examples.

As with the first edition, we welcome feedback, questions, and suggestions that may help shape a future third edition.

0.3 Why You Should Read This Book?

This book is for anyone working with APIs, whether you are part of a platform team, in operations, or focused on API development and architecture. It provides practical guidance, from foundational concepts to advanced configurations and real-world use cases.

You should read this book if:

- You're responsible for securing APIs
- You want to streamline API delivery using OpenAPI and APIOps practices
- You're evaluating or operating an API Gateway
- You're building with microservices, working with the cloud, or integrating hybrid systems

No deep prior knowledge is required. Key concepts are explained step by step, from HTTP fundamentals to JSON Web Tokens and OAuth2.

Part I is particularly useful for API designers, product owners, and project managers who need a clear, high-level understanding of API Gateways and how they fit into modern architectures.

Part II is aimed at operators, developers, and API specialists who want to see how things work in practice, with concrete examples and configuration details.

If your goal is to build secure, maintainable, and scalable API infrastructure, this book is written for you.

0.4 How to Read This Book

If you are new to API Gateways, start with **Part I**. It introduces the core concepts in a structured and easy to follow way and builds a solid foundation.

Part II goes deeper and presents practical examples using the open source Membrane API Gateway to demonstrate how specific problems can be solved. Even if you use a different gateway product, the architectural patterns and techniques discussed here are broadly applicable. Adapting the examples to your own environment should be straightforward.

You can read the book from start to finish, or jump directly to the chapters that are most relevant to your current challenges or interests.

0.5 Why We Wrote This Book

While working on the documentation for Membrane API Gateway, we kept running into the same problem: a reference guide is only helpful if you already know what you're looking for. We found ourselves answering the same kinds of questions, not just “what does this setting do?” but “why would I use it?” and “how does it fit into the bigger picture?”

That’s when we realized something was missing. To use an API Gateway effectively, it’s not enough to understand the individual configuration options. You also need a solid grasp of how gateways work behind the scenes and how they fit into modern architectures.

We wrote this book to fill that gap. It's meant to go beyond the usual documentation and offer practical, hands-on guidance. Whether you’re routing traffic, securing APIs, transforming messages, or exposing legacy systems, you will find patterns and examples to help you along the way.

And yes, we’ll confess. We also wrote this book to give our open source gateway, Membrane, some attention. But we’ve done our best to keep things fair. Part I is vendor-neutral and lays out the general concepts every gateway expert should know. Part II just happens to use Membrane for the hands-on examples. Well, someone had to be the demo gateway anyway. Hopefully, you’ll find value in both parts (and if you end up liking Membrane along the way, we won’t complain).

0.6 How We Wrote This Book

This book is the result of years of hands on experience with API Gateway deployments, the development of our open source API Gateway, and many conversations with the community.

AI served as a patient assistant while writing the book. It helped rephrase, polish, and clean up clumsy English, without complaining about late night edits. The content itself, including the ideas, concepts, and practical experience behind it, is entirely human. Despite the assistance, writing this book still took us more than 1,200 hours alongside our regular daytime work.

0.7 How You Can Help Us

We believe books should be written like software: iteratively, with feedback and continuous improvement. Ebooks make this kind of agile process possible.

We received valuable input and many corrections from readers of the pre-release and the first edition. This second edition has gone through extensive revisions. Still, there are likely errors, omissions, or sections that can be improved.

If you spot a mistake, have suggestions, or want to share general feedback, we would appreciate hearing from you. With your help, the next update will be even better.

Send us an email at:

bayer@predic8.de or **polley@predic8.de**

Thanks for helping us make this book better.

0.8 About Us

Thomas Bayer

I'm Thomas Bayer, CEO of predic8, a software consultancy based in Bonn, the former capital of Germany. My journey into distributed systems began back in the 1990s with FIDO Net, early PC networks, and CORBA. In 1998 I founded my first company, Orientation in Objects, where I embraced Service-Oriented Architectures, built XML-based Web Services, and began exploring the early ideas behind REST.

Since then, I've worked on a wide range of commercial API projects across industries. In 2004, I founded *Osmotic Web* in Boston to promote the still-nascent concept of services, back when APIs weren't yet synonymous with HTTP interfaces. Even then, I believed strongly in the power of open source tools as a foundation for digital transformation. That belief eventually led to the development of Membrane API Gateway.

Since founding predic8 in 2007, I've continued to evolve Membrane, contribute to open source projects, and support clients worldwide in designing, securing, and scaling APIs.

I regularly speak at conferences about software architecture, API design, and security, and I write articles for tech magazines on these topics. On YouTube, I share insights and tutorials on modern API technologies and architectural patterns (the **predic8** YouTube channel, in German).

Outside the world of software, I enjoy learning languages, photography, yoga, and collecting tools.

Tobias Polley

I'm Tobias Polley, co-CEO of predic8 and a software architect with a focus on cloud infrastructure, operations, and API security. Since joining predic8 in 2011, I've helped shape the architecture and security foundations of Membrane, our open source API Gateway. As a consultant, trainer, and international conference speaker, I've supported organizations in securing their APIs and ensuring robust, high-performance deployments.

I studied Mathematics, which continues to influence my analytical approach to software design. Outside of work, I enjoy languages, exploring different cultures, and running. More recently, I've taken up gardening—an unexpectedly rewarding counterbalance to the digital world.

Happy reading, and great success with your API Gateway endeavors!

Part I
API Gateways Fundamentals

This part lays the groundwork for understanding API Gateways. We begin by revisiting the fundamentals, APIs, HTTP, and practical tools such as curl and Postman, to ensure a shared baseline. From there, we take a deeper look at API Gateways: what they are, the problems they address, and how they contribute to security, scalability, and API operations.

Whether you are just getting started or want to sharpen your understanding, Part I provides the essential context needed to make informed architectural and operational decisions.

1 Foundation

Let's begin by establishing a solid foundation. In this chapter, we cover the essential technical concepts you will need throughout the book. If you are already familiar with APIs, HTTP, and common API tools, you can skip ahead to Chapter 2 API Gateways, where we focus on API Gateways.

1.1 Application Programming Interface (API)

When people use an application, they interact through its user interface (UI). But when applications need to communicate with each other, they rely on an Application Programming Interface, or **API**. APIs are designed specifically for machine-to-machine interactions, allowing applications to communicate efficiently at a technical or business level.

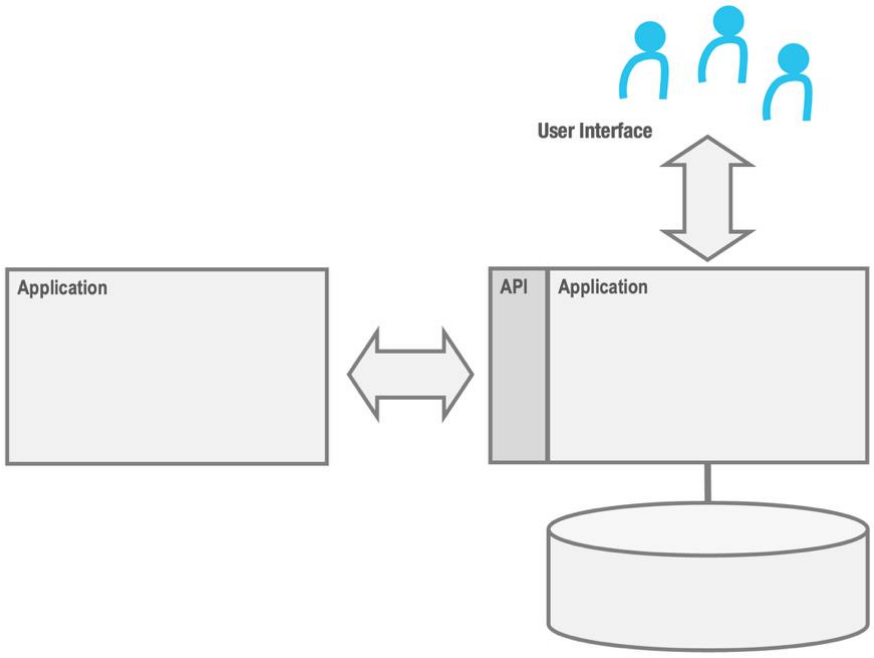


Image: User Interface and API

Today, the most common API style is REST (Representational State Transfer), but alternatives such as GraphQL and other HTTP-based approaches are steadily gaining popularity.

HTTP-based APIs are widely adopted because HTTP simplifies communication between different systems, even across organizational boundaries. HTTP's ability to easily traverse firewalls and network boundaries makes it particularly suited for widespread API implementation. We'll explore HTTP further in the next section.

1.2 Hypertext Transfer Protocol (HTTP)

Most APIs are built on top of the Hypertext Transfer Protocol (HTTP), the backbone of communication on the web. Originally introduced more than 30 years ago, HTTP is the protocol web browsers use to fetch web pages, making it fundamental to how we interact with the Web.

HTTP is known for its simplicity, which contributed to its widespread adoption. Understanding the basics of HTTP is essential for grasping how API Gateways function. In this section, we will explore the core concepts of HTTP to prepare you for the chapters ahead.

HTTP follows the **Client-Server** paradigm, where a client sends a request to a server, and the server responds with the requested resource. For example, suppose a web browser wants to access the URL `https://api.predic8.de`. Here's how this interaction works step by step:

1. **Domain Name Resolution**
   The browser queries the Domain Name System (DNS) to resolve `api.predic8.de` to an IP address.
2. **Connection Establishment**
   Once the IP address is known, the browser opens a connection to the web server.
3. **Sending the Request**
   Then, the browser sends an HTTP request asking for a specific resource.
4. **Receiving the Response**
   The server processes the request and returns an HTTP response.

Exploring HTTP Communication with curl

Instead of using a graphical browser like Firefox, we can use a command-line HTTP client such as `curl` to make a request and observe how HTTP communication works. For example:

```
curl -v https://api.predic8.de/shop/v2/products/7
```

This command initiates an HTTP request to the server. The option `-v` causes curl to show you exactly what is going over the wire. In the output created by `curl`, you will find a request that might look like this:

```
GET /shop/v2/products/7 HTTP/1.1
Host: api.predic8.de
```

Let's break this down:

1. **The Request Line**:
   The first line is the request line:
   - `GET` specifies the HTTP method, which in this case asks for a resource.
   - `/shop/v2/products/7` is the path to the resource on the server.
   - `HTTP/1.1` indicates the HTTP protocol version being used.

    -
2. **Host Header**:
   The `Host` header identifies the server the request is directed to (`api.predic8.de`). This is necessary because multiple domains can share the same IP address, and the server needs to know which site the client wants to access.

After receiving the client's request, the server processes it and sends back a response. For the example above, the server might respond with:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "id": 7,
  "name": "Gac-Fruit",
  "price": 69.99
}
```

This can be broken into the following parts:

1. **Status Line**:
   The first line of the response is the status line:
   - `HTTP/1.1` indicates the HTTP protocol version used for the response.
   - `200 OK` is the **status code** and **reason phrase**. The `200` status code tells the client that the request has been successful.

2. **Headers**:
   HTTP headers provide additional information about the response. In this case:
   - `Content-Type`: Indicates the format of the response body.

3. **Response Body**:
   Following the headers, the server sends the response body, which contains the actual content. In this case, the body includes a JSON document with data about the requested product.

HTTP/2 and HTTP/3

HTTP/2 and HTTP/3 were introduced as successors to HTTP/1.1, aiming to improve performance, especially for loading web pages in browsers. They bring features like multiplexing, header compression, and server push to reduce latency and speed up page loads.

However, when it comes to **machine-to-machine communication**, such as APIs, the benefits are limited.

Despite these improvements, both HTTP/2 and HTTP/3 preserve the core semantics of HTTP: methods like `GET`, `POST`, and status codes like `200` still work the same way. This means your existing HTTP-based APIs don't need to be redesigned to work over newer versions.

Many gateways today support HTTP/2 and even **gRPC**, which takes advantage of some of HTTP/2's features. For general API design and compatibility, at the time of writing, **HTTP/1.1 remains the most widely supported protocol**.

1.3 HTTP Clients

When working with API Gateways, thorough testing is essential. Although a web browser can serve as a basic HTTP client, specialized tools offer enhanced control and deeper insights for API testing and exploration. For most of the examples in this book, the REST Client plugin for **Visual Studio Code** is used. To follow the samples, you can choose from command-line tools like **curl**, graphical interfaces like **Postman**, or editor plugins. The choice depends on your workflow and personal preference.

curl

`curl` is a powerful and versatile **command-line** tool widely used for sending HTTP requests. Its simplicity combined with scripting capabilities makes it perfect for quick testing, automation, and integration in CI/CD pipelines.

Here's a basic example demonstrating how `curl` makes a GET request:

```
curl -v https://api.predic8.de/shop/v2/
```

This produces the output:

```
> GET /shop/v2/ HTTP/1.1
> Host: api.predic8.de
>
< HTTP/1.1 200 OK
< Content-Type: application/json
< Content-Length: 363
<
{
  "links" : {
    "products_link" : "/shop/v2/products",
    "vendors_link" : "/shop/v2/vendors",
    "orders_link" : "/shop/v2/orders",
    "customer_link" : "/shop/v2/customers"
  }
}
```

It shows both the raw HTTP exchange and the JSON response body.

Postman

Postman is a user-friendly graphical tool for exploring and testing APIs. While it started as a simple HTTP client, it has grown into a full-featured API platform with powerful collaboration and automation features.

With Postman, you can group requests into collections, define environments for testing and production, and use variables to manage dynamic data. Its built-in scripting capabilities allow you to write pre-request scripts and tests, automate workflows, and validate responses.

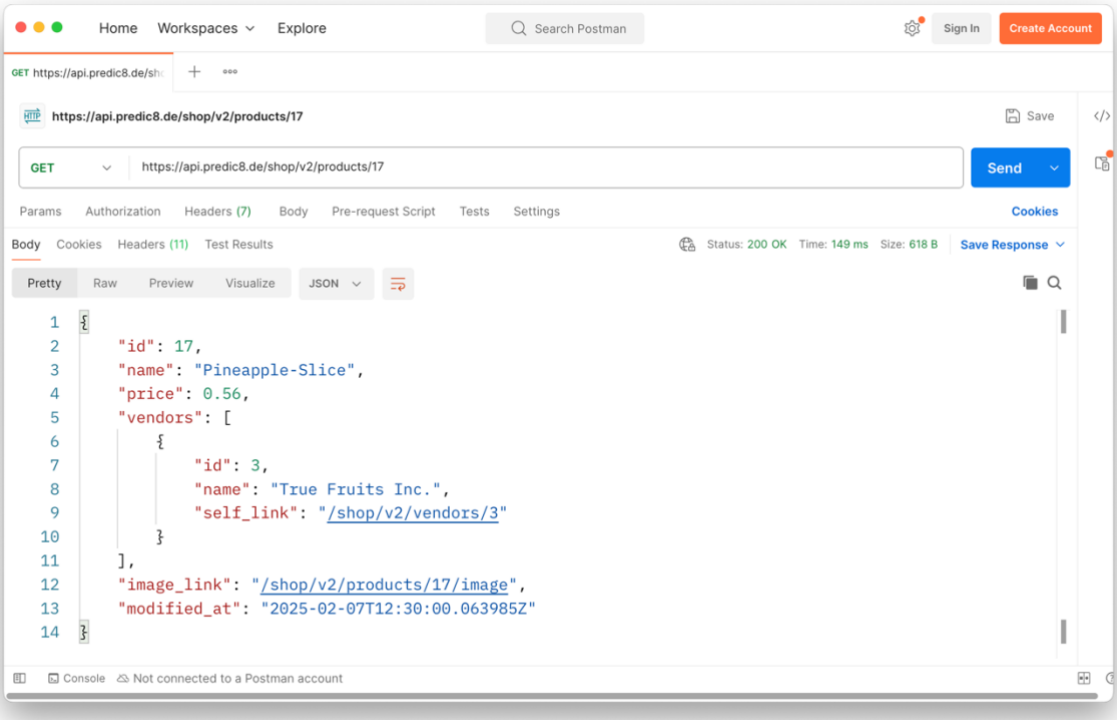


Image: HTTP client in Postman

There are also open source alternatives to Postman. Bruno, for example, offers fewer features than Postman, but it stores collections in local folders and does not require a cloud account. For teams that prefer working with files in git, this can be a crucial advantage.

Resources

command line tool and library for transferring data with URLs (since 1998)
https://curl.se/

Bruno - Opensource IDE for exploring and testing APIs.
https://github.com/usebruno/bruno

Hoppscotch
https://hoppscotch.io/

Postman API Platform
https://www.postman.com

HTTP Client Plugins for Editors and IDEs

HTTP client plugins for editors and development environments like **Visual Studio Code** and **IntelliJ** make it easy to test, debug, and script API calls directly within your IDE. They offer a clean and efficient interface where you can view both the request and the response side by side without hidden headers or metadata in separate tabs.

These plugins also let you:

- Write and organize multiple requests in a single file
- Save and reuse request files across projects
- Share requests with your team using version control (e.g., via **Git**)

They're a great fit for developers who want to stay close to their code while working with APIs.

In this book, you'll find examples like the one below:

```
POST https://api.predic8.de/shop/v2/products/
Content-Type: application/json

{
  "name": "Pineapple",
  "price": 2.79
}
```

This may look like a captured HTTP exchange, but it's a detailed description of an executable request. Unlike a typical HTTP message, which only includes the path (e.g., `/products`) after the request line, this description lets you specify a full URL. That means you can include the protocol (`http` or `https`), hostname, and port, providing the plugin with all the information it needs to send the request to the server.

How to Use This Example

1. **Copy the Example**
   Copy the HTTP request shown above and paste it into your editor.
2. **Set the Language Mode**
   Change the language mode to `HTTP`, or save the file with a `.http` extension so your editor recognizes it.
3. **Send the Request**
   Click the **Send Request** button (usually visible above the request). The response will appear in a panel on the right side of your editor.

These plugins support autocompletion, which makes writing HTTP requests quick and comfortable.

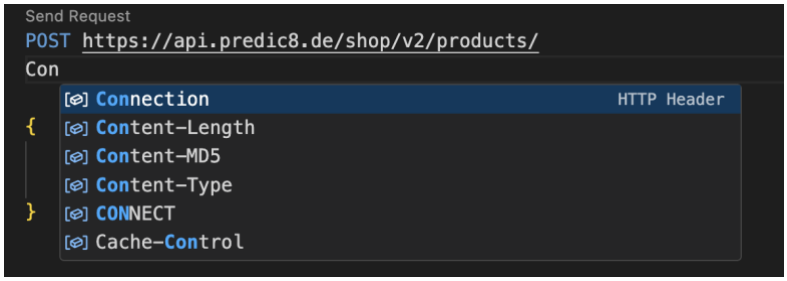


Image: HTTP Autocompletion with the REST Plugin

The screenshot below shows Visual Studio Code after sending a request.

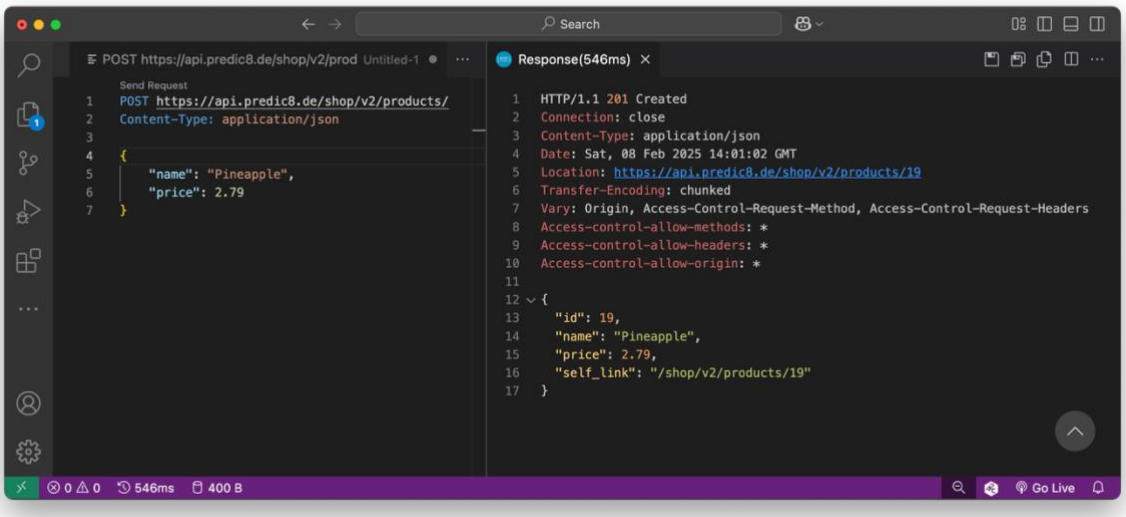


Image: Request and Response in the REST Client Plugin

Installing the REST Client Plugin in Visual Studio Code

It only takes a minute to set up.

1. **Open the Extensions View**
   Click on the **Extensions** icon on the left sidebar in Visual Studio Code (or press `Ctrl+Shift+X` or `Command+Shift+X` on macOS).

2. **Search for "REST Client"**
   Type **REST Client** into the search bar.
3. **Install the Plugin**
   Find the plugin by *Huachao Mao* and click the **Install** button.

Once installed, you're ready to start sending HTTP requests directly from your editor, no terminal or external tools required.

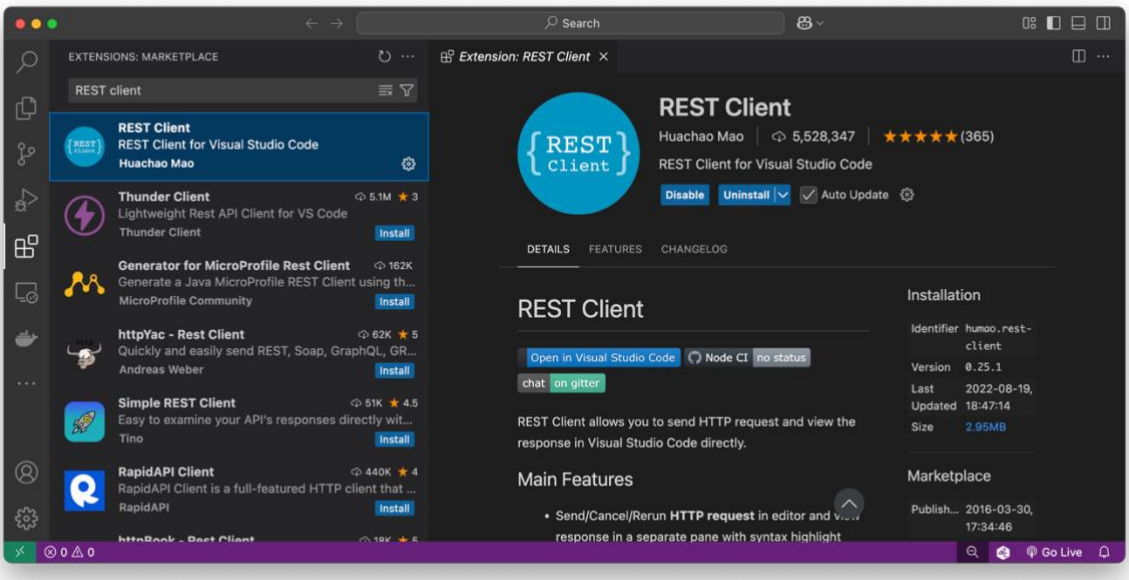


Image: REST Client Plugin in Visual Studio Code

Similar extensions are available for **IntelliJ** and other development environments.

Resources

REST Client, Microsoft Marketplace
https://marketplace.visualstudio.com/items?itemName=humao.rest-client

JetClient - The Ultimate REST Client, IntelliJ
Marketplacehttps://plugins.jetbrains.com/plugin/21173-jetclient--the-ultimate-rest-client

1.4 Reverse Proxies

Now that we're equipped with the right tools, let's turn to a key network component: the reverse proxy. To understand what makes an API Gateway special, and why it plays an important role, it helps to understand the difference between a traditional proxy and a reverse proxy.

Proxies (Forward Proxies)

A traditional proxy, also known as a forward proxy, sits on the client side of a connection, between the client and the public internet.

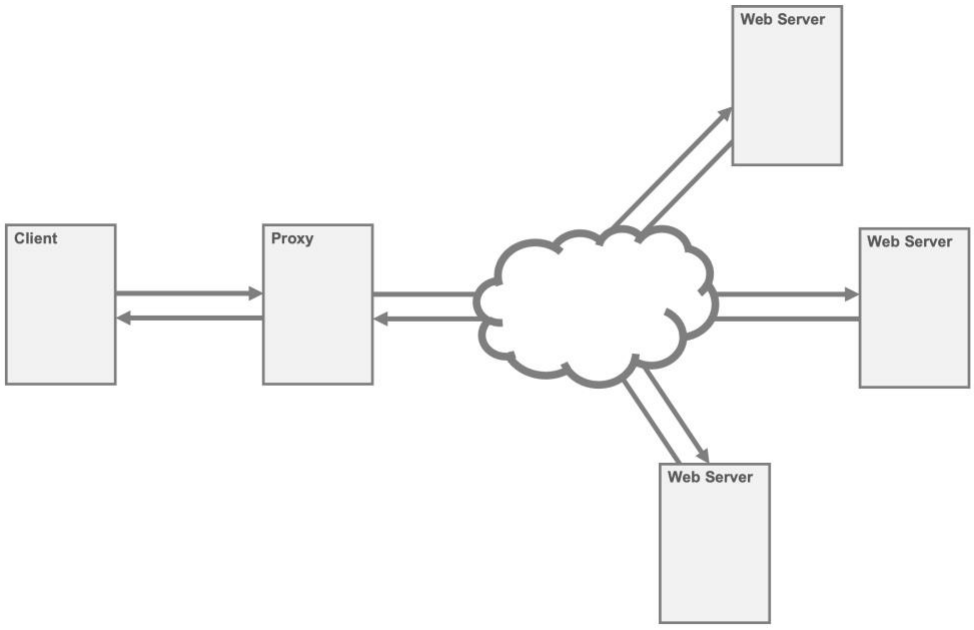


Image: Forward proxy between client and internet

Its primary purposes include:

1. **Performance**: It speeds up communication by caching frequently requested resources.
2. **Access Control**: Filters or restricts access to websites or content.

Reverse Proxies

As the name suggests, a reverse proxy sits on the other side of the connection, directly in front of one or more backend servers. From the client's perspective, it appears to communicate directly with a target server. In reality, the reverse proxy receives the request, applies routing or filtering logic, and forwards it to the appropriate backend server.

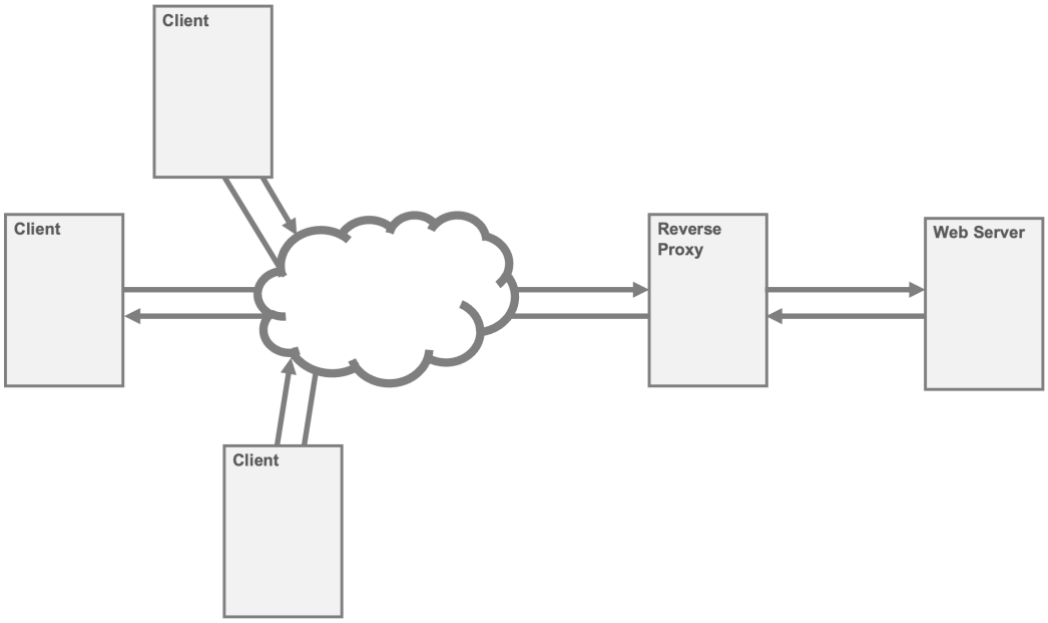


Image: Reverse proxy between internet and server

A reverse proxy:

1. Accepts client requests
2. Forwards them to the appropriate server
3. Returns the backend response to the client

Beyond simple forwarding, a reverse proxy can provide **additional value**:

- **Load Balancing**
  Distributes incoming traffic across multiple servers to improve performance and reliability.
- **Security**
  Hides internal server details and filters potentially malicious requests.
- **TLS Termination**
  Handles encryption and decryption, offloading this work from backend systems.
- **Logging**
  Records information about requests and responses.
- **Monitoring**
  Tracks system health, latency, and error rates.

In the following sections, we will explore how API Gateways build on reverse proxies to address the unique challenges of APIs.

2 API Gateways

API Gateways are essentially reverse proxies but with a twist. They're specialized in handling API traffic and come equipped with API-centric functions. While a traditional reverse proxy might only care about forwarding HTTP requests, an API Gateway understands the nuances of API communication.

It speaks fluent JSON, knows how to decode JWT tokens, manages API keys, and can even handle GraphQL queries. But more importantly, it tackles API-specific challenges like security enforcement, message transformation, and traffic control, all in one place.

Think of it as a smart doorman for your APIs: not only does it open the door, but it also checks IDs, limits the crowd, and makes sure no one is sneaking in anything suspicious.

2.1 Responsibilities of API Gateways

An API Gateway acts as a central control point for API communication between clients and backend services. It provides a range of capabilities that simplify client interaction while protecting backend systems. Key responsibilities include:

Routing
API Gateways forward incoming requests to the appropriate backend services. Clients communicate only with the gateway and do not need to know internal network structures or backend addresses. This abstraction simplifies clients and allows backend systems to evolve without breaking integrations.

Security
Gateways help to enforce security policies by handling authentication, authorization, and message validation before requests hit backend systems.

Logging, Monitoring and Tracing
They collect operational data about API usage by monitoring key performance indicators, recording logs, and tracking request paths.

Message Transformation
API Gateways can transform messages between different formats, such as converting XML payloads into JSON or adapting data to meet client requirements. This helps bridge differences between client and backend representations.

Orchestration
In complex scenarios, the gateway can coordinate multiple backend calls and combine their responses into a single result. For example, it may aggregate data from several microservices to provide a unified response to the client.

Load Balancing
By distributing incoming traffic across multiple backend servers, API Gateways can ensure that no backend becomes overwhelmed. This not only improves overall performance but also enhances system availability and reliability.

Inventory Management
Gateways also aid in inventory management, providing visibility into all exposed APIs, including tracking usage patterns and identifying outdated or deprecated services.

These capabilities make API Gateways a critical piece of modern IT infrastructure, essential for maintaining scalable, secure, and well-managed APIs.

2.2 Kinds of API Gateways

With over sixty API Gateway products listed on the API Landscape web page, choosing the right one for your needs can be a daunting task. Many of these gateways are **tailored for specific scenarios**. For example, there are gateways that focus on edge computing, enterprise API management, or AI integrations.

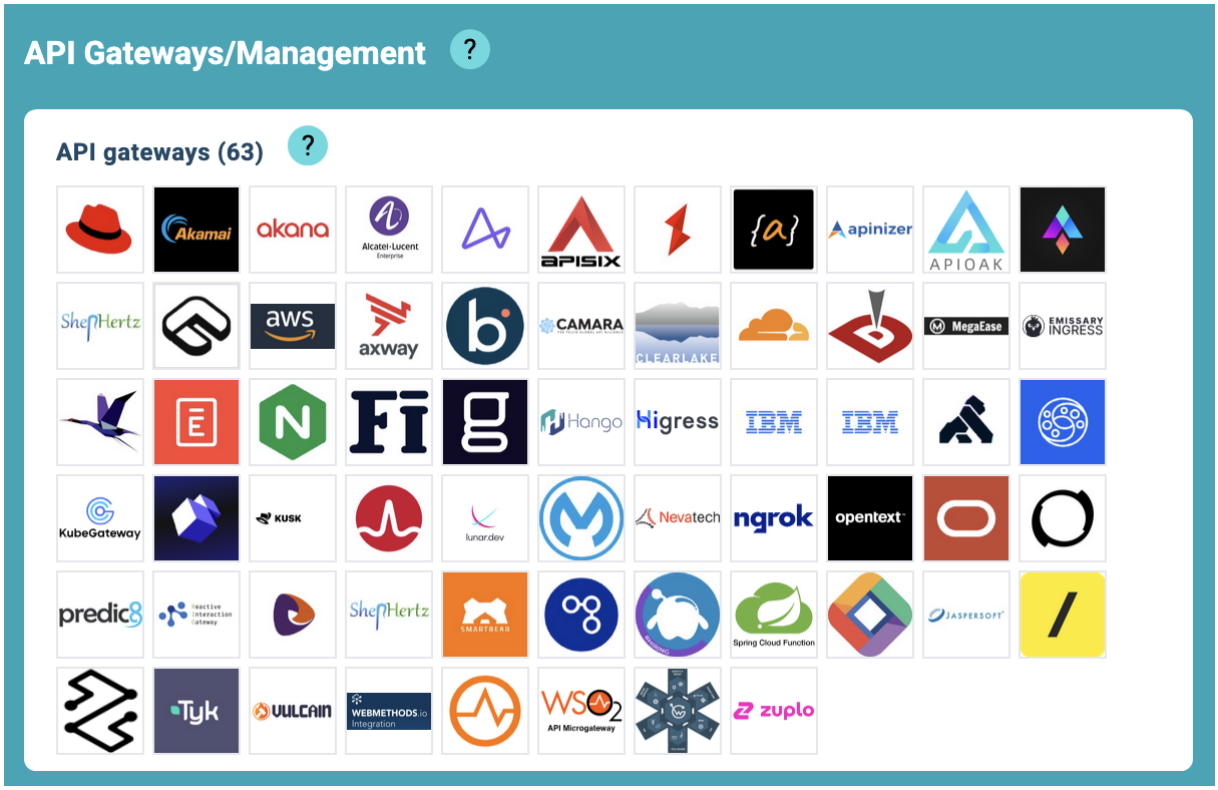


Image: API Gateway products @ API Landscape

The following subsections explore these categories in detail, helping you understand the strengths and use cases for each kind of API Gateway.

Edge Gateways

The **internet protocol** allows a gateway to be reachable from almost anywhere in the world. But global reach doesn't guarantee consistent performance. Latency and bandwidth can vary significantly depending on the user's geographic location. For many business applications, that's acceptable. But some applications require consistently low latency, such as gaming, robotics, or autonomous vehicles. In these cases, even small delays are not acceptable.

Edge computing addresses this challenge by placing services physically closer to where they're needed. This proximity reduces the round-trip time and improves responsiveness. When API Gateways are deployed in a distributed manner, there's always an instance available within the user's region. As a result, requests are handled with lower latency and greater reliability.

Cloud Gateways

Major cloud providers like Amazon AWS and Microsoft Azure offer their own API Gateways that integrate seamlessly into their cloud platforms. These cloud-native gateways benefit from built-in scalability and deep integration with cloud services.

In addition to conventional backends, serverless functions can also be used as backend targets.

However, you're not limited to using a cloud provider's built-in gateway. Almost any other API Gateway can also be deployed and configured to act as the entrance into a cloud environment. This flexibility allows you to choose a gateway based on specific requirements, such as cost, features, or portability across multiple cloud providers.

Gateway Libraries

Gateway libraries let you embed API Gateway functionality directly into your application instead of running a separate gateway process. This approach gives you fine-grained control over routing, filtering, and security behavior, tightly integrated with your application code.

On the Java platform, a well-known example is Spring Cloud Gateway. It provides routing, security, and filtering capabilities within Spring-based applications and integrates naturally with the Spring ecosystem.

Kubernetes API Gateways

Kubernetes is an open source container orchestration platform that enables the management of containerized workloads and services, both on-premises and in the cloud. Within Kubernetes clusters, API Gateways can play a critical role: they manage traffic flowing into the cluster, orchestrate communication between services, and provide security and observability features.

If you are not working with Kubernetes, feel free to skip this section. However, if you use Kubernetes, this section offers insights on how API Gateways can integrate with, and enhance a Kubernetes environment.

Kubernetes Ingress Controllers

Applications running inside a Kubernetes cluster must be accessible from the outside world. Ingress controllers are Kubernetes native components that act as entry points that route external traffic into the cluster. They offer several key features to accomplish this task:

- **Service Discovery:**
  An Ingress controller leverages Kubernetes service discovery to determine which pods can serve as backends for a given API. This dynamic discovery ensures that traffic is routed to the correct and available instances.
- **Traffic Control:**
  To maintain high availability and reliability, requests must be routed only to healthy endpoints. In the event of errors, retries are essential. Product-specific extensions or service meshes often add common patterns such as circuit breakers and rate limiters, managing traffic surges and preventing cascading failures.
- **Observability:**
  All traffic going inside a cluster can be logged and monitored.
- **Protocol Flexibility:**
  Besides HTTP, many Kubernetes API Gateways also support TCP and gRPC. This capability allows nearly any protocol to be proxied, providing flexibility in handling diverse workloads.
- **Tight Kubernetes Integration:**
  The configuration of these special gateways is deeply integrated with and native to Kubernetes.

Several Kubernetes API Gateways, such as Ambassador or EnRoute, are built on top of Envoy Proxy. Envoy offers high performance, extensive observability, and robust traffic management features that are ideal for modern cloud-native environments.

This comprehensive set of features makes Kubernetes Ingress Controllers an essential component for managing external traffic and ensuring that APIs remain scalable, resilient, and secure within the dynamic environment of a Kubernetes cluster.

💡**Sidenote: Kubernetes Gateway API**
Gateway API is a Kubernetes subproject managed by the **Kubernetes Special Interest Group** (SIG Network). It aims to standardize how services are exposed, and traffic is routed within Kubernetes clusters.

The Gateway API provides a set of Kubernetes resource types (like `Gateway`, `HTTPRoute`, and `TCPRoute`) beyond the Ingress API to standardize path rewriting, traffic management, and routing in Kubernetes.

The phrase **Kubernetes API Gateway** can imply something similar to a full-featured API Gateway like Kong, AWS API Gateway, or Tyk. The Kubernetes Gateway API is **not intended** to replace common API Gateways or Kubernetes Ingress controllers. Instead, it's meant as a standard **interface** to define networking and routing behavior, leaving actual implementation to specialized controllers.

Gateway API implementations still rely on concrete networking solutions or ingress controllers (like Istio with Envoy, LinkerD, Contour, Ambassador, or Traefik), which extend and provide functionality behind these standardized interfaces.

Sidecars in Service Meshes

Gateways can hide the complexity of networks and the underlying infrastructure from applications. All traffic to and from an application passes through such a gateway, enabling enhanced security, observability, and traffic management.

In contrast to an ingress gateway positioned at the edge of a Kubernetes cluster, a **sidecar proxy** operates directly alongside each individual application or infrastructure service within the cluster itself.

Because sidecars run alongside every application, it’s critical that they have a minimal resource footprint, often consuming less than 100 MB of RAM. These lightweight proxies manage traffic not only to business applications but also to infrastructure components like databases. As a result, they commonly support not just HTTP but also binary protocols like gRPC or generic TCP connections.

Typical gateways in this category include Envoy and several products built on top of it, such as Istio, Consul, and Ambassador. A key reason for Envoy’s popularity for this use case is its small memory footprint, typically around 10 MB, combined with high performance and efficient resource usage.

Artificial Intelligence Gateways

Interacting with large language models (LLMs) and other AI services can become costly very quickly. **AI Gateways** help manage these costs by monitoring usage, enforcing quotas, and applying rate limits. Some even offer **fallback capabilities**, automatically switching to alternative (and potentially more affordable) models when necessary.

While most API Gateways can, in principle, handle AI-related traffic, **specialized AI Gateways**, like Lunar.dev, are purpose-built for working with AI APIs. These tools come with features designed specifically for LLM workloads, including:

- Detailed usage analytics
- Dynamic routing between models or providers
- Fine-grained access control for different users or teams

These gateways are a great choice for teams building AI-powered applications that need cost control, flexibility, and visibility into usage patterns.

Open-Source API Gateways

When choosing an API Gateway, you've got options. One key decision is whether to go with a commercial product or an open source one. Fortunately, many gateways blend both worlds: they're open source but commercially backed. This means you get the flexibility and transparency of open source, plus the support that often comes with a company behind the scenes.

Examples include:

- **KrakenD**
  A high-performance gateway focused on aggregating and transforming data.
- **Kong**
  One of the most well-known gateways with an active community and a broad plugin ecosystem.
- **Tyk**
  Lightweight and developer-friendly, with great support for hybrid and cloud-native setups.

Then there's **APISIX**, a standout in the pure open source camp, developed under the Apache Foundation. It's built with performance and extensibility in mind and has quickly gained popularity among cloud-native developers.

You'll also come across **Membrane**, our open source API Gateway. To keep the first part of the book relevant to a broader audience, we've kept references to it minimal. In Part II, however, you'll find detailed information and practical examples based on Membrane.

Resources

API Landscape
https://apilandscape.apiscene.io/

Kubernetes Gateway API
https://github.com/kubernetes-sigs/gateway-api

Gateway API FAQ
https://gateway-api.sigs.k8s.io/faq/

3 How API Gateways Work

An API Gateway is essentially **a reverse proxy** with additional capabilities for APIs. You can use a reverse proxy such as Apache mod_proxy to forward API traffic, but it does not provide API-aware features. For example, a typical reverse proxy does not understand API keys or OpenAPI definitions. In other words, it forwards HTTP messages, but it does not understand the API contract or enforce API-specific policies. That is where an API Gateway goes further.

To do its job, the API Gateway sits between API clients and backend services, just like a reverse proxy. The diagram below shows its place in the architecture. The gateway acts as a single entry point and exposes backend services under a unified domain such as `api.predic8.de`.

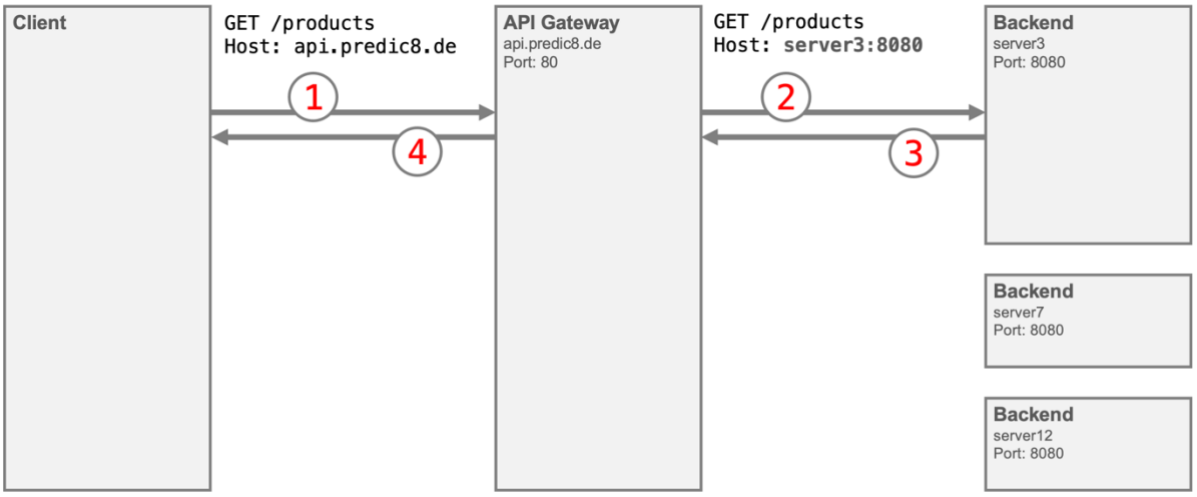


Image: Location of an API Gateway in the message flow

Here is the basic flow:

1. A client sends an HTTP request to the gateway, such as a `GET /products` request.
2. The gateway inspects the request path and forwards it to the appropriate backend (e.g., `server3`).
3. The backend handles the request and sends a response.
4. The gateway then relays that response back to the client.

From the client's perspective, it looks like a direct call to the API. But behind the scenes, the gateway routes traffic, enforces security policies, or logs message data.

At this level, the behavior of an API Gateway is similar to a reverse proxy. The difference lies in its focus. Reverse proxies are built for generic HTTP traffic handling. API Gateways extend this model with API-specific functionality.

In fact, some API Gateways like IBM APIcast, Kong, or APISIX are reverse proxies equipped with plugins and extensions for APIs.

3.1 Plugins and Policies

Plugins and policies are what elevate an API Gateway beyond a simple reverse proxy. They provide the functionality needed to transform, secure, and observe API traffic. The exact terminology varies by product: some call them *plugins*, others *policies, interceptors* or *filters*, but the idea is the same.

Gateways usually come preloaded with a wide selection of plugins. These extensions are often grouped into categories such as:

- **Transformation**: modify headers, or change payload formats
- **Authentication**: validate API key, JWT, or passwords
- **Security**: protect against injection threats or content bombs
- **Observability:** provide logging, tracing, and monitoring
- **Traffic Control:** manage rate limits, quotas, and retries

Some gateways even maintain marketplaces, allowing third-party vendors to publish their own extensions for reuse or commercial distribution.

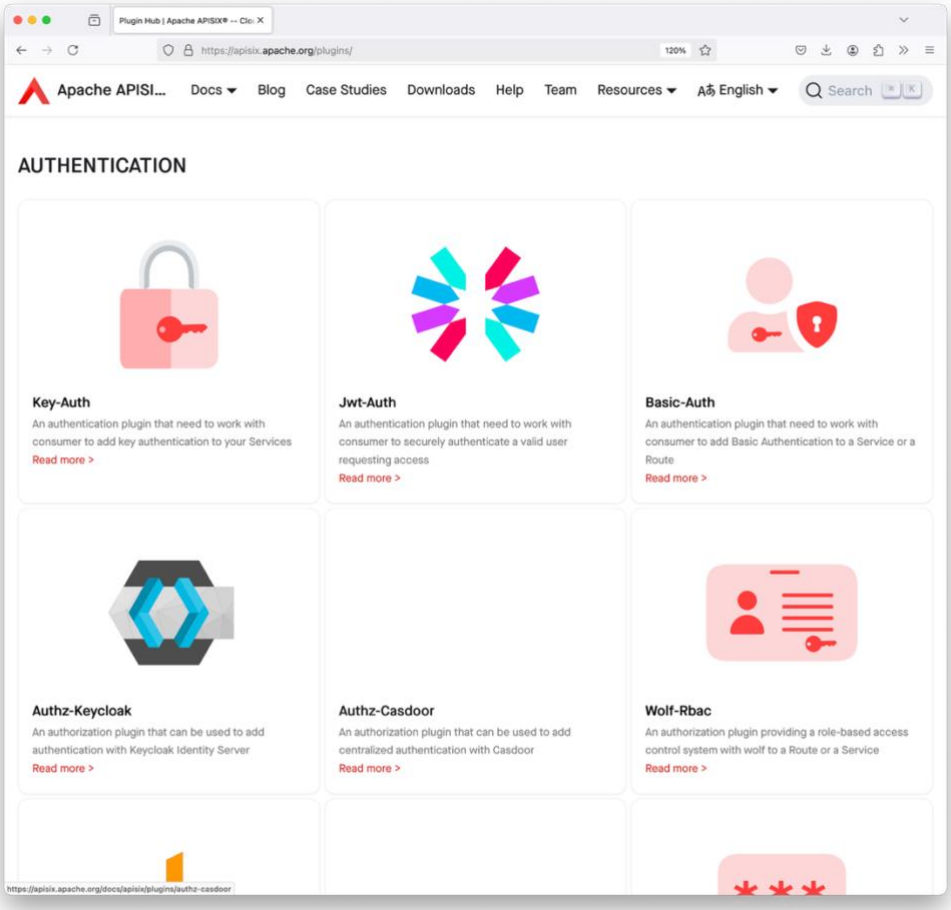


Image: Plugins at APISIX plugin hub

3.2 Message Flow

A call typically passes through the API Gateway **twice** as shown in the image below.

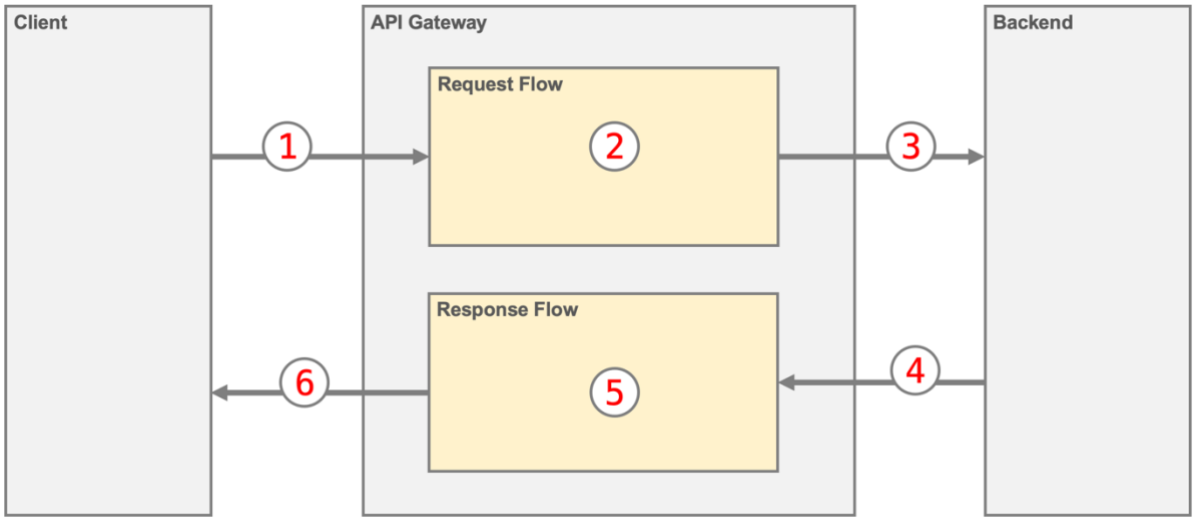


Image: Message flow between client, API Gateway and backend

The sequence is as follows:

1. **Request Flow**

    1. The client sends a request to the Gateway.
    2. The gateway processes the request (e.g., checking authentication)
    3. Then the gateway uses **a second HTTP connection** to the backend and forwards the request.

2. **Response Flow**

    4. After processing, the backend sends a response **in the opposite direction**.
    5. The message passes the gateways response flow in the opposite direction. Further processing can be applied to the response (e.g., transforming the payload or injecting headers).
    6. Finally, the gateway returns the response to the client over the **original connection**.

Plugins, when engaged in the request or response flow, can block requests with invalid credentials or log payloads.

Plugin Placement

API Gateways provide functionality through **plugins**. For a plugin to be effective, it must be plugged into the correct stage of the request or response flow.

Some plugins should be invoked on every request, regardless of which API is being called. Examples include security policies or logging. To accommodate that requirement, most API Gateways offer a **global flow** through which **all messages** pass. By placing a plugin in the global flow, it is applied consistently across all APIs.

In the illustration below, there is a JSON Web Token validator and a Logging plugin engaged in the global flow.

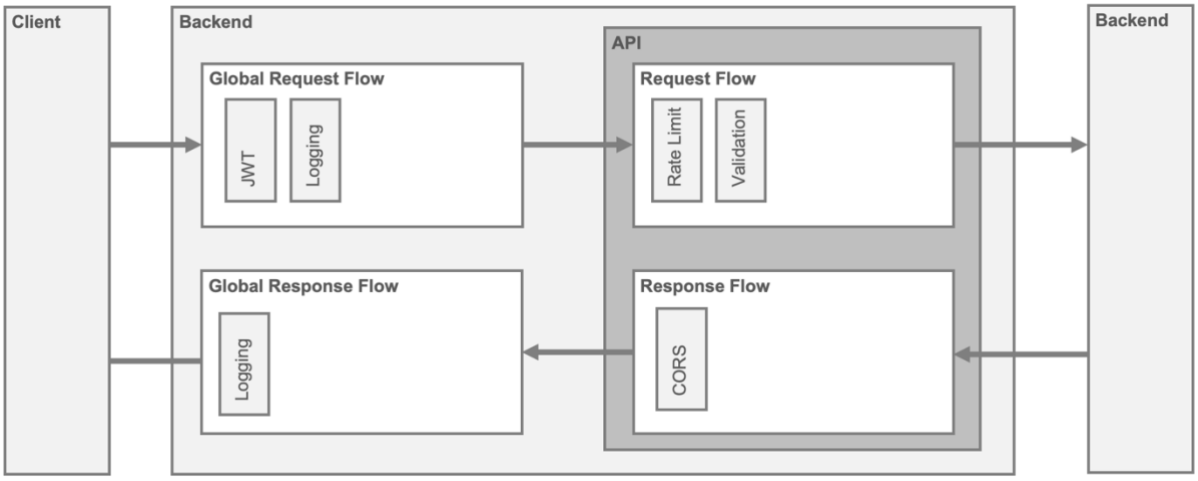


Image: API Gateway with global and API local flows

Other plugins are only relevant for a specific API or endpoint. For example, you might enable schema validation only for one API that must strictly follow its contract. In this case, the plugin is placed in the API local flow.

Gateways usually allow each API to define its own request and response flows. By placing a plugin locally, you restrict its scope to a specific route or backend service.

This flexibility, **choosing between global and local scope** as well as request and response flow, gives you precise control over where a plugin performs its work. Careful planning of plugin placement ensures that each request and response is processed exactly as intended.

Some plugins must participate in both the request and the response flow. In the illustration below, the OpenAPI plugin appears in both flows. This is necessary because it validates not only incoming requests but also outgoing responses against the API contract.

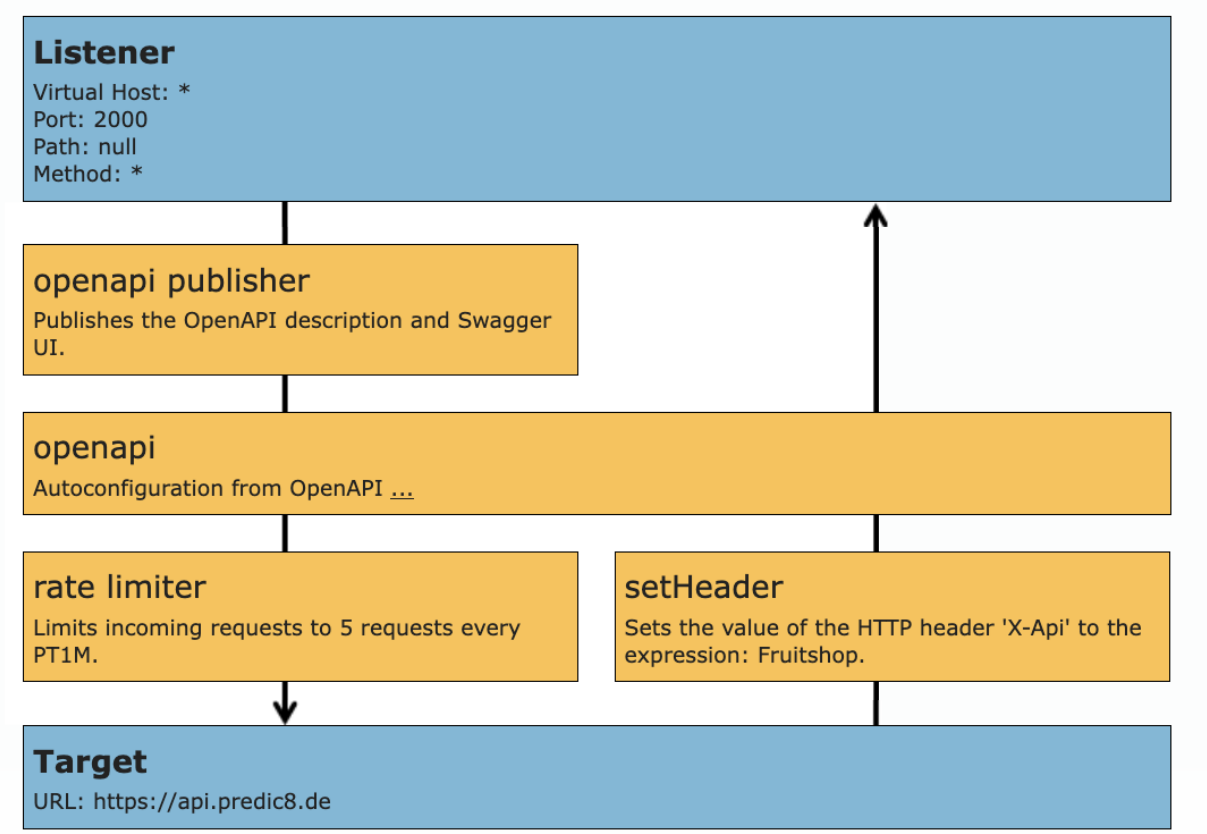


Image: Visualization of an API in Membrane's admin console

Example Throttle for Backend Connections

Let's look at a practical example. Suppose a backend service can handle only ten requests at a time. To protect it from overload, you need to limit the number of concurrent API calls.

A plugin that runs either in the request or in the response flow cannot solve this problem alone. To manage a counter for backend threads, the plugin must have hooks in both flows.

A simple implementation could work like this:

- A counter is **incremented** when a request enters the gateway.
- It's **decremented** when the corresponding response is returned to the client.

If the counter reaches ten, new requests are **blocked** until the number of active backend connections drops below the threshold. To do this reliably, the plugin must track each call across the full request and response cycle. Only then can the counter accurately represent the number of in flight requests. This type of flow-aware logic requires **stateful coordination** and goes beyond simple request filtering.

3.3 Native Plugins and Plugin Runners

Plugins can run directly inside the gateway's runtime environment, sharing its memory and CPU resources. This tight integration allows for fast communication between the gateway and its plugins, enabling high-performance processing. However, for a plugin to run natively, it usually has to be implemented for the same runtime as the gateway itself.

The gateway's underlying technology stack determines which languages are supported:

- **Java-based gateways** support plugins written in Java, Kotlin, or Groovy
- **JavaScript-based gateways** accept plugins developed in JavaScript
- **C-based gateways** allow native C plugins

To make plugin development easier and allow dynamic reloading without restarting the gateway, some products embed a lightweight scripting runtime. A common choice is **OpenResty®**, a Lua-based platform that combines **nginx** with LuaJIT. Gateways like **APISIX**, **Kong**, and **3scale** (now part of IBM) use OpenResty to let developers write and deploy Lua plugins directly into the gateway without recompilation or redeployment.

Plugin Runners

To use plugins written in a language different from that of the gateway core, some API Gateways support a feature called **plugin runner**. This architecture allows, for example, a plugin written in **Python** to integrate with a gateway implemented in **Go**. The gateway communicates with the external plugin over the network, typically using efficient protocols such as **gRPC** instead of HTTP to minimize latency.

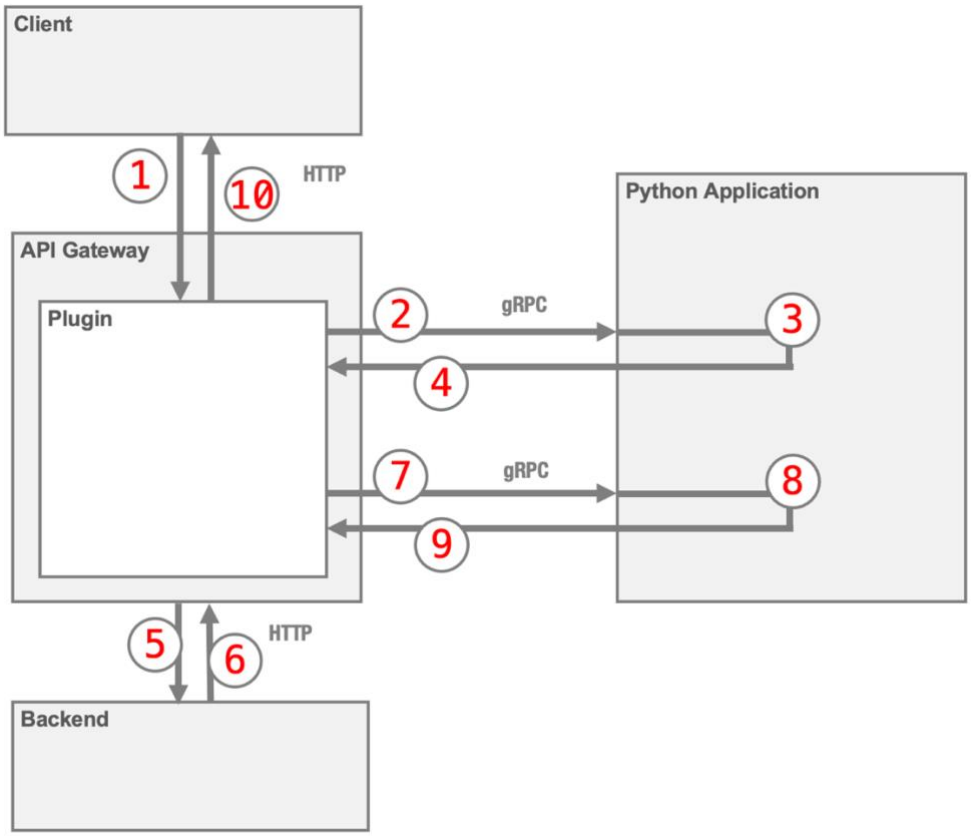


Image: Extending an API with python code via a plugin runner

However, relying on external plugin runners introduces certain trade-offs. Network communication adds potential points of failure, such as latency spikes or connectivity issues, which could lead to delays or even downtime. Additionally, since the plugin runner operates outside the core gateway process, every API call that engages the plugin must be routed across a network boundary.

If your plugin is involved in both **request** and **response** processing (see steps 3 and 8 in the image), it will be **invoked twice** for every single API call, once on the way in and again on the way out. This can amplify latency and increase the complexity of failure handling.

Combining Plugins

Multiple plugins can work together to accomplish a task. **Template, setHeader,** and **extractor plugins** are true team players and are often combined with other plugins.

For example, a request flow consisting of three plugins might extract a year value from an XML body, store it in a variable, then use that variable in a template, and finally prettify the resulting JSON.

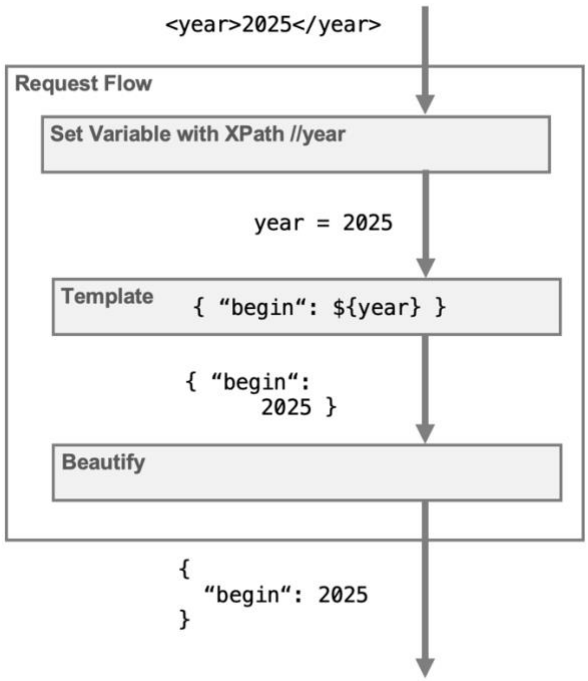


Image: Multiple plugins working together

The glue that binds plugins together is the expression language. It provides the magic that makes collaboration between plugins possible. In the next section, we will take a closer look at how expression languages work and why they matter.

3.4 Expression Languages

Writing a custom plugin for an API Gateway in **Lua**, **Java,** or **Go** isn't rocket science but it requires some ramp-up. You'll need to learn the language, set up a development environment, and probably spend more time than you'd like just getting started.

Luckily, there's a shortcut: **expression languages**.

Most API Gateways include one or more small, embeddable expression languages that let you tweak behavior with just a few characters of code. These languages are more lightweight and focused than general-purpose programming languages.

Think of it like this: instead of writing 50 lines of Java or C to query a database, you just write a short SQL query like:

```
SELECT * FROM products
```

SQL is a **Domain-Specific Language (DSL)**. It's focused on interacting with databases and hides all the low-level plumbing. Expression languages in gateways work the same way, they're DSLs designed for:

- **extracting values** from JSON or XML documents
- **accessing property values** from objects
- **evaluating conditions**

By using an expression language, you can get quick wins without diving into full plugin development.

Popular expression languages include **Google CEL, Jakarta Expression Language, MVEL, JSONPath**, and **XPath**. Full-featured languages like **JavaScript** and **Groovy** are also commonly supported. They're great for quick one-liners or even moderately complex scripts.

Expression languages typically run in a **sandboxed environment**, protecting the host system while giving the script a set of context variables to interact with the gateway. For example, the **Groovy snippet** below calls an `add` method on a header object provided by the gateway:

```
header.add("X-Foo", "42")
```

Let's briefly explore a few popular expression languages. Before you go all-in on one, check that your gateway supports it.

Spring Expression Language (SpEL)

SpEL is part of the Spring Framework and provides more advanced features than the Jakarta Expression Language, commonly used in Jakarta EE projects. It is also a powerful alternative to expression languages such as OGNL or MVEL.

If you are working with Spring Boot, you may already be familiar with SpEL because it is frequently used for configuration or security rules.

SpEL supports property access, method invocation, collection handling, and conditional logic. This makes it suitable for dynamic routing, conditional execution, and of course value extraction inside an API Gateway.

Groovy

Groovy is a full-featured scripting language with seamless Java interoperability. Many Java-based API Gateways, such as **Apiman and Gravitee**, support Groovy, allowing you to extend gateway functionality using the full power of the Java ecosystem. You can even add libraries

to the classpath for advanced tasks like decoding JWTs, transforming XML/JSON, or accessing databases.

Groovy is like Java's laid-back cousin: flexible**,** expressive, and powerful. But with great power comes great responsibility. Groovy scripts might have full access to the underlying operating system. That means they can read files, open sockets, or execute external commands. Capabilities that can make security officers nervous. For this reason, some gateways run Groovy in restricted or sandboxed environments.

JavaScript

Thanks to embeddable JavaScript engines and native support for JSON, JavaScript is a natural fit for API Gateways. Gateways like **Apigee** and **Gravitee** offer built-in support. It's handy for transforming JSON payloads. Consider this transformation example:

```
({
    id: json.id,
    client: json.customer,
    positions: json.items.map(i => ({
         pieces: i.quantity,
         price: i.price
       })
    )
})
```

JSONPath

JSONPath is inspired by XPath and is designed for querying structured **JSON** data. It's commonly used in API Gateways to extract data from incoming or outgoing payloads.

The expression:

```
$.article[].name
```

returns the `name` fields of all elements in the `article` array.

XPath

If you're working with **XML** payloads, XPath is the natural choice. It is often called **SQL for XML** because it allows the extraction of values from XML documents. The learning curve is relatively shallow, yet it provides advanced features to handle even complex querying tasks.

Take this expression, for example. It does quite a lot in one line:

```
//article[@promotion]/color
```

It selects all `color` children of `article` elements that have a `promotion` attribute, regardless of where those `article` elements appear in the document.

In the screenshot below, you can see an online XPath expression tester in action. The **left panel** displays the input XML document, the **top field** contains the XPath expression, and the **right panel** shows the evaluation result. Such tools are useful to develop and test expressions.

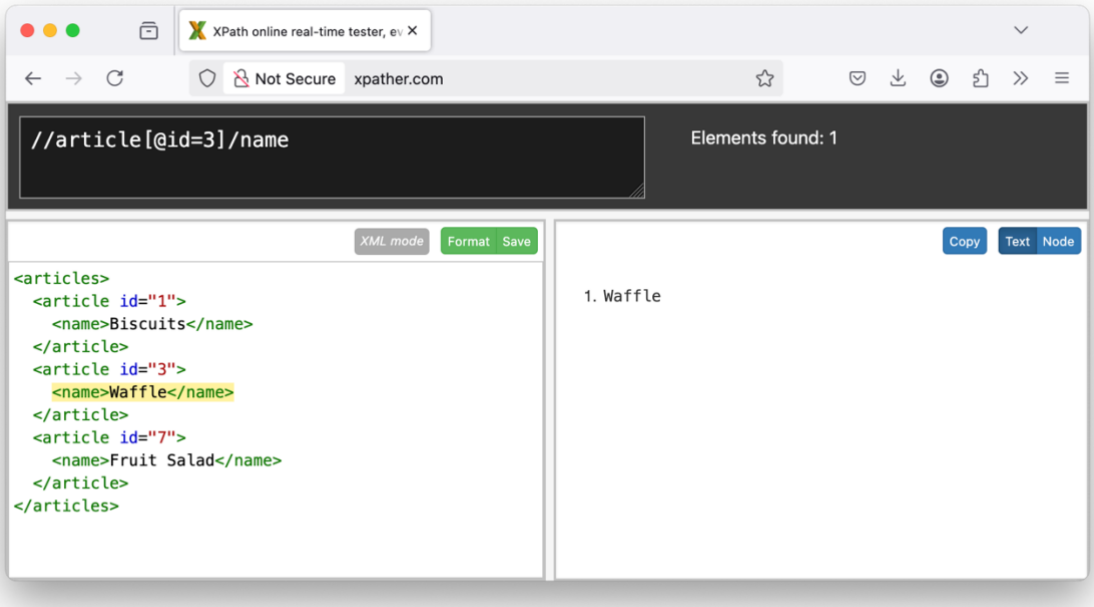


Image: XPath expression tester with document view and result output

Tip: Be cautious when pasting sensitive data into online tools. You never know who might be collecting or logging the input.

For most development environments, there are also **local tools and plugins** available that let you safely experiment with **JSONPath** or **XPath** without sending data over the internet.

Here are a few more useful XPath examples:

Expression	Description
`/articles/article/name`	Retrieves the `name` elements of all `article` elements that match the exact path starting from the root element `articles`.
`//article[1]`	Selects the first `article` element in document order across the entire document.
`article[last()]`	Selects the last `article` element among the current node's `article` children.
`article/@id`	Retrieves the `id` attributes of all `article` child elements of the current node.

Table: XPath examples

3.5 Custom Plugins

Custom plugins let you extend or tweak your API Gateway's behavior seamlessly. They **integrate deeper** than simple scripts, allowing you to run code not only during the request or response flow but also during key **lifecycle** events like initialization and shutdown. Plus, plugins often provide access to inner components like **caches or the routing engine**.

While writing a plugin requires more effort than a quick script, it gives you greater control and the plugin becomes a **first-class citizen of the gateway**.

How you implement a plugin depends on the technology behind the API Gateway. For example, nginx-based gateways often use OpenResty with **Lua**, gateways written in Go are typically extended in **Go**, and Java-based gateways are usually extended with **Java**.

4 Deployment

This chapter introduces the core deployment models of API Gateways and explains how to integrate a gateway into your organization's infrastructure. It covers network placement, **firewall** integration, and operation within a **DMZ**.

You will learn what to consider when positioning gateways at the edge of your network, in the cloud, or in hybrid environments. Each option has implications for security, latency, scalability, and operational complexity.

(Stateless) Standalone Gateway

The simplest deployment model consists of the gateway alone. In this stateless setup, the configuration is usually stored in a local file. Changing the configuration typically means editing that file and restarting or reloading the gateway.

The main advantage is simplicity. A restart brings the gateway back to a clean, well-defined state. There is no external system that must be synchronized or recovered. This makes the setup predictable and easy to operate.

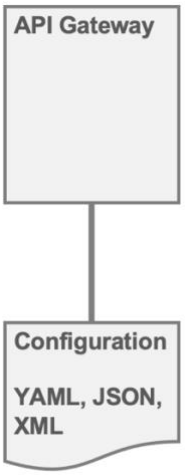


Image: Minimalistic standalone API Gateway

Because no persistent state is stored, horizontal scaling is straightforward. You can start multiple identical instances using the same configuration file and place them behind a load balancer.

Sidenote: Stateless API Gateways
Stateless does not mean that the gateway cannot use memory during request processing. It simply means that no long-term operational state such as rate-limit counters or analytics data is persisted across restarts unless external systems are added.

Gateway with Database Backend

It's also common to store the configuration in a **database**. This enables features like a **graphical user interface (GUI)** for editing the configuration and centralized management of logs, metrics, or usage statistics. Multiple gateways can connect to the same database to stay synchronized.

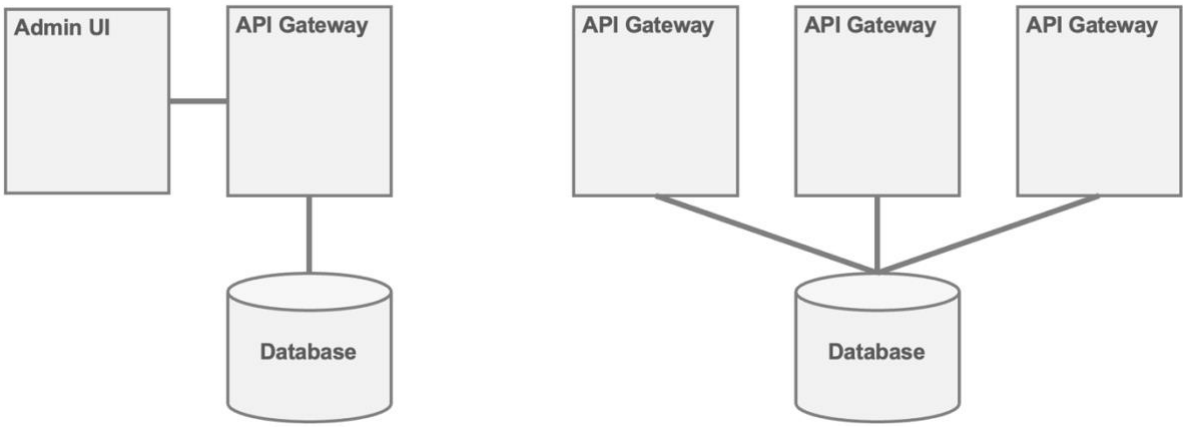


Image: API Gateway and cluster with shared database for configuration and metrics

The **Kong Gateway**, for example, supports both modes: it can run **with** a database to enable full admin functionality, or **without** one for lightweight scenarios. This gives users the flexibility to choose between **feature richness** and **simplicity**.

Extending Gateways with additional Components

Modern API Gateways often support a modular architecture, allowing external components to be integrated as requirements grow.

Common components include:

- **Cache servers** (e.g., Redis, Memcached)
  Used to **store tokens**, **session state**, or counters for **rate limiting** across gateway instances.
- **Monitoring tools** (e.g., Prometheus, Grafana)
  For collecting metrics, visualizing traffic patterns, and triggering alerts.
- **Log aggregators** (e.g., Elasticsearch, Loki)
  To centralize log collection and support advanced search or correlation.
- **Security services**
  Such as external **policy engines** (like OPA), threat detection systems, or **data loss prevention (DLP)** filters.

Some gateways require these components for key features to function. Others provide optional support for them, allowing you to start simple and grow as needed.

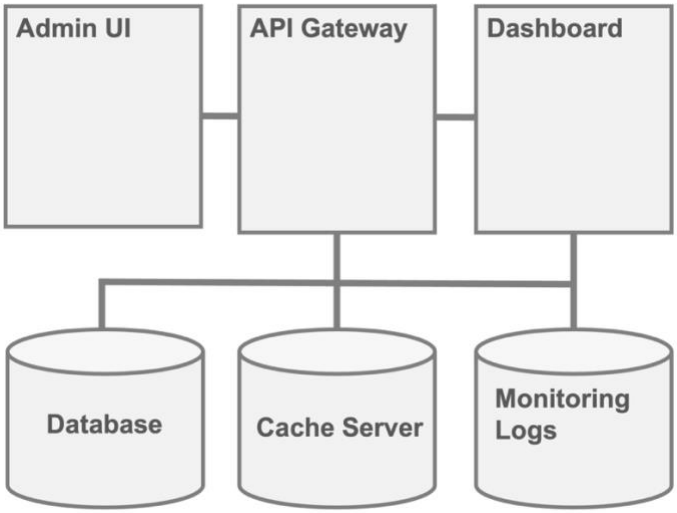


Image: Modular API Gateway setup with optional integrations

Gateways often follow a plug-and-play model: a minimal setup might start with just the gateway and a configuration file, while features like token caching, traffic monitoring, or log aggregation can be **added step by step** as requirements grow.

4.1 Exposing APIs to external Organizations

Organizations frequently need to provide external access to their APIs, for partners, service providers, or customers, while preserving strict security boundaries. This requires thoughtful network design that protects sensitive internal systems from unauthorized access.

Typically, the first line of defense is a firewall, which shields internal networks from external exposure.

Example: Self-Service Portal for an Insurance Company

Imagine an insurance company wants to offer customers a self-service portal to manage their contracts. The challenge is providing external access without compromising internal security.

Demilitarized Zone (DMZ)

To tackle this, companies use a demilitarized zone, a secure buffer network situated between the public internet and internal networks.

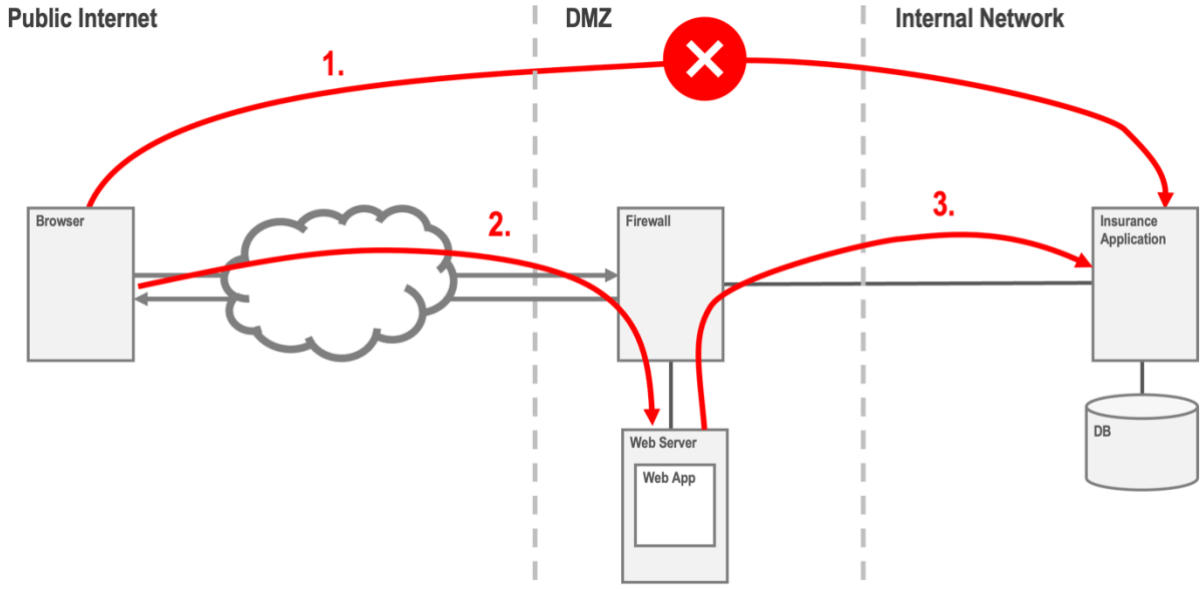


Image: Demilitarized zone between internet and internal network

A DMZ has these key characteristics:

1. Direct routing between internet and internal networks is blocked.
2. Inbound traffic is directed to servers located within the DMZ.
3. DMZ hosts can initiate connections to internal resources.

A secure self-service portal could be set up as follows:

- A **Web application** runs in the DMZ, handling customer interactions.
- This Web app connects to **internal backend services** to access data.

While effective, this setup exposes a complex application directly in the DMZ, creating a sizable attack surface. To mitigate this, companies commonly:

1. Host critical applications within protected internal networks.
2. Use a **reverse proxy** in the DMZ, forwarding external requests securely to internal apps.

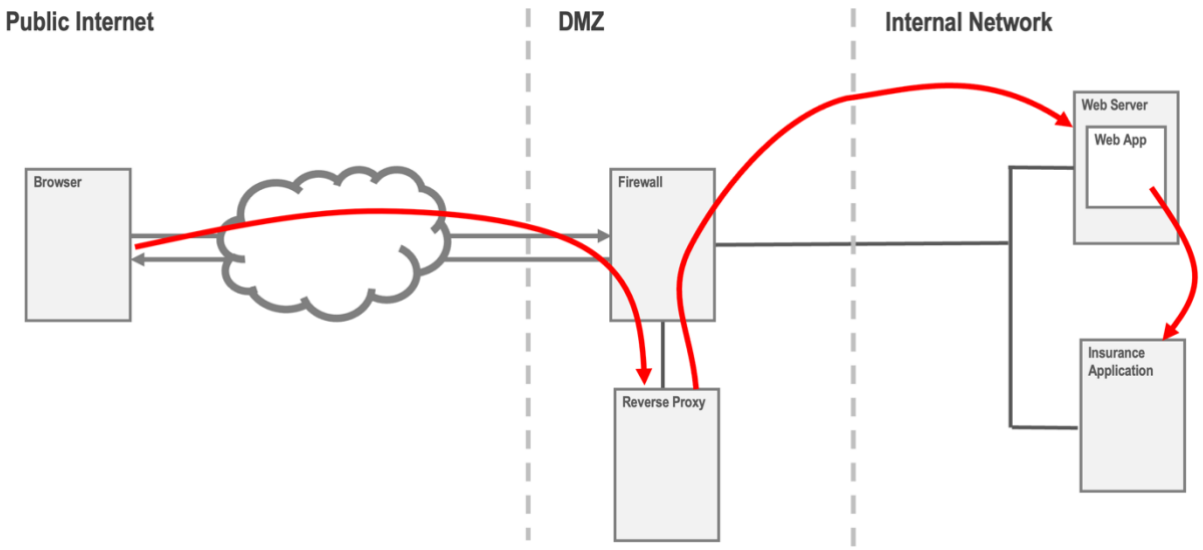


Image: DMZ with reverse proxy

Benefits of a reverse proxy:

- **Reduced Attack Surface**: A simple reverse proxy offers fewer vulnerabilities.
- **Enhanced Security:** Sensitive logic and data remain protected internally.

4.2 Backend for Frontend (BFF) Pattern

Modern web applications, especially **Single Page Applications** SPAs, run in the browser as JavaScript clients. Unlike traditional server rendered applications, SPAs communicate with backend services through APIs to fetch data and trigger functionality.

A Backend For Frontend BFF is a dedicated backend component that acts as a bridge between a specific frontend and internal APIs. It is typically deployed in a secure zone such as a DMZ and exposes only the endpoints required by that frontend.

The key characteristics and features of a BFF are:

- **Dedicated per frontend**
  Each frontend typically has its own dedicated BFF.
- **Tailored requests and responses**
  The BFF aggregates, reshapes, or filters data so the frontend receives exactly what it needs.
- **Request validation**
  Incoming requests are validated before they reach internal services.
- **Authentication and authorization**
  The BFF manages tokens, sessions, and access control policies.

The BFF pattern caters to the separation of concerns principle. Frontend specific logic stays in the BFF, while internal APIs remain clean and reusable. It also strengthens security by limiting direct access from the browser to internal services.

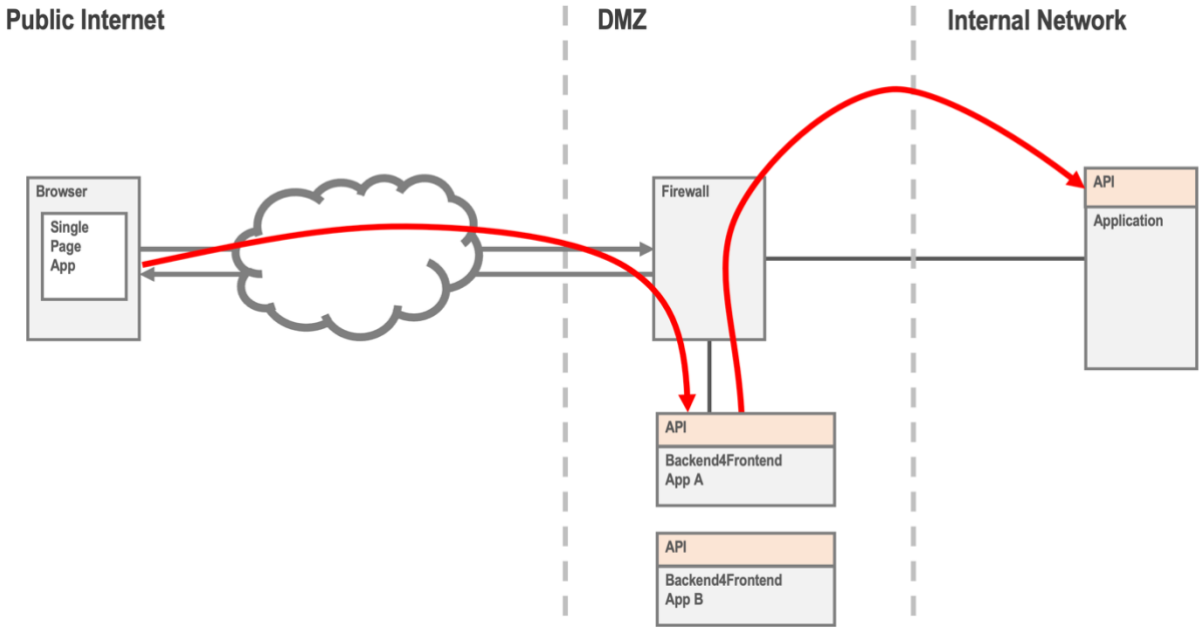


Image: Backend For Frontends (BFF) in the DMZ

Replacing BFF Functionality with API Gateways

When a BFF takes on too many responsibilities, complexity increases quickly. Using a BFF to reshape interfaces and separate frontend concerns is reasonable. But once it also handles cross-cutting concerns such as security and monitoring, it can become more costly and time-consuming than needed.

In many cases, an API Gateway can provide a simpler and more standardized alternative for these concerns.

The key advantages of using an API Gateway instead are:

1. **No custom coding**
   Unlike a custom-built BFF, an API Gateway is typically configured rather than coded. APIs can be exposed and secured through configuration, which reduces implementation effort.
2. **Faster deployment**
   Adding or modifying an API in a gateway often takes only minutes. Rolling out a custom BFF requires development and testing. This can take several days or longer depending on the process.
3. **Standardization**
   Using an API Gateway ensures a consistent, reliable configuration process. This reduces the risk of errors that can occur due to custom coding and manual maintenance in BFF implementations.
4. **Out-of-the-Box features**
   Gateways provide production-grade capabilities such as OpenAPI validation, authentication, and GraphQL protection. There is no need to reinvent common infrastructure features.

5. **Security**
   API Gateways come with battle-tested security functions. Self-implemented infrastructure always comes with the risk of subtle vulnerabilities.

These characteristics turn API Gateways into a robust, efficient, and secure alternative to the Backend for Frontend approach, offering **streamlined API management, accelerated deployment, and a significantly faster time to market.**

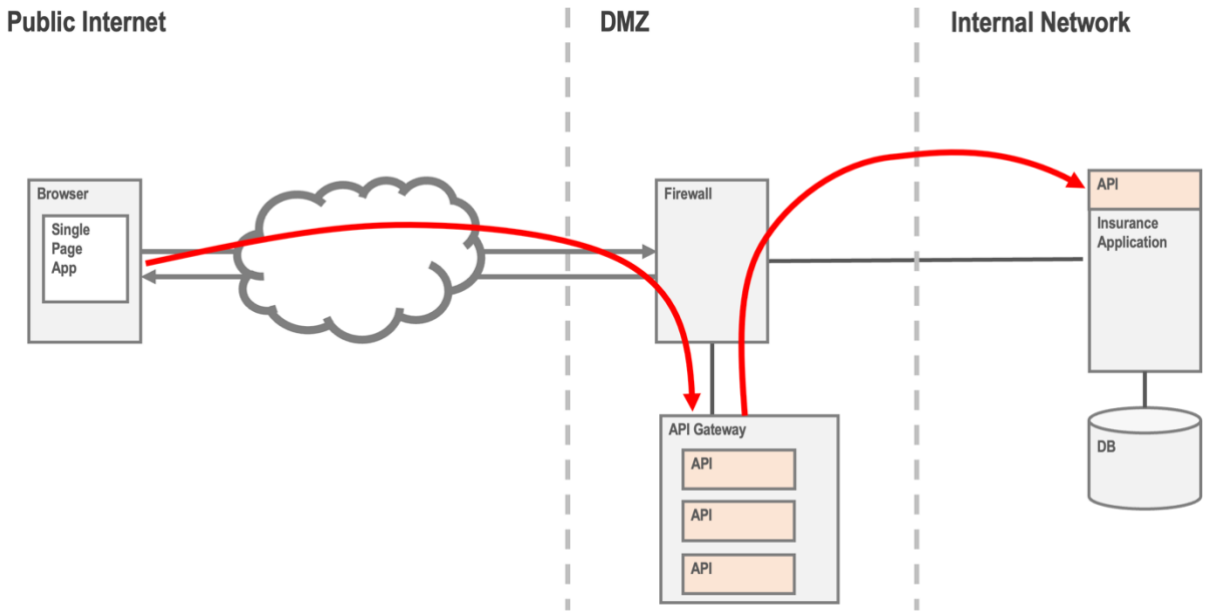


Image: Single API Gateway replacing multiple BFFs

However, combining BFFs and API Gateways can also be a sound architectural choice. The BFF can focus on frontend-specific orchestration and interface reshaping, while the API Gateway handles cross-cutting concerns such as authentication, validation, and traffic management. This separation keeps responsibilities clear and reduces architectural friction.

An additional advantage of this setup is network placement. The API Gateway can be positioned at the edge, for example in the DMZ, where it terminates external traffic and enforces security policies. The BFF can remain inside the internal network, shielded from direct internet access.

This layered approach strengthens security boundaries while preserving flexibility in frontend specific logic.

4.3 Outgoing Gateways

An outgoing API Gateway is the reverse of the typical gateway setup. Instead of managing incoming traffic from the outside world, it handles **outbound** requests from internal systems **to external APIs** like Stripe, PayPal, or Twilio. It's also useful for accessing APIs provided by business partners or public cloud services.

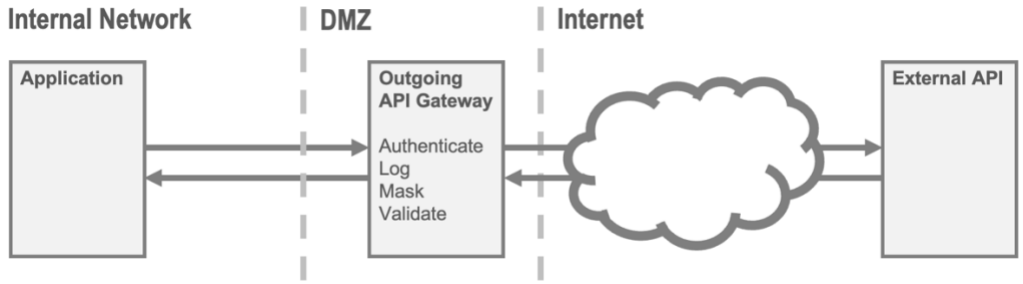


Image: Outgoing gateway routing API requests to external services

Rather than allowing any application in the company to connect to external APIs, an outgoing gateway provides a central, controlled egress point. It helps to:

- **Restrict outbound traffic** to approved external APIs
- **Limit external access** to select internal applications
- **Handle authentication** by adding tokens or API keys
- **Mask or sanitize** sensitive data before it leaves the company
- **Log and monitor** outgoing traffic for auditing or compliance
- **Simplify external API consumption** by handling authentication, versioning, and format conversions
- **Validate responses** before they enter internal systems
- Apply **rate limiting** to control usage and costs (especially useful for pay-per-use APIs like LLMs)

An outbound gateway helps enforce consistent traffic policies and avoids a situation where everyone builds their own outbound solutions.

The Problem with Default Gateway Behavior

Most gateways are designed for inbound traffic. Using them as an outgoing gateway without adjustment can unintentionally leak internal information.

Imagine an internal client sends this request to an outgoing gateway:

```
POST /payments
Host: egress.predic8.com
Content-Type: application/json
User-Agent: SAP (compatible; 750 2.0; abap client 1.0)
X-Api-Key: 6ee7ffc8-5b57-4de6-90cf-02d78591a888
X-Api-Key-Gateway: a79ca858-561a-435e-a5ff-e848c6a2ed3e

{ "payment": "..." }
```

The **X-Api-Key** is needed to authenticate at the external API, and the **X-Api-Key-Gateway** header to authenticate with the outgoing API Gateway to get permission to reach outside.

The gateway now forwards the request to the external API:

```
POST /payments
Host: api.example.com
Content-Type: application/json
User-Agent: SAP (compatible; 750 2.0; abap client 1.0)
X-Api-Key: 6ee7ffc8-5b57-4de6-90cf-02d78591a888
X-Api-Key-Gateway: a79ca858-561a-435e-a5ff-e848c6a2ed3e
X-Forwarded-For: 10.0.3.127

{ "payment": "..." }
```

The external API receives more than it should. Why is that a problem?

- `User-Agent` reveals internal technologies, in this case: SAP and its version.
- `X-Api-Key-Gateway` leaks internal credentials that should **never** reach the outside.
- `X-Forwarded-For` exposes an internal IP address, which could be used for fingerprinting or profiling.

To safely use an API Gateway for outgoing traffic:

- **Prevent the automatic addition** of `X-Forwarded` header fields
- **Don't forward** internal-only headers
- **Only pass** required headers like `Content-Type` or external API credentials

💡 **Sidenote: Outbound APIs in regulated Environments**
Outgoing gateways are especially valuable in regulated industries**,** where strict auditing and control over data flows leaving the organization are required.

4.4 Internal Gateways

API Gateways can be just as valuable *inside* the network, managing service-to-service communication between internal applications.

Two main topologies have emerged: **one central** or **multiple decentralized** gateways.

One Central Gateway

In this model, all internal API traffic flows through a single, centrally managed gateway. This creates a unified control point with several benefits:

- **Centralized Governance**
  Monitoring, security, and version control are handled in one place.
- **Operational Efficiency**
  One single central gateway can reduce operational complexity, especially when each gateway installation incurs costs.

However, this setup also has drawbacks:

- **Single Point of Failure**
  A failure or performance issue in the central gateway can impact the availability of all APIs.
- **Vendor Lock-in**
  Relying on a central gateway product can bring back the same concerns once seen with the monolithic **Enterprise Service Bus (ESB)**. When an entire organization depends on a single critical installation, replacing it later, especially after support ends, can become a costly and risky endeavor.

Decentralized Gateways

In a decentralized setup, multiple lightweight gateways are distributed across the organization. Each gateway serves a specific domain, team, or platform. Modern lightweight gateway solutions make installation and maintenance relatively easy.

Different types of gateways can be deployed depending on the needs of the environment, such as:

- **Internet-facing gateways** for publishing APIs.
- **Cloud gateways** for managing API access within or across cloud environments.
- **Container-native gateways** to handle traffic within platforms like Kubernetes.
- **Integration gateways** with connectivity to messaging systems or legacy protocols such as Web Services

Each type of gateway can be deployed in multiple locations and with multiple instances if needed. The advantages of a decentralized approach include:

- **Increased Resilience**
  Eliminates the risks of a single point of failure.
- **Flexibility**
  Enables teams to select the right gateway for their use case.
- **Scalability**
  Makes it easier to grow the infrastructure alongside API demand and traffic volume.

The main downside is added complexity. Managing and mastering multiple different gateways across teams and environments adds overhead in coordination and monitoring.

Microgateways

Microgateways are purpose-designed for minimal resource usage, often consuming less than 100 MB of RAM. They are ideal for highly scalable, containerized environments where memory and startup time matter. Gateways implemented in efficient languages like Go, Rust, or C tend to perform particularly well in these scenarios.

The table shows the memory footprint of selected gateways:

Gateway	RAM Footprint in MB	Platform	Comment
Microgateways			
Envoy	14	C++	
KrakenD	20	Go	
traefik	23	Go	
tyk	72	Go	
Lightweight Enterprise Gateways			
APISIX	209	nginx, C, Lua	Gateway and etcd registry
Gravitee	416	Java	
Kong	368	nginx, C, Lua	Without database
Membrane	195	Java	

Table: Micro- and enterprise gateways

Note: Memory usage depends on the specific version and setup. These values were obtained through simple measurements and are meant for rough comparison, not as formal benchmarks.

Even the heavier gateways in this list are relatively lightweight compared to traditional enterprise API Gateways, which require significantly more memory, disk space, and external services. All gateways shown here are more or less suitable for microgateway usage depending on the scenario.

4.5 Clustering Gateways

To ensure **high availability** and handle **large volumes of traffic**, API Gateways can be deployed as a cluster. Since gateways are often **stateless**, meaning they don't retain session data or request history, they're well-suited for horizontal scaling. You can simply spin up multiple instances behind a load balancer to distribute incoming requests.

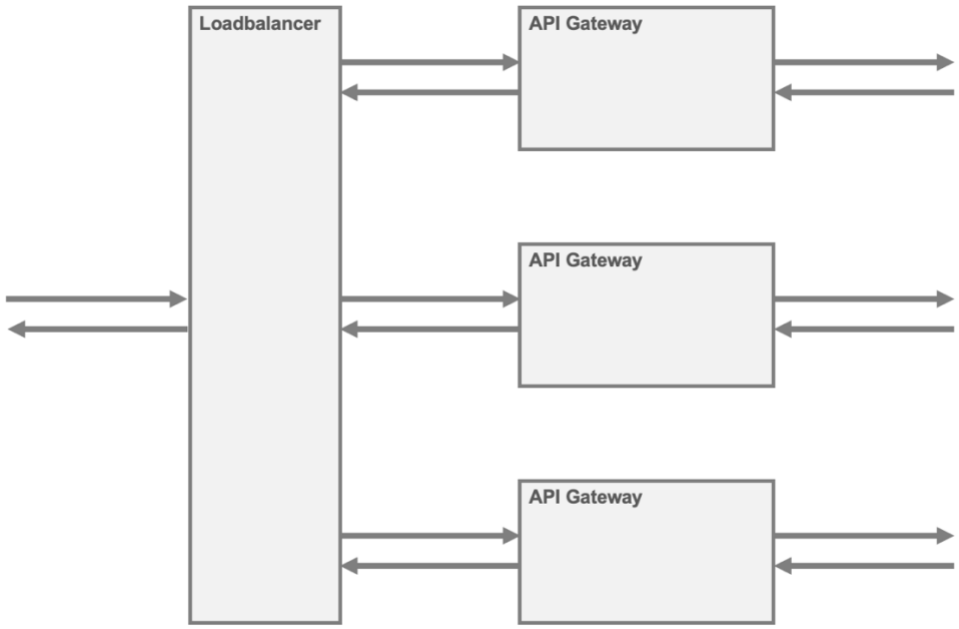


Image: Cluster of API Gateways behind a load balancer.

However, some use cases require **stateful behavior**, such as:

- Session-based authentication (e.g., with cookies)
- Accurate rate limiting

Statelessness makes scaling easy, but in these cases, it can create issues. For example, if a client's requests are routed to different gateway instances, and each instance keeps its own rate-limit counter, the client may effectively bypass rate limits.

To manage state in a clustered gateway setup, two main strategies are used: **shared state** and **sticky sessions**.

Shared State

A shared cache or centralized storage system, such as **Redis or Memcached**, can synchronize state across gateway instances. This allows each instance to access the same session data, rate-limit counters, or authentication tokens, ensuring consistent behavior regardless of which instance handles a request.

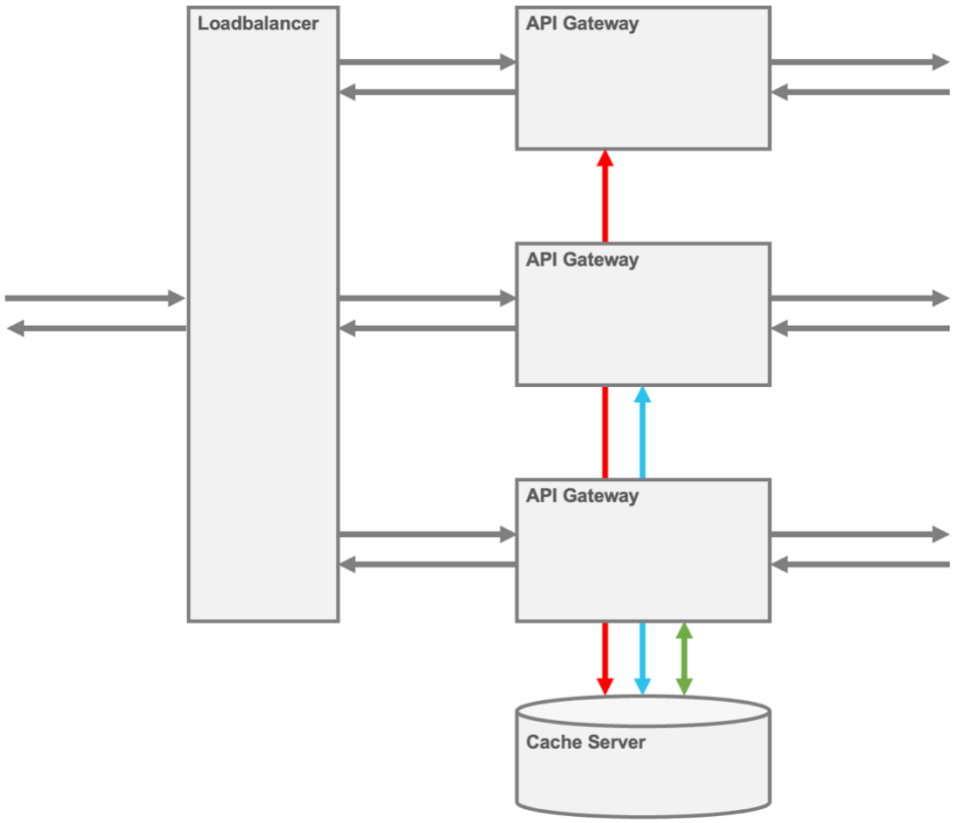


Image: Gateways sharing a cache server like Redis to manage a unified state

💡Sidenote: Why Redis?
Redis is a high-performance, in-memory key-value store often used for caching and transient data. It supports data structures like counters, lists, and expiring keys, making it ideal for tasks like rate limiting or session tracking across distributed systems.

Sticky Sessions (Affinity)

An alternative is to configure the load balancer for session affinity (also known as **sticky sessions**). This ensures that requests from the same client are consistently routed to the same gateway instance, typically using a session cookie. This way, state remains local to each gateway instance but still behaves consistently for each client session.

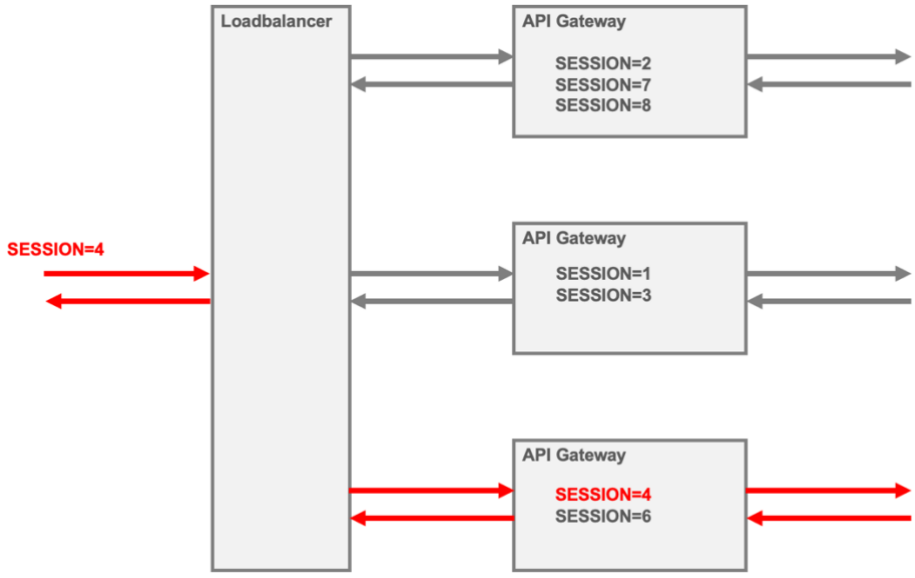


Image: API Gateway instance selection based on session ID

It's common practice to place a **dedicated load balancer** in front of a gateway cluster. However, many modern gateways come with built-in **load balancing capabilities**. In some setups, the gateway itself can act as a load balancer, distributing requests across multiple backend services.

💡 **Tip:** Whenever possible, design and deploy your gateways to be stateless. Stateless gateways simplify scaling, improve reliability, and significantly ease the deployment and operational complexity.

4.6 Chaining Gateways

It's common in real-world architectures to chain multiple API Gateways, each with a distinct role in the infrastructure. Gateways often form a **pipeline**, with each one handling a specific layer of responsibility, from external traffic filtering to internal routing and observability.

A typical gateway chain might include:

- **A load balancer and API Gateway in the DMZ**
  Provides initial security such as authentication, input validation, and protection against malformed JSON or XML payloads.

- **An internal API Gateway**
  Resides within the corporate network, responsible for routing and access policies.
- **An ingress gateway in a Kubernetes cluster**
  Directs external traffic into the cluster and distributes it to the correct services.
- **Sidecar gateways** (proxies in a service mesh)
  Deployed alongside individual services, handling service-to-service communication, traffic shaping, and observability features like tracing and metrics.

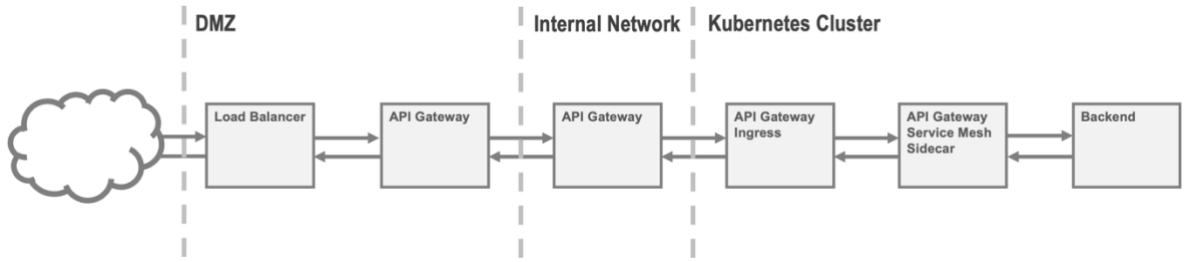


Image: Chain of multiple gateways from DMZ to backend

In microservices architectures, it's also common for each service to be protected by its **own gateway or sidecar proxy**, adding further layers to the chain.

Trade-offs in Gateway Chaining

Routing traffic through multiple gateways naturally introduces a performance overhead. Every hop adds latency and processing time. The challenge is to balance security, observability, and reliability against acceptable performance.

Still, the operational overhead is often lower than expected.

We conducted an experiment chaining **500 gateways sequentially** on a single machine. Each gateway passed the request to the next, using the local operating system's networking stack. The test used a POST request with a 100 KB payload and a 100 KB response. Even with this extreme setup, the total **round-trip** time remained under **200 milliseconds.**

This result demonstrates that even a long chain of gateways introduces only moderate latency. In real-world scenarios, where the number of chained gateways is typically between two and five, the performance penalty is often negligible and outweighed by the benefits of layered control, observability, and modularity.

The surprisingly low latency also supports current architectural trends, especially in **Zero Trust environments**, where clear segmentation and policy enforcement zones are essential.

4.7 Zoning and Zero Trust

While many diagrams in this book show a simplified architecture with just three network zones: **Internet**, **DMZ**, and **Intranet**, this doesn't reflect the reality of most enterprise environments. Nor is it sufficient from a security standpoint.

This basic three-zone model creates a potential **attack path**.

An attacker might breach a single poorly secured system in the Intranet, then use it as a **stepping stone** to move laterally within the network. Given the number and variety of systems inside most corporate networks, it's likely that at least one weak point exists.

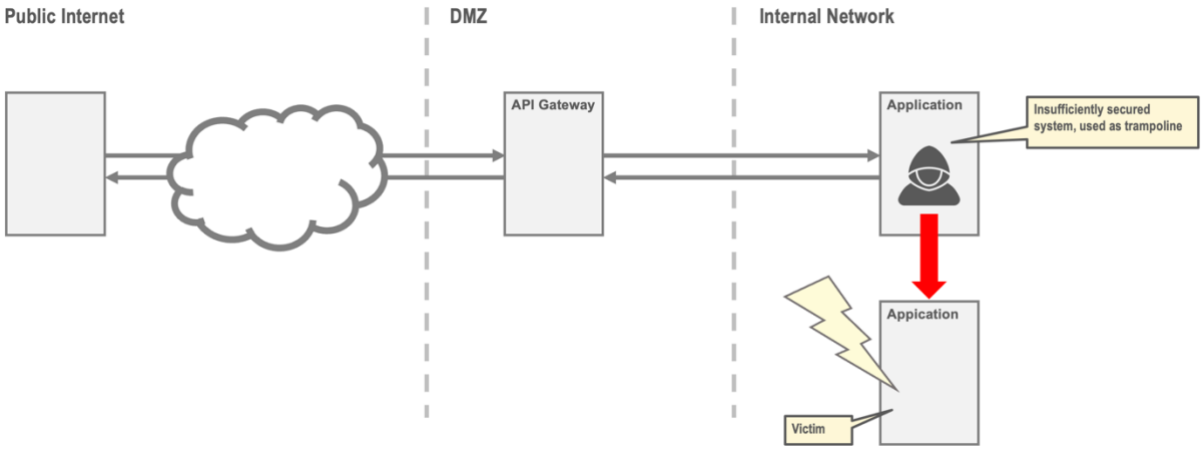


Image: Attacker jumping from compromised Intranet system to internal targets

Fine-Grained Zoning

To strengthen internal security, many organizations introduce **additional internal zones**, each with its own security boundary. This strategy limits lateral movement within the network, helping to contain potential breaches. Technologies like **Software-Defined Networking (SDN)** make it easier to define, manage, and adapt these segmented network zones dynamically.

More zones also mean more complexity. Fine-grained segmentation makes routing between services harder. Systems that previously communicated directly may now need to cross multiple boundaries, and each boundary can enforce different access rules.

That's where **internal API Gateways** come into play. These gateways manage traffic between zones, acting as both **routing hubs** and **policy enforcement points**. They help ensure that only authorized, and validated traffic can pass from one zone to another.

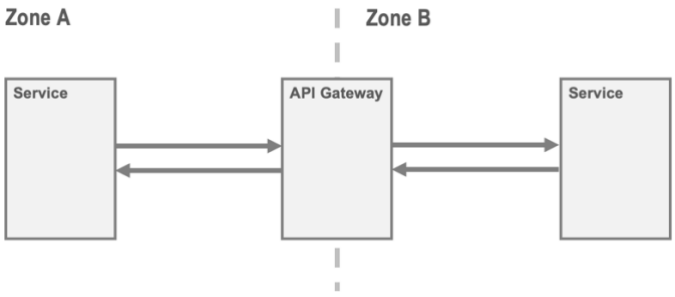


Image: Gateway routing API traffic between internal network zones.

By placing gateways at these boundaries, organizations maintain both **control and visibility** without sacrificing modularity or security.

Zero Trust Networking

Another approach gaining traction is the **Zero Trust** model. Unlike traditional perimeter-based models, Zero Trust **does not automatically trust anything**, not even devices or services inside the internal network.

Under Zero Trust principles:

- Internal network connections are considered untrusted by default.
- Every communication must be **authenticated and authorized**, regardless of its origin.
- All network traffic, especially between services like APIs and gateways, must be encrypted and verified**,** typically using TLS or mutual TLS (mTLS).

The good news? API Gateways are well-suited for Zero Trust environments. They can:

- Terminate and initiate TLS or mTLS connections
- Authenticate and authorize requests
- Enforce fine-grained policies per service or client

This makes them a natural fit for enforcing Zero Trust policies at network and application boundaries.

5 Installation and APIOps

In the past, setting up an API Gateway often meant using a graphical user interface or managing it as part of a larger, sometimes complex **API management** platform.

Today, that's changed. Modern gateways are typically installed and configured using **DevOps**-practices. API Gateways are often packaged as **containers**, making them easy to deploy, update, and scale. This shift enables teams to automate the deployment process and maintain consistent configurations across development, staging, and production environments.

5.1 Containerized Gateways

Packaging API Gateways into containers offers several benefits. It ensures **portability** and **consistency** across development, testing, and production environments. Teams can focus on configuring and scaling the gateway without worrying about the underlying infrastructure.

Containerized gateways are easy to deploy and integrate into CI/CD pipelines. Many gateways can be launched with a **single Docker command**, making it simple to get started or to test locally.

Here are some examples of popular gateways and how to start them using Docker:

Envoy

```
docker run -p 9901:9901 -p 10000:10000
envoyproxy/envoy:v1.73.7
```

Kong

```
docker run \
  -e "KONG_DATABASE=off" \
  -p 8000:8000 -p 8443:8443 \
  kong:latest
```

Membrane

```
docker run --name membrane -p 2000:2000 predic8/membrane
```

Tyk

```
docker run --name tyk -p 8080:8080 tykio/tyk-gateway
```

Certain gateways require additional infrastructure running in separate containers. For example, **APISIX** uses a **Docker Compose** file to launch an **etcd** registry alongside the gateway container.

5.2 APIOps

APIOps applies **DevOps** principles such as automation, version control, and continuous delivery to the **API lifecycle**. It treats both APIs and API Gateways as code, enabling **repeatable, testable,** and **secure** deployments.

By integrating APIOps into your workflow, the configuration and deployment of gateways become significantly more streamlined. Gateway configurations and OpenAPI specifications are stored in **source control** systems like Git, with **pipelines** managing validation, build, and deployment.

A typical APIOps deployment pipeline for updating an API Gateway might work like this:

1. **Merge**
   A configuration change is merged into the main branch of the Git repository, triggering a pipeline.
2. **Verification**
   The gateway configuration is validated for syntax and structural correctness. This may include OpenAPI linting.
3. **Build**
   A container image is built with the updated configuration and pushed to a container registry.
4. **Deployment**
   The image is deployed to the target environment, whether that's Kubernetes, a VM cluster, or a cloud-hosted gateway instance.

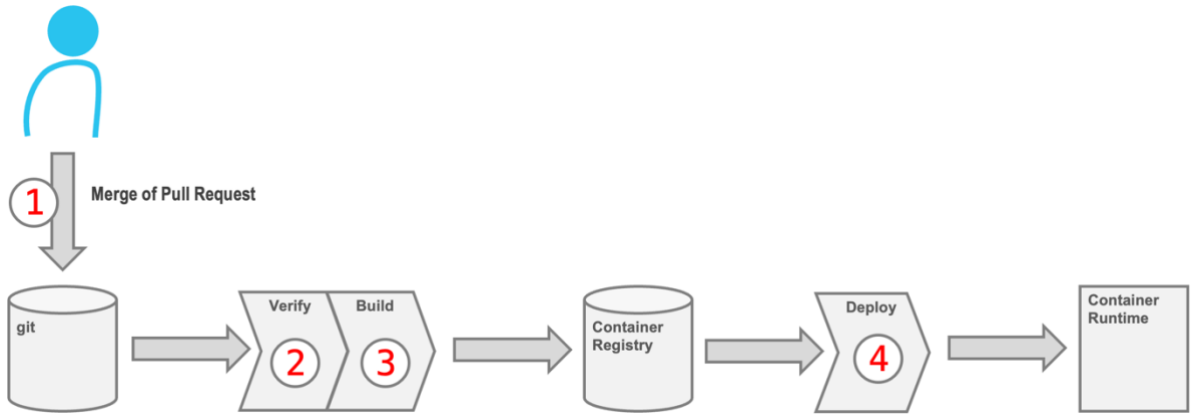


Image: DevOps-based API deployment process

This automated, GitOps-style workflow ensures:

- Consistent configurations across environments
- Fewer manual errors
- Faster and safer rollouts
- A transparent audit trail linked to the source control system

APIOps doesn't just improve efficiency. It also increases confidence in the delivery pipeline.

💡 **Sidenote: Why apply APIOps to gateways?**
Treating gateway configuration as code brings the same consistency and agility that DevOps brought to application code. It also helps **prevent configuration drift** between environments, which often sneaks in through ad hoc changes in UIs or quick fixes in the terminal.

6 OpenAPI

OpenAPI has become the de facto standard for describing HTTP-based APIs. But it's more than just a documentation format. As we already saw in the chapter on configuration, OpenAPI can play a central role throughout the entire API lifecycle.

In this chapter, we'll explore how OpenAPI is used by API Gateways. You'll learn how gateways can:

- Be configured directly from OpenAPI descriptions
- Rewrite addresses in OpenAPI documents on the fly to reflect public-facing endpoints
- Validate incoming and outgoing messages against OpenAPI definitions

These capabilities not only improve the developer experience but also help enforce consistency, contract compliance, and security at runtime.

💡 **Sidenote: What is OpenAPI?**
The **OpenAPI Specification** (originally known as Swagger) defines a standardized way to describe APIs using YAML or JSON. It goes beyond just documentation. OpenAPI descriptions can drive tools for mocking, validation, code generation, testing, and automation. In many setups, these specifications even serve as the configuration source for API Gateways, making them a cornerstone of modern API design and deployment.

6.1 OpenAPI-based Configuration

OpenAPI provides a structured, machine-readable format for describing APIs, covering everything from endpoints and HTTP methods to parameters, authentication requirements, and response formats.

Today, many modern API Gateways support OpenAPI as a first-class configuration source. Instead of setting up routes, authentication, and validation rules manually, you often can just hand the gateway an OpenAPI file and let it handle the rest.

This approach makes the OpenAPI document a **single source of truth** for implementation, documentation, and deployment. It simplifies onboarding, reduces human error, and enables repeatable, automated rollouts of new APIs across environments.

Take **AWS API Gateway** as an example: you can import an OpenAPI file directly in the AWS Console to create and deploy a new API in just a few clicks.

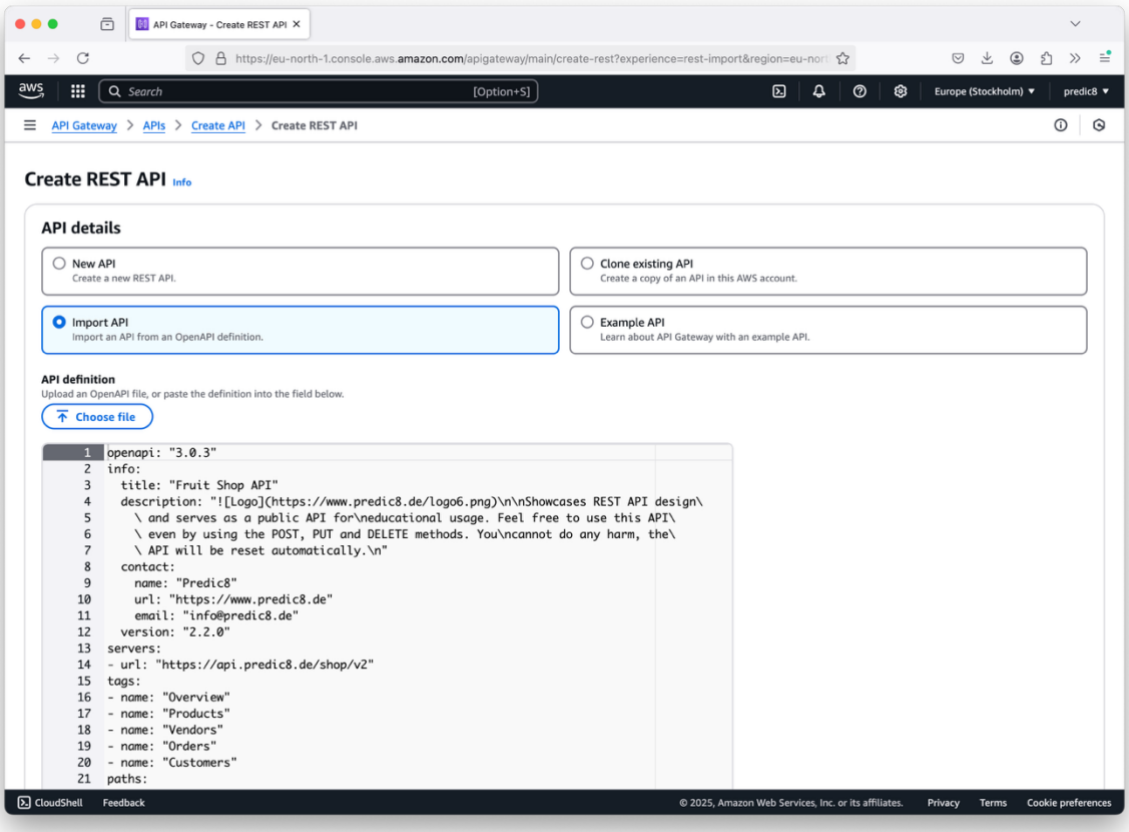


Image: Creating an API from an OpenAPI document in the AWS Console

Once imported from OpenAPI, the API definition appears in the AWS Console. All paths, methods, and parameters are visible and can be further adjusted if needed:

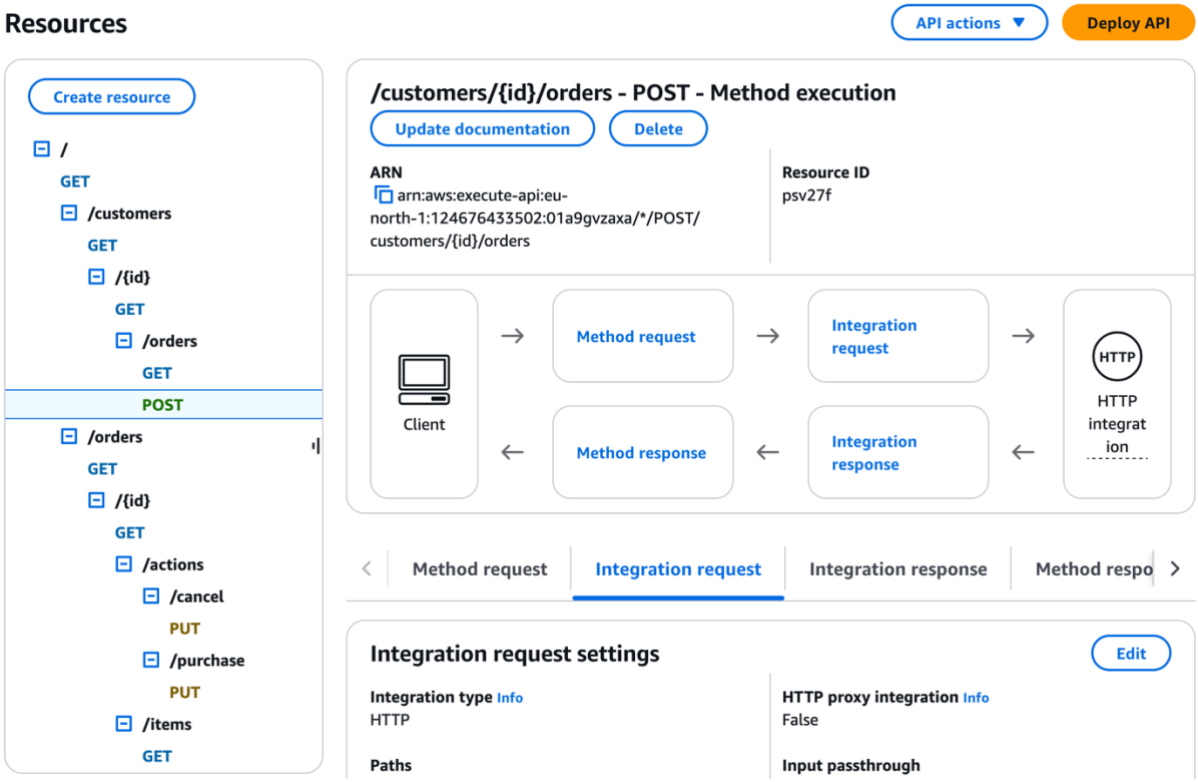


Image: API created from an OpenAPI document in the AWS console

This approach is convenient, and it fits perfectly with automated workflows. Since OpenAPI files are structured and versioned, they can live in Git repositories and serve as a single source of truth. From there, a CI/CD pipeline or the gateway itself can pick up changes and trigger deployments automatically.

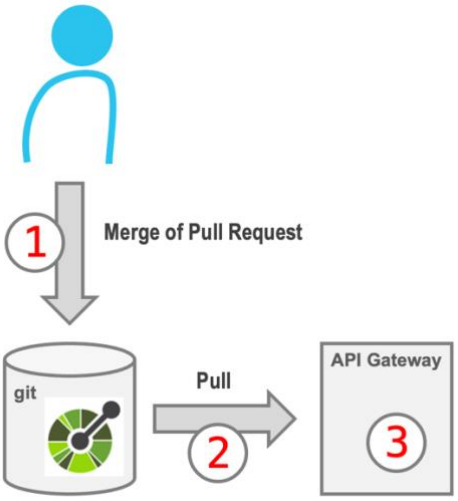


Image: Automated deployment pipeline triggered by OpenAPI changes

To improve reliability and consistency, teams often enforce a pull-request-based workflow for OpenAPI definitions. Any change triggers a CI/CD pipeline that runs automated checks, such as syntax validation, style guide enforcement, and security scanning, and may also require manual review.

One widely used tool in this process is **Spectral**, a linter for OpenAPI documents that supports both custom and community-maintained rule sets. There are even rules for the **OWASP API Security Top 10**.

💡 **Sidenote: What is Spectral?**
Spectral is a customizable linter for OpenAPI that helps to enforce consistency, quality, and security across API definitions. It can validate from basic syntax to more advanced concerns like security policies and **API Style Guides**. For teams working with **APIOps**, Spectral often becomes a central piece in the toolchain, automating governance and ensuring every API stays clean and compliant.

By using OpenAPI as the central artifact for both documentation and configuration, teams can align developers and operations, reduce duplication, and avoid drift between environments.

💡 **Sidenote: Configuration as a document**
Using OpenAPI files as configuration artifacts blurs the line between documentation and deployment. Instead of writing two separate things, API docs and gateway configs, you only need one. That reduces duplication, simplifies maintenance, and gives developers and operations teams a shared artifact to collaborate on.

Resources

OpenAPI Specification
https://swagger.io/specification/

Spectral (OpenAPI Linter)
https://github.com/stoplightio/spectral

OWASP Ruleset for Spectral:
https://github.com/stoplightio/spectral-owasp-ruleset

OWASP Top 10 API Security Risks – 2023
https://owasp.org/API-Security/editions/2023/en/0x11-t10/

6.2 OpenAPI URL Rewriting

An OpenAPI document doesn't just define the structure of requests and responses. It also tells clients **where to find an API**. The `servers` section specifies the base URL that clients should use to connect.

In this sense, OpenAPI provides a lightweight form of service discovery. Instead of querying a registry at runtime, clients read the service location directly from the API description.

Below is an example snippet from an OpenAPI that defines both a production and test environment:

```
openapi: '3.0.3'
info:
  title: Fruit Shop API
  version: '1.0'
servers:
  - url: http://srv5.predic8.de/test/shop/v2
    description: test
  - url: http://srv5.predic8.de/shop/v2
    description: production
```

Now picture this. You have an API Gateway in front of a backend service. A developer downloads the OpenAPI document from the gateway, but the gateway simply forwards the backend's original OpenAPI file unchanged (Steps 1 to 4). The developer then generates a client from that specification (Step 5).

The result is a subtle but real problem. The generated client may use the server URL from the OpenAPI document, which points directly to the backend. In that case, the client starts calling the backend service directly and **bypasses the gateway** entirely (Step 6).

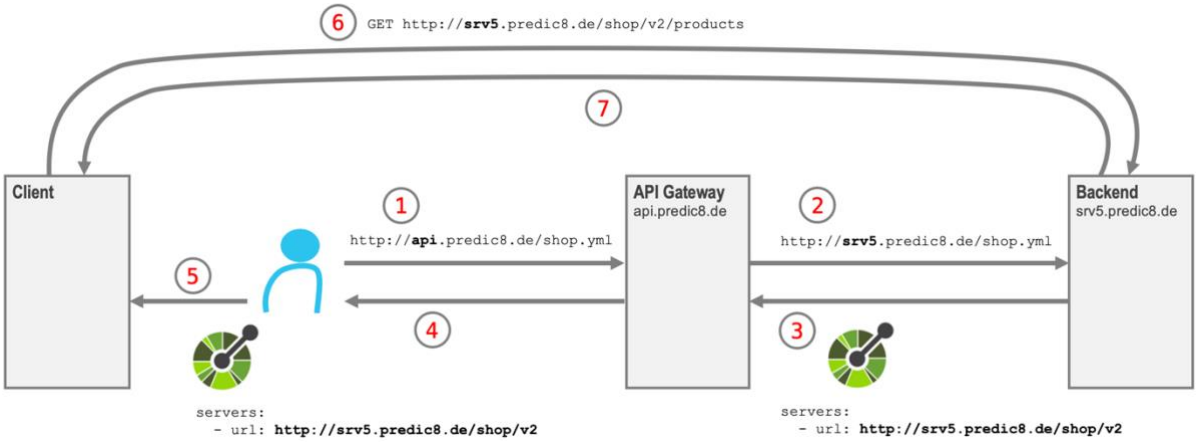


Image: Bypassing the API Gateway caused by an unmodified OpenAPI document

Ideally, the backend should only be reachable through the gateway. If you're lucky, the firewall will block the request.

URL Rewriting

The fix is to change the addresses. The gateway rewrites the OpenAPI document on the fly. More specifically, it replaces the `servers` section with the public-facing URL of the gateway.

You could also keep a modified copy of the OpenAPI document and serve that instead of fetching it from the backend. But that quickly turns into a maintenance burden. Whenever the backend API changes, for example, a new endpoint is added or a parameter is modified, you would have to update your copy manually. That step is easy to miss.

Dynamic rewriting avoids that problem. The gateway forwards the original paths, schemas, and other definitions from the backend, but swaps in the correct base URL. This ensures generated clients always call the gateway rather than reaching the backend directly.

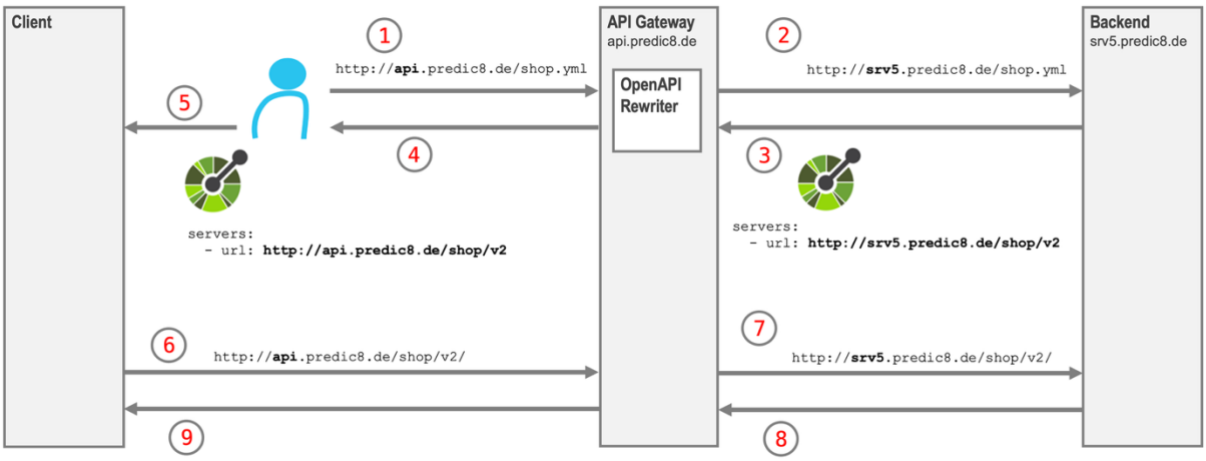


Image: API Gateway rewriting URLs in OpenAPI descriptions

Here's how the modified OpenAPI might look after rewriting:

```
openapi: '3.0.3'
info:
  title: Fruit Shop API
  version: '1.0'
servers:
  - url: https://api.predic8.de/test/shop/v2
    description: test
  - url: https://api.predic8.de/shop/v2
    description: production
```

Clients will now get the correct address, and requests go through the gateway, where they can be secured, logged, or transformed as needed.

The image shows the Swagger UI with the rewritten URLs in the server dropdown.

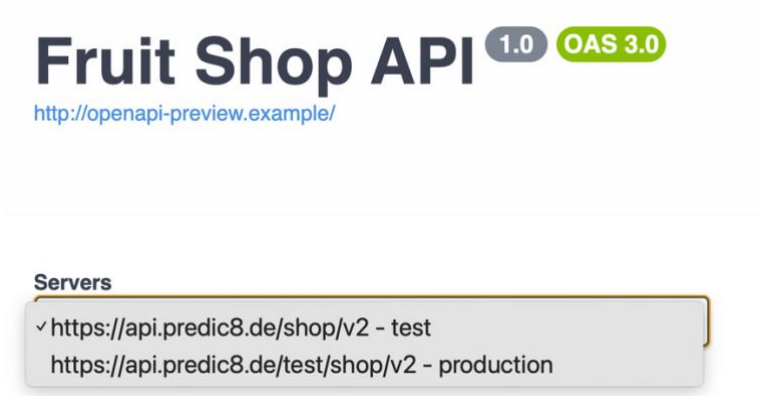


Image: Two rewritten server URLs in the Swagger UI

6.3 Message Validation with OpenAPI

OpenAPI describes an API in detail, which makes it perfect for message validation.

Many API Gateways and testing tools can verify that requests and responses comply with the rules defined in the OpenAPI document. The validation can happen in real-time, right as traffic flows through the gateway.

Gateways can validate every part of an HTTP exchange:

- **HTTP method and path**
  Including path variables and query parameters.
- **Header fields**
  Ensuring required headers are present and correctly formatted.
- **Payload format and structure**
  Verifying the `Content-Type` and validating JSON or XML bodies against the defined schema.
- **Security mechanisms**
  Checking that authentication and authorization requirements defined in the specification are met.
- **Status codes and error responses**
  Ensuring that response codes and error payloads match the documented contract.

By enforcing what's described in the OpenAPI document, gateways increase consistency across environments, help detect integration issues early, and reduce security risks. This makes APIs more predictable, secure, and easier to manage.

The listing below shows an error message returned by the API Gateway after a request failed OpenAPI validation:

```
HTTP/1.1 400 Bad Request
Content-Type: application/problem+json

{
  "title": "OpenAPI Message validation failed!",
  "type": "https://.../problems/validation",
  "validation": {
    "method": "POST",
    "path": "/users",
    "errors": {
      "REQUEST/BODY#/name": [
        {
          "message":
             "'name' exceeds max length of 20 characters!"
        }
      ]
    }
  }
}
```

🔎 **Looking for hands-on examples?**
If you want to see how these OpenAPI features work in practice, head over to Chapter 28 OpenAPI. There, you'll find real-world configurations and use cases like URL rewriting and request validation.

7 Message Transformation

APIs and JSON have made communication between applications much easier. Still, even if two systems both expose APIs, they often cannot talk to each other directly without some kind of translation layer in between. Even within the same business domain, message formats frequently differ in structure, field names, or semantics.

Integration platforms such as **Apache NiFi** or frameworks like **Apache Camel** are built to mediate between many different protocols and data formats. They can bridge HTTP, messaging systems, file transfers, and more. They are also well-suited for mediating between APIs.

API Gateways are more focused. With a few exceptions, they do not bridge HTTP APIs to completely different technologies such as message queues or FTP servers. However, when all participating systems communicate over HTTP, an API Gateway can act as a lightweight integration component between applications. How effective it is in that role depends largely on its message transformation capabilities.

Everything inside an HTTP message can be transformed. Gateways can rewrite paths, add or remove headers, and use transformation components to change the message body itself.

In this chapter, we explore how message transformation works in API Gateways, and which options are available to adapt requests and responses to different applications.

There is also a trade-off. Message transformation increases flexibility, but it also increases complexity. Every additional parsing and transformation step expands the attack surface of the gateway. At the end of this chapter, we look at typical attacks against message transformers and discuss how to mitigate them.

How Message Transformation Works

An HTTP message can contain a body with a specific media type, which is indicated by the `Content-Type` header. In the diagram below, the client sends XML messages, while the backend expects JSON. But the difference is not limited to the data format. The structure and field names may differ as well. For example, the client uses the field name `unit`, while the backend expects `currency`.

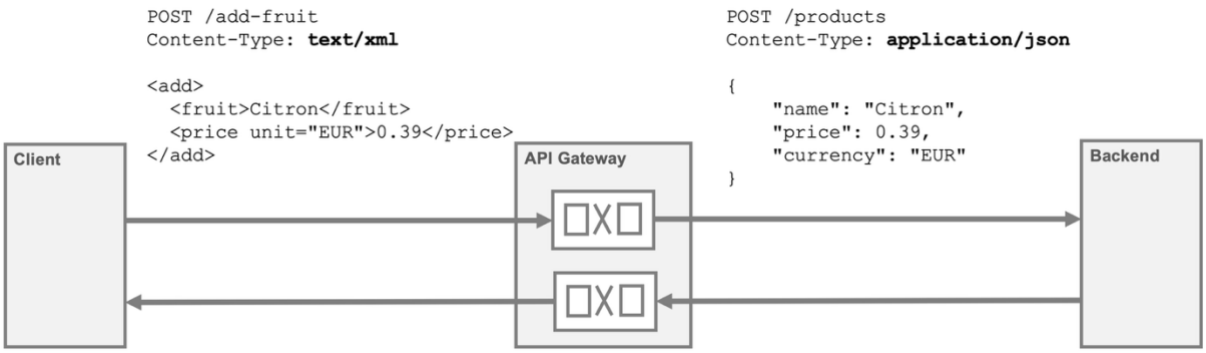


Image: API Gateway transforming XML messages into JSON

The API Gateway between the two parties acts as a mediator. It translates the incoming message into a format that is digestible by the receiver. This translation is performed by a transformer component that is either configured declaratively or implemented using a scripting or templating language.

On the way back from the backend to the client, the process works in the opposite direction. The response message must also be transformed so that the client receives data in the format and structure it can understand.

Recently, artificial intelligence has emerged as a powerful helper for creating message transformations. While it would be technically possible to perform transformations directly using an AI-powered transformation plugin at runtime, this approach would result in poor response times, higher costs, and unpredictable behavior. A more practical approach is to use AI to generate transformation templates, configurations, or scripts that run efficiently at the gateway. We'll look at how to use AI for this purpose at the end of this chapter.

7.1 Template Engines

Many API Gateways use templating engines for message transformation. Templates are *fill-in-the-blanks* documents whose placeholders are evaluated at runtime based on the HTTP message being processed.

AWS API Gateway, for example, provides templating based on the **Apache Velocity** template engine. The following template renders HTTP request headers as a Markdown table:

```
#set($headers = $input.params().header)

| Header | Value |
|--------|-------|
#foreach($name in $headers.keySet())
| `$name` | $headers.get($name) |
#end
```

The output of a template engine is usually text. By defining the appropriate formatting rules, it is possible to generate any text-based format, including JSON, XML, or even CSV. The next example renders HTTP request headers as a JSON document:

```
#set($headers = $input.params().header)
{
  "headers": {
#foreach($name in $headers.keySet())
    "$name": "$headers.get($name)"#if($foreach.hasNext),#end
#end
  }
}
```

HTTP header fields and other metadata are exposed to the template through predefined variables. Some API Gateways also allow access to the message body using expressions such as JSONPath or XPath, enabling even more complex transformations.

7.2 Scripting Transformations

Templates, stylesheets, and generic transformers are useful, but they have limits. Message transformation is often more involved than mapping field A to field B or converting JSON to XML. Typical challenges include:

- **Date format transformations**, for example converting a timestamp into an ISO date like 2026-01-21
- **Date calculations**, such as adding ten working days to a given date
- **Inserting** the current date or time
- **Lookups**, for example translating a code like BOL into *Bill of Lading*
- **Calculating** totals or derived values
- **Generating** sequence numbers

These kinds of tasks quickly go beyond simple declarative mappings. They require conditional logic, arithmetic operations, and sometimes access to external data.

Scripting languages are well-suited for implementing transformation logic directly inside the gateway. They allow you to express complex rules in a compact way, without setting up a compiler or a development environment.

That said, flexibility comes with responsibility. Script-based transformations should be kept readable and well-structured. Otherwise, the gateway can slowly turn into an application server in disguise, which is usually not the goal.

Let's take a look at some popular scripting languages used in API Gateways.

JavaScript

JSON is a subset of JavaScript, which makes it a natural fit for JSON processing. It offers native support for JSON structures and provides a powerful and expressive way to build message transformers.

The example below demonstrates several common transformation techniques: **date conversion**, **restructuring** the document, and **computing** the total value of order positions.

```
function convertDate(d) {
  return d.getFullYear() + "-" +
     ("0"+(d.getMonth()+1)).slice(-2) + "-" +
     ("0"+d.getDate()).slice(-2);
}
({
  id: json.id,
  date: convertDate(new Date(json.date)),
  client: json.customer,
  total: json.items.map(i => i.quantity * i.price)
      .reduce((a,b) => a+b),
  positions: json.items.map(i => ({
    pieces: i.quantity,
    price: i.price,
    article: i.description
  }))
})
```

This example also illustrates the use of functional programming constructs such as `map` and `reduce`, which are well-suited for transforming collections of data.

API Gateways with JavaScript support include **Gravitee API Management**, **Kong**, **Membrane**, **Tyk**, and **Zuplo**.

Groovy Scripts

Groovy is a powerful scripting language that runs on the Java platform. It provides concise and expressive features for reading, manipulating, and writing JSON and XML, coming very close to native support.

The same transformer shown in the JavaScript section above can be written in Groovy like this:

```
import java.text.*

def dfIn = new SimpleDateFormat("d MMM yyyy",Locale.US)
def dfOut = new SimpleDateFormat("yyyy-MM-dd")

[
  id: json.id,
  date: dfOut.format(dfIn.parse(json.date)),
  client: json.customer,
  total: json.items.collect { it.price * it.quantity }.sum(),
  positions: json.items.collect {
    [
      pieces: it.quantity,
      price: it.price,
      article: it.description
    ]
  }
]
```

This example highlights Groovy's clean syntax and its strong support for collection processing. Operations like `collect` and `sum` make transformations both readable and compact, especially when dealing with structured data.

We like Groovy's expressive style and often use it for message transformations in gateways and integration scenarios.

API Gateways with Groovy support include **Boomi API Management**, **Gravitee API Management**, and **Membrane**.

Lua Scripts

API Gateways such as **Apache APISIX**, **Kong**, and **IBM APIcast** are built on top of **nginx** and **OpenResty**. As a result, they support scripting with Lua.

The following example shows the same transformation as in the previous sections, implemented as a Lua script:

```
ngx.req.read_body()
local date = require "date"
local json = cjson.decode(ngx.req.get_body_data())

local total = 0
local positions = {}
for _, item in ipairs(json.items) do
  total = total + (item.price * item.quantity)
  table.insert(positions, {
    pieces = item.quantity,
    price = item.price,
    article = item.description
  })
end

local output = {
  id = json.id,
  date = date(json.date):fmt("%Y-%m-%d"),
  client = json.customer,
  total = total,
  positions = positions
}

ngx.req.set_body_data(cjson.encode(output))
```

7.3 Transformation between JSON and XML

XML and JSON share some similarities. Both are human-readable text formats, both support nested structures, and both are widely used for data exchange between systems. Still, converting between them is not as simple as it looks. A true one-to-one translation is often not possible.

JSON is not just XML with a nicer syntax. XML offers features that JSON does not have, such as namespaces, attributes, and mixed content. JSON, on the other hand, follows a simpler structural model. Object properties are unordered, and duplicate field names are not allowed. In XML, element order is significant, and the same element name can appear multiple times as sibling elements.

These differences matter. Any transformation between JSON and XML has to make assumptions about how certain structures are represented.

When designing an API that relies on format conversion, these structural differences should be considered early. Otherwise, small inconsistencies can turn into interoperability issues later.

7.3.1 XML to JSON

In this section, we look at XML peculiarities and how they can be mapped to JSON in a predictable and practical way.

XML Attributes

JSON has no equivalent to XML attributes. Consider the following XML document:

```
<person id="1721">
  <name>Jose</name>
</person>
```

One possible transformation into JSON looks like this:

```
{
  "person": {
    "name": "Jose",
    "_id": 1721
  }
}
```

The XML attribute `id` is mapped to a JSON object property named `_id`. To distinguish attributes from properties that originate from child elements, attribute names are often prefixed with an underscore or an at-sign.

Mixed Content

JSON is sometimes described as the successor of XML. In reality, they are fundamentally different. XML is a **markup language**, while JSON is a data **notation for structured objects**.

In typical API and integration scenarios, XML's markup capabilities are rarely used. Most XML messages represent structured data, not formatted text. However, XML supports markup, and this feature may appear in documents that need to be converted.

Consider the following example:

```
<log>Access from <ip>192.168.2.28</ip> blocked.</log>
```

Here, the `log` element contains text, and inside that text an `ip` element is embedded. This structure is known as **mixed content**. Character data and child elements are interleaved within the same parent element.

JSON has no concept of markup or mixed content. The closest representation is to model the content as an ordered array of text fragments and objects:

```
{
  "log": [
    {
      "_text": "Access from "
    },
    {
      "ip": "192.168.2.28"
    },
    {
      "_text": " blocked."
    }
  ]
}
```

The array is necessary to preserve the original order of text and embedded elements. Without it, the relative position of the IP address within the surrounding text would be lost.

Fortunately, mixed content is rarely used in typical data integration scenarios, where XML documents usually represent structured data rather than text markup.

Namespaces

Namespaces allow mixing the vocabulary of different **XML languages** within a single document. They make it possible to combine independently defined schemas without naming collisions.

At the same time, namespaces significantly increase XML's complexity and are a reason why XML is perceived as difficult to work with.

```
<ns1:person xmlns:ns1="uri:person" xmlns:ns2="uri:address">
  <ns2:address>Madrid</ns2:address>
</ns1:person>
```

JSON has no concept of namespaces. As a result, XML-to-JSON conversions must either drop namespace information or encode it in some ad hoc way. A common approach is to retain namespace prefixes as part of the property names:

```
{
  "ns1:person": {
    "xmlns:ns1": "uri:person",
    "xmlns:ns2": "uri:address",
    "ns2:address": "Madrid"
  }
}
```

Array Detection

One of the hardest problems in XML-to-JSON conversion is determining whether an element represents a single value or a list.

Consider the following XML document, which contains one animal:

```
<animals>
  <animal>dog</animal>
</animals>
```

A converter might reasonably transform this into:

```
{
  "animals": {
    "animal": "dog"
  }
}
```

In a later message, a second animal appears:

```
<animals>
  <animal>dog</animal>
  <animal>cat</animal>
</animals>
```

Now the same converter produces a list:

```
{
  "animals": {
    "animal": [
      "dog",
      "cat"
    ]
  }
}
```

From the perspective of a backend or client, this is a **breaking change**. The structure of the document has changed. Code written to handle a string cannot process an array.

Without explicit knowledge of the intended structure, provided by a schema or configuration, it is impossible for a converter to reliably decide whether an XML element should be mapped to a JSON array. This ambiguity is one of the fundamental impedance mismatches between XML and JSON and a frequent **source of integration errors**.

Element Order

In XML, the order of elements matters. After parsing, an application receives the elements in exactly the order in which they appear in the document. In many applications, this order is essential for correct processing.

The following document describes a workflow whose steps must be executed in sequence:

```
<workflow>
  <check/>
  <translate/>
  <compute/>
  <archive/>
</workflow>
```

After transforming this document using the `xml2json` component of our gateway, the result was:

```
{
  "workflow": {
    "compute": "",
    "archive": "",
    "check": "",
    "translate": ""
  }
}
```

The element order is lost. Executing the workflow in this order does not make any sense.

While a converter could attempt to preserve the original order when producing JSON output, JSON objects themselves are unordered by definition. There is no guarantee that the property order in the serialized document will be preserved when it is parsed by the receiving application.

To reliably preserve ordering, the workflow steps must be represented as an array, since array elements in JSON are ordered:

```
{
  "workflow": [
    "check",
    "translate",
    "compute",
    "archive"
  ]
}
```

This example highlights another fundamental mismatch between XML and JSON and shows

Missing Types

In XML documents, data does not carry type information. While types can be defined in external XML Schema definitions (XSD), most XML-to-JSON converters do not use schemas as guidance during transformation.

Instead, converters often try to **guess** data types based on the actual values they encounter. Consider the following XML document for an address in Milan:

```
<address>
    <city>Milano</city>
    <zip>20121</zip>
</address>
```

A converter may infer that the ZIP code is numeric and produce the following JSON. Notice that `20121` isn't wrapped in quotes.

```
"address": {
  "zip": 20121,
  "city": "Milano"
}
```

Now look at an address from Tokyo, where ZIP codes contain a dash:

```
<address>
    <city>Tokyo</city>
    <zip>100-0005</zip>
</address>
```

This time, the same converter will likely infer that the ZIP code is a string:

```
  "address": {
    "zip": "100-0005",
    "city": "Tokyo"
  }
```

The result is an inconsistent data type for the same field. At the receiving end, this inconsistency can easily break the contract.

Without explicit type information, XML-to-JSON conversion is always a guess. You can never be sure whether a value that looks like a number in one request might suddenly appear as a string in the next. One possible workaround is to convert all values to strings, but that leads to poor API design.

7.3.2 JSON to XML

The transformation from JSON to XML also comes with challenges, although typically fewer than in the opposite direction.

JSON follows a simpler structural model and does not support features such as attributes or namespaces. This reduces ambiguity during conversion. Still, there are structural differences that must be addressed, especially around root elements, array handling, and element ordering.

In the following sections, we examine the typical issues that arise when converting JSON to XML and how they can be handled in a consistent way.

JSON Documents are not Trees

At first glance, JSON documents look like trees. However, JSON itself does not define a tree model in the same way XML does. XML always has a single root element, and a clearly defined parent child hierarchy. A JSON object, by contrast, is defined as a set of name value pairs and arrays. Object properties are unordered by specification.

Another important difference is that a JSON document does not have to be an object. Any valid JSON value can be a complete document. All the following samples are valid JSON documents:

Sample Document	Description
`{ "foo": 1 }`	Object
`[1,2,3]`	Array
`42`	Number
`true`	Boolean
`null`	No value
`"foo"`	String

Table: JSON data types

In integration scenarios, most JSON messages are objects. Arrays are also common, and in rare cases even a single number or string may be exchanged. These values can still be mapped to XML by wrapping them inside a root element.

For the following examples, we focus on JSON objects. Consider this simple document:

```
{
  "name": "Elena"
}
```

A JSON-to-XML converter could transform this into:

```
<name>Elena</name>
```

Now consider a slightly extended document:

```
{
  "name": "Elena",
  "city": "Buenos Aires"
}
```

Here, `name` and `city` are sibling properties at the same level. XML, however, requires a single root element. To produce valid XML, both elements must be enclosed within a common parent:

```
<root>
  <city>Buenos Aires</city>
  <name>Elena</name>
</root>
```

Although both XML outputs represent similar data, their structure differs. The shape of the resulting XML depends on whether the JSON input contains one property or multiple properties.

This variability can break interfaces that rely on a stable XML structure. For that reason, many JSON to XML converters always wrap the generated content in a predefined root element, regardless of the number of properties in the original JSON document.

Property Order

Properties in JSON are unordered by definition. Even if many implementations preserve insertion order, the JSON specification does not guarantee it.

XML, in contrast, is order-sensitive. In many XML schemas, elements are defined as sequences with a strict order. Changing the position of an element can lead to validation errors or cause problems in systems that rely on a fixed structure.

When converting JSON to XML, this difference becomes visible. If the converter processes properties in arbitrary order, the generated XML may not match the expected element sequence. For loosely defined XML formats this may not matter, but for schema-driven interfaces it can be critical.

To avoid issues, converters may enforce a predefined element order or apply a post-processing step, for example using a script or XSLT, to rearrange elements into the required sequence.

Null Handling

JSON has a `null` data type to represent unknown or missing values. XML itself does not have a built-in concept of `null`.

In the following JSON document, the field `a` contains an empty string, while the value of `b` is unknown:

```
{
  "a": "",
  "b": null
}
```

A JSON-to-XML transformation might render both values as empty elements:

```
<root>
  <a></a>
  <b></b>
</root>
```

7.4 Combining generic with concrete Transformations

The shortcomings of generic transformations can be compensated for by adding an additional transformation step that corrects or refines the result.

A common pattern is to first apply a generic transformation, for example JSON to XML, and then follow it with a more specific transformation that adjusts structure, naming, or ordering. This keeps the initial step simple while still allowing precise control where needed.

In a transformer pipeline, each step builds on the output of the previous one. A generic converter can handle the heavy lifting, such as structural format changes, while a second step enforces schema requirements, renames elements, reorders fields, or enriches the message with computed values.

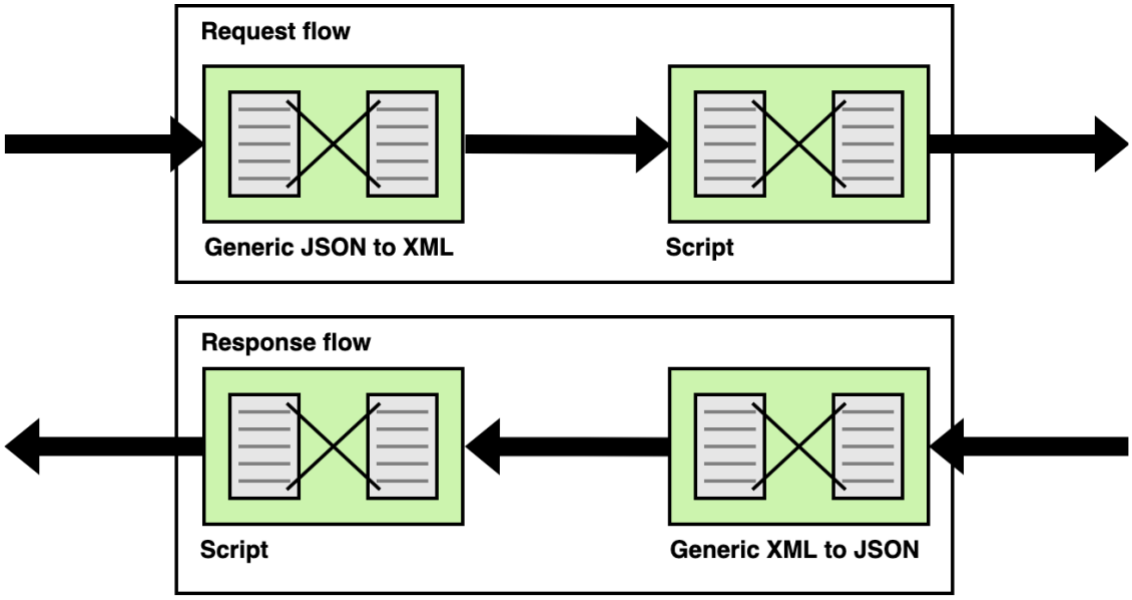


Image: Generic transformation followed by specific post-processing step (Tool: draw.io)

This layered approach keeps configurations manageable. Instead of replacing a generic converter with a complete custom implementation, you only refine the parts that require special handling.

Best Practice: Start generic, then specialize
Start with a generic conversion, such as XML to JSON. If the result already covers about 80% of what you need, add a second step that makes the remaining adjustments. Typical fixes include renaming fields, converting data types, or reordering elements

7.5 XSLT Transformation

XML messages can be transformed using templates or scripting languages. If you already have experience with XSLT or have existing stylesheets, XSLT can be an additional option for message transformation.

We have migrated several legacy applications that relied heavily on XSLT stylesheets. In many cases, these stylesheets were copied from closed-source solutions and reused almost unchanged with a modern open source gateway.

XSLT, short for *eXtensible Stylesheet Language Transformations*, is a language for transforming XML documents into other text-based formats. The output can be XML, HTML, CSV, or virtually any other text format. The input, however, must always be an XML document.

XSLT is not easy to learn. It is not imperative like most programming languages and does not use commands in the traditional sense. Instead, it is rule-based and declarative. In this respect, it resembles logic programming languages such as Prolog and shares characteristics with functional programming, including the absence of side effects. Because XSLT requires a

significant learning effort and is considered a legacy technology, you should **think twice before starting to learn XSLT** today.

If that does not discourage you, XSLT offers several advantages for the transformation of XML messages:

- **Standardized**: XSLT is a W3C standard, and stylesheets can be reused across different tools and platforms without modification.
- **XML-native**: XSLT fully supports XML features such as namespaces.
- **Rule-based model**: Pattern matching enables concise and powerful transformation logic.

One major benefit is the ability to reuse existing **stylesheets from legacy systems**, which can significantly reduce migration effort.

Only a subset of API Gateways, primarily those based on the Java platform, provide built-in XSLT support. The following gateways offer XSLT capabilities (non-exhaustive list):

- Azure API Management
- Axway API Gateway
- Google Apigee
- Gravitee.io API Management
- IBM API Connect / DataPower
- Membrane API Gateway
- MuleSoft Anypoint Platform
- Oracle API Gateway
- WSO2

Most gateways support XSLT versions 1.0 and 2.0. IBM API Connect also supports XSLT 3.0, which can be used to transform JSON input in addition to XML.

XML to JSON Transformation with XSLT

In this section, we'll look at how **XSLT** can be used for message transformation in an API Gateway. The use case is a common one: integrating a backend API that returns XML with a client that expects a specific JSON format.

Assume the backend responds with the following XML document containing a list of books:

```
<books>
  <book id="1">
    <title>The Mythical Man-Month</title>
    <author>Frederick P. Brooks Jr.</author>
    <year>1975</year>
  </book>
  <book id="2">
    <title>Code Complete</title>
    <author>Steve McConnell</author>
    <year>1993</year>
  </book>
</books>
```

The API client, however, expects the data in this JSON format:

```
{
  "books" : [ {
    "title" : "The Mythical Man-Month",
    "author" : "Frederick P. Brooks Jr.",
    "year" : "1975"
  }, {
    "title" : "Code Complete",
    "author" : "Steve McConnell",
    "year" : "1993"
  } ]
}
```

To perform the transformation, the following stylesheet can be used:

```
<xsl:stylesheet version="1.0"
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text" encoding="UTF-8"/>

  <xsl:template match="books">
    { "books": [
      <xsl:apply-templates/>
    ] }
  </xsl:template>

  <xsl:template match="book">
    {
      <xsl:apply-templates select="*"/>
    }
    <xsl:if test="position()!=last()">,</xsl:if>
  </xsl:template>

  <xsl:template match="*[text() and not(*)]">
    "<xsl:value-of select="name()"/>":
    "<xsl:value-of select="normalize-space(.)"/>"
    <xsl:if test="position()!=last()">,</xsl:if>
  </xsl:template>

</xsl:stylesheet>
```

I struggled whether to include XSLT in this book at all. It's undeniably a true legacy technology. However, in practice we still encounter many existing systems that rely on large collections of XSLT stylesheets. In recent projects, we've reused and migrated quite a few of them. Since the stylesheet is included here, it's worth explaining how it works for anyone who wants to adapt it for their own project.

The stylesheet consists of three templates, each responsible for a specific part of the transformation. The XSLT processor selects templates based on the current XML node being processed.

Because `<books>` is the root element of the XML document, the following template is applied first:

```
<xsl:template match="books">
  { "books": [
    <xsl:apply-templates/>
  ] }
</xsl:template>
```

This template creates the outer structure of the JSON document. The `<xsl:apply-templates/>` instruction tells the processor to continue processing the child elements of `<books>`.

Next, the processor applies the following template once for each `<book>` element:

```
<xsl:template match="book">
  {
    <xsl:apply-templates select="*"/>
  }
  <xsl:if test="position()!=last()">,</xsl:if>
</xsl:template>
```

This template wraps each book in a JSON object. The `position()!=last()` check ensures that commas are placed correctly between book entries, but not after the last one.

Finally, the third template renders the individual properties of a book:

```
<xsl:template match="*[text() and not(*)]">
  "<xsl:value-of select="name()"/>":
  "<xsl:value-of select="normalize-space(.)"/>"
  <xsl:if test="position()!=last()">,</xsl:if>
</xsl:template>
```

This template matches elements that contain only text and no child elements, such as `<title>`, `<author>`, and `<year>`. It outputs them as JSON key-value pairs and again takes care of comma placement. With this generic template, large numbers of fields (elements) can be converted in a generic matter. For fields that are not converted as intended like numbers that do not require quotes or fields that need to be renamed you can just add additional rules like this one to make the needed adjustments:

```
<xsl:template match="id | price | count | age">
  "<xsl:value-of select="name()"/>":
  <xsl:value-of select="normalize-space(.)"/>
  <xsl:if test="position()!=last()">,</xsl:if>
</xsl:template>
```

XSLT may be a legacy technology, but it is also a very powerful one. It can handle complex XML transformations efficiently and at scale.

To balance out the **"old-school XSLT"** content, the last section in this chapter explains using AI to generate transformation templates. In practice, tools like ChatGPT or Claude can be quite helpful when designing or debugging stylesheets.

7.6 Security Considerations for Templating

Templates make message transformations easy to implement. That simplicity, however, can lead to careless usage. Consider the following template:

```
{
  "city": "{{params.city[0]}}"
}
```

At first glance, this looks harmless. It simply inserts a query parameter such as:

```
?city=Vienna
```

into a JSON document:

```
{ "city": "Vienna" }
```

Now consider a carefully handcrafted request from a hacker:

```
?city=Roma%22,%22admin%22:true,%22x%22:%22d"
```

The encoded value `%22` represents a double quote. After decoding, the template produces:

```
{
  "city": "Roma",
  "admin":true,
  "x":"d"
}
```

The attacker closes the original string with the first injected quote and then adds additional JSON properties. The extra `"x":"d` fragment balances the trailing quote from the template so that the final JSON is syntactically valid.

This is a classic **injection attack**. The template assumes that the inserted value is safe and properly formatted. But **unescaped user input can break the surrounding structure** and **inject additional fields**.

To prevent this, values must be escaped according to the target format. In JSON, this means escaping quotation marks and other characters with special meaning.

With proper escaping, the template could look like this:

```
"city": {{toJSON(params.city[0])}}
```

Because the `toJSON()` function renders a valid JSON string **literal**, additional quotes in the template are no longer needed. The malicious input is then transformed into:

```
"Roma\",\"admin\":true,\"x\":\"d"
```

When this escaped value is inserted, the JSON structure remains intact. The `city` field contains one long string with JSON gibberish, but no additional properties are injected. The result is syntactically valid and safe to process.

Security Hint: Never trust external input
The principle "never trust external input" applies to templating as well. Treat all user-supplied data as untrusted and take appropriate measures, such as escaping and validation, before including it in messages.

Escaping

Escaping can be implemented through helper functions or built directly into the template engine. A JSON aware template plugin can automatically escape inserted values before rendering them.

Escaping must match the target format. JSON, XML, and URLs all require different characters and sequences to be escaped.

Consider this input:

```
Bob "The <Boss>" & Sons
```

Depending on the context, it must be escaped differently:

Context	Escaped Sample
JSON	`Bob \"The <Boss>\" & Sons`
XML	`Bob "The <Boss>" & Sons`
URL	`Bob%20%22The%20%3CBoss%3E%22%20%26%20Sons`

Table: Escaping for JSON, XML and URLs

The correct escaping depends on the downstream consumer and how it interprets the content. Missing escaping, or escaping for the wrong context, can break parsing or open injection vulnerabilities.

Using the wrong format in a template, or forgetting to specify the format at all, often leads to incorrect or missing escaping.

Even better than relying on string-based templates and manual escaping is constructing structured data using JSON or XML libraries instead of concatenating strings. Scripts can

help implement this approach, but they introduce a different attack surface and must be secured carefully.

Security Hint: How to escape
Escape for the final output context, using the correct encoding, at the last possible moment.

Combine Transformation with Validation

Injection attacks often change the structure of a message or make it invalid for the expected data format. A validator that rejects unexpected or additional fields can help detect such manipulations early.

Validating input types and enforcing length constraints adds another layer of protection. Some injection payloads simply do not fit into short strings or numeric fields. While validation does not prevent injection completely, it raises the bar and makes exploitation harder.

Validation can be performed at the backend, at the gateway, or at both layers. Performing it at the gateway has the advantage that malformed or malicious input can be rejected before it reaches internal systems.

OpenAPI specifications and **WSDL** descriptions already define message structures and data types. They can therefore serve as a basis for request and response validation. Alternatively, **JSON** and **XML Schemas** can be used to validate payloads against precise structural rules.

Transformation and validation work best together. Transformation adapts messages to the required format, while validation ensures that only valid and expected data is processed further.

Resources

Alice & Bob learn Application Security
Tanya Janca @shehackspurple, Wiley,

RFC 8259 (JSON) section on strings and escaping
https://www.rfc-editor.org/rfc/rfc8259#section-8.1

Extensible Markup Language (XML) 1.1 (Second Edition), 2.4 Character Data and Markup
https://www.w3.org/TR/2006/REC-xml11-20060816/#syntax

7.7 Best Practice

Here are a few practical guidelines that help with complex transformations and legacy integrations.

Start simple, then refine
Begin with a minimal, working configuration that solves one small part of the overall task. Then add transformation steps, security checks, and orchestration gradually and iteratively. This keeps complexity under control, makes errors easier to trace, and gives you a chance to learn from each step before moving on.

Prefer structured processing over string concatenation
When generating or transforming content, use dedicated JSON or XML libraries instead of building documents through string concatenation or regular expressions.

These libraries understand the target format, handle data types correctly, and apply the appropriate encoding rules automatically. This reduces the risk of injection vulnerabilities, malformed output, and subtle parsing errors.

Separate wiring from logic
Keep orchestration and routing in the gateway configuration, but move complex transformation logic into external scripts or dedicated files. This keeps API definitions concise and makes the flow easier to understand.

Externalizing transformation logic also enables isolated testing without running the full gateway, which improves maintainability and supports automated tests.

An API Gateway is not the right place to host application logic. Do not misuse transformations and integrations to implement business rules. Keep domain logic in backend systems or microservices, and let the gateway focus on mediation, security, and protocol adaptation.

Build transformation libraries
When mapping large interfaces with many endpoints, it pays off to extract reusable transformation logic into libraries. In real-world APIs, the same data structures and fields often appear again and again.

Instead of duplicating mapping logic across multiple configurations, encapsulate it in reusable scripts or templates. These libraries can be implemented using the gateway's scripting capabilities and referenced from different APIs.

This reduces duplication, keeps configurations consistent, and makes changes easier and safer.

Generic transformations
If many fields are copied or transformed in the same way from input to output, generic transformers can significantly reduce effort. Generic JSON to XML and XML to JSON converters, as well as reusable scripts, are often sufficient for large parts of an interface.

In many cases, most of a transformation can be handled generically. Only the fields and structures that deviate from the norm require individual rules. This keeps configurations shorter and easier to maintain.

Scripting languages such as JavaScript, Lua, or Groovy are suitable for implementing generic transformers for JSON and other formats. They allow you to traverse structures dynamically and apply consistent rules across large payloads.

When working with XML, XSLT can be a powerful option. Its template matching model enables compact and expressive rule-based transformations.

7.8 Creating Transformation Templates with AI

Writing transformation templates, scripts, or stylesheets for API Gateways is usually not hard, but it takes time. That makes it a good use case for AI assistance. With a short prompt and a representative sample, you can often get a working template, or at least a solid starting point, in a few iterations.

The following sample prompt asks an AI model to create a transformation template for the Tyk Gateway. The template converts a JSON document into CSV.

Prompt:

```
Create a Tyk response transformation template using Go
templates that converts documents with the following JSON
structure into CSV:

{
  "animals": [
    {"name": "Skye", "species": "dog", "legs": 4},
    {"name": "Molly", "species": "cat", "legs": 4}
  ]
}

Requirements:
- Output CSV with headers
- Comma-separated CSV format
- Include complete API definition snippet
```

Privacy Hint
When using a public AI model, use sample data and avoid sharing sensitive or personal information.

8 API Orchestration

A single API can aggregate or coordinate multiple underlying APIs. This is called API orchestration. Instead of working in isolation, the orchestrating API **coordinates** multiple internal APIs to fulfill a request, acting like a conductor leading an ensemble of backend services.

Orchestration is especially common in **service-oriented architectures (SOA)** and **microservices** environments, where functionality is split into small, focused services. Because each of these services is highly specialized, useful business processes often require combining several of them.

Take the example below: an order API depends on customer, article, and price APIs. Rather than duplicating functionality, the order API serves as a composite that orchestrates responses from all three.

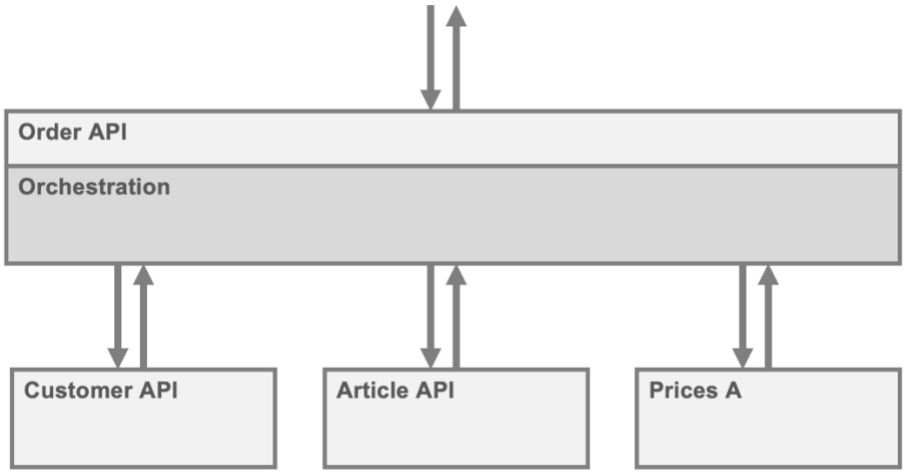


Image: Orchestration of fine-grained APIs

Orchestration can improve **structure**, **encapsulation**, and **reusability**. But it also comes with trade-offs: the orchestrator becomes a **point of dependency**, and if not carefully managed, this can increase **coupling** and **fragility** in the system.

Some API Gateways come with features such as:

- Calling multiple backend services per request
- Merging responses
- Executing conditional logic

Alternatively, API orchestration can be implemented outside the gateway using tools such as:

- **Workflow engines,** e.g., Camunda, Zeebe
- **Integration frameworks,** e.g., Apache Camel, Spring Integration
- **Low-code platforms,** e.g., AWS Step Functions, Azure Logic Apps

These tools can manage more complex flows, long-running processes, or asynchronous tasks that go beyond what an API Gateway is designed for.

Typically, in API orchestration, the response from one API call becomes the input for the next. In practice, this is rarely straightforward. The involved APIs often speak **different languages**: one might return JSON, while another expect XML, and their field structures may vary widely.

This means orchestration often requires:

- **Data format conversion** (e.g., JSON to XML)
- **Field mapping** (e.g., renaming, flattening, or restructuring data)

When using an API Gateway for orchestration, make sure it supports robust querying and transformation features such as JSONPath, XPath, and templating capabilities. Without these, you'll likely run into limitations when integrating APIs with mismatched expectations.

Sidenote: Orchestration vs. Choreography
In orchestration, a central API Gateway or workflow engine explicitly controls the interactions between services. In choreography, services react to events and coordinate themselves without a central controller. Gateways typically implement orchestration, not choreography.

9 Security

API Gateways sit between clients and backend services, placing them in a natural point to enforce and enhance API security. As the central entry point for API traffic, a gateway can address a wide range of concerns, starting with transport-level protections and extending through authentication**,** authorization, and application-level defense**.**

By offloading these responsibilities from backend services, the gateway helps **standardize** and **centralize** security policies across APIs, reducing complexity and improving the organization's ability to respond to evolving threats.

An API Gateway can:

- **Handle transport encryption** using TLS
- **Authenticate and authorize** client requests
- **Validate input and output** messages against business rules or schemas
- **Protect against message-based attacks**, including those targeting XML, JSON, or GraphQL
- **Log and audit** traffic, users, and sensitive events

The following sections introduce key security concepts such as **integrity**, **confidentiality**, **authentication**, and **authorization**. These concepts form the foundation for understanding how secure API communication works in practice. Together, they define how data is protected, how identities are verified, and how access decisions are enforced in distributed systems.

9.1 Integrity

Digital messages are easy to change but hard to trust. Without protection, a receiver has no way to tell whether a document was altered. That's where integrity checks come in. A signature or cryptographic hash can ensure that the content hasn't been tampered with.

Integrity is a fundamental requirement in API security. One of the most common use cases is **token validation**. Tokens such as JSON Web Tokens (JWTs) are signed so that the recipient (usually the API Gateway or backend) can verify that they haven't been changed and that they were indeed issued by a trusted authority.

9.2 Confidentiality

In 2017, a routing incident caused internet traffic for Google, Facebook, and Microsoft to be redirected through Russia. The event underscored a critical risk: traffic can be silently detoured without the sender or receiver knowing.

On the open internet, the route a message takes is unpredictable. It might pass through dozens of routers and networks, including infrastructure owned by third parties or even potential attackers. Without proper protection, anyone along that route could intercept and read the data.

This is where **encryption** comes into play. Encryption ensures that only authorized parties can read and understand the protected content.

To ensure **confidentiality with APIs**, two main strategies are used:

1. **Establishing a secure communication channel**
   Confidentiality is achieved by creating a secure, encrypted point-to-point connection between the client and server. This is typically done using **Transport Layer Security (TLS)**, the successor to SSL. TLS encrypts the entire communication channel, including paths, headers, and payloads. But only while the data is in transit. Once it reaches the endpoint and is decrypted, the protection ends.
2. **Encrypting the message**
   This secures the message itself, so it stays protected even after transmission. This is useful when messages pass through intermediaries or when they are stored for later processing. This approach is used for **data at rest** or in very sensitive scenarios.

9.3 Authentication & Authorization

In API security, it's important to distinguish between two foundational concepts: **authentication** and **authorization**.

Authentication is the process of verifying **who someone is**. For example, when you're asked to show an ID card, the goal is to confirm that you are the person you claim to be. In the world of APIs, this often means logging in with credentials, using an API key, or presenting a client certificate.

Once a subject is authenticated, we know their identity, but we do not yet know what it is allowed to do.

Authorization determines **what actions the authenticated subject is permitted to perform**. For instance, an API might verify that a user is authenticated as "Tobias" but only allow users with the "admin" role to perform a DELETE request on a certain endpoint.

Authentication = Who are you?

Authorization = What are you allowed to do?

Authorization in APIs

In the context of APIs, authorization typically governs actions like:

- Can a user perform a POST?
- Is this token allowed to access `/admin`?
- Does this client have permission to read a certain resource like `/contracts/334`?

Gateways, API backends, or security policies often enforce these rules by checking roles, scopes, or claims within a token.

Resources

'Suspicious' BGP event routed big traffic sites through Russia, The Register 2017/12/13
https://www.theregister.com/2017/12/13/suspicious_bgp_event_routed_big_traffic_sites_through_russia/

10 Transport Layer Security (TLS/SSL)

API security relies heavily on **Transport Layer Security (TLS)** to securely transmit data and tokens between systems. In this chapter, we explain why transport-level security is essential, clarify the difference between SSL and TLS, and show how TLS is used by API Gateways.

Man-in-the-Middle Attacks

API communication involves two parties: the client and the server. To protect integrity and confidentiality during communication, it's essential that **no one in between** can **read, manipulate, or redirect the data.**

TLS protects against **man-in-the-middle (MitM)** attacks, where an attacker silently intercepts or modifies messages in transit. Without TLS, any router, proxy, or network node along the path could potentially tamper with the communication.

TLS is the de facto standard for securing internet traffic. In fact, higher-level security mechanisms like **OAuth2** or **OpenID Connect** assume that the transport layer is already secure. In other words: **TLS is a foundation, not an option.**

SSL and TLS

You might still hear the term **SSL (Secure Sockets Layer)**, but it's outdated. SSL was the original protocol developed by Netscape in the 1990s to secure internet communications. However, due to serious vulnerabilities, it has long been deprecated. Its successor, **Transport Layer Security (TLS)**, is now the modern standard for encrypted connections.

Despite this, many people still refer to TLS as SSL out of habit.

Transport Layer Security is known for providing confidentiality by encrypting data in transit. However, TLS can **also provide authentication**. It uses certificates and certificate authorities (CAs) to verify the identities of communicating parties, ensuring that both the server and optionally the client are who they claim to be. This dual function of TLS helps prevent man-in-the-middle attacks and unauthorized access.

API Gateways and TLS Connections

API Gateways sit between clients and backend services and play an active role in securing communication.

In most setups, **two separate TLS connections** are established:

- One between the **client** and the **gateway**
- One between the **gateway** and the **backend service**

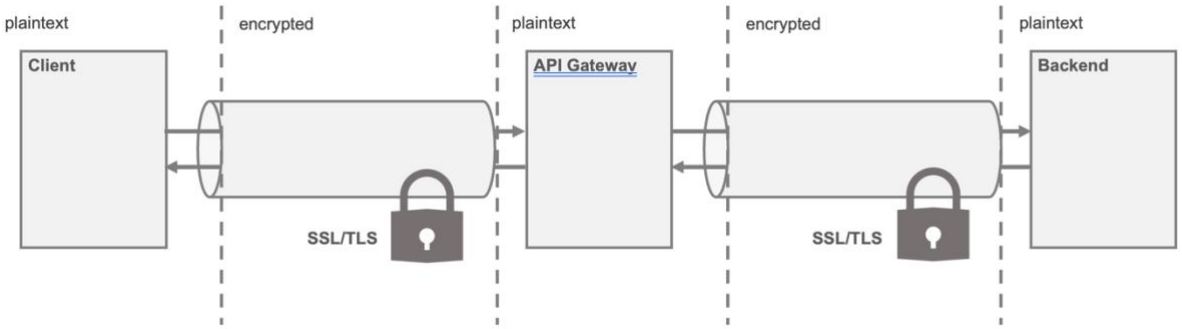


Image: TLS communication between client, gateway, and backend

This design allows the gateway to **terminate TLS**, inspect traffic, apply security policies, validate tokens, and perform transformations before forwarding requests.

💡**Sidenote**: **TLS Passthrough**
Some gateways also support **TLS passthrough**, where the TLS session is not terminated at the gateway. Instead, the encrypted connection is forwarded directly from the client to the backend.
While this approach offers end-to-end encryption, it comes with trade-offs: the gateway cannot inspect, transform, or route traffic based on content. As a result, this mode is less common and used only in specific scenarios where full privacy is prioritized over control.

11 Content Protection

A well-worn adage is: **"never trust user input."** In the world of APIs, that becomes: "never trust the request." This holds especially true for structured data formats like **XML**, **JSON,** and **GraphQL**. Their flexibility and expressiveness also make them attractive targets. Attackers can exploit specific characteristics of these formats to overload systems, bypass validation, or trigger unintended behavior.

The importance of content protection is underscored by the number of known parser vulnerabilities. According to cve.org, there are nearly **2,000 documented vulnerabilities** related to JSON, and **more than three times as many for XML**.

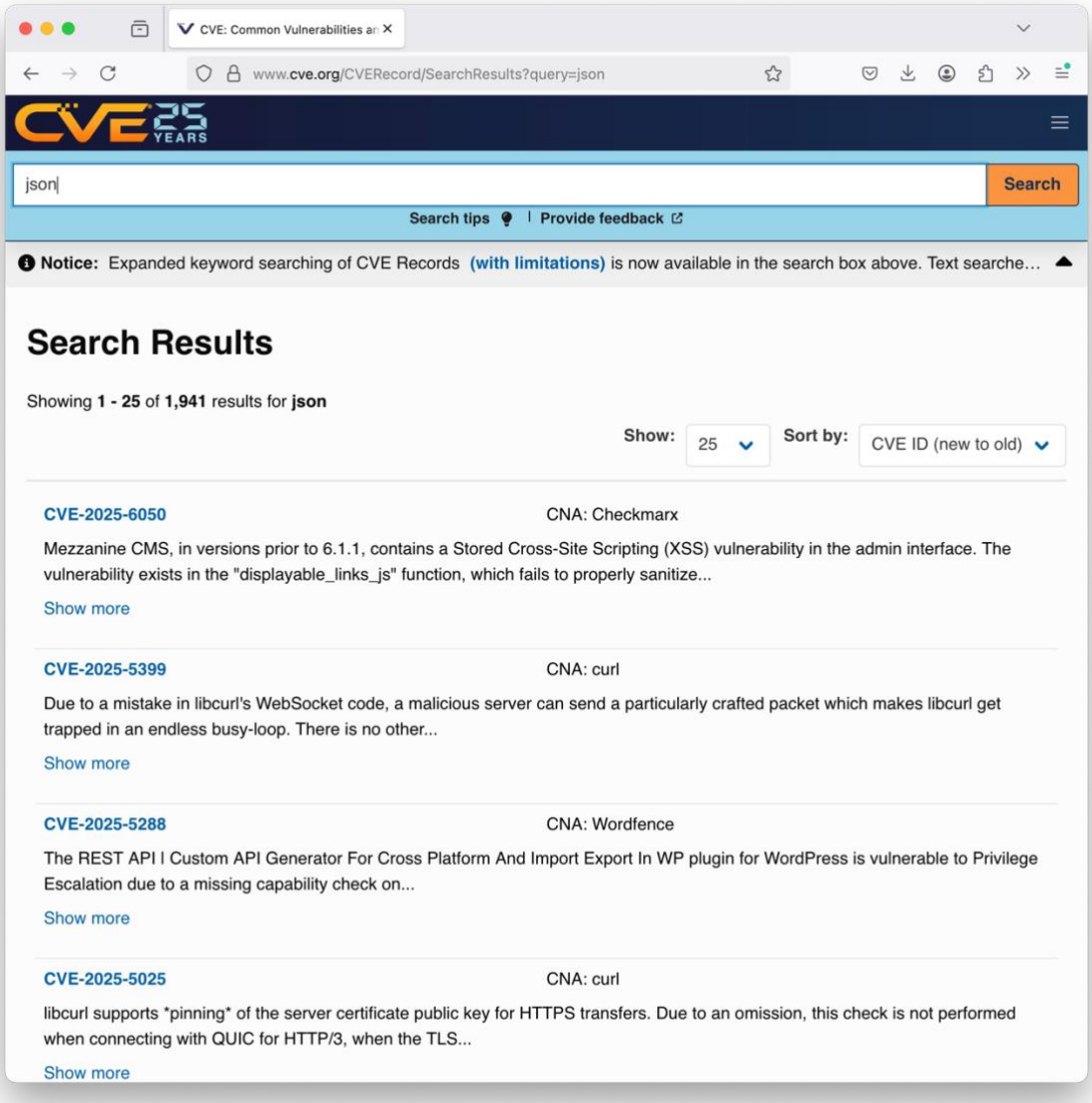


Image: Search result for JSON vulnerabilities on cve.org

In the following sections, we'll take a closer look at some common attacks targeting JSON, XML, and GraphQL.

11.1 JSON Attacks

JSON-related attacks exploit the way JSON payloads are parsed and processed. They can take advantage of parser inconsistencies, implementation flaws, or overwhelm systems with excessive or maliciously structured data. Below are some of the most common techniques:

Duplicate Fields
JSON objects are not supposed to contain duplicate keys. But if they do, parsers handle them differently. Some take the first occurrence; others take the last. Consider this example:

```
{
  "article": "Smartphone",
  "quantity": 1,
  "price": 998,
  "price": 10
}
```

If validation checks only the first `price` field, but the business logic uses the second one, an attacker could exploit this inconsistency to manipulate prices or bypass validation.

Excessively Large Arrays or Strings
Oversized arrays or unusually long strings can consume significant memory and processing time. JSON, for example, does not impose limits on string length or array size. So a value might be just a few bytes or several gigabytes. Without safeguards, this kind of input can overwhelm the receiver, degrade performance, or even cause denial of service (DoS) conditions through memory exhaustion or processing timeouts.

Deeply Nested Structures
Nesting JSON structures deeply may look innocent but can be devastating. Even small payloads with excessive depth can slow down or crash parsers by exhausting stack space or consuming excessive memory.

Each additional level of nesting increases the processing effort. As the snippet below illustrates, adding another level requires only a few extra characters in the input, yet it increases structural depth significantly.

```
{
  "a": {
    "b": {
      "c": {
        "d": {
          "e": {
            "f": {
            }
          }
        }
      }
    }
  }
}
```

11.2 XML Attacks

XML offers a rich set of features, but that richness also creates risk. Its extensibility and flexibility make it prone to several attack vectors, especially when parsers are overly permissive or insecurely configured.

One of the most notorious threats comes from **Document Type Definitions (DTDs)**, which allow the definition of entities that can reference external resources. If DTD processing is enabled, attackers may exploit **XML External Entities (XXE)** to access sensitive files or trigger unexpected network requests.

External Entity Injection

An attacker might send a request containing an XML payload like the following:

```
POST http://localhost:2000
Content-Type: application/xml

<!DOCTYPE foo [
   <!ELEMENT foo ANY >
   <!ENTITY e SYSTEM "file:///etc/passwd" >]>
<foo>&e;</foo>
```

XXE attacks are considered **so dangerous** that we couldn't include this example as plain text. Security tools and virus scanners would flag or block the book, and most likely your corporate firewall would even prevent you from downloading your copy of the book. The safest way to include it was as an image. That's why this listing looks different from the others.

If the backend XML parser accepts this input and **external entities are enabled** (which they often are by default in older systems), the parser will replace the `&e;` entity with the content of the referenced file `/etc/passwd`. The result: sensitive data is silently leaked.

Beyond leaking files, XXE attacks can also be used for the disclosure of confidential data, network port scanning, SSRF (Server-Side Request Forgery), or denial of service.

💡 **Sidenote: Why is XML still a risk?**
While many systems have moved on to JSON, XML is still in use, especially in enterprise and legacy systems. That makes XML security just as relevant as ever.

11.3 GraphQL Exploits

GraphQL offers powerful capabilities for building flexible APIs, but its dynamic nature also introduces unique security challenges. Without proper safeguards, GraphQL endpoints can be abused, leading to excessive backend load, **denial of service (DoS)** conditions, or bypassing rate limits.

Recursive Queries

One of the most common attack patterns is a **recursive query**, where a client uses circular references to force the server into excessive work. Here's an example:

```
{
  products {
    vendor {
      products {
        vendor {
          products {
            vendor {
              products {
                vendor {
                  products {
                    vendor {
                      products {
                        name
                      }
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}
```

At first glance, the query seems innocent, only about 400 bytes. But it triggers a chain of lookups across the `product` and `vendor` relationship that can rapidly expand the response size. Even on a small demo API with fewer than 20 products, this query can generate more than **3 megabytes** of response data.

This kind of pattern is a textbook example of a denial of service risk: the server does a lot of work, while the client does very little. With even deeper recursion, the backend can quickly become overloaded.

Introspection Abuse

GraphQL provides powerful introspection features that allow clients to query metadata about the GraphQL schema. While useful during development, these features can become a **valuable reconnaissance** tool for attackers. By leveraging introspection, a hacker can:

- Discover all available types, fields, queries, and mutations
- Map out relationships between objects
- Identify undocumented or internal API functions
- Construct highly targeted attacks

The query below retrieves all types defined in the GraphQL schema of an API which includes hidden and internal fields.

```
{
  __schema {
    types {
      name
      fields {
        name
      }
    }
  }
}
```

Batching

In GraphQL, multiple queries or mutations (i.e., remote function calls) can be grouped into a single message. Consider the following document:

```
{
  q1: products(id: "2") { name }
  q2: products(id: "3") { name }
  q3: products(id: "7") { name }
  q4: products(id: "8") { name }
  q5: products(id: "10") { name }
  q6: products(id: "11") { name }
  q7: products(id: "13") { name }
}
```

Although it's just one message, the server will treat this as seven separate queries. Because GraphQL queries can be written very compactly, a small payload of just a few kilobytes may contain hundreds of queries or mutations.

Rate-limiting plugins at the API Gateway, which are often unaware of GraphQL internals, typically count the entire batch as a single call. This can be exploited by attackers to:

- Generate heavy workloads with minimal effort to cause denial of service attacks
- Bypass rate limits and perform brute-force attacks

11.4 Applying Content Protection

To mitigate content-based attacks, you can add **content protection rules** directly into your API Gateway configuration. Once enabled, the gateway inspects incoming payloads before they reach backend services.

If a message matches known attack patterns or violates configured constraints, the gateway can take one of two actions:

- **Block** the message entirely, responding with an appropriate 4XX HTTP error code
- **Sanitize** the content by removing or replacing harmful elements (e.g., prototype fields, or XML DTDs)

This layer of protection is especially valuable when working with **legacy systems**, which may lack input validation or use outdated parsers with known vulnerabilities.

Gateway Support for Content Protection

Different gateway products offer different levels of support for content inspection and validation. Here's a quick comparison:

Gateway	JSON Protection	XML Protection	GraphQL Protection
Apigee	✓	✓	✓ passthrough only
Envoy	✓	☐	☐
Gravitee	✓	✓	✓ (in beta June 2025)
Kong	✓	☐ Not natively supported	☐ Via community plugins
Tyk	✓	☐ Limited XML support	☐ GraphQL introspection filtering in Enterprise
AWS API Gateway	✓ by JSON Schema validation	☐ Limited XML support	✓ GraphQL support via AppSync

Table: Support for content protection in different gateways

Properly configured content protection ensures that APIs do not become a backdoor for parser bugs, protocol tricks, or malformed payloads. For high-risk formats like XML or GraphQL, **limit what the gateway will accept** before passing it on.

Vendor-Specific Media Types

Vendor-specific media types introduce another detail. A type such as `application/vnd.predic8.product+json` uses the `+json` suffix to indicate that the payload is JSON-encoded, even though the full media type is custom. Security mechanisms must check not only for `application/json`, but also for any media type ending in `+json`.

Resources

XML External Entity (XXE) Processing
https://owasp.org/www-community/vulnerabilities/XML_External_Entity_(XXE)_Processing

11.5 Content-Type Confusion

When content protection is enabled, API Gateways typically apply validations such as blocking DTDs or detecting recursive structures based on the declared `Content-Type`. However, these checks are only executed if the `Content-Type` header is correctly set.

Take the following example:

```
POST /api/user HTTP/1.1
Content-Type: text/plain

{
  "role": "customer",
  "role": "admin",
  "name": "Tobias"
}
```

The payload is clearly JSON. But since the `Content-Type` is declared as `text/plain`, the gateway treats it as plain text, where virtually any byte sequence is considered valid, and skips all JSON-specific inspections. This opens a loophole that attackers can use to bypass payload validation and security filters.

One might argue that the backend should reject the request based on the incorrect `Content-Type` header. But in practice, many backend implementations either ignore the content type altogether or assume the payload is JSON by default. This behavior makes them vulnerable to so-called **content-type confusion** attacks.

How to Guard Against Content-Type Confusion?

Ensure the content type of incoming requests matches the expected format. This can be enforced using a policy in the API Gateway, or more effectively by validating requests against an OpenAPI description.

In OpenAPI, every request body is tied to a declared content type. The validator checks whether the `Content-Type` header in the request matches what is defined in the API spec. If not, the request is rejected.

In the snippet below, the request body is explicitly declared to use the content type `application/json`.

```
requestBody:
  content:
    application/json:
      schema:
        $ref: '#/components/schemas/Product'
```

A **web application firewall (WAF)** positioned in front of the API Gateway adds extra protection. By inspecting parts of the payload, it may detect a mismatch between the declared media type and the actual message structure.

💡 **Tip: Use content protection in combination with a strict content type check.**
OpenAPI validation ensures that the `Content-Type` header in incoming requests matches the expected value. This prevents attackers from bypassing security filters by mislabeling payload formats.

12 Injection Attacks

In an injection attack, malicious input is inserted into a system in such a way that it becomes part of a command or query. As a result, the system may execute unintended instructions instead of treating the input strictly as data.

Common types include SQL injection, code injection, XPath injection, LDAP injection, and XML external entity (XXE) attacks. The impact ranges from leaking or modifying sensitive data to bypassing authentication, escalating privileges, or even full system compromise.

Injection Attacks on APIs

APIs can unintentionally open channels for attackers to inject malicious code into backend systems. For instance, consider the following HTTP request where an injection is part of the query string:

```
GET /rest/products/search?q=apples')) UNION SELECT id, email,
password, '4', '5', '6', '7', '8', '9' FROM USERS--
```

In this example, the search string is prematurely terminated by the `'` character, and an SQL injection follows, designed to extract sensitive user information from the database. This attack succeeds if the backend service dynamically builds an SQL query without proper parameterization (e.g., using prepared statements).

Beyond query parameters, virtually any part of an HTTP request, such as path parameters, headers, payloads, or even JSON Web Tokens (JWTs), can be a vehicle for injection attacks.

12.1 Input Validation with OpenAPI

Input validation reduces the risk of injection attacks. It can block malicious input outright or at least make exploitation harder. Take the search parameter `q` from the previous example. Suppose it is defined in OpenAPI like this:

```
parameters:
  - in: query
    name: q
    schema:
      type: string
      maxLength: 20
      pattern: '[A-Z0-9]*'
```

Only uppercase letters and digits would be accepted, and the input must not exceed 20 characters. That already prevents many injection attempts. The SQL injection from the earlier example, for instance, used 85 characters, far beyond the 20-character limit enforced here.

To make APIs more secure, every parameter and field should be defined as precisely as possible. Use:

- **Length limits** (`minLength`, `maxLength`)
- **Enumerations** (`enum`)
- **Regular expressions** (`pattern`)
- **String formats** (`email`, `uuid`, `date-time`, etc.)

These constraints aren't just helpful for documentation. They actively guard against attacks.

To enforce these practices, static analysis tools like **Spectral** can lint OpenAPI definitions. There is a ruleset available that includes checks for the **OWASP API Security Top 10**, and you can add custom rules too.

Here's an example Spectral rule that ensures all string-based query parameters include a `maxLength`.

```
rules:
  query-parameters-should-have-maxLength:
    description: Query parameters should define maxLength
    message: '"{{property}}" is missing maxLength.'
    given: '$.paths[*][*].parameters[?(@.in=="query")]'
    then:
      field: schema.maxLength
      function: defined
```

Validation is your first line of defense, but it should be combined with other techniques like proper escaping, parameterized database queries, and contextual output encoding.

12.2 Why Validation Alone Isn't Enough

Validation contributes to an effective defense against injection attacks. However, validation alone is not sufficient. Consider this email address, for example:

```
"'OR 1=1--"@predic8.de
```

At first glance, it does not appear to be valid. However, it is syntactically correct according to the email specification, and many validators will accept it without complaint.

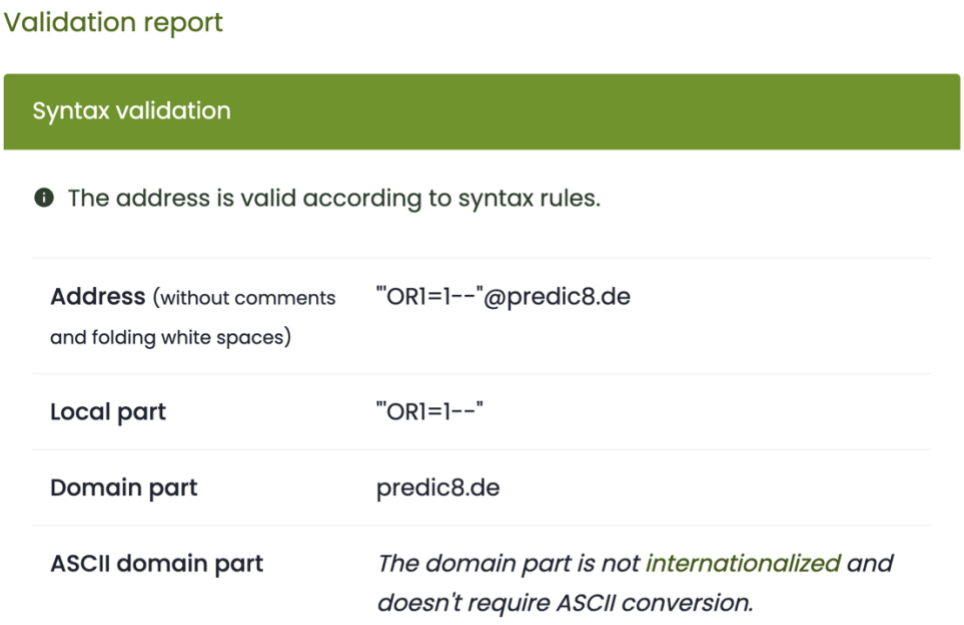


Image: Excerpt of validation result at `https://verifalia.com/validate-email`

Although the string passes email validation, it contains a classic SQL injection fragment:

```
' OR 1=1--
```

If such a value is inserted into a dynamically constructed SQL query, it can alter the logic of the statement. In an authentication context, this may allow an attacker to **bypass password checks**. In poorly designed systems, the query could return the first user in the database, which is often an administrator account. In that case, the attacker has not only bypassed authentication but has also gained access to the most privileged account in the system.

Validation at the gateway or service layer is valuable. It raises the bar and filters out many malformed or obviously malicious inputs. But it does not replace secure coding practices.

Effective protection against SQL injection requires parameterized queries, also known as prepared statements. Instead of concatenating user input into SQL strings, the database driver binds values safely and treats them strictly as data.

In environments constrained by legacy systems or tight budgets, external protection mechanisms such as web application firewalls or API Gateways can add defensive layers. However, they should complement secure backend coding, not replace it.

12.3 Effective Injection Protection

As discussed earlier, secure backend coding practices and a sound architecture are the most effective defenses against injection attacks. Still, additional protective layers can be placed in front of backend services to reduce risk.

Injection Signature Detection

Injection scanners analyze incoming requests for patterns that indicate potential attacks. These patterns may include SQL fragments, comment markers, or suspicious operator combinations.

Signature-based detection can be effective, but it has limitations. It is not feasible to block every character or sequence that might be used in an injection attack. For example, blocking the `'` character would reject legitimate input such as the name `O'Reilly`. Likewise, blocking keywords such as `union`, `drop`, or even `or`, which are common in SQL injections, could also interfere with valid user input and business data.

If everything that could possibly resemble an injection attack is blocked, normal production traffic may no longer function correctly. Injection protection is therefore always a compromise. It must be carefully tuned and combined with additional safeguards.

Curated rule sets, such as those provided by **Snort** or similar security communities, help reduce this risk. They are continuously updated and refined to reflect evolving attack techniques.

Machine Learning and Artificial Intelligence (AI)

AI-driven tools can detect anomalies and suspicious patterns more flexibly than fixed, signature-based rules. Instead of relying only on predefined attack strings, they analyze behavioral patterns, statistical deviations, or contextual features in requests.

There are gateway plugins available that incorporate machine learning techniques for advanced injection detection. These systems can identify attack variants by recognizing unusual input or traffic patterns.

12.4 API Gateway vs. Web Application Firewall (WAF)

Both API Gateways and web application firewalls can help protect against injection attacks. However, their responsibilities differ.

- WAFs typically focus on generic injection detection. They are often placed in front of the API Gateway and inspect incoming traffic using signature-based or anomaly-based techniques.
- API Gateways are well suited for API-specific validation. Using JSON Schema, XML Schema (XSD), or OpenAPI definitions, they can validate requests precisely per endpoint and per contract.

Some WAF products also support JSON and XML schema validation. In that case, every API change requires updating the schema in both components, the WAF and the API Gateway. This increases coordination effort and the risk of configuration drift.

WAF products typically excel at signature-based detection and threat intelligence integration. Many API Gateways, on the other hand, focus more on authentication and contract validation, and therefore usually do not provide the same depth of injection protection as dedicated WAFs.

This separation also allows independent update cycles. The gateway configuration changes whenever the API contract changes. The WAF rules are only updated when signature sets evolve.

Resources

OWASP Top 10 API Security Risks – 2023
https://owasp.org/API-Security/editions/2023/en/0x11-t10/

SPECTRAL, JSON/YAML Linter with Custom Rulesets Documentation
https://docs.stoplight.io/docs/spectral/674b27b261c3c-overview

Spectral OWASP API Security
https://github.com/stoplightio/spectral-owasp-ruleset

SNORT, Open Source Intrusion Prevention System (IPS)
https://www.snort.org/

13 Message Validation

Message validation is a core defense mechanism in API security. Because APIs often operate at system or organizational boundaries, they should inspect incoming data carefully before passing it to backend services.

Effective validation starts with a clear definition of what a valid request looks like. These expectations could originate from the business domain. Business stakeholders can define which values, formats, and constraints are meaningful. Once specified, these rules can be formalized as schemas and implemented in security infrastructure such as an API Gateway.

API Gateways can enforce structural definitions based on specifications such as JSON Schema, XML Schema, OpenAPI, or WSDL. With precise schemas in place, invalid input can be rejected at the edge, before it reaches internal systems.

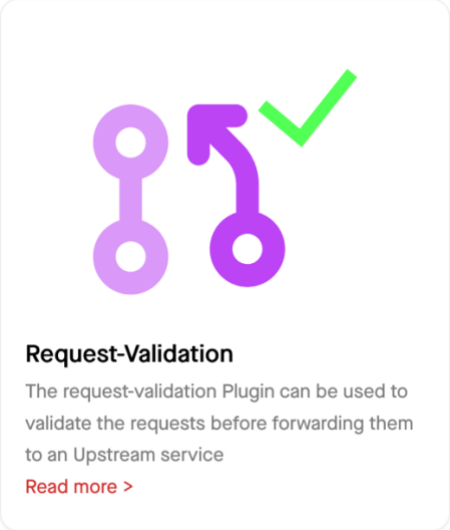


Image: The request validation plugin for the APISIX gateway

13.1 Response Validation

Input validation receives most of the attention. Response validation is often overlooked. It is tempting to assume that, because the backend generates the output, it must be safe. That assumption can lead to unintended **information disclosure**.

Why Is Response Validation Important?

Consider the following error response:

```
{
"status": 400,
"trace":
      "org.springframework.http.converter.HttpMessageNotReadableException:
JSON parse error: Unexpected character ('\"' (code 34)): was expecting
      comma to separate Object entries; nested exception is
      com.fasterxml.jackson.core.JsonParseException: Unexpected character
      ('\"' (code 34)): was expecting comma to separate Object entries\n at
      [Source:
      (org.springframework.util.StreamUtils$NonClosingInputStream); line:
      3, column: 4]
at
      org.springframework.http.converter.json.MessageConverter.readJavaType
      (MessageConverter.java:391)
at
      org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(Appl
      icationFilterChain.java:227)
... 51 more",
}
```

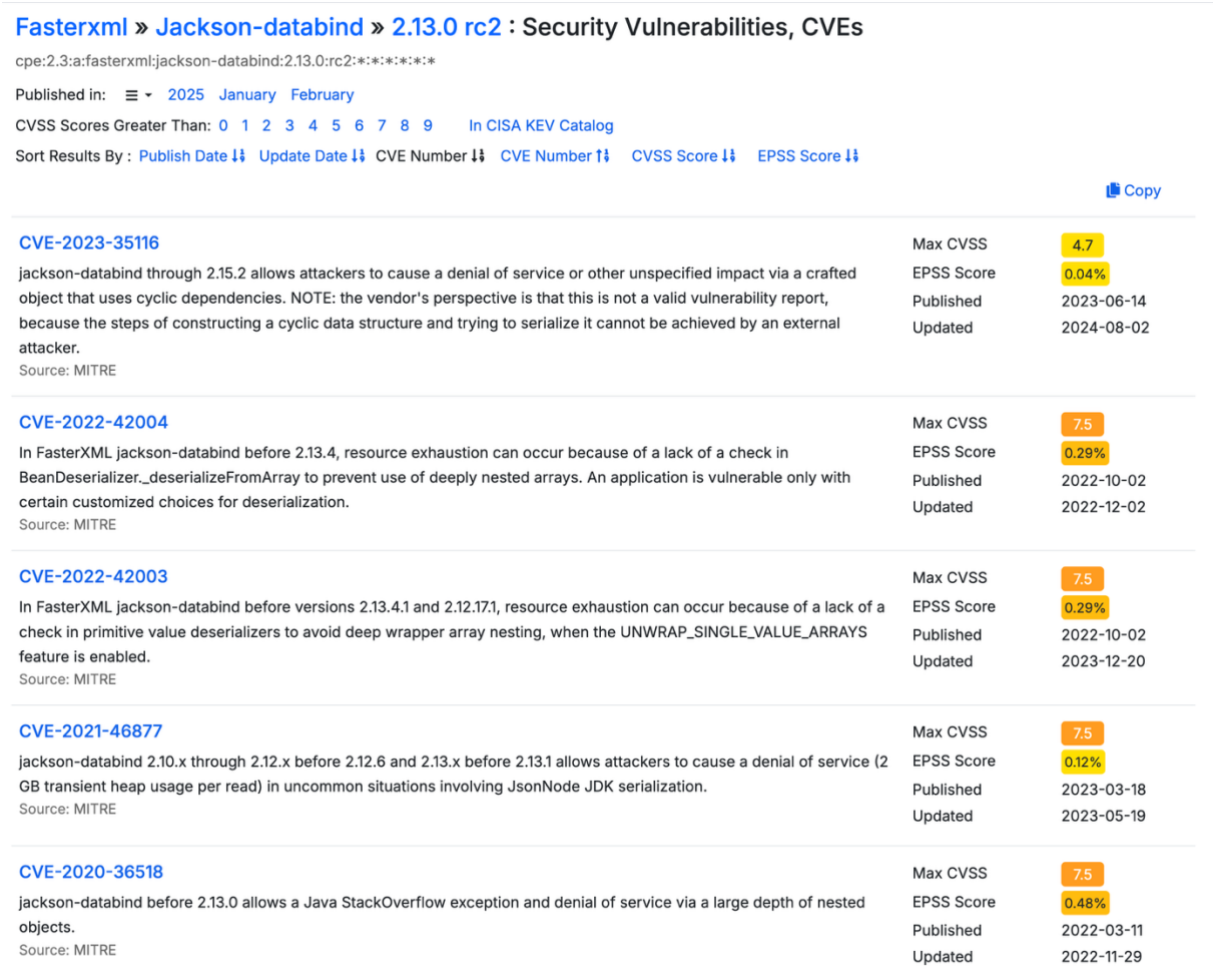
This stack trace exposes detailed information about the internal structure of the server application. An attacker performing reconnaissance can extract valuable clues about the technology stack and libraries in use.

It is obvious that the application is implemented in Java. The class names and line numbers help narrow down framework versions. When such a stack trace is analyzed with automated tools, it is often possible to infer likely versions of Spring, Jackson, or the underlying application server.

Pasting this stack trace into an AI tool revealed that:

- the application uses **Spring Framework 5.3.x**
- **Jackson 2.13.x** is used for JSON deserialization
- the system runs on **Tomcat or Catalina 8.5.x or 9.0.x**

With this information, an attacker can search the **Common Vulnerabilities and Exposures (CVE) Database** for known vulnerabilities.



Fasterxml » Jackson-databind » 2.13.0 rc2 : Security Vulnerabilities, CVEs

cpe:2.3:a:fasterxml:jackson-databind:2.13.0:rc2:*:*:*:*:*:*

Published in: 2025 January February

CVSS Scores Greater Than: 0 1 2 3 4 5 6 7 8 9 In CISA KEV Catalog

Sort Results By : Publish Date Update Date CVE Number CVE Number CVSS Score EPSS Score

Copy

CVE-2023-35116 jackson-databind through 2.15.2 allows attackers to cause a denial of service or other unspecified impact via a crafted object that uses cyclic dependencies. NOTE: the vendor's perspective is that this is not a valid vulnerability report, because the steps of constructing a cyclic data structure and trying to serialize it cannot be achieved by an external attacker. Source: MITRE	Max CVSS EPSS Score Published Updated	4.7 0.04% 2023-06-14 2024-08-02
CVE-2022-42004 In FasterXML jackson-databind before 2.13.4, resource exhaustion can occur because of a lack of a check in BeanDeserializer._deserializeFromArray to prevent use of deeply nested arrays. An application is vulnerable only with certain customized choices for deserialization. Source: MITRE	Max CVSS EPSS Score Published Updated	7.5 0.29% 2022-10-02 2022-12-02
CVE-2022-42003 In FasterXML jackson-databind before versions 2.13.4.1 and 2.12.17.1, resource exhaustion can occur because of a lack of a check in primitive value deserializers to avoid deep wrapper array nesting, when the UNWRAP_SINGLE_VALUE_ARRAYS feature is enabled. Source: MITRE	Max CVSS EPSS Score Published Updated	7.5 0.29% 2022-10-02 2023-12-20
CVE-2021-46877 jackson-databind 2.10.x through 2.12.x before 2.12.6 and 2.13.x before 2.13.1 allows attackers to cause a denial of service (2 GB transient heap usage per read) in uncommon situations involving JsonNode JDK serialization. Source: MITRE	Max CVSS EPSS Score Published Updated	7.5 0.12% 2023-03-18 2023-05-19
CVE-2020-36518 jackson-databind before 2.13.0 allows a Java StackOverflow exception and denial of service via a large depth of nested objects. Source: MITRE	Max CVSS EPSS Score Published Updated	7.5 0.48% 2022-03-11 2022-11-29

Image: Known vulnerabilities in an outdated Jackson library

Once vulnerable versions are identified, the attacker can study published exploits and craft targeted requests. Verbose error responses significantly reduce the effort required for this type of targeted attack.

Response validation helps to prevent this. Detailed stack traces and internal exception messages should be logged internally, not exposed to external clients.

Security Hint: Block stack traces in responses
Always block stack traces and verbose error messages from backend systems. These can reveal sensitive internal details, such as class names, frameworks, and line numbers, that attackers can use to identify vulnerabilities. Use API Gateways or a middleware to catch and rewrite such responses before they reach the client.

Resources

Keyword search for CVE Records @MITRE Corporation.
https://www.cve.org/

Response Validation and Error Messages

A message validator can only validate responses that are defined in a schema or contract. When a backend returns an unexpected error message, for example a raw database exception, the response will fail validation.

As described earlier, this is often desirable. The unexpected error is blocked (5) and does not reach the client. Instead, the client receives a validation error generated by the gateway.

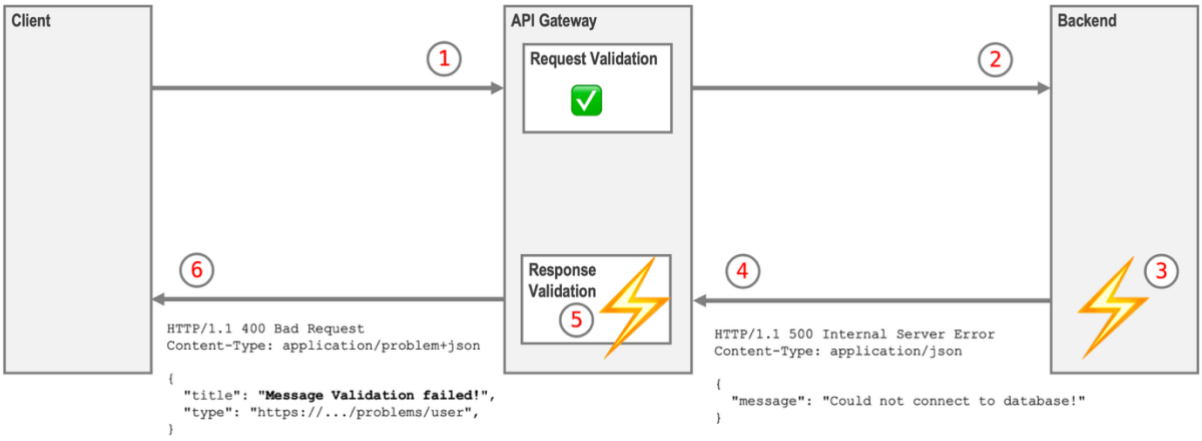


Image: Failed response validation after an error at the backend

However, this can lead to confusion. The client developer may assume that the validation error is caused by the request, while the real issue is an unmodeled backend response (6).

Some gateways allow validation to be disabled for error responses. While this avoids validation failures, internal error details may pass through the gateway and be exposed to the client.

Ideally, backend services should return only error responses that are defined in the schema used by the gateway. In practice, especially with legacy or off-the-shelf systems, error formats are often inconsistent and not described in OpenAPI or JSON Schema.

One solution is to introduce an additional transformation step at the gateway (4). The gateway can normalize backend error responses into a defined and documented error format before validation is applied.

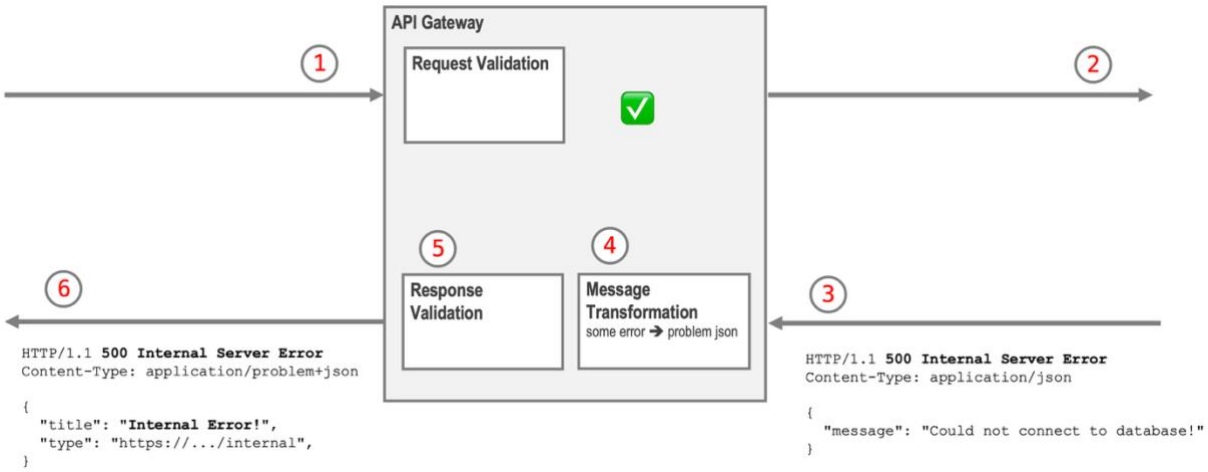


Image: Transforming custom error messages before validation

For response validation to work reliably, error messages must be explicitly defined in a JSON Schema, OpenAPI specification, or XML Schema. The next section explains how to model such error responses properly.

13.2 Describing Error Messages

To validate responses properly, schema definitions are needed not only for successful messages but also for error responses.

Problem Details (RFC 9457)

RFC 9457, **Problem Details for HTTP APIs**, defines a standardized format for error messages in HTTP-based APIs. The format is designed to be both human-readable and machine-readable, making it easier for clients to process errors programmatically.

A typical example looks like this:

```
HTTP/1.1 404 Not Found
Content-Type: application/problem+json

{
  "title" : "Product 7 not found",
  "type" : "https://membrane-api.io/problems/user",
  "uri" : "/products/7"
}
```

According to RFC 9457, fields such as `type` and `title` are typically present. The `uri` field shown here is a custom field. That flexibility is one of the strengths of **Problem Details**. You start with a standardized structure and extend it to fit your domain.

Validating Problem Details with OpenAPI

OpenAPI allows you to define structured error responses in addition to successful ones. This enables an API Gateway to validate error messages just like success payloads and prevents unintended data exposure.

Below is an example OpenAPI snippet for an endpoint that returns either a successful response or a 404 error:

```
paths:
  /products:
    post:
      responses:
        '200':
          description: Ok
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Product"
        '404':
          description: Not Found
          content:
            application/problem+json:
              schema:
                $ref: '#/components/schemas/Problem'
```

Notice that the 404 response uses the `application/problem+json` media type and references a shared `Problem` schema. The schema might look like this:

```
components:
  schemas:
    Problem:
      type: object
      additionalProperties: false
      properties:
        type:
          type: string
          description: URI reference identifying the problem
        title:
          type: string
          description: Human-readable summary
        uri:
          type: string
          description: Request URI
```

The `Problem` schema defines the required `type` and `title` fields from RFC 9457 and includes a custom `uri` field. The line `additionalProperties: false` is important. It ensures that unexpected fields in error messages are rejected during validation. This prevents internal details such as stack traces or debug information from leaking to clients.

💡**Security Tip**: **Error Validation with OpenAPI**
Define error schemas with the same precision as success responses. OpenAPI validation does more than improve documentation. It can enforce clean, consistent, and secure API behavior at runtime.

Status Code Wildcards

Describing every possible HTTP error code in an OpenAPI spec can get tedious. Fortunately, OpenAPI supports wildcards that keep things concise and maintainable.

Here's a sample:

```
paths:
  /products:
    post:
      responses:
        '200':
          description: Ok
          ...
        '404':
          description: Not found
          ...
        '4XX':
          description: Bad Request
          ...
        '5XX':
          description: Server Error
          ...
        'default':
          description: default
          ...
```

Using wildcards:

- `'4XX'` matches any client-side error (400–499)
- `'5XX'` matches server-side errors (500–599)
- `'default'` catches any other status codes not explicitly defined

This makes API definitions compact while still covering a wide range of potential responses. It's especially useful when combined with response schemas like Problem JSON.

Resources

Problem Details for HTTP APIs
https://datatracker.ietf.org/doc/html/rfc9457

13.3 JSON Message Validation

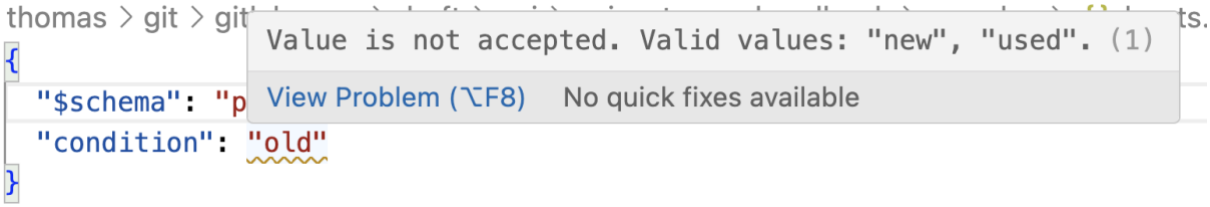
JSON Schema is a widely adopted standard for describing the structure of JSON data. It allows you to define data types, required fields, formats, and constraints such as minimum and maximum values, regular expression patterns, or enumerated values.

Let's look at a simple example:

```
{
  "type": "object",
  "properties": {
    "condition": {
      "type": "string",
      "enum": [ "new", "used" ]
    }
  }
}
```

This schema defines an object with a single field called `condition`. The value must be a string and can only be either `new` or `used`.

Now consider a request payload where `condition` is set to `old`. Validation fails because `old` is not part of the allowed enumeration. The gateway can reject this request, preventing invalid or unexpected data from reaching the backend.



By enforcing strict schemas, gateways can block malformed or invalid input early in the processing chain.

Many API Gateways support JSON Schema validation directly or indirectly through OpenAPI.

Security Hint: Unknown properties
By default, JSON Schema allows unknown object properties. If you declare `additionalProperties: false`, only explicitly defined fields are permitted.

Without this constraint, clients can inject additional fields that may be ignored by validation but still processed by backend logic. This can open the door to subtle exploits.

On the response side, unexpected properties may expose internal domain fields or technical error details that were never intended for external clients.

Resources

JSON Schema Reference
https://json-schema.org/understanding-json-schema/reference

Michael Droettboom, et al, Space Telescope Institute, Understanding JSON Schema
https://json-schema.org/UnderstandingJSONSchema.pdf

13.4 XML Schema Validation

Just as JSON uses JSON Schema, XML relies on XML Schema for validation. An XSD defines the expected structure of an XML document, including its elements, attributes, data types, cardinality, and element order.

An API Gateway can use XSD validation to verify that an incoming or outgoing XML message conforms to the defined schema. If the structure, required elements, or data types do not match, the message is rejected before it reaches the backend.

This type of validation is particularly important in legacy environments and B2B integrations, where XML is still widely used and strict contract adherence is often mandatory.

13.5 OpenAPI Validation

In OpenAPI, message bodies are described using **JSON Schema**, even if the OpenAPI document itself is written in YAML. When an API Gateway validates a message against an OpenAPI definition, it performs JSON Schema validation under the hood.

The error message below shows a validation failure caused by a negative `price` value that violates the embedded JSON Schema:

```
{
  "title": "OpenAPI message validation failed",
  "type": "https://membrane-api.io/problems/user/validation",
  "validation": {
    "errors": {
      "REQUEST/BODY#/price": [
        {
          "message": "-2.79 is smaller than the minimum of 0"
        }
      ]
    }
  }
}
```

The error response itself conforms to the **Problem Details** specification, which keeps error handling consistent and machine-readable.

OpenAPI validation goes beyond the message body. It can also validate:

- HTTP method
- Request path and path parameters
- Query parameters
- Header fields
- Status code and content types

With JSON Schema alone, you cannot validate all of that.

You can try this curl command:

```
curl "https://api.predic8.de/shop/v2/products?limit=No"
```

The request fails OpenAPI validation and produces this error response:

```
{
  "title" : "OpenAPI message validation failed",
  "type" : "https://membrane-api.io/problems/user/validation",
  "validation" : {
    "errors" : {
      "REQUEST/QUERY_PARAMETER/limit" : [ {
        "message" : "No is not an integer."
      } ]
    }
  }
}
```

One more advantage of OpenAPI over standalone JSON Schema validation is its integration with HTTP semantics. The gateway can automatically determine which schema applies based on the path and method of the request. With plain JSON Schema, you would need to manage separate schemas manually or define generic ones that provide less precise validation.

If an OpenAPI description is available, it can be used directly for validation. Typically, you configure the gateway with a URL pointing to the specification or reference a local copy. Once enabled, validation is enforced automatically for matching endpoints.

Some API Gateways support OpenAPI validation natively. Others convert OpenAPI documents into internal validation policies or configuration artifacts.

🔒 **Security Hint: OpenAPI validation enforces the full contract**
OpenAPI validation is particularly powerful because it enforces the full API contract. It validates not only structure but also the interaction model defined by paths, methods, and response codes.

📌 **Note**: Part II explores how to configure OpenAPI-based message validation in practice.

14 API Keys

With API keys you can secure APIs without the need for complex security protocols. They allow control over who can access an API, enforce basic authorization rules, and track usage for monitoring or billing. This straightforward approach makes API keys appealing when you need rapid integration and minimal overhead, even if API keys do not provide the fine-grained control or dynamic capabilities of more advanced mechanisms like JSON Web Tokens (JWT).

What are API Keys?

There is no universal standard for API keys. Each product handles them in slightly different ways. However, one common characteristic remains: an API key is a secret value sent together with a request, which is used to authenticate the caller.

In the images below, the client's request contains an `X-Api-Key` header. The API or gateway verifies the key by matching it against a list of known keys. If the key is found, the request is authenticated; if not, the request is denied.

Image: API Key HTTP header

Sometimes, one API key is shared among multiple clients to simplify setup and operation. However, this approach is only appropriate for low-risk, uncritical applications. If the key is compromised, the damage can be huge because all clients using the shared key would be affected. Therefore, it is best practice to assign each client its own unique key, ensuring that a breach only impacts a single client rather than the entire system.

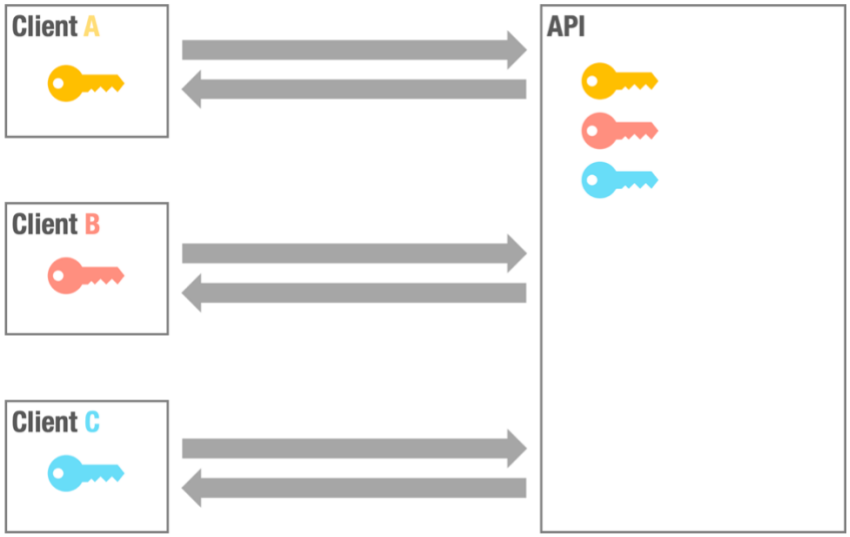


Image: Multiple clients with individual API keys

API keys are **stateless**. No security context is maintained between calls. Each request is authenticated independently by verifying the key included in that specific request.

This means that **every single call must include the key.** There's no session, cookie, or persistent connection involved.

At first glance, this may seem inefficient. Why include and validate the key every time? Isn't that unnecessary overhead?

In practice, the overhead is minimal. The key adds only a small number of bytes to the request, and the validation step is lightweight and fast. Most gateways or backend services can validate keys in microseconds, either against a local list, a database, or a cache.

By sending and validating the key with each call, the API remains **stateless**, which brings significant architectural benefits:

- Easier horizontal scaling
- Improved reliability (no session storage needed)
- Better fault tolerance across distributed systems

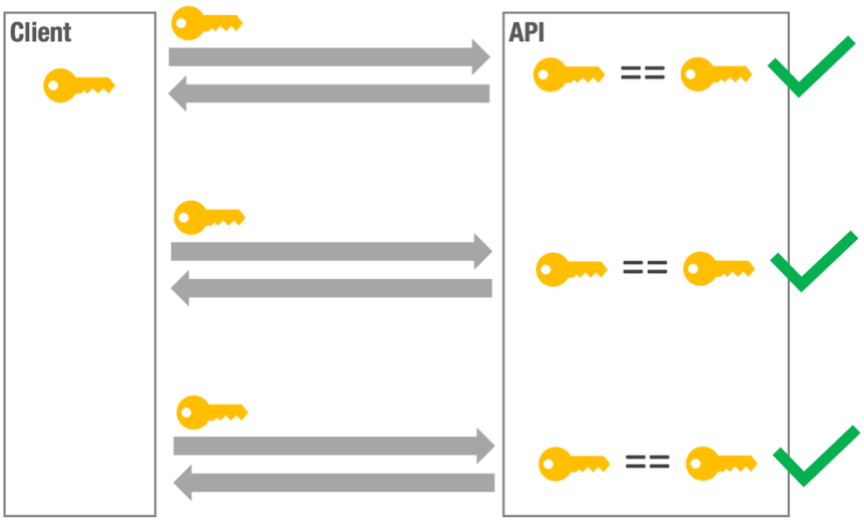


Image: Stateless Security, each request must provide a key

API keys and basic authentication share several similarities:

- **Stateless operation:** Both methods are stateless, meaning a secret is sent with each request without maintaining a persistent session.
- **Need for transport encryption:** As both are effectively transmitted in plaintext, they require transport encryption (e.g., TLS) to safeguard the secret against interception.

However, there are important differences stemming from the infrastructure that supports these methods. **Basic authentication** is typically provided by web servers or proxies. **API keys** are managed by API Gateways or API management solutions. They not only authenticate requests but also enable additional functionality such as analytics, rate limiting, and more granular security policies based on API keys.

Best Practices for API Keys and Roles

Here are some best practices that help mitigate common shortcomings of API keys:

- **Always use TLS**
  API keys are not encrypted and must never be transmitted over plain HTTP. Always protect them with HTTPS.
- **Do not hardcode keys in the configuration**
  Load keys from environment variables, external files, or a database. This improves security and simplifies key management.
- **Do not pass API keys to the backend**
  Keep them at the gateway. If the backend needs to know the client's access rights, forward derived information such as an `X-Scopes` or `X-Role` header instead.
- **Use long, unguessable keys**
  Randomly generated UUIDs or cryptographically secure random strings are good choices. Avoid short or predictable values like `test123`, which are vulnerable to guessing or brute force attacks.

- **Rotate API keys**
  Rotate API keys regularly, especially for public or long-lived clients. Regular rotation limits the impact of leaked or compromised keys.

15 Tokens and API Security

Traditional username–password authentication has several well-known drawbacks, especially in distributed systems and APIs:

1. **All-or-nothing access**
   You either share your full credentials with someone (which is a security risk) or you don't. There's no way to share just part of a password or restrict access to only a subset of functionality.
2. **Tedious recovery**
   If a password is lost or forgotten, the recovery process is time-consuming and frustrating.
3. **Password reuse is risky**
   Using the same password across multiple services increases the impact of a single breach.
4. **Managing unique passwords is hard**
   Using a different, strong password for every service is more secure, but also difficult to manage without additional tools.

To mitigate these problems, many people use a **password manager**. It stores individual passwords securely and unlocks them with a single master password. If one stored password is compromised, the others remain safe.

Tokens go even further in solving these challenges: they act as **temporary, scoped credentials** that are easier to manage, revoke, and control without exposing the underlying username and password.

15.1 What is a Token

A helpful way to understand what tokens are and how tokens improve API security is by imagining how payments work at a festival.

Image: Exchanging a token for food at a festival

When you arrive at a large fair or music festival, you typically don't pay cash at every food truck or vendor. Instead, you go to a central booth and exchange your money for **festival tokens**. These tokens are then used throughout the event: at food stands, drink booths, or game stations.

This system has several benefits:

1. **You don’t expose your real payment method.**
   Carrying around tokens is safer than carrying a credit card. If you lose tokens, the damage is limited.
2. **Tokens can be scoped.**
   Some tokens may be valid only for food, others only for drinks.
3. **Tokens can be limited.**
   You might buy ten tokens for the day, usable only during that event. If someone finds your leftover tokens tomorrow, they won’t work.
4. **You can delegate access without full trust.**
   You can give a few tokens to a friend to get food without giving them your entire wallet or PIN.

These same properties map directly to API authentication scenarios.

This same idea applies to APIs. Instead of handing out your password repeatedly, you exchange it once for a **token**. That token:

1 Represents your identity or the client's identity
2 Can be scoped to specific actions (In APIs, scopes represent permissions such as `read:users`, `write:orders`, or `admin:all`.)
3 Expires after a set time
4 Protects your password by avoiding repeated exposure to many systems

In short, tokens make systems **more secure**, **easier to manage**, and **more flexible**, especially when working across **multiple services, users, or devices**.

15.2 How (Bearer) Tokens Work

Most API tokens are **bearer tokens**, and they work a lot like those festival tokens.

A bearer token is a credential that grants access to whoever presents it. There’s no extra identity check. If the token is valid, access is allowed. The system assumes the caller is the legitimate holder of the token.

The sketch below shows a typical flow using bearer tokens:

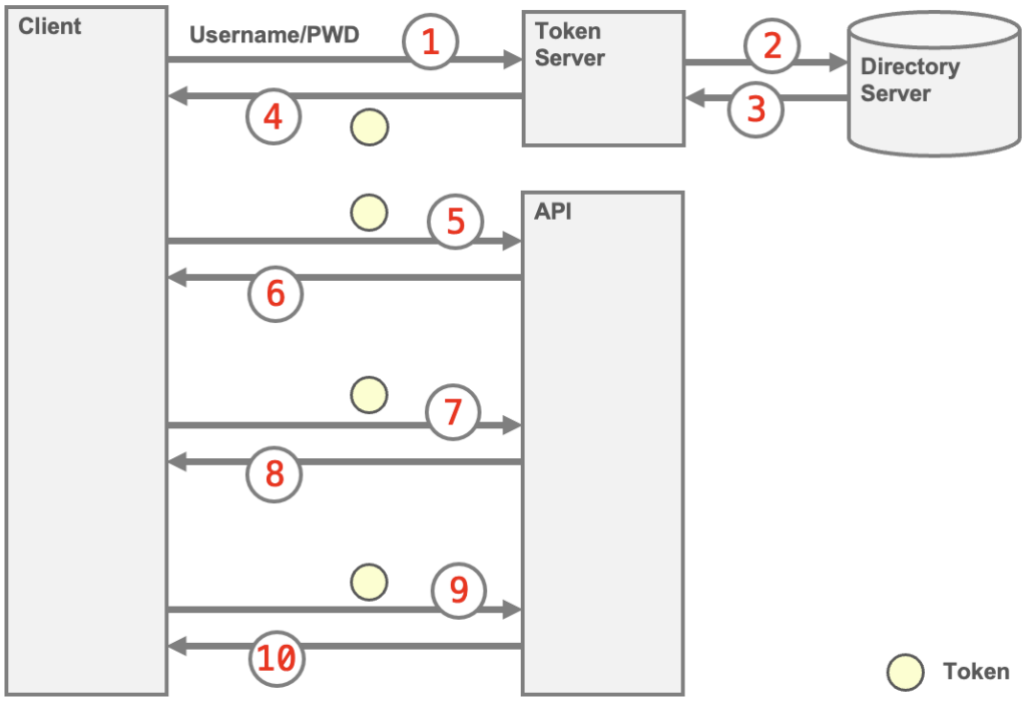


Image: Issuing and presenting a bearer token

Here's what's going on:

Step 1
The client **authenticates** by sending a username and password to the token server.

Steps 2–3
The token server might retrieve additional information, like roles or groups, from a directory service such as LDAP.

Step 4
It generates a token and sends it back to the client.

Steps 5–10
The client makes requests to the API, **including the token each time** for authentication and authorization.

💡 **Sidenote: Why "bearer"?**
The term comes from the idea that **anyone who bears (holds) the token** can use it. This makes bearer tokens convenient, but also risky: if someone else gets hold of the token, they can use it too.

To stay safe, bearer tokens should always be:

- Used only over **encrypted TLS** connections
- **Stored securely**, especially in frontend code and browser apps
- Configured with **expiration times**

Bearer tokens are everywhere for a reason. They are simple and flexible. But they do rely on keeping the token out of the wrong hands.

15.3 Types of Tokens

Bearer tokens are the most common type used in APIs, but there are important differences in how tokens are structured and verified. Not all tokens are created equal. How a token behaves depends on what it contains and how it is processed.

Bearer vs Non-Bearer Tokens

Most tokens, including festival tokens, API keys, and JSON Web Tokens (JWTs), are **bearer tokens**. This means that whoever presents the token is granted access. No additional proof of ownership is required. The system assumes that if a valid token is presented, the client is authorized.

However, not all tokens rely solely on possession. Some require the client to actively **prove possession** of a secret. These are sometimes referred to as **proof-of-possession (PoP) tokens**.

A common example is a **private key** held by the client. Instead of sending the key, the client is challenged by the server. The server sends a cryptographic challenge that only the holder of the private key can answer: Typically, the client signs the challenge with his private key and sends the result back to the server. If the decrypted value matches, the server knows the client holds the correct private key. This process is known as *proof of possession* or *proof of ownership*.

💡Sidenote: HTTP bearer tokens
"Bearer" is also used as an authentication scheme name in the `Authorization` header of HTTP requests:

```
Authorization: Bearer <token>
```

Opaque and Structured Tokens

Another important distinction is whether a token is **opaque** or **structured**.

Opaque tokens contain no readable information. They are usually just random strings or UUIDs, and their meaning cannot be inferred from their content. To verify an opaque token, the recipient must contact a central **token server** that stores the relevant metadata, such as the token's expiration time, the user it was issued to, and its current validity.

For example, this HTTP header carries an opaque token:

```
X-Api-Key: C32abQ-5031-42a0-bcea-7c839b5c6062
```

You can't tell what this token is for just by looking at it. Verification requires a round trip to the original token server.

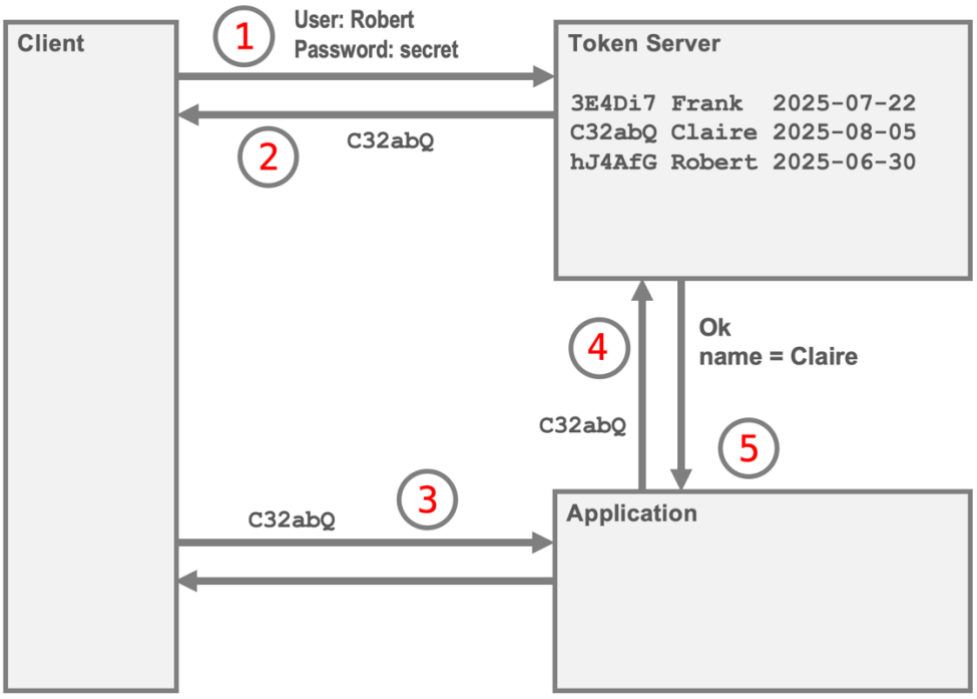


Image: Verification of an opaque token

The image above illustrates a typical verification flow. The client attaches the token to a request and sends it to the application (step 3). The application then forwards the token to the token server that originally issued it (step 4). The token server holds information about the token's context, such as the associated user and expiration time. Based on this information, it determines whether the token is valid. Finally, the result of the verification, along with any relevant metadata, is returned to the application (step 5).

In contrast, structured tokens like JWTs are self-contained. They carry information such as user ID, roles, and expiration time directly within the token payload. A receiving service can verify the token's digital signature locally without needing to contact a central token server.

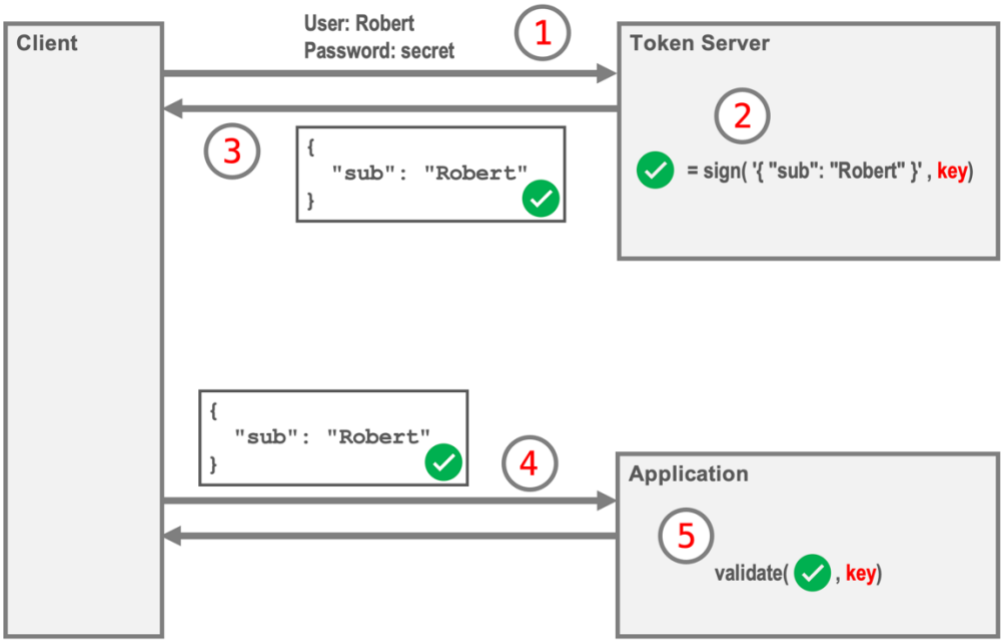


Image: Verification of a structured token by the receiver without consulting a token server

In a typical flow, as illustrated above:

1. The client authenticates with the token server.
2. The token server signs the token.
3. The signed token is returned to the client.
4. The client attaches the token to an API request.
5. The receiving application verifies the token's signature locally and decides whether to accept or reject the request.

For local verification to work, the receiver of the token must trust its issuer. This trust can be established in different ways, for example, by sharing a secret key or through a more sophisticated public-private key infrastructure. We'll explore how this trust is built, how tokens are signed and verified, and how these mechanisms work together in the chapter on JSON Web Tokens (JWTs) later in the book.

Trade-offs

The key trade-off between opaque and structured tokens lies in **revocation versus efficiency**:

- **Structured tokens** allow fast, local verification but are difficult to revoke, making short expiration times essential.
- **Opaque tokens** enable centralized revocation and tighter control but require a network call for each validation.

You may have noticed that we haven't talked about API Gateways in this chapter yet. In the next section, we'll look at how gateways can support token handling and where they fit into the overall architecture.

16 JSON Web Tokens

The predecessors of modern APIs, the XML-based Web Services, used powerful but extremely verbose token formats. Tokens adhering to the **Security Assertion Markup Language (SAML)** specification had numerous features, but the tokens were notoriously difficult to read, create, and verify. Perhaps most annoyingly, a typical SAML token could span multiple pages, making it impossible to pass along in an HTTP GET request due to length limits.

Why does that matter? Passing tokens via GET requests is particularly useful during login processes, where the token server redirects clients to their requested resource. Because XML tokens were too large for this, developers had to resort to complex workarounds like JavaScript hacks in the browser to trigger HTTP POST requests instead.

Today, many systems have embraced **JSON Web Tokens (JWT)**, a compact alternative. JWTs aren't just shorter and simpler; they're also easily serialized into concise strings. That makes JWTs perfect for use in URLs, even as simple query parameters:

```
GET /order/7?token=eyJhbGciOiJIUzI..
```

However, this is discouraged for security reasons: JWTs in URLs can leak via referrer headers, logs, browser history, and analytics tools.

Let's dive deeper into these simple yet powerful tokens.

What is a JSON Web Token?

JSON Web Tokens (JWTs)**,** pronounced "jot", are compact, structured, self-contained, URL-safe bits of information transferred securely between parties. They are commonly used to authenticate users in web applications and are increasingly popular for authenticating API clients as well.

The following request includes a JWT in the `Authorization` header using the Bearer token scheme:

```
GET /customers/ HTTP/1.1
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ
pc3MiOiJodHRwczovL2FjY291bnRzLnByZWRpYzguZGUiLCJzdWIiOiIxMjM0N
TY3ODkwIiwibmFtZSI6IlRvYmlhcyBQb2xsZXkiLCJpYXQiOjE3NTAxNzA5OTZ
9.RNCFqlC5Dt3-jNE68pmFe9yk3JsUWSp43pV4o2CQhLE
```

The highlighted part is just one very long line.

After verifying the token's authenticity and checking **trust** in the token's issuer, a server might decide to allow the request to proceed and return the list of customers to the caller.

🔎 **Quick tip for JWT experts**
You can recognize a JWT string by:

- The presence of **two** or **four dots** (`.`) separating the sections
- The prefix **`ey`** at the beginning, which roughly corresponds to `{"` when Base64URL-decoded (i.e., the start of a JSON object)

Two Kinds of JWTs

In many API systems, it's important to pass around information about the caller, such as their identity or access level, without having to query a central identity service for every request. JSON Web Tokens make this possible by embedding such information directly in the token itself.

Depending on whether the data needs to be visible or kept confidential, JWTs come in two flavors: **JWS (JSON Web Signature)** and **JWE (JSON Web Encryption)**.

With **JWS**, the token's payload is only **signed**, not encrypted. This means that any recipient can read the content but cannot alter it without breaking the signature. In contrast, **JWE** encrypts the payload, so its content remains confidential and can only be decrypted by trusted recipients.

In practice, most systems use **JWS tokens**, because the payload is **not considered sensitive** in the context of the application, such as a username or access role like `["read"]`. The payload does not need to be hidden from the API client. The important part here is that the information is **trusted and tamper-proof**, not necessarily secret.

Decoding JWTs

JWTs are used most often in their **compact serialization** format, especially when authenticating towards an API.

A JWS might look like this:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJodHRwczovL2FjY
291bnRzLnByZWRpYzguZGUiLCJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IlR
vYmlhcyBQb2xsZXkiLCJpYXQiOjE3NTAxNzA5OTZ9.RNCFqlC5Dt3-jNE68pmF
e9yk3JsUWSp43pV4o2CQhLE
```

At first glance, it might seem illegible. But don't worry, JWTs contain dots (.) that split the token into parts. A JWS always consists of three parts, while a JWE always consists of five parts. In edge cases, some parts may be empty, but we can safely ignore them for now.

Each individual part is encoded using the Base64URL scheme.

Base64URL Encoding

Base64URL encoding is a variant of Base64-encoding that uses URL-safe characters. This means that the encoded data can be safely included in URLs without needing additional encoding. Base64URL encoding is like Base64 but replaces the + and / characters with - and _, respectively, and omits padding characters =.

In the context of JWTs, Base64URL ensures that any payload — even if it contains characters like " { } or unicode letters — is safely representable without the +, /, or = characters used in standard Base64.

16.1 JSON Web Signature

JSON Web Signature (JWS) is a compact, URL-safe way to ensure the integrity and authenticity of its payload. It consists of three parts:

```
<Header>.<Payload>.<Signature>
```

Let's break those down.

The header typically includes information about the **type** of token and the **algorithm** used to sign it. The header of the JWS shown above is:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
```

which decodes to (pretty printed here for readability):

```
{
  "alg":"HS256",
  "typ":"JWT"
}
```

This example header only contains standardized parameters:

- `typ: "JWT"` tells us this is a JSON Web Token. This field is optional, though. The structure itself already gives that away.
- `alg: "HS256"` refers to the signing algorithm. It is defined in the JSON Web Algorithms (JWA) standard.

The payload contains the **claims**, which are statements about an entity (often the user or client presenting the token) and additional data. The payload of the JWS shown above is:

```
eyJpc3MiOiJodHRwczovL2FjY291bnRzLnByZWRpYzguZGUiLCJzdWIiOiIxMj
M0NTY3ODkwIiwibmFtZSI6IlRvYmlhcyBQb2xsZXkiLCJpYXQiOjE3NTAxNzA5
OTZ9
```

When decoded, it looks like this:

```
{
  "iss": "https://accounts.predic8.de",
  "sub": "1234567890",
  "name": "Tobias Polley",
  "iat": 1750170996
}
```

We will discuss the claims shown here further down in this chapter.

Signature

The final part is the **signature**. It's created by taking the header and payload, encoding them, and then signing them using the specified algorithm and a secret key. The result is a binary signature that ensures the token hasn't been altered.

If anyone tries to change the header or payload, the signature won't match anymore: The token will be rejected.

16.2 JSON Web Encryption

While JSON Web Signature (JWS) ensures that a token hasn't been tampered with, **JSON Web Encryption (JWE)** goes a step further. It hides the content entirely. If JWS is like sealing a letter with wax, JWE is like locking it in a safe.

Here's what a real JWE might look like:

```
eyJhbGciOiJSU0EtT0FFUC01MTIiLCJlbmMiOiJBMjU2R0NNIiwidHlwIjoiSl
dUIn0.eRmAQpf5PlWWITJlwfDL1Zi1LfF7R-Ut_smjDs-UuqAUqg3RMXu28nEV
utyJn1f1GHwIr4m9enmvIm4GWH84cxL7MNnhqsHXOsqMeMP1w8YMNQ9UC7SRas
p7f6Xr9cKpEneeclYEUrLUbdc-tm5UjEso1CCy1DvQaCsk2VgNU7aS1971ACYa
SAqmpkUc4bNQ2BP8euPkZUicYRg6pReqgsTb48nTA7ihJQH_uKm3Ps8NvpNXYA
TmPZlSyipLN46OvaIYgQeUzpY517juRt6Lm1xWqYNuOpJrU1dGwQH7ZPjMRT6R
fjEKz-PP8ij86tq1BRFWXxQfXj8lfVfykkACKA.zbG6eOT8NzvZTFWf.k_q3dt
GDSb3_PUsftJrhwQHhTXO1wmDymN4sJbfJ0PEMnoAaKGdTrTES2_DkQecTg9kU
76gWJOXcXo3gVM-AiaZayX2CIi1PPM042o4.Gnb0R__E_nIkkux9v26QWA
```

JWEs consist of five parts, separated by dots:

```
<Header>.<Encrypted Key>.<Initialization Vector>
.<Ciphertext>.<Authentication Tag>
```

Again, let's break down what each part does:

- **Header**: Contains metadata about the encryption, such as the algorithm used. Example (decoded):

```
{
  "alg": "RSA-OAEP-512",
  "enc": "A256GCM",
  "typ":"JWT"
}
```

- **Encrypted Key**: A symmetric key, encrypted with the recipient's public key. This key is used to decrypt the actual data.
- **Initialization Vector (IV)**: A random value that ensures the same plaintext will encrypt to different ciphertexts each time.
- **Ciphertext**: The encrypted payload that can't be read without the key.
- **Authentication Tag**: A value produced by the encryption process. It ensures the integrity of the encrypted content. If anything was tampered with, decryption will fail.

The last four parts except the header are binary data. All parts are Base64URL encoded to make them safe for transport in URLs and HTTP headers. To decrypt the token, you'd need the appropriate private key, which isn't included in this book.

Interpreting the Payload

The meaning of the claims is a **contract** between the party issuing the tokens and the parties verifying the tokens. Both parties must have a common understanding of what a certain field (like `sub`) means. Otherwise, things can get confusing really fast.

The JWT specification defines a few **standard claim names** with well-known semantics. Here are some common ones:

- `iss` (issuer) identifies the token server.
- `sub` (subject) identifies the entity the token refers to. It usually refers to the user account or client the token belongs to. The exact meaning of the field value is not defined, although the value should be **locally unique**. It often refers to an employee or customer number or their email address: All are unique values within a given organization.
- `name` is a custom claim and not standardized.
- `iat` (issued at) is the timestamp when the token was issued. It's expressed as a **Unix timestamp**. A Unix timestamp counts elapsed seconds since January 1, 1970, at 00:00 UTC, excluding leap seconds. A value of 1750170996 therefore refers to June 17, 2025 at 14:36 UTC.

16.3 How to Protect an API with JWT

To secure an API using JWTs, whether in your server implementation or at the API Gateway, you need to **inspect every incoming HTTP request** for a valid token. Only after verifying the token should you allow the request to proceed. The goal is simple: make sure the request is **authentic**, **untampered**, and **authorized**.

Here's how it works:

1. **Check for a JWT** in the request, typically in the `Authorization` header as a `Bearer` token.
2. **Verify the signature** using a key you already have and trust (either a shared secret or a public key).
3. **Reject the request** if the signature is invalid.
4. **Continue processing** if the signature is valid.

The **signature check** is what makes the token trustworthy. It proves that the token was issued by someone you trust and hasn't been altered in transit.

Depending on your exact use case, you might want to further limit the JWTs you accept by adding additional **claim checks**.

Adding a Time Restriction

Imagine an attacker managed to exploit a vulnerability in your API version 1 and got hold of a valid JWT from another user. Later, you upgrade to version 2 and patch the security hole. Problem solved? Not quite.

If that stolen token is still valid, the attacker might continue using it. Even though the vulnerability is gone. Sneaky, right?

To prevent this, you need to limit how long a JWT is valid. Here's how:

1. **Limit the validity period**
   Keep the token's lifespan as short as your use case allows. In many cases, 5 minutes is enough. For example, a token might be valid from:

   2025-03-14 09:30 UTC → 2025-03-14 09:35 UTC

   In JWTs, you express this using Unix timestamps with the `nbf` (Not Before) and `exp` (Expiration) claims:

   ```
   {
     ...,
     "nbf": 1741941000,
     "exp": 1741941300
   }
   ```

2. **Check the validity period**
   When your API receives a token, it should check whether the **current time** falls within the `nbf` and `exp` range. If it doesn't, reject the request.

   Allow a small clock-skew window (typically ±30 seconds) to accommodate minor time differences between systems. A token might have been valid when sent but expired by the time it arrived.

3. **Ensure accurate timekeeping**
   Your server needs to know the current time to verify tokens correctly. That means your system clock must be accurate. Use NTP (Network Time Protocol) or a similar service to keep it in sync. Of course, NTP itself might have security issues.

   A broken clock is only right twice a day, and that's not good enough for API security!

Adding a Spatial Restriction

When you're protecting multiple APIs with JWTs issued by the same authority, it's smart to add extra security checks. Imagine you're running two servers:

- `https://finance.predic8.de/` handles sensitive financial data
- `https://lunch-menu.predic8.de/` serves less critical information (unless someone's really hungry)

Naturally, you'll invest more in securing the finance API. But here's the catch: an attacker might target the weaker link. If they manage to get a valid token from the lunch menu API, they could try to use it to access the finance API.

That's the catch.

To prevent this kind of **cross-API token abuse**, you can enforce a **spatial restriction** using the `aud` (audience) claim in JWTs.

Here's how:

1. **Issue tokens for a specific API**
   The token issuer must include the `aud` claim in the payload to indicate which API the token is meant for. For example:

   ```
   {
     "aud": "lunch-menu",
     ...
   }
   ```

2. **Verify the `aud` claim**
   Each API must check the `aud` claim and reject tokens not intended for it. So, the **finance API** would reject any token with `"aud": "lunch-menu"`.

To access both APIs, a caller would need to acquire two tokens: one for the finance API and one for the lunch menu API.

If you require tokens to be valid for more than one API, you might consider issuing tokens with multiple audiences:

```
"aud": ["finance", "lunch-menu"]
```

However, be aware of the security implications. If you operate several APIs, e.g., `wiki.predic8.de`, `tickets.predic8.de`, and `crm.predic8.de`, using separate tokens for each audience is generally the more secure approach.

17 OAuth2 and OpenID Connect

OAuth2 and OpenID Connect underpin modern delegated authorization and identity federation. They allow apps to rely on trusted identity providers instead of implementing their own login and credential management.

Both standards help delegate the responsibility of managing user identities and access rights to a central authority, so your app doesn't have to reinvent the wheel.

Let's explore how they work, how they differ, and why they matter.

17.1 OAuth2

Back in Chapter 15.2 How (Bearer) Tokens Work, we talked about using tokens at a festival to buy food. Let's stretch that analogy just a little further. It still holds up.

Just as festivals standardize how you acquire and spend tokens, OAuth2 standardizes how clients obtain access tokens to call APIs.

Have you ever stood in front of a food truck, craving fries, only to realize you need to pay with tokens instead of cash? The question becomes:

- Where do you get the tokens?
- How do you get them?
- Do you pay with cash or with credit card?

This is exactly what OAuth2 standardizes: **the process of acquiring a token**.

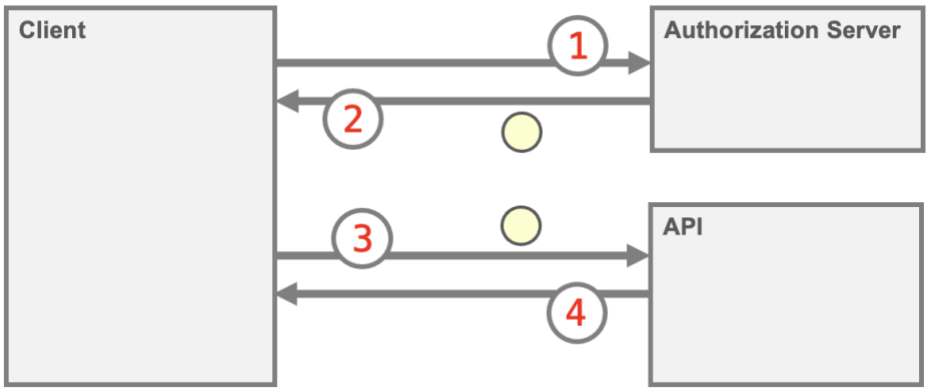


Image: Acquiring a token using OAuth2 and using it for API access

OAuth2 standardizes how clients obtain tokens, steps 1 and 2, but intentionally leaves many client authentication details flexible. For example, it does not mandate a specific authentication method that the Authorization Server must require from the client in step 1. In fact, OAuth2 leaves a lot of practical questions open:

- What does the token even look like?
- How should it be verified?
- Is the client a browser? Or an application? Is there a user involved at all?

17.2 Securing APIs with OAuth2

OAuth2 supports a wide range of scenarios, from web applications to mobile apps and even physical devices. With all its different flows, choosing the right one can feel intimidating at first. But if we narrow the focus to APIs, the picture becomes simpler. For API access, the most relevant flows are:

- Client Credential Flow
- Authorization Code Flow
- Authorization Code Flow with PKCE

The right choice depends on who is calling the API. To make it easier, we can group things into two principal use cases: applications acting on their own behalf, and applications acting on behalf of a user.

Applications Acting Alone

API stands for *Application Programming Interface*. The literal meaning is an interface between applications. Not a graphical user interface where a human interacts with software. Here, applications talk directly to other applications without a user being involved.

For example, a logistics application might notify an ERP system like SAP that a delivery has been completed. Since no user is directly involved, we describe this scenario as the client *acting on its own behalf*.

In the sketch below, the client application first authenticates with an Authorization Server (step 1). After successful authentication, it receives a token, represented as the yellow coin in the sketch (step 2). The client can then present this token when making the API call (steps 3 and 4).

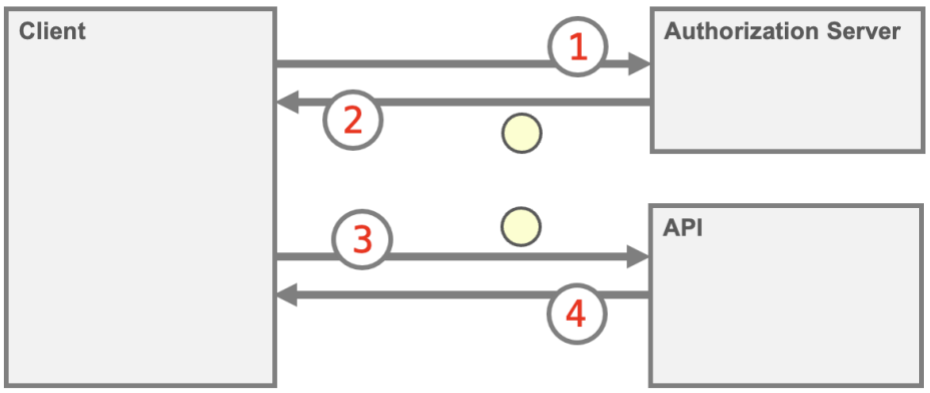


Image: API Access *on behalf of the client*

Since there is no user typing in a password, the client must prove its identity to the Authorization Server in another way. This can be achieved by storing a secret (similar to a password) or by using a certificate with a private key that is kept securely on the client.

Applications Acting on Behalf of the User

In this scenario, an application makes API calls as a delegate of a user. A common example is a web application such as an email client that communicates with a backend service. The client needs to prove not only its own identity but also that it is acting on behalf of the logged-in user. This isn't limited to web apps. Standalone desktop or mobile applications can do the same.

Here's how it works in practice: after opening a web application in the browser, the user is redirected to an Authorization Server (for example, Microsoft Entra). There, the user authenticates, typically with a password and possibly a second factor. Once that succeeds, the application requests a token from the Authorization Server. That token is then used to access the API, proving that the client is authorized to act on the user's behalf.

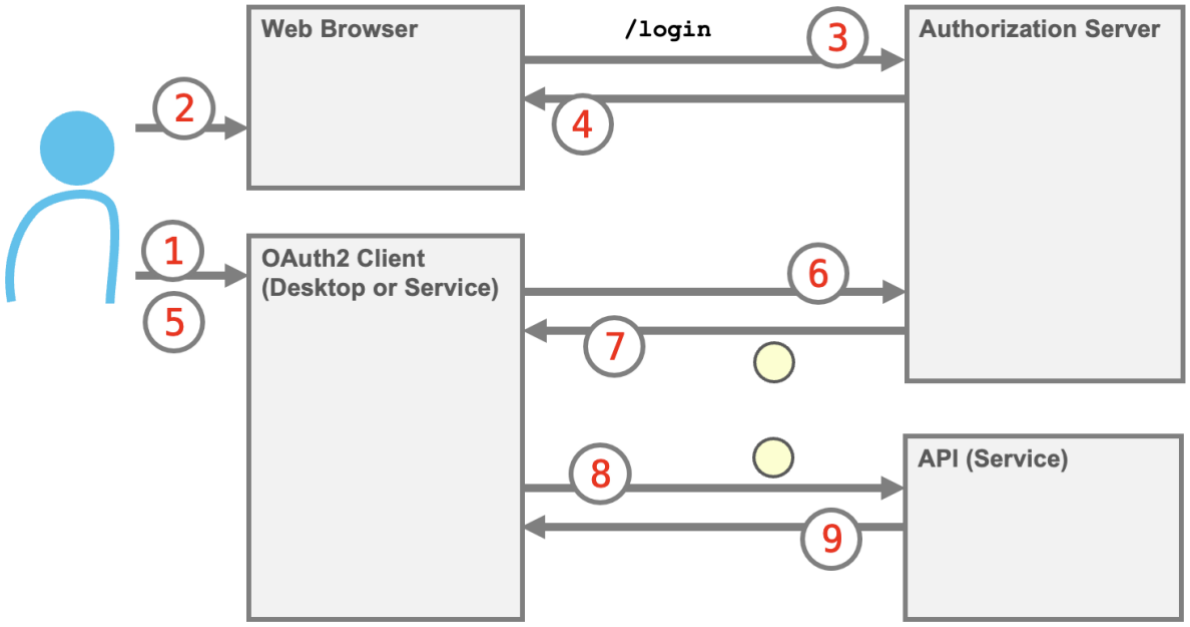


Image: A user authenticates through a desktop app, which then accesses the API on their behalf.

That's the rough overview of the two principal OAuth2 use cases for APIs. Next, we'll tackle the big questions, starting with: *How does the API know who is calling?* OAuth2 itself doesn't answer that. For identity, we need another standard: **OpenID Connect**.

17.3 OpenID Connect

OpenID Connect (OIDC) is a family of standards built on top of OAuth2. While OAuth2 focuses on *delegating authorization* and granting access to resources, OIDC adds *authentication*. The core specification, **OpenID Connect Core**, helps answer questions like:

- How does the API know who is calling?
- How can we retrieve the caller's username, email address, or phone number (if available)?

OIDC introduces several supporting standards as well. The most important one is **OpenID Connect Discovery**: It simplifies configuration by allowing clients to automatically find endpoints and capabilities of the Authorization Server.

Other OIDC specifications exist, but most are either highly specialized or rarely used in typical API scenarios, so we won't dive into them here.

Tokens in OIDC: ID vs Access

OIDC references the JWT standard, which might lead you to think: "So the tokens are JWTs, right?". Well, … maybe!

OIDC distinguishes between two types of tokens:

- **ID Tokens**: These are JWTs and are used to **identify** the user. They contain claims like the caller's name, email, and authentication time.
- **Access Tokens**: These come from OAuth2 and are used to authorize **access** to APIs. They may or may not be JWTs, depending on the implementation and configuration.

In practice, the line between these tokens can get blurry, especially when developers try to use ID tokens to access APIs (which they in theory shouldn't).

Before things get too tangled, let's take a step back and look at how OAuth2, OIDC, and JWTs work together in a typical API authentication flow.

17.4 OAuth2 and OIDC In Practice

Let's walk through a real-world example where a backend service takes the role of the **client** in an OAuth2 and OpenID Connect setup.

Don't worry about the complexity. This section gives you a detailed description of what is going on under the hood. The good news is that even though the communication flow is complicated, OIDC makes the configuration of API Gateways really simple!

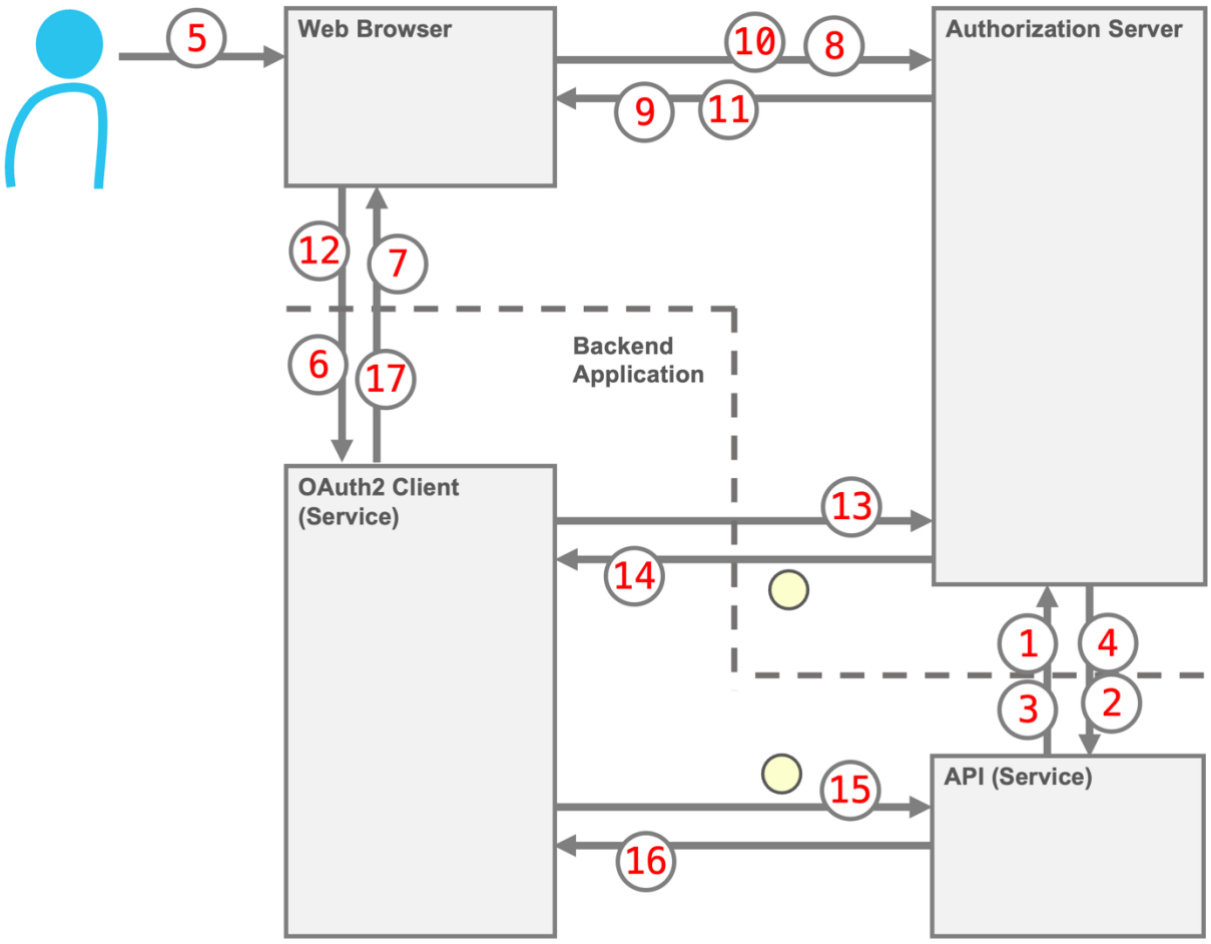


Image: A demo case of OAuth2, OpenID Connect (core and discovery)

In the image, a lot is going on. So, let's break it down step by step. On the left, we describe what's **happening (H)**. On the right, we explain which parts are **standardized (S)**, and where assumptions or implementation details come into play.

H: What Happens	S: What's Standardized
Step 1-4: When the API backend starts, it connects to the Authorization Server over HTTPS. It discovers the URL where the server's public keys can be downloaded. The backend downloads and caches those keys.	✅ Standardized by **OIDC Discovery**
Step 5-6: The user opens their browser and navigates to the client (e.g., https://clients.predic8.de/customers).	❌ Not standardized. This is just how users interact with web apps.
Step 7-12: Since the user isn't logged in, the client redirects the browser to the Authorization Server. (7+8) The server displays a login dialog. (9) After login, the user may see a consent dialog explaining what personal data (e.g., username, email) will be shared with the client. The user accepts (10) and is redirected back (11+12).	✅ Redirects: **OAuth2 + OIDC Core** ❌ Login method (password, passkey, multi-factor authentication (MFA)?): not standardized ⚠️ Consent dialog: part of **OIDC Core**, but often configured by the enterprise admin to be hidden (that means "auto accept")
Step 13-14: The client retrieves an access token and an ID token from the Authorization Server. Let's **assume** the access token is configured to be a JWT. Now the client knows the user is authenticated.	✅ 100% **OAuth2**, extended by **OIDC Core** ⚠️ Access token format is an implementation detail. It's often configurable, JWT is a common choice.
Step 15-16: The client calls the API on behalf of the user (e.g., GET /customers) and attaches the access token in the `Authorization: Bearer ...` HTTP header. The API verifies the token using the public key retrieved in Step 4	✅ The HTTP header format is standardized by the **OAuth2 Bearer Token** spec ✅ If the access token is a JWT, verification is also standardized (e.g., using the JWT spec and public key verification) ⓘ `GET /customers` is just an example and is application-specific

This flow illustrates how OAuth2, OpenID Connect, and JWTs work together in practice, while also showing where the standards end and implementation details begin.

💡 **Sidenote: Delegation vs. authentication**
OAuth2 is primarily about *delegation.* It lets a user grant limited access to their resources without sharing credentials. OpenID Connect builds on OAuth2 to add *authentication*, meaning it can also tell you who the user is. Think of OAuth2 as the valet key to your car, and OpenID Connect as the valet also showing you their ID badge.

17.5 Reasons to use OAuth2 and OpenID Connect

Why go through the trouble of setting up OAuth2 and OpenID Connect? Here are a few compelling reasons:

- **Separation of concerns**
  OAuth2 separates authentication and authorization from the actual API implementation. Your API doesn't need to know how users log in. It just checks the token.
- **Centralized control**
  A central Authorization Server is easier to maintain, upgrade, and audit. New policies can be rolled out in one place and take effect across all clients and APIs.
- **Consistent enforcement**
  All APIs, whether you have 10 or 1000, can use a common authorization mechanism (e.g., JWTs as access tokens). This makes it easy to define policies like:
  "Every HTTP request must carry either no token or a valid JWT."
- **Scalability**
  As your system grows, you don't want every API to manage its own user database. OAuth2 and OIDC allow you to scale authentication and authorization independently of your services.
- **Interoperability**
  OAuth2 and OIDC are widely adopted standards. They're supported by major identity providers (Google, Microsoft, Okta, etc.) and integrate well with third-party tools and libraries.
- **Security best practices**
  Tokens should be short-lived and scoped. This reduces the blast radius of a compromised credential and supports the principle of least privilege.
- **User experience**
  With OIDC, users can log in once and access multiple services without re-authenticating. This enables single sign-on (SSO) and smoother user journeys.
- **Auditability and compliance**
  Centralized login and consent flows make it easier to track who accessed what, when, and how. This is very useful for compliance and incident response.
- **Flexibility for different clients**
  OAuth2 supports different flows for different types of clients: web apps, mobile apps, backend services, and even IoT devices.

17.6 Setting Up a JWT Verifier with OIDC

OAuth2 and OpenID Connect might look complex from the outside, but the good news is: setting up JWT verification in an API Gateway is surprisingly straightforward.

When securing APIs with JSON Web Tokens (JWTs), hardcoding public keys or token verification URLs is brittle and hard to maintain. **OIDC Discovery** introduces a mechanism that automates all of this dynamically.

In practice, this means you only need the **base URL of the Authorization Server**. All other configuration is discovered dynamically. The gateway or verifier takes care of the rest.

According to the OpenID Connect OIDC Discovery specification, the discovery document is located at:

```
/.well-known/openid-configuration
```

By appending this path to the base URL, the gateway can automatically retrieve the provider metadata. This document contains all the information required to configure the token verifier at the gateway, including token endpoints and supported algorithms.

The illustration below shows the discovery flow:

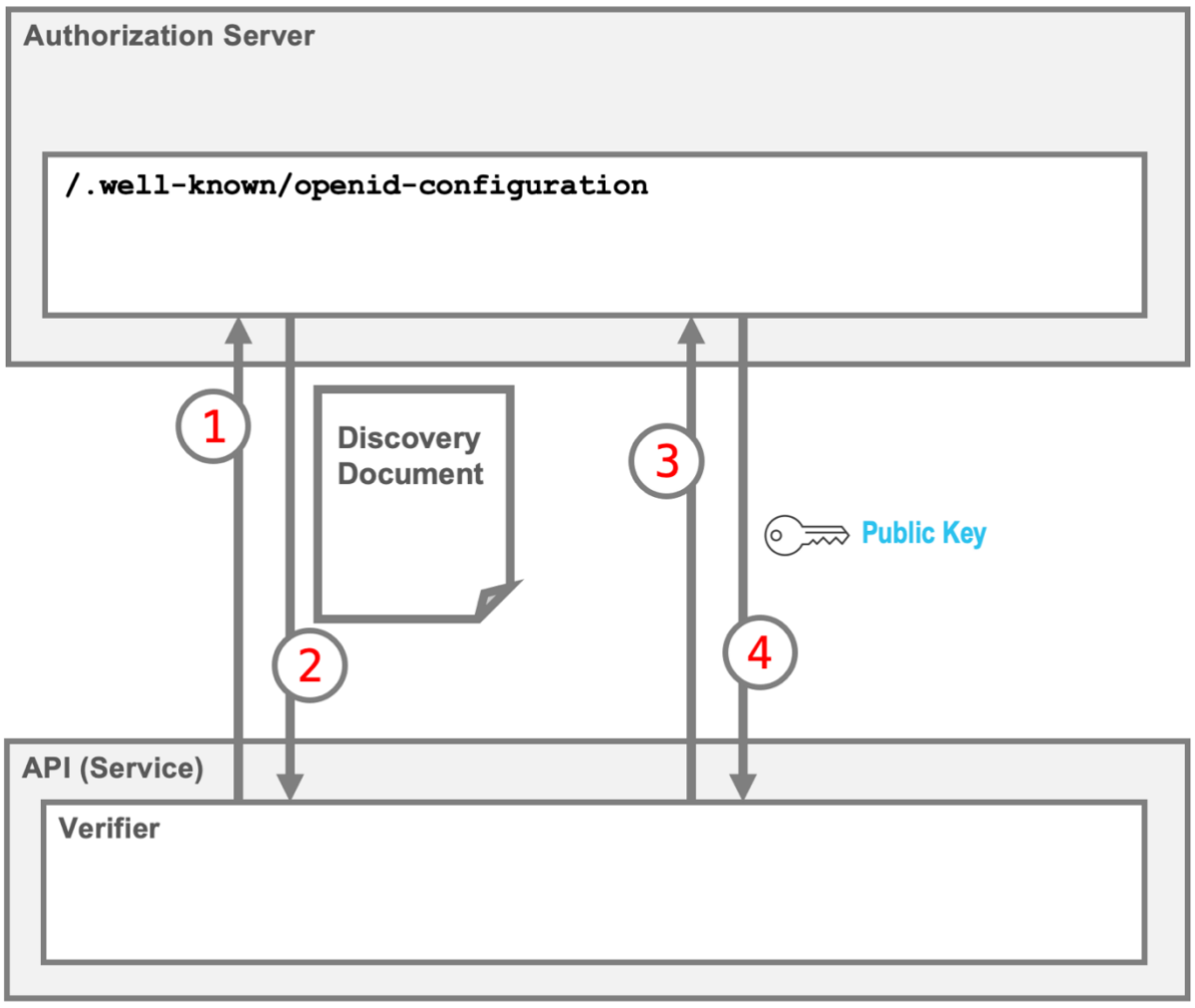


Image: How an API retrieves public keys and configuration from an OpenID Auth Server

Let's walk through the steps:

1. Initial discovery request

The verifier (usually part of the API Gateway or backend) starts by querying the OpenID Provider's standard discovery endpoint:

```
https://<auth_server_base_url>/.well-known/openid-
configuration
```

This is a fixed URL pattern. The `<auth_server_base_url>` is the base URL of your Authorization Server, such as Keycloak, Auth0, or Google Identity.

2. Fetching the discovery document

The Authorization Server responds with a JSON document describing its capabilities and important endpoint URLs. For example:

```
{
  "issuer": "https://accounts.google.com",
  "authorization_endpoint":
         "https://accounts.google.com/o/oauth2/v2/auth",
  "token_endpoint": "https://oauth2.googleapis.com/token",
  "jwks_uri": "https://www.googleapis.com/oauth2/v3/certs",
  ...
}
```

Of particular interest is the `jwks_uri` field, that's where we'll find the keys.

3. Requesting the JWKS (public keys)

The verifier downloads the JSON Web Key Set (JWKS) from the `jwks_uri`. Here's an example:

```
{
  "keys": [
    {
      "kid": "882503a5fd56e9f734dfba5c50d7bf48db284ae9",
      "kty": "RSA",
      "alg": "RS256",
      "use": "sig",
      "n": "woRUr445_ODXrFeynz5L208aJkABOKQHEzbfGM_V1...",
      "e": "AQAB"
    }]
}
```

The verifier typically caches these keys.

Because the JWKS is retrieved over HTTPS with validated server certificates, the verifier can trust the keys originate from the Authorization Server.

This dynamic setup is considered a best practice when working with modern identity providers such as **Keycloak, Auth0, Azure AD,** or **Google Identity**. It eliminates the need for manual key management and simplifies key rotation.

⚠ **Note**: **Reloading of keys**
How often the JWKS is reloaded depends on the implementation. Some gateways only fetch it at startup. If the keys rotate, the verifier might continue using stale keys unless periodic background reloading or manual refresh is supported.

What is JWKS?

The **JSON Web Key Set (JWKS)** document is a JSON structure that lists public keys used to verify JWTs. Each JWK includes:

- `kty`: Key type (e.g. RSA or EC)
- `alg`: Algorithm (e.g. RS256)
- `kid`: Key ID used to match with the JWT header
- `n`, `e`: RSA public key values, if it is an RSA key

The JWKS is fetched over a trusted TLS connection from the issuer, so the verifier can treat the keys as authoritative for that issuer.

17.7 Verification of JWT Signature and Claims

In contrast, the API gateway receives JWTs from untrusted clients via an *untrusted inbound* TLS connection *from anyone*. Trust in the JWT has to be established by verifying it.

The JWTs often include a `kid` (Key ID) field in their header like so:

```
{
  "alg": "RS256",
  "typ": "JWT",
  "kid": "882503a5fd56e9f734dfba5c50d7bf48db284ae9"
}
```

After receiving a JWT, the verifier reads the token's header to find the `kid` (Key ID) and selects the corresponding public key from the JWKS document. With the correct key in hand, the verifier checks the signature of the token. If the signature is valid, it proceeds to examine the claims inside the JWT, such as:

```
{
  "iss": "https://accounts.google.com",
  "aud": "my-api-client-id",
  "exp": 1716549780,
  "sub": "1234567890"
}
```

The verifier must check that:

- `iss` (issuer) matches the expected identity provider
- `aud` (audience) matches your API's configured client ID
- `nbf` (not before) is in the past
- `exp` (expiration) is in the future and hasn't yet passed

Only when the signature is correct **and** the claims are valid is the request allowed to proceed.

💡Sidenote: Expired or mismatched claims
Signature verification might succeed, but if any claims don't match, like an expired token (`exp`), the wrong audience (`aud`), or an unexpected issuer (`iss`), the token is still considered invalid. In such cases, most gateways will return a `401 Unauthorized` or `403 Forbidden` response depending on the context.

18 Rate Limiting

API calls happen fast, often so fast that you do not notice how many are being sent until the system is under pressure. Consider this simple Python script. It attempts to create one million products in the Fruitshop API:

```
import requests

for i in range(1000000):
    requests.post("https://api.predic8.de/shop/v2/products/",
                  json={ "name": f"Fruit-{i}", "price": 1.99}
                 )
```

Even though the script is small, it can generate a massive amount of traffic in a short time. Without rate limiting or other protective measures, such a burst can exhaust backend resources, saturate database connections, or trigger cascading failures.

You are welcome to experiment with this script against our demonstration API, but do not run it against production systems or third party services.

Why would someone hammer an API like that? There are several reasons:

- **Brute-force attacks**
  Continuously guessing passwords, API keys, or tokens.
- **Data scraping**
  Grabbing an entire database by repeatedly sending queries.
- **Heavy legitimate usage**
  Sometimes a customer genuinely needs heavy API usage.
- **Resellers**
  Users who build their business around your API.
- **Developer tests**
  A harmless coworker or student trying out the API.

Whatever the reason, the remedy is the same: set limits on how many requests clients can send or slow them down so your system can breathe. Rate limiting protects your backend, ensures fair usage, and prevents accidental overload.

Next, we'll dive into how gateways enforce these limits, track usage, and deliver helpful errors when clients exceed their allowance.

Hitting the Limit

What happens when an API is hit hard and the rate limit kicks in? The server enforces its rules. Here's the response a client receives after sending more than 500 requests:

```
HTTP/1.1 429 Too Many Requests
Content-Type: application/problem+json
RateLimit-Policy: "unauthenticated";q=500;w=3600
RateLimit: "unauthenticated";r=0;t=2351

{
  "title": "Rate limit exceeded.",
  "type": "https://iana.org/assignments/http-problem-
types#quota-exceeded",
  "violated-policies": ["unauthenticated"]
}
```

The `429 Too Many Requests` status code signals that the client has crossed the threshold. The **RateLimit-*** headers give the client guidance on what to do next. They indicate whether the client should back off or wait before trying again.

Here's what those headers mean:

Header Field	Value	Description
RateLimit	`"unauthenticated"; r=35;t=129`	35 calls remain within the next 129 seconds
RateLimit-Policy	`"unauthenticated"; q=500;w=3600`	Limit is 500 requests every 3600 seconds (1 hour)

Table: Rate limiting HTTP header

💡 Sidenote: RateLimit or X-RateLimit?
The IETF draft *RateLimit header fields for HTTP* is a standard in the making that defines a common set of rate-limit-related response headers such as `RateLimit` and `RateLimit-Policy` which are used in this chapter.
However, many APIs still use nonstandard headers with the `X-` prefix, such as:

```
X-RateLimit-Limit: 5
X-RateLimit-Remaining: 0
X-RateLimit-Reset: 48
```

These headers are widely used in practice, but their exact semantics can vary between providers.

How Rate Limiting Works

At first glance, rate limiting might sound simple. Just count the number of requests, right? But in distributed systems, things get complicated quickly.

Typically, you want to count requests per unique client. If clients are authenticated, that's straightforward: you can track calls by username, API key, or token.

Client	Number of Calls
Fredo	13
Sophia	7
Marc	93

The challenge comes with anonymous clients. In those cases, you often end up counting requests by IP address. That works to some degree, but it is not perfect: IP addresses can change when users reconnect, and multiple users inside the same organization might appear under a single shared IP.

Client	Number of Calls
192.168.2.99	324
10.7.75.102	10
10.2.99.3	25

Despite its shortcomings, IP-based counting is still one of the most common fallbacks when no other client identifier is available.

Flexible Counting

When requests include API keys, tokens, or other identifying data, you can use that information for counting instead of relying on IP addresses. Many API Gateways even let you configure custom expressions with an expression language, JSONPath, or XPath.

Here are a few examples:

Expression	Description
`header['Authorization']`	Count requests by authenticated user (HTTP header).
`jwt.claims['sub']`	Count by JWT subject (user ID).
`request.path`	Count by request path. The user or ID does not matter.
`$.product.id`	Count by product ID in the JSON payload using JSONPath.

Table: Expressions for counting requests

This flexibility allows rate limits to be applied in very specific ways. For instance, you might set tighter limits on sensitive endpoints such as `/login` or `/change-password`, while keeping more relaxed limits elsewhere.

Combined Aggregation

Counting can also be based on a combination of multiple values. For example:

```
jwt.claims['sub'] + method + request.path
```

This expression combines the user (from the JWT `sub` claim), the HTTP method, and the request path. Each unique combination is counted separately.

That means a `GET` and a `POST` to the same endpoint are treated as two different calls. Similarly, if a user accesses multiple paths, each path maintains its own rate limit.

This fine-grained approach gives you flexibility: users can interact with different endpoints without being unfairly throttled, while still protecting the API from cases where one user repeatedly calls the same method on the same path.

Counting with Distributed API Gateways

Imagine trying to count every car entering a city. You could place a student with a clipboard at each road. At the end of the day, you add up the tallies, and everything looks fine. But real-time counting? That's much harder. Each student would need to constantly sync their clipboard with a central counter.

Image: Counting cars entering the city

Distributed API Gateways face the same challenge. When multiple gateways serve requests, rate limiting requires synchronization. Without it, clients could sidestep the limits simply by routing calls through different gateways.

To solve this, gateways often rely on **shared counters**:

- **Databases** (such as PostgreSQL) for reliable, persistent counting.
- **Caches** (like Redis or Memcached) for fast, in-memory counting.

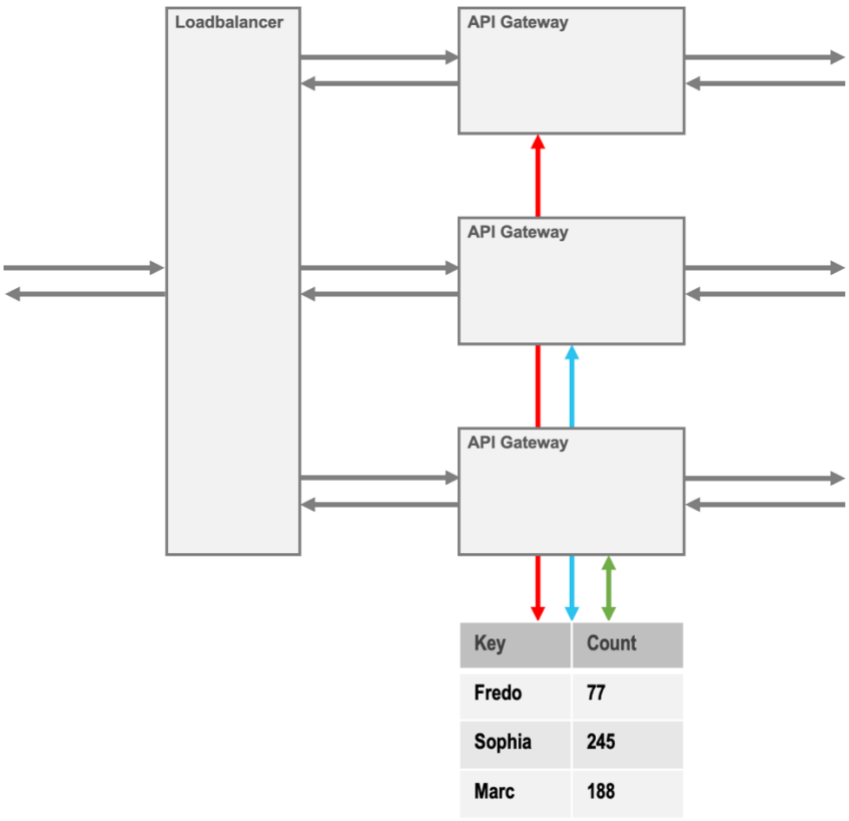


Image: Shared counters for rate limiting

This approach ensures global rate limits across all gateways, but it comes at a cost: added complexity and operational overhead.

Distributed Counting without Shared State

Some load balancers use **hash-based routing**. By hashing an identifier such as the client's IP address or token, they can ensure that a client is always routed to the same gateway instance. This way, each gateway can enforce its own limits independently, without the need for shared counters or storage.

💡**Best Practice: Multiple gateways without shared counters**
Another workaround is simply being generous. Instead of enforcing a total limit (e.g., 1000 requests/hour shared across two gateways), let each gateway enforce its own limit (like 750 or even 1000). This eliminates the need for shared state.

If you do this, avoid returning `RateLimit-*` headers, since their values would differ between gateway instances, potentially confusing clients.

Resources

RateLimit header fields for HTTP, Roberto Polli,
Alex Martínez Ruiz, Darrel Miller, 18 March 2025
https://datatracker.ietf.org/doc/draft-ietf-httpapi-ratelimit-headers/

19 Data Masking

The **General Data Protection Regulation (GDPR)** defines how organizations can collect, process, store, and share personal data of individuals in the European Union. These rules apply to APIs as well. Even if a gateway doesn't permanently store data, it still processes it which can introduce privacy risks.

A common example is **logging**. API Gateways often record requests and responses for auditing, debugging, or monitoring. But those logs may unintentionally contain personal data, especially with RESTful APIs.

Take this request:

```
GET /employees/34234
```

If logging is enabled, the employee ID will show up in the log. Under GDPR, such identifiers can be considered personal data. Just like names, addresses, or phone numbers, it requires protection.

Data masking addresses this problem by obscuring or anonymizing sensitive values before they are written to logs. For example:

```
127.0.0.1 - - [2025-05-07T12:03:57] "GET /employees/XXXXX"
```

This keeps the log useful for operations and debugging while reducing the risk of exposing personal data.

Masking can be applied not just to path parameters but also to query strings, headers, and even request or response bodies.

20 Security for Legacy Protocols (SOAP)

Some technologies like XML-based Web Services just refuse to die. Loved by few, maintained by many, SOAP and other verbose legacy protocols still lurk in the infrastructure of large organizations.

Exposing these services through modern API Gateways can be a challenge. Compared to JSON-based APIs, SOAP brings extra baggage: complex message structures, bloated payloads, and unique XML-specific risks like XML bombs or injections.

To address this, some API Gateways offer specialized support for securing legacy protocols, including:

- WSDL and XML Schema validation
- WS-Security enforcement
- XML Signature verification and XML Encryption
- Content-based filtering or transformation

This kind of support allows organizations to modernize gradually, integrating old and new systems without compromising on security.

20.1 WSDL Validation

For SOAP-based services, the **Web Services Description Language (WSDL)** acts as the contract between the client and the service. It defines which operations (remote functions) are available and what the expected messages look like. Just as OpenAPI describes REST APIs, WSDL defines what a SOAP service accepts and returns.

API Gateways can use WSDL documents to validate SOAP messages, ensuring they follow the expected structure and constraints. Validation checks that required elements are present, appear in the correct order, and contain values that match the defined types.

WSDL validation relies on **XML Schema Definition (XSD)**, which can be embedded directly in the WSDL or referenced externally. One advantage of WSDL over plain XSD validation is that it ties each operation to a specific XML element. This allows the gateway to validate messages precisely based on the operation being called.

In practice, WSDL validation is usually applied by attaching a validation policy to a service proxy. A service proxy in this context, works like an API definition in a gateway, just for SOAP instead of REST.

The stricter the WSDL and its associated schemas, the more effective validation becomes, not only for enforcing message structure but also as a defensive layer that improves security.

21 Cross-Origin Resource Sharing (CORS)

Modern web pages and single-page applications (SPAs) in particular access data and functions dynamically from backend systems by making HTTP calls from the browser. At the same time, browsers enforce the **same-origin policy**, a strict security mechanism designed to protect users and their data.

The same-origin policy prevents scripts loaded from one origin from reading responses from another origin. This is a fundamental defense against attacks such as **cross-site scripting.** It is an essential protection for users, even though it has a reputation for driving developers to despair.

CORS applies to cross-origin HTTP requests initiated from web pages, including APIs. Whenever an API is accessed from browser-based JavaScript, CORS rules come into play. Understanding these rules is essential when exposing APIs to web applications.

Cross-origin resource sharing provides a controlled way to relax this restriction. It defines how browsers and servers can safely interact across different origins, under clearly defined rules.

CORS is only relevant when APIs are called from **web pages running inside a browser**. Server-to-server communication and native mobile apps are not affected.

Cross-Site Request Forgery (CSRF) Attacks

CSRF attacks belong to a broader category known as **confused deputy attacks**. In this type of attack, a less-privileged actor tricks a more-privileged one into **performing an action on their behalf.**

Think of a thief convincing a company employee to open a locked door by claiming, “I work here too, but I forgot my key card.” The employee unknowingly becomes an accomplice, doing something they’re authorized to do, but for the wrong person.

Now apply this idea to web sessions, as illustrated in the image below.

Let's say a user logs into her bank account but forgets to log out, leaving a valid session cookie in the browser. Later, while casually browsing the web, she visits a **malicious website** (1). This site includes **hidden JavaScript** (2) that silently sends a forged request to the bank's API (3), instructing it to transfer money to the attacker's account.

The user's browser, unaware of the trick, **automatically attaches the valid session cookie** to the request (4). If the bank's session is still active, it processes the request and transfers the money (5).

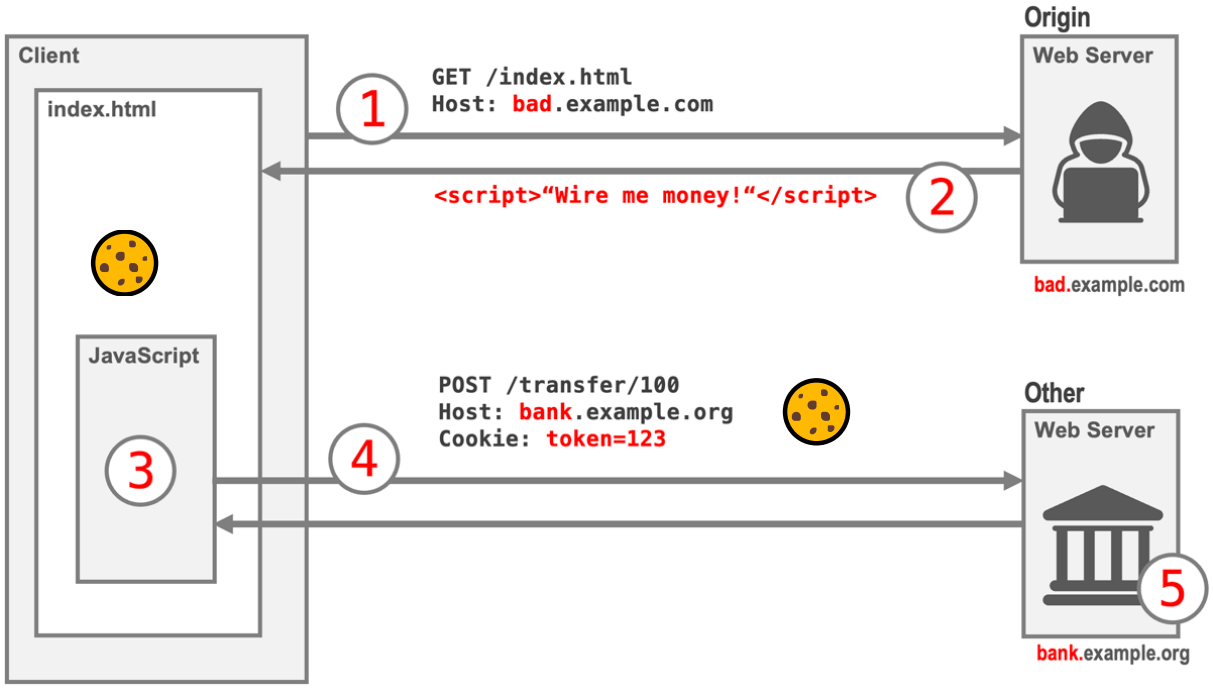


Image: CSRF attack against an API

This is a classic CSRF attack, where a victim's browser is abused to perform actions they didn't intend, using their valid credentials.

Since around 2020, modern browsers have tightened CSRF protections by introducing stricter default cookie settings (e.g., SameSite=Lax). These protections prevent cookies from being sent in cross-origin requests unless explicitly allowed, significantly reducing the risk of CSRF.

💡**Sidenote**: **CSRF and bearer tokens**
Cross-site request forgery mainly affects session-based authentication mechanisms where browsers automatically attach cookies. When APIs use bearer tokens in headers (such as OAuth2 access tokens or JWTs), the risk of CSRF is greatly reduced.

Why the Same-Origin Policy Blocks API Calls

Modern browsers protect users from CSRF attacks by enforcing the **same-origin policy**. This policy restricts scripts running in the browser from reading responses from a different origin than the one that served the page. While this is great for security, it can get in the way of legitimate use cases, like calling APIs from single-page applications.

Here's a typical scenario:

1. A web page is loaded from `www.predic8.de` (steps 1 and 2).
2. The JavaScript on that page attempts to call an API hosted at `api.predic8.de` (steps 3 and 4).
3. Since the domain differs, even slightly (due to the subdomain), the browser treats this as a cross-origin request and **blocks the calling script from accessing the response** unless CORS allows it.

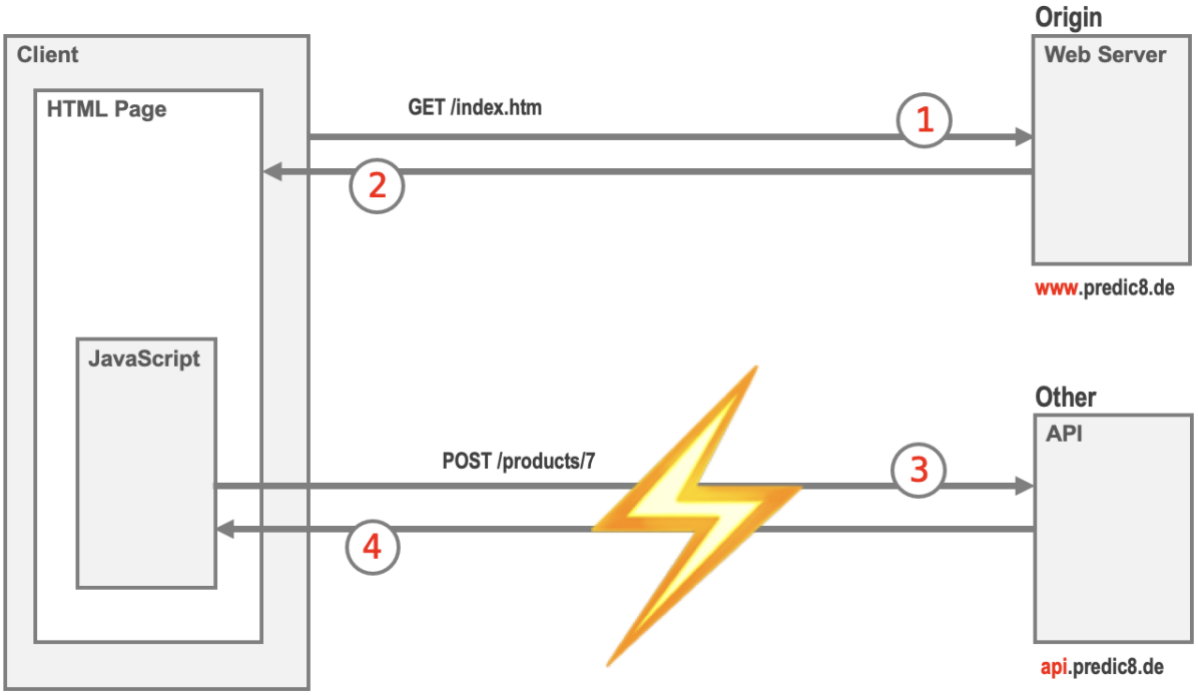


Image: Browser enforcing **same-origin policy** on a POST request to a different server.

That's where cross-origin resource sharing comes in.

CORS allows servers to indicate which cross-origin requests the browser may expose to the calling script. The browser enforces the policy, but CORS opens the door safely.

💡Sidenote: Policy enforcement at the browser
The same-origin policy is enforced by the **browser**, not the server. Without it, malicious sites could reuse authentication cookies or credentials to access private APIs behind the scenes.

How CORS Works

CORS lets a server explicitly say, “It’s okay for this resource to be used by code from origin xyz.” This permission is communicated through HTTP headers automatically added by the browser and corresponding response headers returned by the server.

When a browser makes a cross-origin request, it automatically includes an origin header such as:

```
Origin: https://www.predic8.de
```

It tells the server which origin the request is coming from. An *origin* is defined by the combination of **protocol**, **hostname**, and **port**. If the server allows that origin, it responds with:

```
Access-Control-Allow-Origin: https://www.predic8.de
```

This tells the browser: “You may expose the response to code originating from https://www.predic8.de”.

For potentially unsafe operations, such as `POST`, `PUT`, or any request with custom headers, the browser **must ask the server for permission in advance**. It does this using a **preflight request**.

💡**Sidenote: Same-origin policy vs. CORS**
The **same-origin policy** is a built-in browser security mechanism that blocks cross-origin response access by default.
CORS is the opt-in mechanism that allows a server to tell the browser, “This origin is allowed.”

Preflight (OPTIONS) Requests

CORS includes a handshake mechanism called **preflight**, where the browser first asks the server if a particular cross-origin request is allowed.

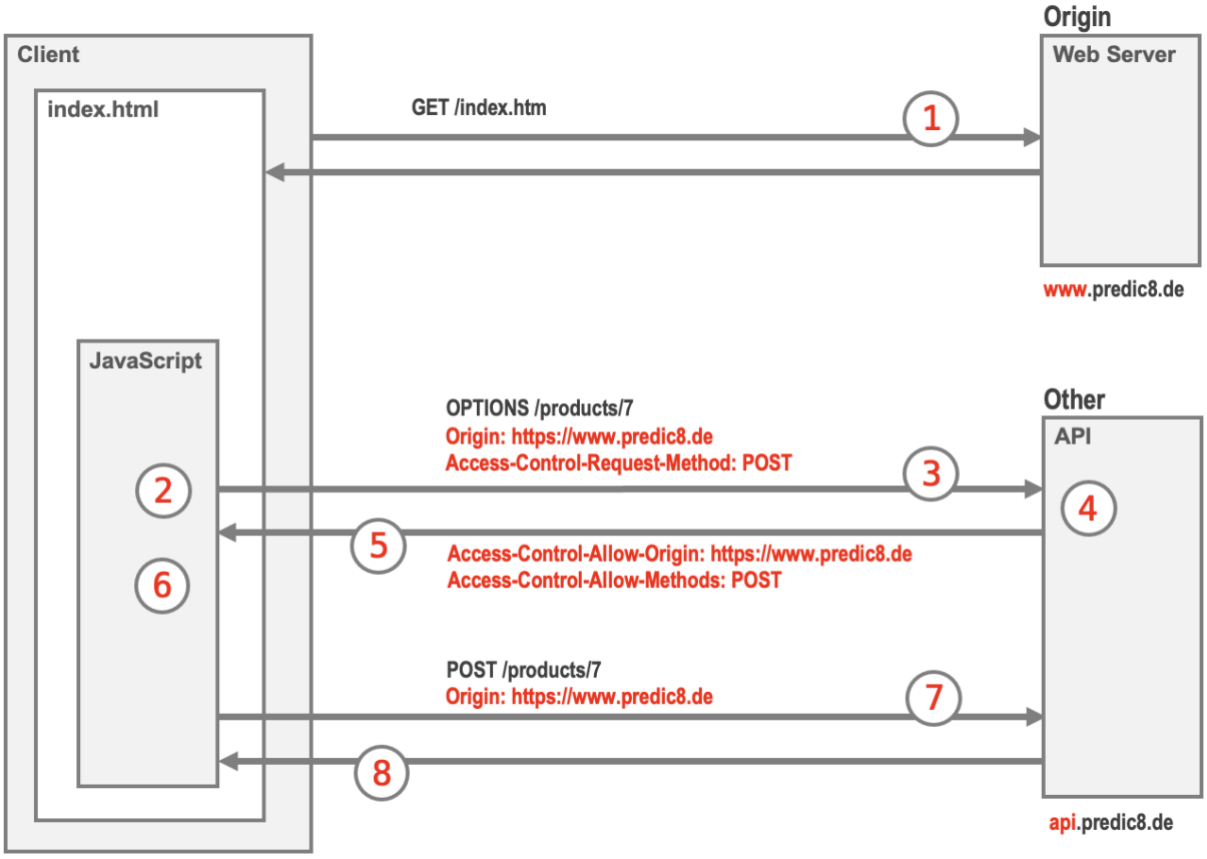


Image: How a browser performs a CORS preflight before a POST request

Here's how it works, step by step (illustrated in the image above):

1. A page with JavaScript is loaded from the origin.
2. Then the script uses the `fetch()` function to send a `POST` request with a `Content-Type` header:

```
fetch('https://api.predic8.de/shop/v2/products', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({ name, price })
});
```

3. Before the browser makes this potentially sensitive call, it sends an `OPTIONS` request to check whether the API allows this cross-origin request:

```
OPTIONS /products/7 HTTP/1.1
Host: api.predic8.de
Origin: https://www.predic8.de
Access-Control-Request-Method: POST
Access-Control-Request-Headers: content-type
```

   This request is automatically triggered and executed by the browser. It happens behind the scenes as part of the CORS preflight mechanism. The client developer does not need to write any additional code to initiate it.

4. The server evaluates the `OPTIONS` request based on its CORS policy.
5. If allowed, it responds with something like:

```
HTTP/1.1 200 OK
Access-Control-Allow-Headers: content-type
Access-Control-Allow-Methods: POST
Access-Control-Allow-Origin: https://www.predic8.de
```

6. The browser reviews this response.
7. If everything is fine, the browser proceeds to send the original `POST` request.
8. The server processes the request and returns the actual response.

Browsers handle preflight requests quietly in the background, so users usually don't notice. But you can inspect them using the **Network tab** in your browser's **Developer Tools**. Just open the DevTools (usually by hitting the **F12** key) and look for the `OPTIONS` request.

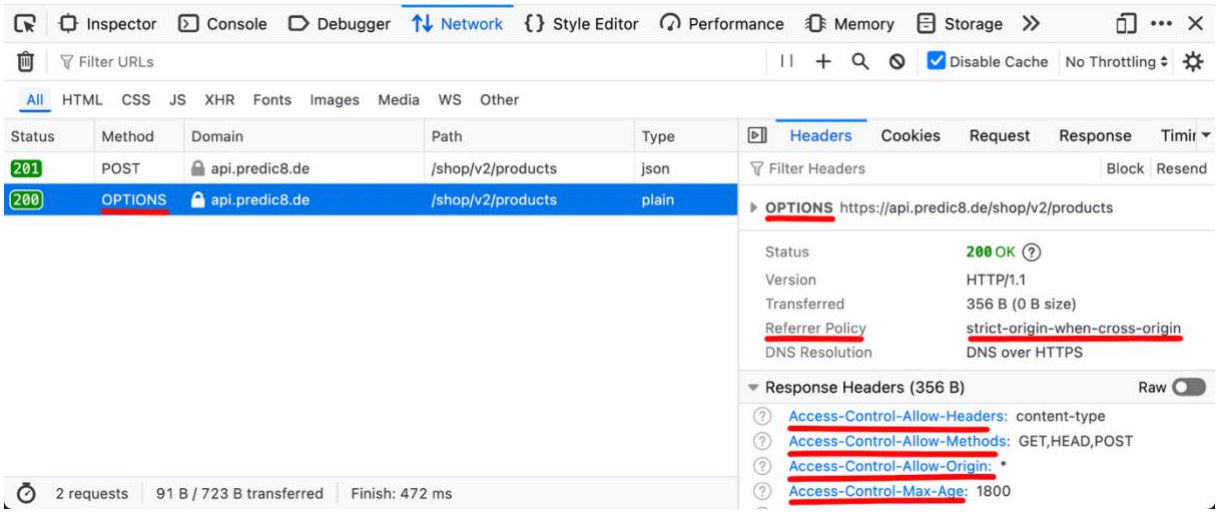


Image: Preflight request shown in the developer tools

To handle preflight requests efficiently, use a **CORS plugin** in your API Gateway. Once configured, it will add the required headers automatically.

If you want to avoid dealing with CORS altogether, you can solve the issue at the gateway level as described in the next section.

Preventing CORS Problems Using a Gateway

One simple and effective way to avoid CORS issues is to serve both the **web application** and the **API from the** same origin. If you control both components and can host them under the same domain, you can skip all the CORS complexity.

This works especially well when an **API Gateway** or **load balancer** is placed in front of both the web server and the API. Even though the web app and the backend API might live on different machines, the browser only communicates with the gateway.

From the browser's point of view, everything comes from a single origin, so **no CORS restrictions apply**.

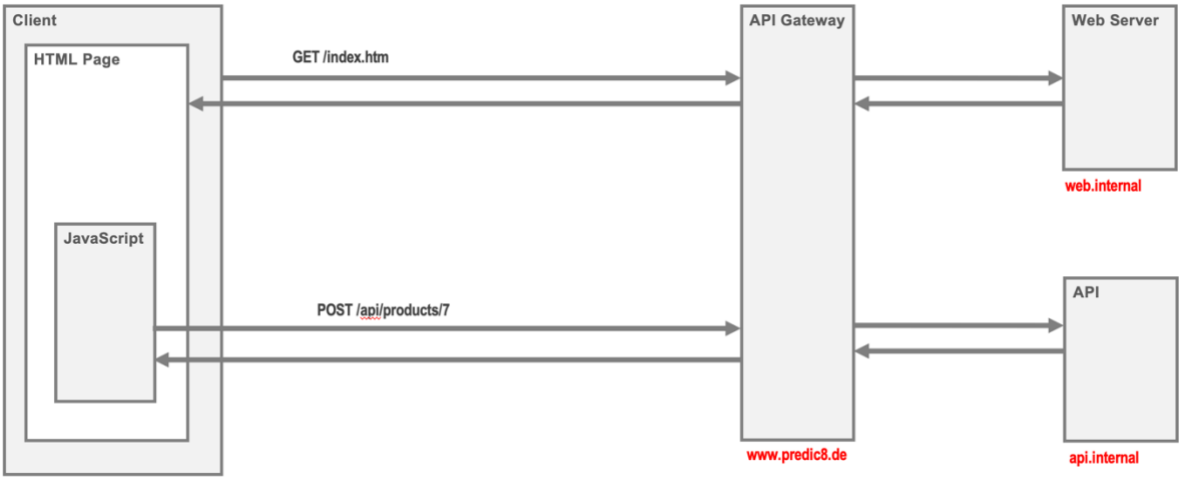


Image: Gateway setup avoiding CORS by hiding web server and API behind one address

This setup is common in enterprise or internal applications where you control the full stack. Routing both static content and API calls through the same gateway simplifies the architecture and removes the need to fiddle with CORS headers.

However, this strategy falls short when you offer APIs to others. If developers are embedding API calls into apps hosted on different domains, proper CORS configuration becomes unavoidable. In that case, you'll need to set up CORS rules at the API Gateway or directly on the backend.

CORS Support in Gateways

Most API Gateways provide support for CORS and can:

- Intercept and respond to preflight **OPTIONS** requests
- Add the correct **CORS response headers** to both preflight and actual requests
- Enforce CORS rules without requiring changes to the backend

This makes it easy to support cross-origin requests without modifying the application providing the API. You simply configure the gateway with the rules you want to allow, such as permitted origins, HTTP methods, or headers, and the gateway applies them consistently.

The example below shows how to configure an API in the Apache APISIX Gateway with the CORS plugin enabled:

```
{
  "uri": "/products/*",
  "plugins": {
    "cors": {
      "allow_origins": "https://www.predic8.de",
      "allow_methods": "POST",
      "allow_headers": "Content-Type,Authorization",
      "expose_headers": "Content-Length,Content-Type",
      "max_age": 3600,
      "allow_credentials": true
    }
  },
  "upstream": {
    "type": "roundrobin",
    "nodes": {
      "fruitshop2.prod.local:8080": 1
    }
  }
}
```

Resources

Cross-Origin Resource Sharing (CORS), @mozilla.org
https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CORS

CORS Online Demo for REST APIs
https://www.predic8.de/api-cors

22 API Load Balancing

Load balancers distribute work across multiple resources. Their main purpose is to scale applications and even out load. By spreading requests across servers, they also improve **availability** and **reliability**. For that reason, load balancers are also used when the volume of work or traffic isn't the problem.

What Is an API Load Balancer?

An API load balancer distributes incoming API requests across multiple backend servers. Unlike general-purpose network load balancers, it operates at the **application level** with the HTTP protocol. That means it can make routing decisions not only based on network or transport information but also on **API-specific details** such as status codes, HTTP headers, or tokens like API keys and JSON Web Tokens (JWTs).

22.1 Load Balancing Algorithms

When a request arrives, a load balancer has to decide which backend should handle it. Most balancers provide several algorithms for node selection, and some even let you define your own. At a high level, there are two different styles. Static balancers follow simple strategies such as round robin or random robin. They do not take the actual state of the backends into account, but they are easy to understand, robust in practice, and often good enough for many scenarios. Dynamic balancers, in contrast, use more complex algorithms that consider the current conditions. They might look at the health of each backend, the number of active connections, or the response time before making a decision. This allows them to distribute requests intelligently and adapt to changing conditions.

This section describes algorithms that are supported by many load balancing products. Each balancer comes with its own twists and variations, but the basic ideas are the same.

Round Robin

Round robin is probably the most widely used load balancing algorithm. It cycles evenly through all backend servers: the first request goes to server A, the next to server B, then to server C, and once the list is exhausted it simply starts over again. In the illustration below you can see how requests one through six are distributed across the backends in this predictable pattern.

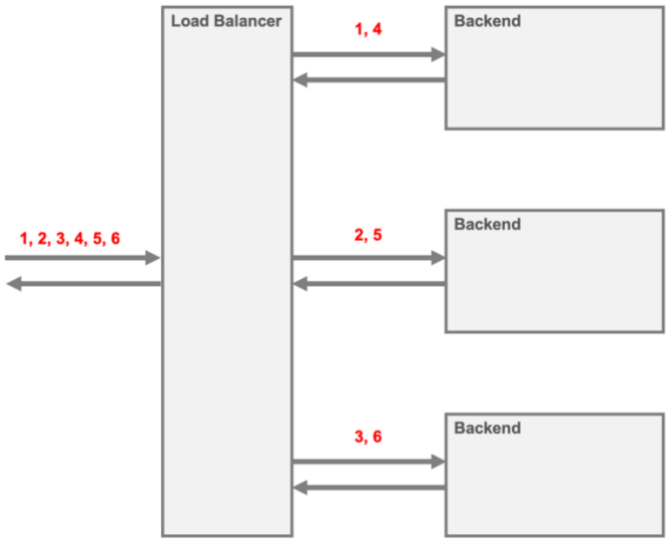


Image: Round robin distribution of requests across servers

An extension of this approach is **weighted round robin**, where each backend is assigned a weight that reflects how many requests it should handle relative to the others. A server with a higher weight will receive more requests than a server with a lower one. This allows stronger machines to carry more of the traffic, while weaker ones are still utilized without being overloaded.

Although round robin is trivial, it does involve keeping track of a counter that identifies the next node. That counter must be shared across concurrent request handlers. Even though it's only a counter, sharing and synchronizing access to it comes at a cost. This overhead is small but worth noting. To avoid it, some balancers use random robin.

Random Robin

Random robin, also known as random selection, is even simpler than round robin. Instead of maintaining a counter, it rolls the dice to pick the next server. Because it does not keep any state, it is lightweight, robust, and easy to implement.

Round robin guarantees predictability and a fair rotation across servers. Random robin trades that predictability for simplicity. Still, when you look at a large number of requests, the random distribution usually balances out well enough.

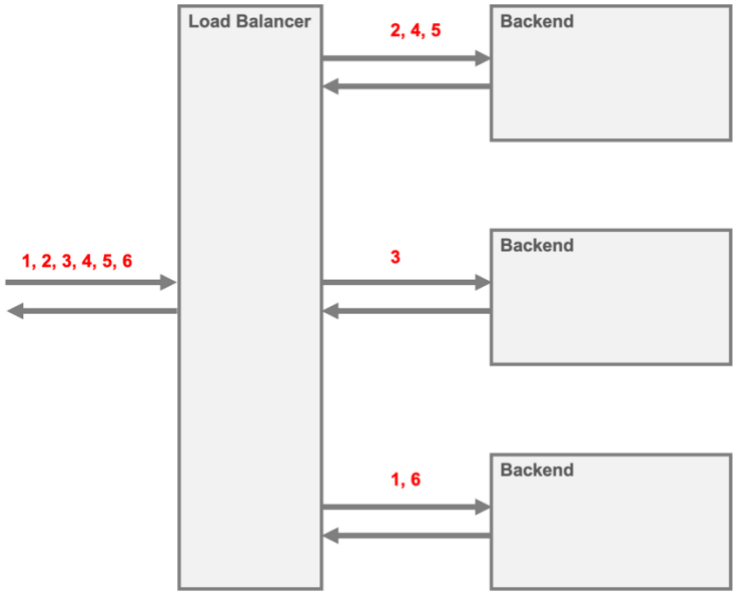


Image: Random distribution of requests

Session-Based Routing (Sticky Sessions)

Sometimes requests from one client must always be routed to the same backend. For example, the first request of a client session might be routed to backend C, where the user authenticates. As long as the next requests in that session go to the same backend, the user stays authenticated, and the calls are fast. But if a request is suddenly sent to a different backend, the client may need to authenticate again, which takes extra time and consumes resources.

A common solution is that the backend sets a **cookie** after successful authentication. The load balancer then reads this cookie and uses it as a session identifier, ensuring that all future requests from that client are routed to the same server.

For APIs, cookies are not the only option. Other identifiers are often used as well, such as JSON Web Tokens (JWTs), API keys, or other forms of client identity. Sometimes values inside JSON or XML payloads, like a user ID, can serve as a session identifier.

In the illustration, the numbers represent session IDs. The load balancer sends requests with the same ID to the same backend, ensuring session continuity. For example, a client with session ID 3 is always routed to the second backend.

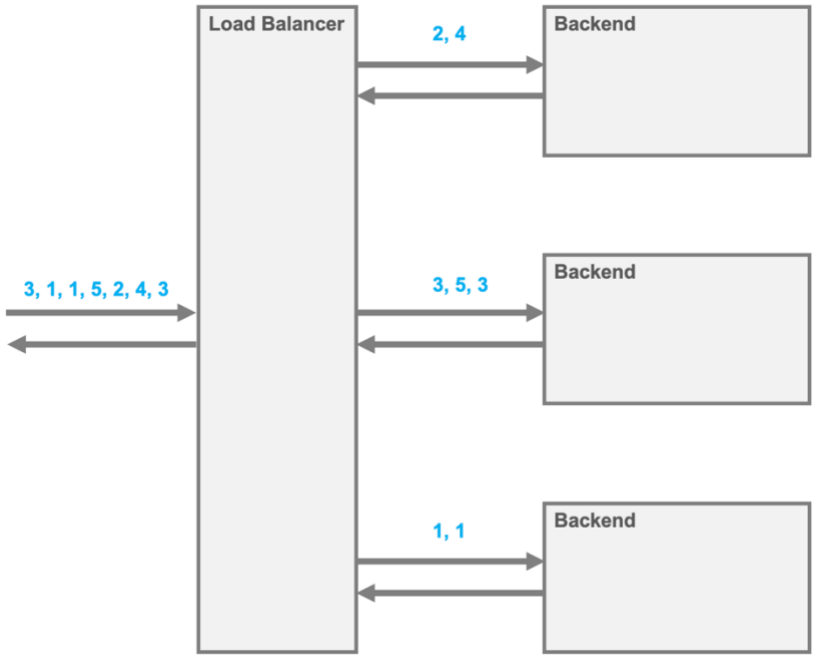


Image: Load balancing with sticky sessions

Priority-Based Balancing

Priority-based balancing is useful when certain servers should be preferred over others. A common example is when servers are spread across different locations. Local servers are usually given the highest priority, since they minimize latency and reduce bandwidth costs. Remote servers, in contrast, are treated as a fallback option that only comes into play if the local ones fail.

In normal operation, the load balancer routes all traffic to the priority 1 servers in the local data center.

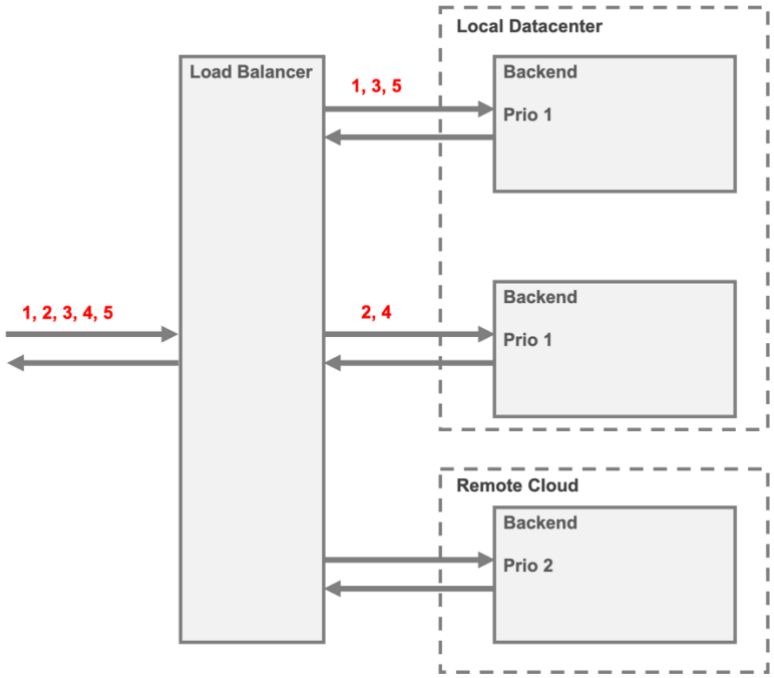


Image: Using priorities to route requests to backends in a local data center.

If the local servers go down, the balancer simply switches to the next available priority. In the second illustration, all local servers are down, so the requests are rerouted to the priority 2 backends in the remote cloud.

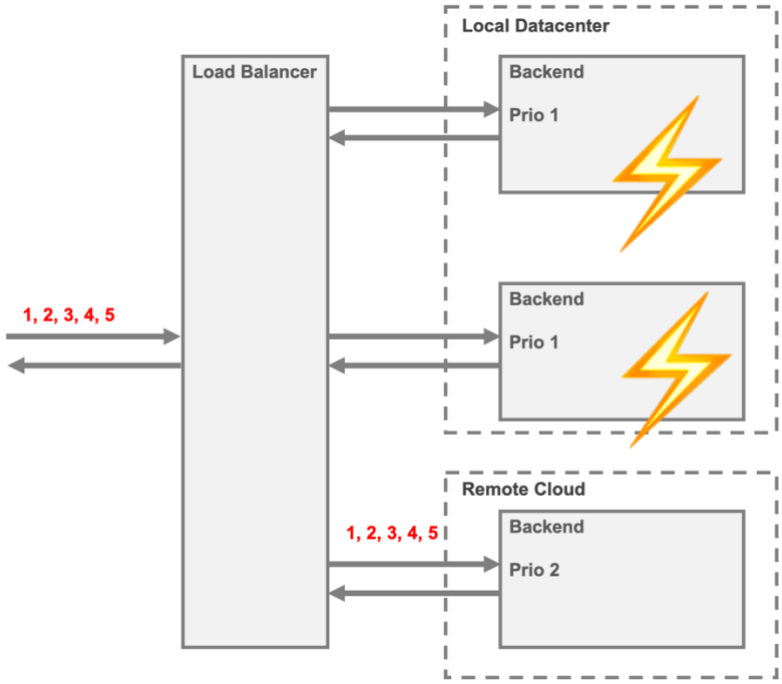


Image: Requests rerouted to remote cloud with lower priority

22.2 Dynamic Balancers

A dynamic balancer adjusts the distribution of traffic based on the current state of the backend servers rather than following a fixed scheme. This makes it more adaptive than static approaches.

It can take into account the health of nodes, response times, the number of active connections, or even the ratio of failures to successes. Using this information, it can direct requests to the servers that are able to handle them fastest, bypass servers that are unhealthy, and make the most efficient use of limited resources.

Dynamic balancers add complexity, and that complexity is not always necessary. Static balancers should not be underestimated. A simple round robin or random scheme, especially when combined with retries, can still support systems that are both scalable and resilient.

💡**Sidenote: When to use a static or dynamic balancer?**
Static balancers are best for simple setups where backends have similar performance. They're easy to configure, easy to maintain, and can provide reliability, especially when combined with retries.
Dynamic balancers are the right fit when backend performance or availability fluctuates. They are especially valuable if downtime must be detected and avoided. For resource-intensive workloads such as models, dynamic balancing can help squeeze the most out of each server by optimizing performance and utilization.

22.3 Health Monitoring

The health of backend systems is probably the most important information for a dynamic load balancer. After all, routing traffic to a dead or overloaded server defeats the whole purpose. There are two common ways to gather health information:

Periodic Health Checks

A balancer can periodically probe backends by calling their health endpoints. If a server fails the check, it is taken out of rotation until it recovers.

This approach is common in microservices and Kubernetes setups, where pods usually provide dedicated health endpoints such as `/healthz` or `/ready`.

As an advantage, servers can be taken out of rotation **before client calls hit them**. But health checks themselves generate traffic. Sometimes the volume of health checks exceeds the volume of actual client requests.

Sidenote: What if there is no health endpoint?
If there is no dedicated health endpoint, you can simply send a GET request against an existing resource. Ideally, it should be one that touches critical dependencies such as the database or external services. That way the check validates not only that the server is up but also that the resources it depends on are functioning.

Health Statistics

Instead of actively probing, the balancer can rely on real traffic data. By monitoring the outcome of backend calls (successes vs. failures), it can identify unhealthy servers. Backends with repeated failures can be removed from rotation.

The statistics come for free from normal request processing. No additional traffic is needed. As a downside, clients may experience failed requests before the balancer learns a server has a problem.

One way to reduce the impact: **combine health statistics with retries**. If a request fails, the balancer retries it against another backend. In many cases, the client never even notices the failure.

Sidenote: Hybrid health monitoring
Many balancers use a mix of both approaches. **Active health checks** detect problems early and prevent traffic from hitting unhealthy servers. **Health statistics from real traffic** give additional feedback and catch issues that may not show up in simple checks (like degraded performance or partial failures). By combining the two, a load balancer can make smarter decisions. Add retries into the mix, and clients often won't notice a failing server at all.

22.4 Availability and Failover

Critical applications that depend on APIs need strong guarantees, and two of the most important are availability and failover. **Availability** for an API means that it can be reached and will accept a request. If the server that is handling the request crashes midway, the client will still receive an error message. Availability does not promise that the processing itself will succeed. What it does guarantee is that there is another server standing by, ready to take the next request.

Failover, in contrast, is about shielding clients from technical issues such as server crashes, downtimes, or network errors. If a server fails while processing a request, the balancer can hand the request over to another backend that is healthy and able to complete it successfully.

Even a simple static balancer can provide availability by spreading requests across multiple backends. As long as at least one backend is alive, the API remains reachable. Compared to failover, ensuring availability is the easier problem to solve.

Failover for APIs is typically realized through retries. If a request to one backend fails, the balancer can attempt the same request against another server. Retries can be very effective when used under the right conditions.

Client Retries

Retries are not limited to API Gateways or load balancers. Many HTTP clients also repeat failed calls. In fact, most HTTP client libraries already include retry logic, and even browsers quietly retry certain requests without anyone noticing. When a retry succeeds, it is invisible to the application, and the user experiences a smoother interaction. Because networks are inherently unreliable, retries help to mask those imperfections.

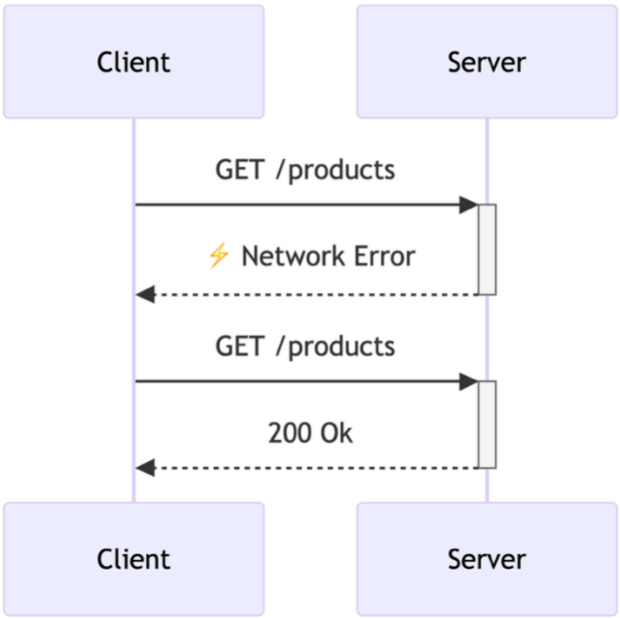


Diagram: Client retrying a request after a network error

The catch is that retry behavior varies widely, sometimes even between different versions of the same library. Most clients only retry certain network-level failures, and usually only for **GET requests**, since they are safe to repeat. Very few clients automatically retry on server-side errors such as `500` or `503`.

If clients do not provide the desired retry behavior, or if configuring them is not possible or becomes too cumbersome, the responsibility can be shifted to a load balancer between the client and the server. From that position, it can handle retries centrally and enforce consistent behavior across all clients.

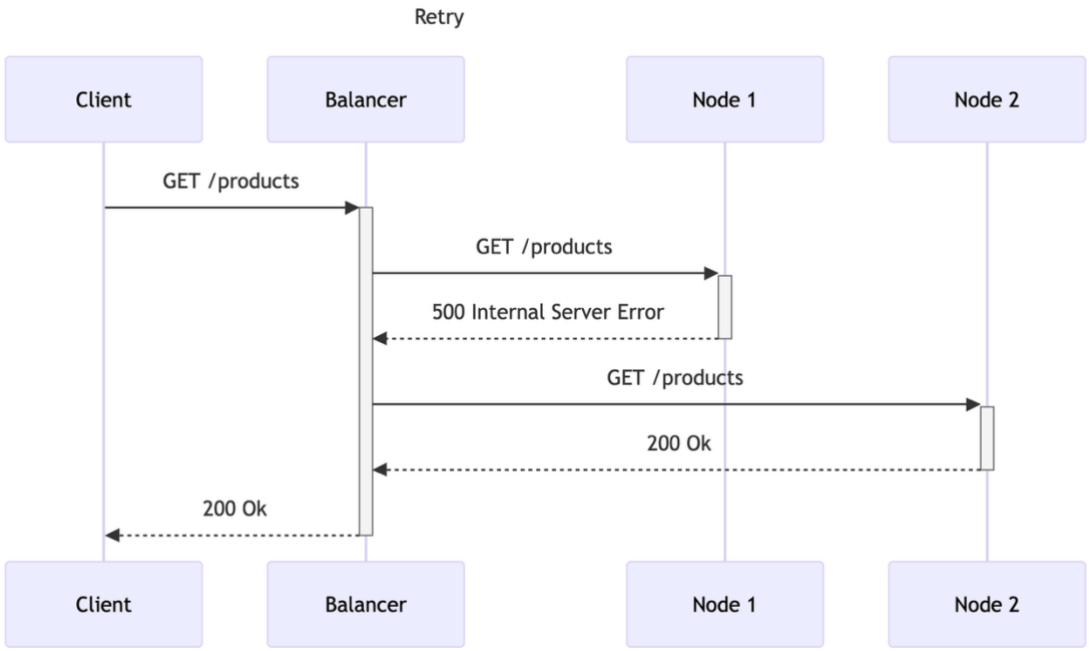


Image: Balancer retries failed request on second node

When applied carefully, retries can hide temporary glitches and keep systems running smoothly. When applied blindly, they risk corrupting data by creating duplicates or leaving a backend in an inconsistent state.

There are also situations where retrying makes no sense at all. Think of a phone call: if you get a busy signal, calling again later might work. But if you dial the wrong number, calling it over and over will not succeed. It will only annoy whoever answers. With APIs it is the same. Knowing when a retry is worthwhile and when it is pointless is essential to building resilient systems.

Harmless and Harmful Methods

Not all HTTP methods behave the same when it comes to retries. A **GET** request does not alter the state of the server, which makes it harmless. If a GET call fails, it can be repeated safely without any consequences.

Other methods, however, do change the server. After a **PUT** or **DELETE**, the server's state is no longer the same. But if the exact same request is repeated, there is no additional state change.

Take the following example. This PUT request changes the product with ID 15 so that its name is *Lemon* and its price is *0.79*:

```
PUT /shop/v2/products/15 HTTP/1.1
Host: api.predic8.de
Content-Type: application/json

{
  "name": "Lemon",
  "price": 0.79
}
```

If the request is sent once, twice, or even three times in a row, the result is the same: the product remains Lemon priced at 0.79. The server's state does not diverge further. The same applies to **DELETE**. Once a resource is deleted, deleting it again has no effect.

This property is called **idempotence**. Methods such as PUT and DELETE are idempotent because repeating them results in the same server state as executing them once. A load balancer can therefore retry those requests without concern.

POST, on the other hand, is not idempotent. Each call usually changes the server's state in a new way. For example, the following request creates a new product with every call:

```
POST /shop/v2/products HTTP/1.1
Host: api.predic8.de
Content-Type: application/json

{
  "name": "Apricot",
  "price": 1.49
}
```

Calling it three times will create three different products, all named Apricot but with unique IDs:

```
{
  "products": [
    ...
    {
      "id": 19,
      "name": "Apricot"
    },
    {
      "id": 20,
      "name": "Apricot"
    },
    {
      "id": 21,
      "name": "Apricot"
    }
  ]
}
```

Because POST is not idempotent, retries must be avoided or applied with great care. That said, there are conditions under which retrying a POST can be safe, as we will see later.

Besides the HTTP method, the status code also provides important clues about whether a failed call can be retried.

HTTP Status Codes

Failed requests are answered with a status code of 400 or greater. The **4xx codes** are usually not worth retrying, because they indicate user or client errors. A *404 Not Found* will return the same result regardless of how many times the request is repeated.

The **5xx codes**, on the other hand, signal server-side errors. Some of them make retries pointless. *501 Not Implemented*, for example, is not going to succeed on the second try unless the developers implement the missing feature in the meantime. Others, such as *500 Internal Server Error*, *502 Bad Gateway*, or *504 Gateway Timeout*, could succeed on retry, especially if the problem was transient or limited to one backend node.

The most common case is the generic ***500 Internal Server Error***. It typically occurs when the server has problems connecting to a database, cannot reach a downstream API, or has run out of memory. Retrying the same node in those conditions is unlikely to help. But trying another backend may work if that server is healthy.

The table below shows common 5xx codes and whether a retry with the next node is generally considered sensible:

Status Code	Description	Retry is sensible
500	Internal Server Error	yes
501	Not Implemented	no
502	Bad Gateway (backend failed)	yes
503	Service Unavailable	yes
504	Gateway Timeout	yes
507	Insufficient Storage	yes

Table: 5XX status codes and retries

Many API Gateways and load balancers allow you to configure retry behavior for 5xx codes, since not all applications need the same handling.

There are even cases where retrying a **POST** request after a 500 can be safe. With proper use of transactions and a suitable framework, a server application can guarantee that no state change occurred before the error was raised, or that the transaction was rolled back entirely. But this requires strong guarantees from the transaction model. If the server state has changed, repeating the POST risks corrupting data or creating duplicates.

Network Errors

Not every failure comes from the backend server itself. The network can also be the culprit. A network error can occur before a request reaches the server, while the server is processing it, or even after the server has finished its work.

If the error happens **before the server was reached**, it is generally safe to retry, even for a non-idempotent request such as POST, since the backend never saw it. But if the error occurs **during or after processing**, the request may already have caused a change on the server, which makes retries risky.

Understanding the exact meaning of network-related error codes is therefore important: they provide clues about *when* the error occurred. Some codes clearly indicate that the request was never delivered at all. For example, the TCP error ***Connection Refused*** means that no process was listening on the target port. In that case, it might even be safe to retry a POST because the server never received the call.

The table below lists common TCP error messages and whether retrying is generally considered safe:

TCP Error	Meaning	Safe retry?
Connection Refused	No process is listening (server down or a blocking firewall)	Yes
Connection Reset	Connection closed by the peer (crash, overload, firewall)	No
Connection Aborted	Connection closed unexpectedly (often local socket issue)	Yes
Connection Timed Out	No response within timeout	No
Host Unreachable	No route to the host	Yes
Network Unreachable	Routing issue, request never left the client.	Yes

Table: Network errors and safe retries

If there is even the slightest doubt that the request processing has already started, a non-idempotent call must not be repeated. The risk of duplicating or corrupting data is too high.

Load balancers are typically aware of common network errors and react accordingly. In many cases, you can rely on their default behavior without needing to adjust any configuration. This makes handling network glitches largely transparent, so developers can focus on the application logic rather than fine-tuning error handling in the balancer.

22.5 Single Point of Failure

High availability means avoiding a single point of failure. Instead of relying on a single backend, systems typically run multiple nodes. A load balancer can then distribute requests across healthy servers.

But what if the load balancer itself goes down? In that case, it becomes the weakest link. To avoid this situation, there are different approaches.

DNS Load Balancing

The Domain Name System (DNS) can be used to achieve load balancing on the client side, removing the need for a separate load balancer in the middle that could otherwise become a single point of failure.

When a client wants to connect to a server, it first resolves a hostname like `api.predic8.de` to an IP address, for example `20.113.32.106`. Instead of returning just a single address, a

DNS server can return multiple IPs for the same hostname. If one of those addresses does not respond, the client can try the next one.

Here's an example with Cloudflare, which provides two IP addresses for the same hostname:

```
$ dig www.cloudflare.com +short
104.16.123.96
104.16.124.96
```

DNS load balancing happens on the **client side**, which means there is no central load balancer that could fail. This makes it a very robust option. Many large-scale websites rely on DNS-based balancing as part of their overall traffic distribution strategy.

Support for multiple IP addresses, however, depends on the client software. Some HTTP libraries, such as Java's `HttpClient` (since Java 11), Go's `net/http`, or `curl`, handle multiple IPs gracefully and retry with another if one fails. Other clients may only use the first IP address provided by the operating system and ignore the rest.

Another factor is caching. DNS records have a time-to-live (TTL), and depending on the value, clients may hold on to old IP addresses for minutes or even hours. That means changes to the DNS configuration are not always reflected instantly.

💡**Sidenote**: **DNS load balancing in Kubernetes**
Kubernetes relies heavily on DNS. Services inside a cluster are assigned stable DNS names, and kube-proxy ensures traffic gets routed to the right pods. This illustrates that DNS balancing plays a central role not just for public websites, but also for container orchestration systems.

Anycast Routing

The next approach, **anycast routing**, also avoids a central balancer by allowing multiple servers to share the same IP address. When a client connects, the internet's routing infrastructure automatically directs the request to the server that is closest in network terms. Similar to DNS load balancing, anycast reduces the risk of a single point of failure and distributes traffic across multiple endpoints.

The setup, however, is more complex and comes with some caveats. Connections may break if routing changes cause traffic to be directed to a different node during an active session. Because the same IP is served by multiple nodes, TLS keys must be shared across all endpoints.

Despite these challenges, anycast is widely used and supported by major cloud providers. Services such as *AWS Global Accelerator*, *Google Cloud's Global Load Balancer*, *Azure Front Door*, and *Cloudflare* all offer anycast-based load balancing.

Keep Load Balancing Simple

If DNS load balancing or anycast is not an option, you may have to accept that the load balancer itself can become a single point of failure. In general, components that are complex and prone to failure should be made redundant. Consider backend applications that depend on databases and external services. However, components that are simple, unlikely to fail, and easy to recover can often be deployed as a single instance. Load balancers and API Gateways sometimes fall into this category.

A balancer running in a small, stateless container without a database can often run for extended periods of time without issues. And if it fails, restarting the container typically takes only a few seconds. Because it is stateless, the restarted balancer comes back as pristine as a new one. Standby virtual machines can provide a similar level of resilience where containers are not available.

To minimize the impact of failures, monitoring is key. Detecting problems early and reacting quickly keeps downtime short.

23 Performance

An API Gateway sitting between client and backend adds an extra hop to the communication path. That additional hop comes at a cost. The question is how significant that cost is and under which conditions it becomes noticeable.

This chapter introduces key performance metrics such as latency and throughput. It examines how API Gateways influence overall system performance, explains how to design and interpret load tests, and outlines best practices for tuning and optimizing performance.

23.1 Latency

Latency is the time it takes for a request to travel from the client to the server and for the response to return. It typically includes:

1. Network time from client to server
2. API processing time on the server, including business logic, validation, and database access
3. Network time back from server to client

In the context of APIs, latency is usually measured from the moment the client sends a request until the full response is received.

Compared to a local function call, for example within the same Java Virtual Machine, network latency is several orders of magnitude larger.

Type of Call	Typical Latency
In memory, same process (e.g., C#, Java)	~ 5-20 ns
To localhost (HTTP)	~ 0.2-1 ms
Same data center (HTTP)	~ 1-10 ms
Cross region (HTTP)	~ 25-150 ms
Mobile (HTTP)	~ 50-500+ ms

Table: Typical latencies

In distributed systems, network latency is often the dominant factor influencing overall performance. It has a major impact on architectural decisions, such as service granularity and the placement of gateways.

Low latency is important for user-facing applications such as web frontends or mobile apps, where even small delays affect perceived responsiveness.

How API Gateways Affect Latency

An API Gateway introduces an additional hop between the client and the backend. Instead of a single direct connection, there are now two network segments: client to gateway and gateway to backend.

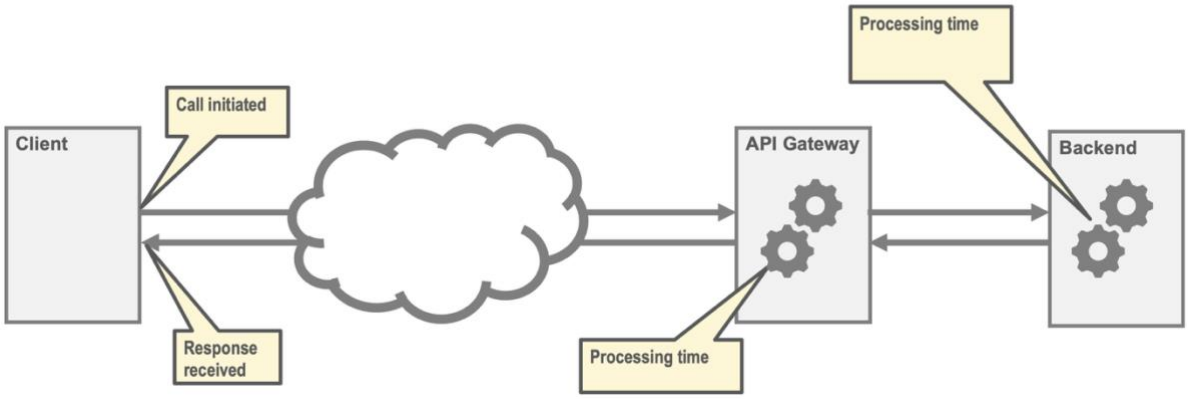


Image: Measuring the latency of an API call including API Gateway

In addition to the extra network hop, each message passes through several processing stages inside the gateway. Each step adds a small amount of delay:

1. Reading the request from the network
2. Decryption if TLS is used
3. Parsing HTTP headers and optionally deserializing the payload
4. Routing
5. Executing plugins such as validation or authentication
6. Serializing the outgoing message
7. Encryption if TLS is used

Many of these steps are performed twice: once on the request path and again on the response path.

What Is the Typical Overhead of an API Gateway?

In simple routing scenarios, most API Gateways add less than **1 to 10 milliseconds** of latency. Even with TLS termination, basic validation, and lightweight transformations, the overhead usually remains in the low millisecond range.

In many real-world systems, **backend services contribute far more to total response time** than the gateway itself. Database access and business logic are typically the dominant factors.

Performance can degrade, however, if the gateway is under heavy load or if many resource intensive plugins are enabled. Operations such as deep payload inspection, full schema validation, or complex message transformations can noticeably increase processing time.

23.2 Throughput

Throughput describes how many requests an API can process per unit of time. It is usually measured in **requests per second (RPS)**. The achievable RPS of a gateway depends on several factors:

- **Enabled plugins and processing tasks**
  Features like OpenAPI validation, JSON/XML transformations, or content filtering can noticeably affect throughput.
- **Message size and complexity**
  Small payloads under 1 KB are processed much faster than large or deeply nested JSON documents.
- **Applied security mechanisms**
  Token validation, rate limiting, and TLS encryption all add processing overhead.

Beyond configuration, hardware characteristics and the gateway implementation also influence throughput. Some **Go**-based gateways such as **KrakenD** or **Tyk** publish benchmarks exceeding 80,000 RPS. Even on my five-year-old laptop, I achieved more than 20,000 RPS using our Java-based gateway.

These numbers may appear impressive, but they are typically achieved under controlled benchmark conditions. Real-world results are often lower. Production environments include TLS, authentication, validation, and logging, all of which affect throughput and latency.

In practice, the enabled plugins and policies usually have a greater impact on performance than the gateway product itself. The more processing steps a request must pass through, the lower the achievable maximum throughput.

23.3 Load Testing

To get a realistic understanding of a gateway's performance, measure it under load. Run targeted load tests, enable or disable specific features, and observe how latency and throughput change. Modern gateways are often highly efficient at routing and JSON processing.

There are many tools available for load testing, but not all are suitable for high performance measurements. Some tools with graphical user interfaces may exhaust client-side resources before the gateway reaches its limits. In such cases, the measured throughput reflects the capacity of the load testing tool, not that of the API. It is equally important to use a backend that is capable of handling a high number of concurrent connections without becoming the bottleneck.

A practical tool for load testing HTTP servers and API Gateways is **`hey`**, a modern alternative to Apache Bench. It is lightweight, runs from the command line, and allows you to control the number of requests and the concurrency level.

Example load test with 10,000 JSON messages and a concurrency level of 100:

```
$ hey -n 10000 -c 100 -m POST -D "fruits.json" -H "Content-
Type: application/json" http://localhost:2000

Summary:
  Total:  0.3181 secs
  Slowest:     0.0272 secs
  Fastest:     0.0003 secs
  Average:     0.0031 secs
  Requests/sec:     31431.7845
```

These results provide a quick overview of latency distribution and overall throughput. They are useful for comparing configurations, evaluating the performance impact of validation or transformations, and verifying that a setup meets performance expectations.

23.4 Performance Tuning

In many projects, performance tuning is not required. Modern API Gateways are generally well optimized out of the box. Still, if you need to squeeze out additional RPS or reduce latency, the following measures can help:

- **Disable or reduce logging**
  Logging can add noticeable overhead, especially at high throughput.
- **Measure plugin performance**
  Some plugins are resource-intensive. Identify which ones add noticeable latency or limit throughput.
- **Use built-in metrics**
  Most gateways expose runtime statistics such as request rates, error counts, and response times.
- **Set up monitoring**
  Tools such as Prometheus and Grafana help visualize trends and identify bottlenecks over time.

If you want to analyze performance across multiple gateways and microservices, consider integrating **OpenTelemetry**. Distributed tracing allows you to measure each segment of the call graph and pinpoint where latency is introduced.

And if you still reach capacity limits, scale horizontally. Many API Gateways are designed to be stateless, which makes it straightforward to add more instances and distribute traffic.

Streaming Optimizations

Some gateways support streaming, where the request payload is piped to the backend without full deserialization. By bypassing body parsing and deserialization, latency can be reduced.

However, streaming is only possible when **no plugin needs access to the payload**. As soon as a plugin performs JSONPath evaluation, schema validation, or content transformation, full message processing is required, and streaming is disabled.

Resources

hey, an HTTP load generator
https://github.com/rakyll/hey

KrakenD Homepage
https://www.krakend.io/

Tyk Performance Benchmarks
https://tyk.io/performance-benchmarks/

Part II
API Gateways in Practice

This part takes a practical approach. Real-world examples illustrate key concepts and common scenarios, from basic routing and load balancing to advanced topics like request transformation, service orchestration, legacy system integration, and token validation.

As mentioned earlier, **Membrane API Gateway** will be used for demonstrations. It is a lightweight, open source tool that makes practical experimentation easy and transparent.

That said, most of the examples presented here are not specific to Membrane. The patterns and configurations apply broadly and can be adapted to other API Gateways. The goal is to explain the underlying principles in a clear, hands-on manner.

24 Membrane API Gateway

Membrane is a lightweight and flexible API Gateway designed with a strong focus on simplicity and extensibility. Its clear, declarative configuration style and broad feature set make it well suited for illustrating a wide range of API Gateway concepts and implementation techniques.

Membrane is fully open source and released under the Apache License 2.0. This permissive license allows unrestricted use, including in commercial environments, without licensing fees or usage limitations. As a result, Membrane is equally suitable for learning, prototyping, and running production-grade API infrastructures.

This section guides you through installing Membrane and getting it up and running so you can follow the examples in the next chapters. You will learn how to start Membrane and verify that it is working correctly.

The setup is straightforward and typically takes less than ten minutes.

Membrane is written in Java and supports several deployment options. Here, we focus on two approaches: the standalone Java distribution and running it as a Docker container. For other deployment options, see: https://www.membrane-api.io/deployment/

Code Samples for Copy-and-Paste

This part of the book contains many configuration and request examples. YAML requires correct indentation, but copying code from a PDF may sometimes break the formatting.

To make it easier to try out the examples, we have collected all configuration snippets, code samples, Dockerfiles, and `curl` commands in a single text file that can be copied and pasted without formatting issues.

The file can be downloaded here:

```
https://www.membrane-api.io/gw.txt
```

24.1 Standalone Java Installation

If your system already has a Java installation or if you can install one, the standalone distribution is an excellent way to get started with Membrane. It allows you to explore the gateway quickly and run the included examples.

This approach is particularly convenient for learning and experimentation. For production deployments, you can later switch to container-based setups.

Step 1: Java installation

Make sure Java 21 or newer is installed:

```
java -version
```

The output should resemble:

```
openjdk version "21.0.5" 2024-10-15 LTS
```

If Java is not installed, you can download and install it within a few minutes. Visit one of the distribution sites below and follow the instructions for your operating system.

Important: Install the JDK, not just the JRE. The JDK includes development tools that Membrane requires, for example to support Groovy scripting.

- https://adoptium.net
- https://www.azul.com/downloads/
- https://aws.amazon.com/corretto/

Step 2: Download and unzip Membrane

Download the latest version from:

https://github.com/membrane/api-gateway/releases

Once downloaded, unzip the file.

Step 3: Start Membrane

Navigate to your Membrane installation directory and start the gateway:

```
cd membrane-api-gateway-X.X.X
./membrane.sh
```

or on Windows:

```
membrane.cmd
```

After starting, Membrane lists the deployed APIs:

```
11:48:52,370  INFO 1 main HttpEndpointListener:92 {} - listening at '*:2000'
11:48:52,372  INFO 1 main HttpEndpointListener:92 {} - listening at '*:9000'
11:48:52,373  INFO 1 main ApiInfo:34 {} - Started 4 APIs:
11:48:52,373  INFO 1 main ApiInfo:36 {} -   0.0.0.0:2000 /fact
11:48:52,373  INFO 1 main ApiInfo:36 {} -   Fruit Shop API
                                              - "fruit-shop-api-v2-2-0" @ openapi/fruitshop-v2-2-0.oas.yml
11:48:52,373  INFO 1 main ApiInfo:36 {} -   0.0.0.0:2000
11:48:52,373  INFO 1 main ApiInfo:36 {} -   0.0.0.0:9000
11:48:52,374  INFO 1 main RouterCLI:419 {} - Membrane API Gateway 7.1 up and running!
```

Image: Membrane Gateway startup log

Step 4: Accessing an API

Open your browser and navigate to:

http://localhost:2000

You should see a response like this:

```
{
  "time": "2026-02-24T11:16:07.031796+01:00"
}
```

You can also try these endpoints from the command line:

```
curl http://localhost:2000/fact
curl http://localhost:2000/shop/v2/
```

In the next section, we'll take a closer look at the API configuration and explore how to customize the behavior of the gateway.

💡Troubleshooting
If these examples do not work, first verify that you are using a recent version of Membrane. Older distributions may not include the same sample APIs.

Then check the file `apis.yaml` in the `conf` folder of your local installation directory. You can also consult the latest **Getting Started Guide**:

https://www.membrane-api.io/getting-started.html

💡**Hint**: **No direct internet connection**
If the examples still fail, make sure your machine has direct internet access. Some corporate environments use outbound proxies that block external connections.

If you are behind such a proxy or have no direct internet access, review the file `offline.apis.yaml` in the `examples/configuration` folder. It explains how to run Membrane in offline mode or how to configure a proxy server.

24.2 Docker Installation

Even if Docker is part of your long-term deployment strategy, it's best to start with the full Membrane distribution running locally on Java, as described in the previous section. This makes it easier to explore the many bundled examples, most of which are not containerized and run best in a local environment.

That said, you can absolutely use Docker from the beginning if you prefer. For production deployments, Docker or Kubernetes is often the right choice, and Membrane supports both out of the box.

```
docker run -it -p 2000:2000 predic8/membrane
```

This starts Membrane using the default configuration and exposes port 2000 on the local machine.

However, to follow the examples in this book effectively, we recommend a setup that gives you direct access to the `apis.yaml` configuration file. This makes it easier to modify routes and experiment with different gateway features. The next steps show how to run Membrane with Docker while keeping the configuration under control.

Step 1: Download and unzip Membrane

Get a recent Membrane distribution from:

https://github.com/membrane/api-gateway/releases

Unzip the archive into a directory of your choice.

Step 2: Start a Membrane Container

Open a terminal and navigate to the Membrane installation folder:

```
cd membrane-api-gateway-X.X.X
```

Start the container, mounting the `apis.yaml` file from the `conf` folder. Make sure you are in the `membrane-api-gateway-*` folder:

On macOS/Linux:

```
docker run -it -p 2000:2000 \
 -v "$(pwd)/conf/apis.yaml:/opt/membrane/conf/apis.yaml" \
  predic8/membrane
```

On Windows (PowerShell or CMD):

```
docker run -it -p 2000:2000 -v
${PWD}\conf\apis.yaml:/opt/membrane/conf/apis.yaml
predic8/membrane
```

Troubleshooting

🔧 **Dockerized Samples**
Starting with version 7.1.0, the Membrane distribution includes scripts to run the samples located in the `tutorials` folder in a Docker container.

🔧 **Note: PWD**
If *`${PWD}`* does not work in PowerShell, replace it with the full path manually, e.g. *`C:\Users\YourName\Downloads\membrane-api-gateway-7.0.5\conf\apis.yaml`*

🔧 **Right Directory**
Verify that `conf/apis.yaml` is accessable.

Step 3: Testing the Installation

Open the following URL in the browser:

http://localhost:2000

You should see a JSON document like this:

```
{
  "time": "2026-02-24T11:16:07.031796+01:00"
}
```

💡**Hint:** If the browser shows a connection error, make sure Docker is running, and you have a working internet connection.

25 API Configuration

Membrane's behavior is configured in the `apis.yaml` file located in the `conf` folder. You can edit this file with any text editor. However, using an IDE or an editor with YAML support, such as **Visual Studio Code** or **IntelliJ** is strongly recommended. These editors provide useful YAML features, including:

- Autocompletion (CTRL+SPACE)
- Syntax highlighting and validation
- Inline documentation based on the JSON Schema declared at the top of the configuration file

These capabilities make configuration more efficient and help prevent common mistakes such as indentation errors or invalid attributes.

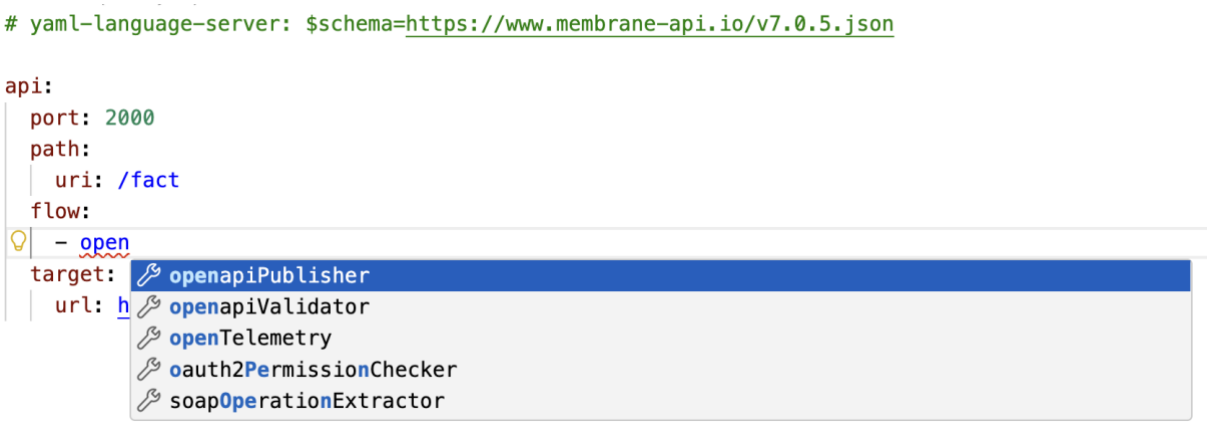


Image: Help from autocompletion

First API Configuration

The following steps show how to extend a basic API configuration to enable logging of request and response messages.

Step 1: Open apis.yaml

Navigate to the `conf` subfolder in the Membrane distribution and open the `apis.yaml` file in your preferred editor.

Step 2: Create an API

We will start from the ground up. Delete all lines in `apis.yaml` except the schema declaration on the first line. Then define the following API. Use autocompletion in the editor to reduce typing and avoid mistakes.

```
# yaml-language-server: $schema=https://www.membrane-
api.io/v7.1.1.json

api:
  port: 2000
  target:
    url: https://api.predic8.de
```

This minimal configuration tells Membrane to listen on port `2000` and forward incoming requests to the backend at `https://api.predic8.de`.

Step 3: Save and Start Membrane

Make sure to save your changes in `apis.yaml`. Then open a terminal in the installation directory and start Membrane.

On Linux or Mac:

```
cd membrane-api-gateway-X.X.X
./membrane.sh
```

On Windows:

```
cd membrane-api-gateway-X.X.X
membrane.cmd
```

Once started, Membrane listens on port `2000` for incoming connections.

Step 4: Test the API

Open a browser and navigate to:

http://localhost:2000

If everything is set up correctly, you should see a JSON response returned from the backend service.

Finding Configuration Errors

If the `apis.yaml` file contains an error, Membrane will refuse to start and report the issue in the log output. The error message usually points to the problematic section and helps identify the issue.

For example, the following message indicates a typo in the configuration:

```
20:27:34,873 ERROR 1 main RouterCLI:135 {} - Invalid field 'fort' in api
Config:
  api:
>   fort: 2000
    path:
      uri: "/token"
    flow:
    - request:
      - $ref: "#/components/get-token"
      - log: {}
    - return:
        status: 200
```

In this case, `fort` is not a valid configuration attribute; it should be `port`. Although Membrane provides helpful diagnostics, YAML or schema validation errors can still be difficult to interpret, especially when indentation or overall structure is incorrect.

Fortunately, modern editors such as IntelliJ IDEA or Visual Studio Code offer YAML support with schema validation. In some cases, you may need to install a YAML plugin first to take advantage of these features.

Image: YAML plugin for Visual Studio Code

With the plugin enabled, errors are highlighted in the editor while you type, often before you even start Membrane. This saves time and reduces unnecessary restarts.

To benefit from this feature, keep the first line that references the JSON Schema describing the configuration format. This allows the editor to validate the file against the correct schema.

```
# yaml-language-server: $schema=https://www.membrane-api.io/v7.0.5.json

ap  Property fort is not allowed. yaml-schema: membrane(513)
    View Problem (⌥F8)    Quick Fix... (⌘.)
  fort: 2000
  target:
    url: https://api.predic8.de
```

Image: YAML validation error displayed in editor

26 Routing Traffic

Routing forwards incoming requests to backend services based on defined matching criteria. These criteria typically include the HTTP method, request path, or other attributes.

The API configuration below listens on port 2000 for `GET` requests whose path starts with `/shop/v2` and forwards them to the backend at `api.predic8.de`:

```
api:
  port: 2000
  method: GET
  path:
    uri: /shop/v2
  target:
    url: https://api.predic8.de
```

You can try this configuration yourself. After saving the `apis.yaml` file and starting Membrane, test the route from the command line:

```
curl http://localhost:2000/shop/v2/
```

Alternatively, you can use the **REST Client** extension in **Visual Studio Code** to send requests and inspect responses directly from your editor.

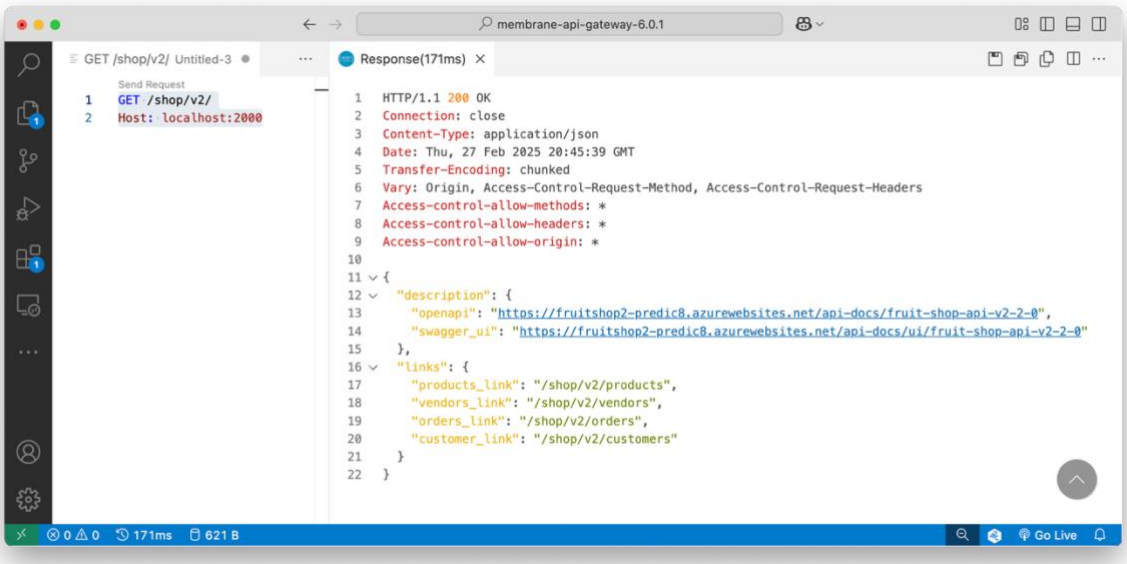


Image: Using Visual Studio Code as API client with the **REST Client** plugin

Order of API Matching

Gateways often manage dozens or even hundreds of APIs. When a request arrives, the gateway evaluates the API definitions from top to bottom. The first API whose matching criteria are satisfied is selected and handles the request.

This means the order of API definitions matters. More specific routes should usually appear before more generic ones to avoid unintended matches.

Consider the following example with two APIs:

```
api:
  name: API 1
  port: 2000
  method: GET
  flow:
    - log:
        message: "Method: ${method}"
  target:
    url: https://api.predic8.de

---
api:
  name: API 2
  port: 2000
  method: POST
  flow:
    - log:
        message: "Method: ${method}"
  target:
    url: https://api.predic8.de
```

(To avoid typing, download the samples from `https://membrane-api.io/gw.txt`)

Note that the APIs are separated by a line containing three dashes (---), which is the YAML document separator.

Now let’s test how requests are routed with this configuration.

Case 1: Sending a GET request

After sending this request:

```
GET /shop/v2/products
Host: localhost:2000
```

or by using curl:

```
curl http://localhost:2000/shop/v2/products
```

The console displays two entries:

```
10:38:33,081  INFO 69 ... {api=API 1} - Method: GET
10:38:33,220  INFO 69 ... {api=API 1} - Method: GET
```

API 1 is selected because it is the first matching API in the configuration.

Why are there two entries? The `log` interceptor runs once during the request and once during the response flow. We will explore flows in detail later.

Case 2: Sending a POST request

Now, let's look at a POST request:

```
POST /shop/v2/products
Host: localhost:2000
Content-Type: application/json

{
  "name": "Biscuits", "price": 1.99
}
```

or:

```
curl -X POST -H 'Content-Type: application/json' \
     -d '{"name": "Biscuits", "price": 1.99}' \
     http://localhost:2000/shop/v2/products
```

Here's what happens:

- **API 1** is skipped because it requires `GET`.
- **API 2** matches the `POST` request

The log output confirms this:

```
... {api=API 2} - Method: POST
```

Case 3: Sending a request with an unsupported method

When sending:

```
PUT / HTTP/1.1
Host: localhost:2000
```

Neither API matches, so Membrane returns: **404 Not Found**.

Default API for Unmatched Requests

In some cases, it is useful to define a fallback for requests that do not match any API. This can be done by placing a default API at the end of `apis.yaml` to catch all unmatched requests.

```
api:
  port: 2000
  flow:
    - static:
        src: Not on my server!
    - return:
        status: 404
```

The `return` interceptor reverses the flow and sends a response immediately. Return handling and short-circuiting are explained later in this chapter.

💡**Tip**: **Custom not found**
Place a fallback API **after** all other definitions to provide a custom error message.

Routing Criteria

An API Gateway can base routing decisions on almost any part of a client's request, not just the path, method, or port. Routing criteria include:

Criteria	Description
Port	Port on which the request was received
IP	IP address of the client
Method	The HTTP method used (e.g., GET, POST)
Host	The value of the **Host** header. Useful for supporting multiple virtual hosts.
Path	The requested path (e.g., `/shop/v2/`)
Header	Any HTTP header field
Content	The request payload

Table: Routing criteria

Content-Based Routing Pattern

Routing decisions do not have to rely only on method or path. An API Gateway can also inspect the request body and route traffic based on its content.

The following example evaluates a JSON field in the request body:

```
api:
  port: 2000
  test: json['name'] == 'Lolly'
  flow:
    - static:
        src: "Sweets this way!"
    - return:
        status: 200
```

For this API to be selected, the request must contain a JSON body with an object whose `name` field has the value `Lolly`. The condition in the `test` attribute is written using the **Spring Expression Language (SpEL**).

Try it with the following request:

```
POST /shop/v2/products
Host: localhost:2000
Content-Type: application/json

{
  "name": "Lolly"
}
```

Send the request and inspect the response.

💡**Sidenote: Test expressions**
Membrane supports **SpEL, JSONPath, XPath,** and **Groovy** expressions in the `test` attribute to route based on headers, query parameters, or message body content. See the chapter on expression languages for details.

26.1 Short-Circuit Routing

Short-circuit routing is a simple but powerful communication pattern in which the gateway skips the backend call and immediately returns a response to the client. It allows the API Gateway to act as a self-contained endpoint, handling requests entirely on its own without involving downstream services.

This pattern is useful in several scenarios:

- **Mocking and testing**
  Simulate a backend service that does not exist yet.
- **Early error handling**
  Stop request processing when a precondition is not met, for example when authentication fails.
- **Lightweight APIs**
  Implement simple endpoints directly at the gateway, such as health checks, status endpoints, or feature flags.

As the diagram illustrates, the request flow is connected directly to the response flow. No backend call is made in between.

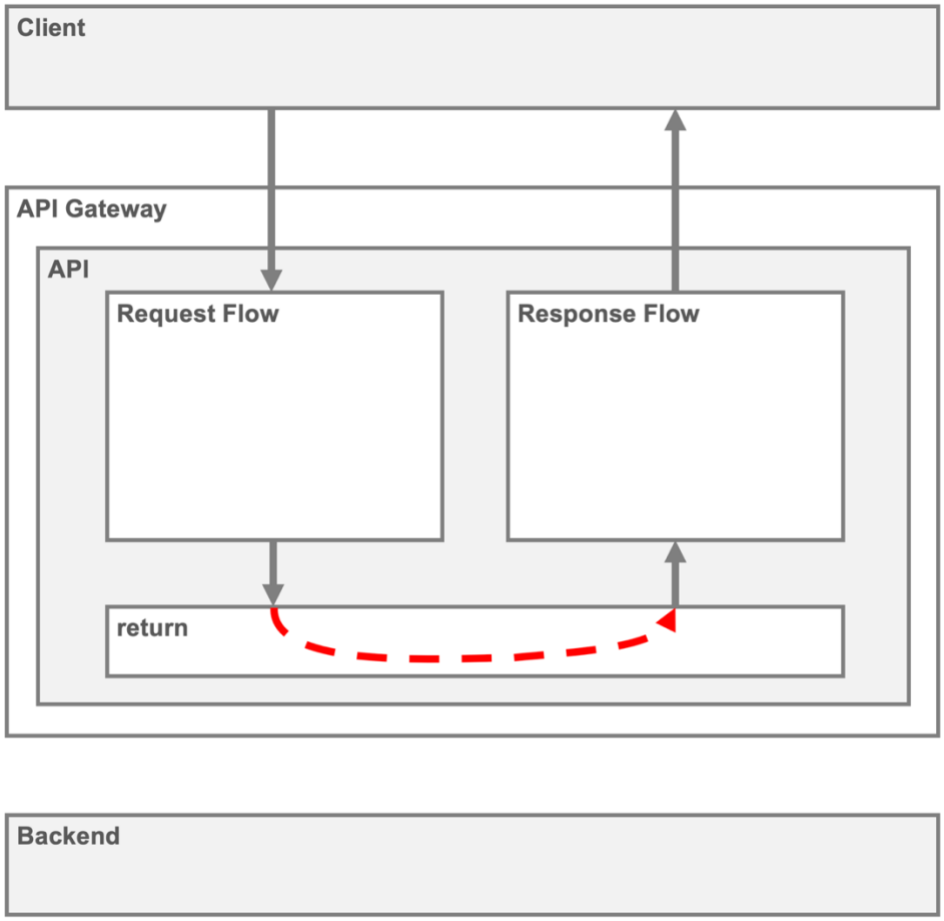


Image: Short-circuit routing

Short-circuit routing is used throughout this book to simplify examples.

Temporarily Blocking Access

One application of short-circuit routing is blocking access to an endpoint for a limited time, for example during maintenance or while handling an incident.

The following configuration blocks all requests to `/products` and immediately returns an error response from the gateway:

```
api:
  port: 2000
  path:
    uri: /products
  flow:
    - request:
        - static:
            contentType: application/json
            src: |
              {
                "message": "Temporarily unavailable"
              }
        - return:
            status: 503
```

If this blocking API is placed before the regular API in the configuration file, it will be matched first and effectively override the normal route. Once the backend is available again, remove or comment out this configuration.

26.2 URI Templates

In OpenAPI and many REST frameworks, URI templates allow capturing values from the request path using placeholders. This technique is equally useful in API Gateways.

For example, the template:

`/products/{pid}`

matches requests such as:

`/products/7`

The value `7` is extracted and assigned to the placeholder `pid`. In Membrane, it becomes available via the `pathParam` object.

URI templates can also serve as routing criteria. The following configuration routes requests based on the structure of the incoming path:

```
api:
  port: 2000
  path:
    uri: /customers/{cid}/orders/{oid}
  flow:
    - log:
        message: Order ${pathParam.oid} for ${pathParam.cid}
    - return:
        status: 200

---
api:
  port: 2000
  path:
    uri: /customers/{cid}
  flow:
    - log:
        message: 'Customer: ${pathParam.cid}'
    - return:
        status: 200
```

With this setup, the following sample requests produce these log statements:

- **GET http://localhost:2000/customers/88**
  Log: "Customer: 88"

- **GET http://localhost:2000/customers/88/orders/3**
  Log: "Order 3 for 88"

26.3 Naming APIs

Clear and descriptive names make it easier to keep track of APIs, especially when a gateway handles dozens or even hundreds of them.

If no name is assigned, Membrane generates one automatically based on routing parameters, for example:

```
0.0.0.0:2000 /shop/v2
```

While this works, it is neither easy to read nor easy to remember. Giving each API a distinct name improves clarity.

```
api:
  name: Fruitshop
  path:
    uri: /shop/v2
```

API names appear in **logs, monitoring dashboards**, and the **web console**:

```
13:11:13 INFO 5 Log:148 {api=Fruitshop} - Path: /shop/v2/
```

Named APIs make debugging and monitoring easier. Instead of decoding port and path combinations, you immediately see which API handled the request.

27 Message and Exchange

Most API Gateways wrap an incoming request and its corresponding response into a shared data structure. In Membrane, this structure is called an **exchange**. The exchange object is the central hub through which the gateway and its plugins read and modify messages.

This design simplifies processing and enables features such as authentication, rate limiting, and message transformation using a single unified object.

When a request arrives, the gateway wraps it in an exchange and sends the exchange through the **request flow**. At this point, the exchange contains only the request. After the backend replies, the response is added to the exchange and processing continues through the **response flow**.

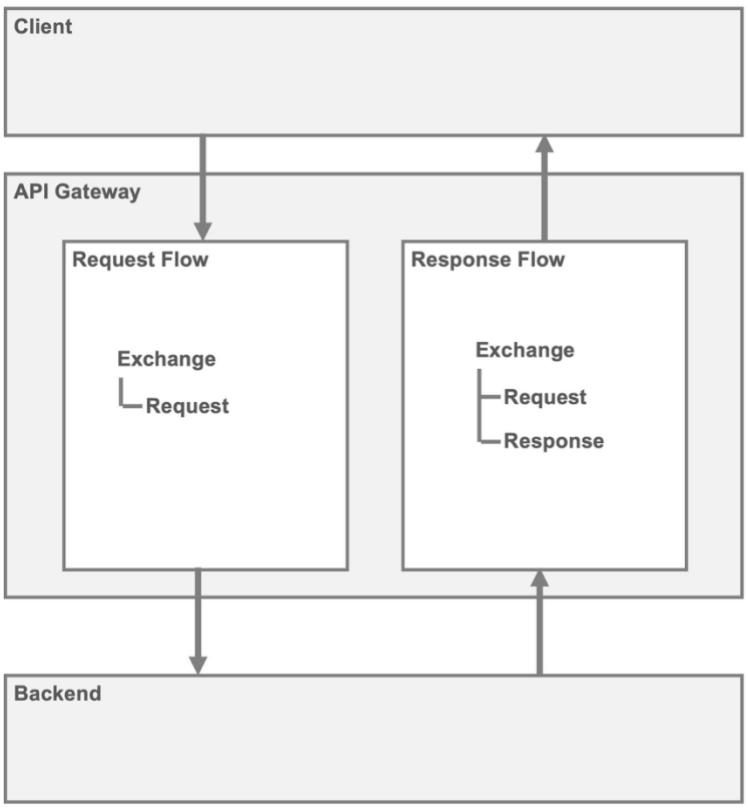


Image: Structure of an exchange in the request and response flow

The request and response parts of the exchange look similar:

- Both contain HTTP headers
- Both can carry a body (e.g., JSON, XML, binary data)

However, they also differ:

- The **request** includes the HTTP **method** and the **path**
- The **response** adds the HTTP **status code**

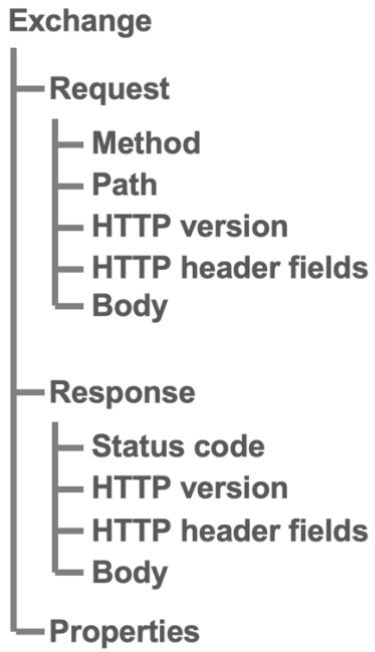


Image: Structure of an exchange

In addition to the request and response, an exchange also maintains a set of **properties**. These are **key-value pairs** used to **share information across plugins** and processing stages. For example, one plugin can store the result of authentication so that later plugins can use it for permission checks. Other plugins may use properties for dynamic routing or for tracking custom metrics.

The next section explains how exchange properties enable multiple plugins to work together.

Exchange Properties

APIs often execute multiple plugins in sequence, with each plugin performing a specific task. For example, one plugin extracts a value from the request, and a subsequent plugin uses that value to build a JSON message. This chain of operations requires a mechanism for sharing data across processing stages. That is where **exchange properties** come into play.

Consider the following two APIs. The first API listens on port `2000`. It sets an exchange property named `color` and an HTTP header named `X-Size`. It then forwards the request to a second API running on port `2001`, which simulates a backend.

```
api:
  port: 2000
  flow:
    - setProperty:
        name: color
        value: RED
    - setHeader:
        name: X-Size
        value: XL
    - log:
        message: |
          Header ${header['X-Size']} Prop ${property.color}
  target:
    url: http://localhost:2001

---
api:
  port: 2001
  flow:
    - request:
      - log:
          message: |
            Header ${header['X-Size']} Prop ${property.color}
      - return:
          status: 200
```

After calling the API on port `2000`, the following log entries are produced:

```
{api=0.0.0.0:2000} - Header XL Prop RED
{api=0.0.0.0:2001} - Header XL Prop null
{api=0.0.0.0:2000} - Header null Prop RED
```

The first log entry comes from the API listening on port `2000`. At this point, both the HTTP header and the exchange property are available. The `X-Size` header is then forwarded with the request to the API on port `2001`.

```
GET / HTTP/1.1
Host: localhost:2001
X-Size: XL
```

As shown in the second log entry, the API on port `2001` can access the `X-Size` header passed from the first API. However, the exchange property `color` is not available, because exchange properties are local to a single API and are not transmitted over HTTP.

The third log entry is produced when control returns to the API on port `2000` and the response flow is executed. At this point, the `${header}` expression refers to the **response headers**, which do not contain `X-Size`. The exchange property `color`, however, is still available within the first API and can be accessed.

Without storing the value as an exchange property, it would not be possible to access it in the response flow of the first API. **Exchange properties connect the request flow to the response flow**. By sharing properties between both flows, complex plugins can be built. For example, you could implement a plugin that counts the number of concurrent backend connections.

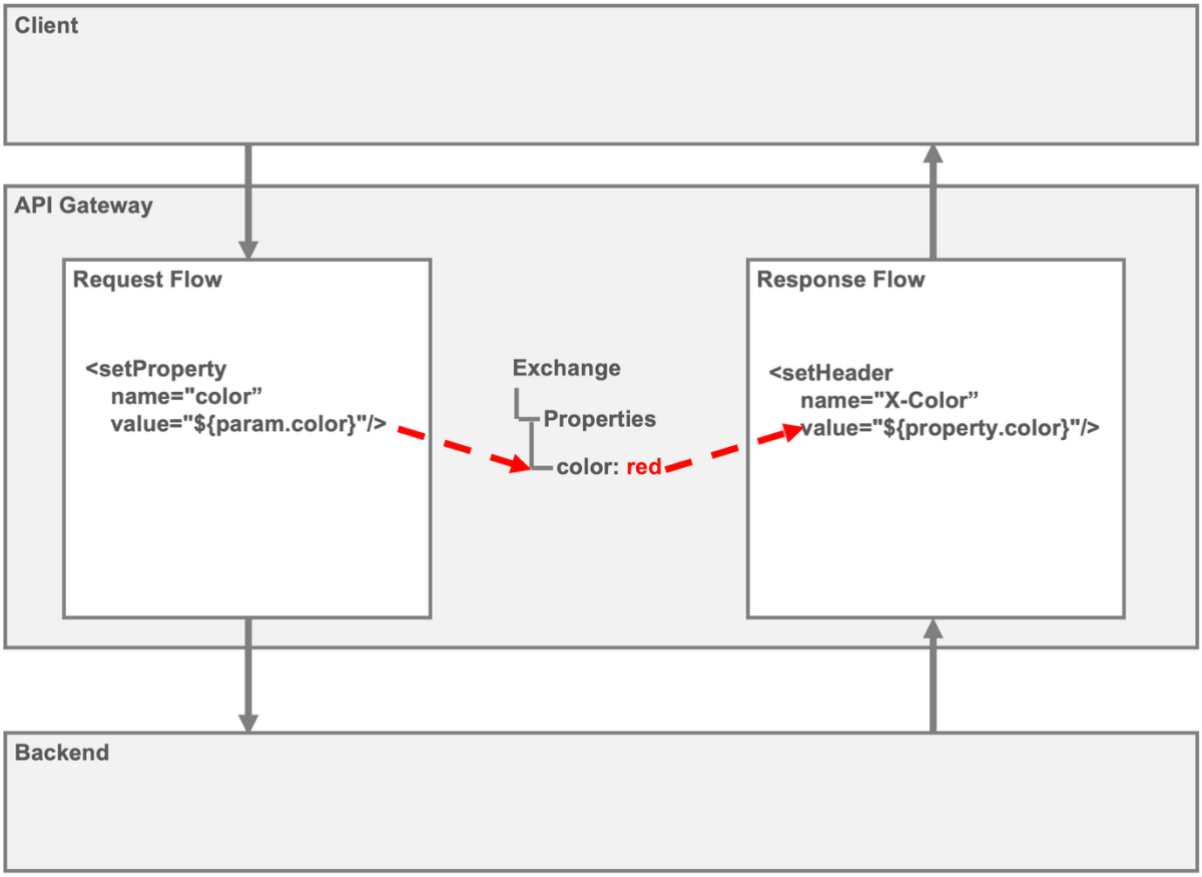


Image: Shared property between request and response flow

28 OpenAPI

In this chapter, we explore how to use OpenAPI documents with an API Gateway. You will learn how to:

- Automatically set up APIs using OpenAPI as configuration
- Validate messages against an OpenAPI definition
- Establish APIOps best practices by using OpenAPI
- Let the gateway rewrite server addresses inside OpenAPI documents

Native OpenAPI support at the gateway streamlines workflows, improves reliability, and reduces manual effort when managing APIs at scale.

📖 **Background first?**
This chapter builds on the concepts introduced in Chapter 6: *OpenAPI*. If you are new to OpenAPI or want an overview, start there before continuing.

28.1 Using OpenAPI for Gateway Configuration

OpenAPI documents describe an API's paths, methods, parameters, and data formats. They can also include information about security mechanisms, versioning, and deployment environments. This information is not limited to documentation. It can be used directly to configure an API Gateway. By relying on the OpenAPI specification, the runtime behavior of the gateway stays aligned with the declared contract. This reduces configuration errors and simplifies setup.

Consider the following example. Here, the gateway configuration is driven almost entirely by an OpenAPI document. The only additional information required is the port on which the gateway should listen:

```
api:
  port: 2000
  openapi:
    - location: openapi/dlp-v1.0.0.oas.yml
```

Place this API definition in `conf/apis.yaml`. The `openapi` folder is located relative to `conf/`, so the gateway can load the specification directly.

Once deployed, open the generated overview page in the browser:

```
http://localhost:2000/api-docs
```

The gateway reads the OpenAPI file, extracts metadata such as the title, version, and paths, and then presents them in an overview page.

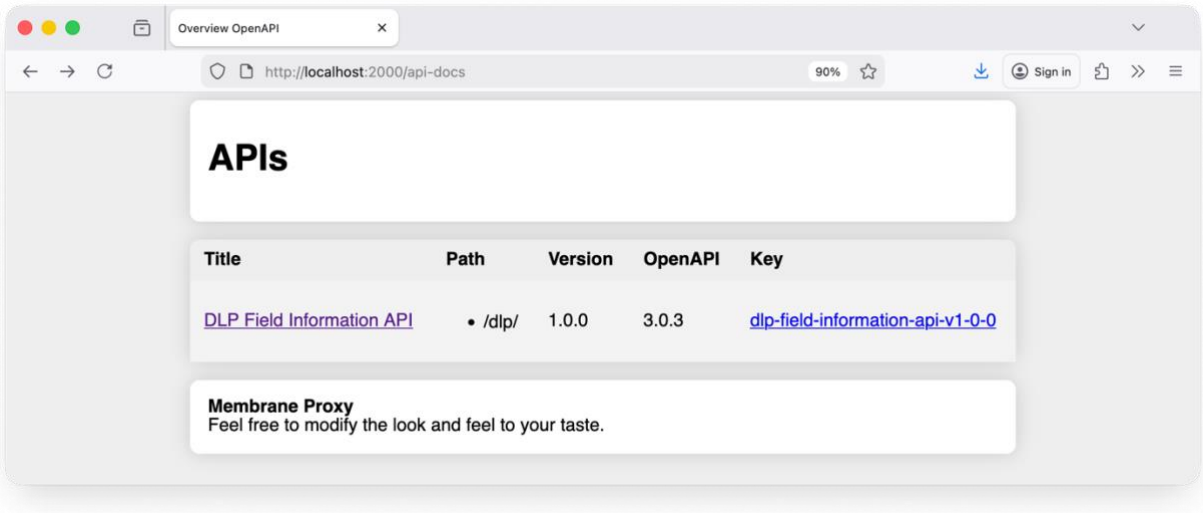


Image: Overview of APIs deployed from OpenAPI

From here, developers can download the OpenAPI document or launch **Swagger UI** to explore and test the API.

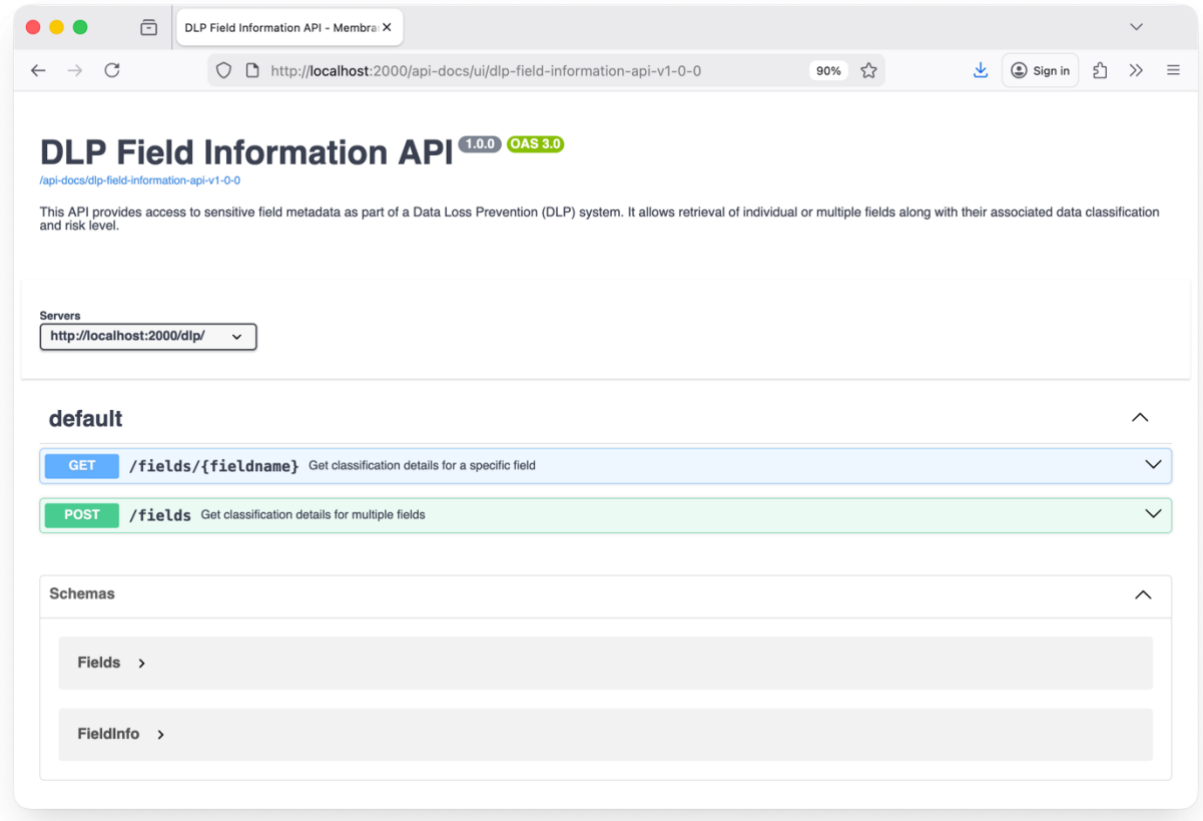


Image: Swagger UI with deployed OpenAPI

Loading Multiple APIs

Each OpenAPI document can be specified individually:

```
api:
  port: 2000
  openapi:
    - location: openapi/dlp-v1.0.0.oas.yml
    - location: openapi/fruitshop-v2-2-0.oas.yml
```

However, if you are exposing dozens of APIs, there is a simpler and more scalable approach:

```
openapi:
  - dir: conf/openapi
```

This configuration scans the `conf/openapi` directory and automatically loads every OpenAPI file it finds there.

Referencing Remote Documents

Membrane can also fetch OpenAPI specs from remote locations. Just provide a URL:

```
location: https://api.predic8.de/api-docs/fruit-shop-api-v2-2-0
```

This is useful when OpenAPI documents are **stored in a central repository** or managed in a version control system like **Git**. When the specification is updated, you can roll out the new configuration by restarting the gateway without modifying local configuration files.

💡 **Sidenote: Declarative configuration**
Declarative configuration is a cornerstone of modern API operations. When the same specification defines both documentation and runtime behavior, it becomes much easier to ensure consistency, validate changes, and automate deployments through CI/CD pipelines.

28.2 Rewriting of OpenAPI Addresses

As described in *Part I, Chapter 6.2* OpenAPI URL Rewriting, the `servers` field in an OpenAPI document must represent the **public** address of the API, not the backend address. Otherwise, clients generated from that OpenAPI document might bypass the gateway and connect directly to the backend.

To prevent this, Membrane automatically rewrites the `servers` section. It derives the protocol, hostname, and port from the incoming request and replaces the corresponding entries in the OpenAPI document before returning it to the client.

This ensures that clients always communicate with the gateway, preserving routing, security policies, and observability controls enforced at the gateway.

The table shows how the rewritten URLs depend on the incoming request:

Request to Fetch an OpenAPI	Rewritten URL in the Returned OpenAPI
http://localhost:80/shop-v1.oas.yml	http://localhost:80/shop/v2
https://api.predic8.de/shop-v1.oas.yml	https://api.predic8.de/shop/v2

This behavior works well in development environments, where internal and external addresses are often the same.

In real-world deployments, however, gateways typically sit behind firewalls, inside private networks, or run in containers. The address visible to external clients may differ from the gateway's internal address.

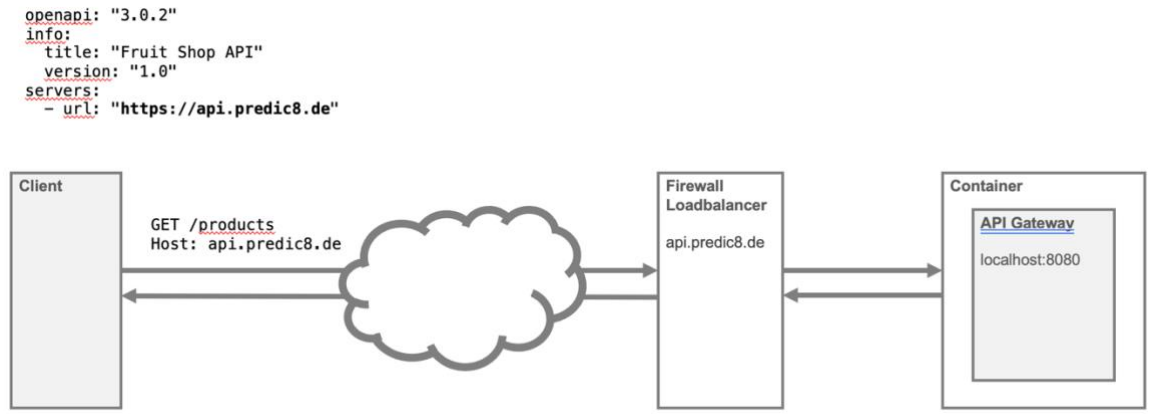


Image: External and internal address of an API

To ensure that external clients receive the correct **public-facing address**, you can override the default rewrite behavior with explicit parameters:

```
openapi:
  - location: openapi/fruitshop-v2-2-0.oas.yml
    rewrite:
      host: api.predic8.de
      protocol: https
      port: 443
      basePath: /store
```

With this configuration, the gateway serves an OpenAPI document that contains the following public endpoint:

```
servers:
  - url: https://api.predic8.de/store
```

With dynamic rewriting, the gateway can retrieve the current OpenAPI document from the backend and rewrite only the address information. This avoids duplication and keeps the specification aligned with the actual API.

💡**Sidenote**: **Why not serve a manually updated OpenAPI?**
You could, but this quickly creates manual maintenance overhead. Whenever the backend API changes, for example when a new endpoint is added, both the backend and the stored OpenAPI document at the gateway must be updated.

28.3 OpenAPI Message Validation

Because OpenAPI describes messages in detail, it can serve as a contract against which messages can be validated.

To enable validation in Membrane, set the corresponding option in the `openapi` configuration:

```
api:
  port: 2000
  openapi:
    - location: openapi/fruitshop-v2-2-0.oas.yml
      validateRequests: true
```

Once enabled, only requests that conform to the OpenAPI contract are accepted.

In the **Fruitshop** OpenAPI document, the `price` property is defined as a non-negative number:

```
price:
  type: number
  minimum: 0
```

If a client sends a negative price, the gateway rejects the request and returns a detailed validation error:

```
HTTP/1.1 400 Bad Request
Content-Length: 640
Content-Type: application/problem+json

{
  "title": "OpenAPI message validation failed",
  "type": "https://membrane-api.io/problems/user/validation",
  "validation": {
    "errors": {
      "REQUEST/BODY#/price": [
        {
          "message": "-10 is smaller than the minimum of 0"
        }
      ]
    }
  }
}
```

Membrane uses **Problem Details for HTTP APIs (RFC 9457)** for validation errors. This standardized format includes structured fields that help clients understand and correct issues.

By default, Membrane provides a detailed explanation of what went wrong, including which part of the request failed and why. While this is great for debugging and development, it might reveal too much information in a production environment.

If you prefer to keep error messages more generic, you can suppress detailed output using the `validationDetails` attribute:

```
openapi:
  - location: openapi/fruitshop-v2-2-0.oas.yml
    validateRequests: true
    validationDetails: false
```

💡Sidenote: Why OpenAPI validation is disabled by default
Validation is not disabled for performance concerns. For typical request sizes and configurations, the overhead is minimal.

It is disabled to avoid unexpected behavior. When response validation is enabled, error messages returned by the backend may be blocked if they do not match the OpenAPI contract. From the client's perspective, this can look like the gateway is causing the problem, even though the root cause lies in the backend.

Keeping validation off by default provides a smoother initial experience. It allows you to introduce request and response validation deliberately, once your API contracts and error formats are clearly defined and aligned.

Response Validation

While request validation is widely used, response validation is often overlooked. It is just as important for catching backend bugs, improving client compatibility, and preventing accidental information disclosure.

In Membrane, response validation can be enabled independently of request validation:

```
openapi:
  - location: openapi/fruitshop-v2-2-0.oas.yml
    validateRequests: true
    validateResponses: true
```

When enabled, the gateway verifies that responses conform to the OpenAPI definition, including status codes, content type, and payload structure.

A detailed discussion of response validation and its security implications can be found in Chapter *13.1* Response Validation.

28.4 APIOps with OpenAPI

APIOps applies DevOps principles to the API lifecycle. It emphasizes automation, testing, and repeatable processes from design and development to deployment and monitoring.

When OpenAPI specifications are integrated into CI/CD pipelines, any change to an API can automatically trigger validation, testing, and deployment steps. This reduces manual effort and increases consistency.

Gateways such as Membrane support this model by allowing APIs to be deployed directly from OpenAPI documents. This keeps implementation and documentation aligned.

Deploying Membrane with OpenAPI in Docker

Membrane can be packaged as a container image that already includes one or more OpenAPI descriptions. This approach enables controlled rollouts via CI and CD pipelines, supports versioning, and makes API definitions immutable.

The following Dockerfile builds a Membrane image with a predeployed API definition based on an OpenAPI document hosted on GitHub:

```
FROM predic8/membrane

USER root

RUN apt-get update && \
    apt-get install -y wget && \
    wget "https://github.com/predic8/rfq-
api/releases/latest/download/rfq-api-v1.oas.yml" \
    -O /opt/membrane/conf/openapi/rfq.oas.yml

USER membrane

COPY apis.yaml /opt/membrane/conf

EXPOSE 2000

ENTRYPOINT ["/opt/membrane/membrane.sh"]
```

Let’s break down what this Dockerfile does:

- **Base image:** `predic8/membrane` provides the Membrane runtime.
- **Switch to root:** Required to install additional packages.
- **Install tools:** Updates package lists and installs `wget`.
- **Download OpenAPI spec:** Retrieves the latest OpenAPI document from GitHub and stores it in the `conf/openapi` directory.
- **Switch to non-root:** Avoids running the container with elevated privileges.
- **Add custom configuration:** Copies the `apis.yaml` configuration into the container.

- **Expose port:** Exposes port 2000.
- **Entrypoint:** Starts Membrane when the container launches.

The `apis.yaml` file, copied into the image, defines the gateway configuration. It should be version-controlled together with the OpenAPI specification. In this example, the configuration deploys all OpenAPI definitions found in the `conf/openapi` directory:

```
api:
  port: 2000
  openapi:
    - dir: conf/openapi
```

The image can now be built and run as a container:

```
docker build -t membrane:1 .
docker run -it -p 2000:2000 membrane:1
```

Once the container is running, open the following URL in the browser:

http://localhost:2000/api-docs

Membrane provides a simple UI listing deployed APIs.

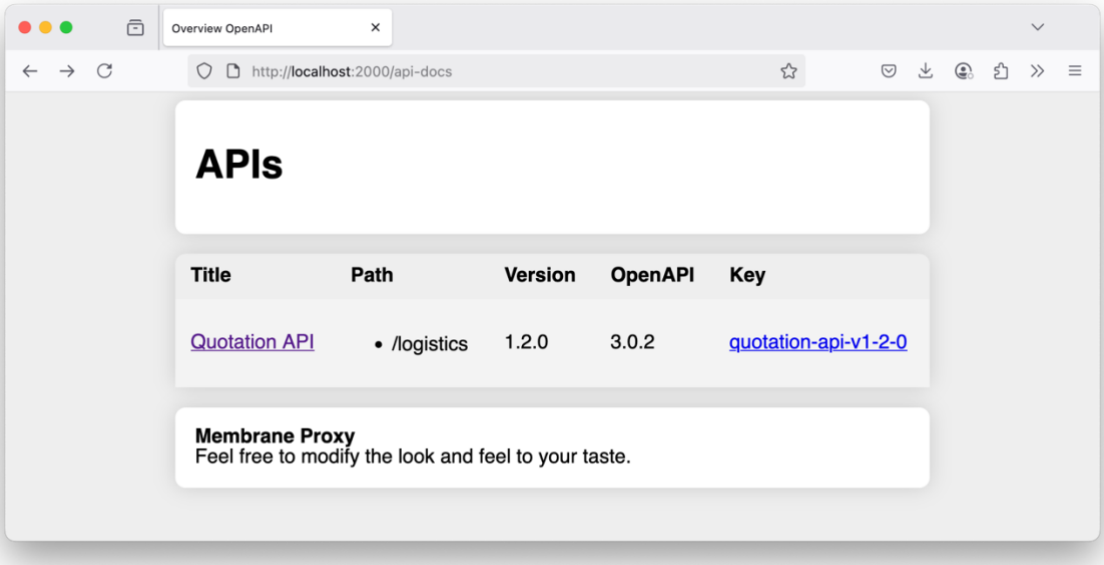


Image: List of APIs deployed from OpenAPI in the container

Storing OpenAPI Descriptions in Git

Managing OpenAPI documents in a source code repository unlocks automation and collaboration. When the OpenAPI YAML or JSON file is treated as code, it can be integrated seamlessly into CI and CD pipelines.

For example, a GitHub Actions workflow or GitLab CI job can be triggered whenever the OpenAPI specification or the gateway configuration changes on the main branch. The pipeline can then build and publish a new API Gateway Docker image and optionally deploy it to a staging environment.

Using Git allows API changes to be reviewed through pull requests. Team members can discuss and approve modifications to the API contract before they are merged into the main branch. This helps prevent accidental breaking changes and ensures updates are intentional and documented.

Storing OpenAPI files in Git supports collaborative API design, automated deployments, and consistent rollouts across environments. When a pull request is merged, the pipeline can deploy the updated specification with confidence, knowing it has been reviewed and tested. This reduces surprises in production and shortens the feedback cycle during API development.

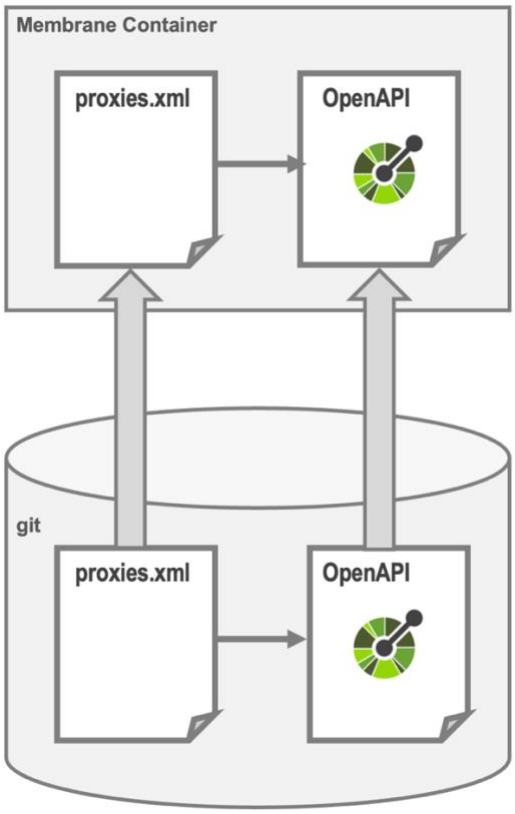


Image: Container from versioned configuration and API description

28.4.1 Best Practices for OpenAPI Deployments

When containerizing an API Gateway that includes OpenAPI specifications, consider the following best practices to improve reliability, security, and maintainability:

Keep gateway configuration and OpenAPI specifications under version control
Store the gateway configuration and the OpenAPI files in Git. This way, changes to the API contract are tracked and can undergo peer review just like code changes. This also enables CI/CD pipelines to automatically rebuild and redeploy the gateway whenever the specification or configuration changes. This keeps the gateway in sync with the API contract.

Pin versions for reproducibility
Avoid floating versions such as `latest` for base images or external downloads. Use a fixed version of the gateway's base image and reference a specific release of the OpenAPI document. Pinning dependencies ensures reproducible builds and prevents surprises.

Minimize the image footprint
Remove build-time tools and clean up caches in the same Docker layer. A smaller image reduces the attack surface and speeds up deployments.

Run as non-root
Switch to an unprivileged user for the gateway. If the process is compromised, the impact is limited to the container and its user privileges. Do not run the gateway as the root user.

Use health checks and monitoring
When running in Docker or orchestration platforms such as Kubernetes, configure liveness and readiness probes. Combine this with monitoring to detect failures early.

Plan for API versioning
Version OpenAPI files explicitly, for example `rfq-api-v2.1.1.oas.yml`. A gateway can host multiple versions at the same time. Use different base paths that include a version number for separation.

Automate testing
Include automated tests in the pipeline. After building the container, start it in a CI environment and execute contract tests or sample requests. Use OpenAPI linters such as Spectral or openapi-cli to validate the specification before deployment.

Avoid bundling secrets into the image
Never bake credentials into Docker images. Inject them at runtime using environment variables, Docker secrets, or Kubernetes mechanisms. This reduces the risk of accidental exposure.

Regularly update dependencies
Monitor updates for the gateway and its base image. Updating to a newer stable version provides security patches and improvements. With CI/CD in place, rolling out an updated gateway becomes a controlled and repeatable process that requires little effort.

By following these practices, deploying an API Gateway with OpenAPI becomes part of your standard delivery process. The gateway benefits from the same discipline in versioning, testing, and automation as the rest of your system.

Resources

Should I Set docker image version in docker-compose? (Stackoverflow)
https://stackoverflow.com/questions/70424052/should-i-set-docker-image-version-in-docker-compose#:~:text=

29 Transformation and Message Manipulation

Message transformations in an API Gateway range from simple operations, such as adding or modifying HTTP headers, to more advanced conversions of JSON or XML payloads.

These transformations make it possible to adapt requests and responses to specific client or integration requirements without modifying the backend services. This separation keeps backend implementations stable while allowing flexible adjustments at the edge.

29.1 HTTP Header Manipulation

Let's begin with a simple but practical example: adding and removing HTTP header fields.

Adding an HTTP Header

API Gateways often add HTTP headers to outgoing messages, for example to enable CORS or to supply credentials.

The following configuration adds a custom header to the response:

```
api:
  port: 2000
  flow:
    - response:
        - setHeader:
            name: X-Foo
            value: '42'
    - return:
        status: 200
```

When `setHeader` is placed in the **response** flow, the header is added to the response message. The quotes ensure that 42 is interpreted as a string rather than a numeric value.

To test this configuration, use `curl` with the `-i` option to display HTTP headers:

```
curl -i http://localhost:2000
```

The response now includes the custom header:

```
HTTP/1.1 200 OK
Content-Type: text/plain
X-Foo: 42
```

29.2 Passing HTTP Headers to Backends

An API Gateway can forward HTTP headers to an upstream backend. This technique is commonly used to propagate authentication information, correlation IDs, or tenant identifiers across service boundaries.

In the example below, this behavior is demonstrated using two APIs.

Gateway API: Adding a Token

The first API listens on port `2000`. It adds a header named `X-Token` to requests going to the backend on port `3000`:

```
api:
  port: 2000
  flow:
    - request:
        - setHeader:
            name: X-Token
            value: abc123
  target:
    url: http://localhost:3000
```

Backend: Logging the Token

The second API runs on port `3000` and simulates a backend service. It logs the received token.

```
api:
  port: 3000
  flow:
    - request:
        - log:
            message: 'Got: ${header["X-Token"]}'
        - return:
            status: 200
```

For demonstration purposes, both APIs can be deployed on the same Membrane instance. Even though they run on the same machine, the request is routed through the operating system's network stack and back.

Testing the Configuration

To test this setup, send a request to port `2000`:

```
curl http://localhost:2000
```

The console log should look similar to this:

```
14:42:06 INFO 70 {api=0.0.0.0:3000} - Got: abc123
```

Although this example is simple, HTTP headers are widely used in real-world systems for security, correlation, and tracing.

💡**Note:** Sensitive tokens are logged here for demonstration purposes only. In production environments, sensitive information should never be written to logs.

29.3 Computing Header and Property Values

Header and property values can be computed dynamically using expression languages such as **Groovy**, **JSONPath**, or the **Spring Expression Language (SpEL)**. This moves APIs beyond static routing and enables flexible behavior at the gateway.

The example below demonstrates how to compute header values using simple expressions:

```
api:
  port: 2000
  flow:
    - request:
        - setHeader:
            name: X-Prod-Env
            value: ${header['host'] matches 'prod.*'}
            language: spel
        - setHeader:
            name: X-Date
            value: ${java.time.LocalDate.now()}
            language: groovy
        - setHeader:
            name: X-Address
            value: ${$.address.zip} ${$.address.city}
            language: jsonpath
        - log:
            message: "Headers:\n ${header}"
        - return:
            status: 200
```

(To avoid typing, download the samples from `https://membrane-api.io/gw.txt`)

Suppose the following request is sent to the API:

```
POST http://localhost:2000
Content-Type: application/json
Host: prod.example.com

{
  "address": {
    "city": "Boston",
    "zip": "02108"
  }
}
```

The computed headers in the response are:

```
X-Prod-Env: true
X-Date: 2026-01-16
X-Address: 02108 Boston
```

This example illustrates how header values can be computed at runtime based on request metadata, environment information, or message content.

The `return` interceptor at the end copies the request headers and the body into the response. That is why the headers appear in the API response, even though they were set in the request flow.

How the Expressions Work

If these expressions look cryptic at first glance, the following explanations break down what each of them does.

```
${header['host'] matches 'prod.*'}
```

This is a **SpEL** expression. It reads the value of the incoming `Host` header and matches it against the regular expression `prod.*`. If the hostname starts with `prod`, the expression evaluates to `true`.

```
${java.time.LocalDate.now()}
```

This is a **Groovy** expression that calls a Java library to obtain the current date.

```
${$.address.zip} ${$.address.city}
```

This example uses **JSONPath** to extract values from the request body.

To use an expression in a `value` field, it must be wrapped in `${...}`. In this case, the header value contains two expressions. Each expression is evaluated separately and then combined into a single string.

29.4 Removing HTTP Headers

Setting headers is only half the story. Removing them can be just as important, especially when it comes to **privacy and security**. Backend systems often return headers that reveal implementation details, such as the server type or framework version. Attackers can use this information to search for known vulnerabilities.

To reduce this risk, API Gateways allow you to control which headers are forwarded. You can define allowlists or blocklists to explicitly determine which headers should pass through and which should be removed.

The following configuration places an API in front of `wikipedia.org` and removes all response headers except those that start with `Content` or are named `Last-Modified`:

```
api:
  port: 2000
  flow:
    - response:
        - headerFilter:
            - include: Content.*
            - include: last-modified
            - exclude: .*
  target:
    url: https://www.wikipedia.org
```

To see what headers Wikipedia normally returns, try the following command:

```
curl -sS -D - https://www.wikipedia.org -o /dev/null
```

Then try again by sending the request to the gateway:

```
curl -sS -D - http://localhost:2000 -o /dev/null
```

With the `headerFilter` in place, only explicitly allowed headers (`Content.*` and `last-modified`) are included in the response to the client. Everything else is dropped. This helps keep the system's **internals private**.

29.5 Body Transformation

Just like headers, the message body can be transformed by an API Gateway. This is useful when integrating systems that expect a different data format or when tailoring responses for frontend applications.

Gateways typically support several approaches for body transformation:

1. **Format converters**
   Automatically convert between formats, for example from XML to JSON or vice versa.
2. **Templates**
   Replace placeholders dynamically with values from headers, query parameters, or the message body.
3. **Beautifiers and formatters**
   Pretty-print JSON or XML by adding indentation and line breaks. This does not change the data itself, but it improves readability for developers.

In the following sections, we will walk through each of these techniques with practical examples. By the end, you will be able to reshape API payloads to meet the needs of both clients and backend systems.

29.6 Make It Nice

Message payloads are often stripped of unnecessary whitespace to save bandwidth. In most cases, both client and server are applications, not human entities, and they do not care whether a payload is neatly formatted or compressed into a single line.

For developers, however, analyzing a JSON document that consists of one very long line can be painful. To make debugging easier, payloads can be beautified. An API Gateway can format JSON or XML messages before sending them to the client.

The following API formats the payload before returning it to the caller:

```
api:
  port: 2000
  flow:
    - request:
        - beautifier: {}
        - return:
            status: 200
```

When this API is invoked with the following request:

```
POST http://localhost:2000
Content-Type: application/json

{ "a":{ "b":{"c":7}}}
```

The caller receives a nicely formatted response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "a": {
    "b": {
      "c": 7
    }
  }
}
```

Isn't that pretty?

You can also try the sample API with XML payloads, since the Membrane beautifier supports both JSON and XML. Just make sure to set the correct **Content-Type** header. Without it, Membrane cannot determine which beautification algorithm to apply.

Should you care about how the payload looks? For APIs with large payloads or very high request volumes, it’s usually better to skip beautification and save bandwidth. However, when an API is explored, tested, or debugged heavily by developers, improved readability can justify the small overhead. For payloads smaller than a few kilobytes, the performance impact is typically negligible. In practice, beautification is often enabled only in development or test environments and disabled in production.

29.7 Transformation Between JSON and XML

Membrane can perform generic format transformations between JSON and XML. For background information and limitations of this approach, see Chapter *7.3 Transformation Between JSON and XML*.

Here, we focus on how such transformations are applied in practice.

Generic JSON to XML Transformation

The API below converts a JSON response from the **Fruitshop** into XML before returning it to the client:

```
api:
  port: 2000
  flow:
    - response:
        - json2Xml:
            root: fruit
  target:
    url: https://api.predic8.de
```

The Fruitshop backend returns JSON, as shown earlier. When calling it through this API:

```
curl http://localhost:2000/shop/v2/products/7
```

the client receives an XML response like this:

```
<fruit>
  <unit>€</unit>
  <price>69.99</price>
  <name>Gac-Fruit</name>
  <id>7</id>
</fruit>
```

The JSON object returned by the backend contains several properties. Because XML requires a root element, the properties must be wrapped inside a single root element. To avoid the converter falling back to a generic default name, we explicitly define the root element as `fruit`.

One important detail is element order. JSON object properties are unordered by definition. As a result, the generated XML elements may appear in any order. In the example above, the `id` element appears last, even though many XML schemas would expect it first.

In XML, element order usually matters. A different order can cause schema validation errors or break the interface.

If element ordering is important, the response should be post-processed before it is returned to the client. This can be done using a transformation step, for example with a script or an XSLT stylesheet that explicitly controls the structure and order of the elements.

XML2JSON Transformation

A conversion in the other direction is also possible: from XML to JSON.

In the following API, the gateway converts the incoming XML request body to JSON before forwarding it to the backend:

```
api:
  port: 2000
  flow:
    - request:
        - xml2Json: {}
  target:
    url: https://api.predic8.de
```

Valid XML documents always have exactly one root element. In this example, `add` is the root element that wraps the actual data:

```
POST http://localhost:2000/shop/v2/products
Content-Type: text/xml

<add>
    <name>Citron</name>
    <price>0.39</price>
</add>
```

The converter transforms the XML root element into a JSON equivalent, represented as an object with a single property.

```
{
  "add": {
    "price": 0.39,
    "name": "Citron"
  }
}
```

The payload is now JSON, but there is a catch. The backend API expects the fields directly, not wrapped inside an `add` object. Before the request can be processed successfully, that wrapper must be removed, for example with an additional transformation step.

29.8 Rendering Dynamic Content

Templates generate message bodies dynamically and can produce JSON, XML, or legacy formats such as SOAP.

`setBody` is a simple templating interceptor. In contrast to the `template` interceptor, which only supports Groovy as expression language, `setBody` additionally supports SpEL, XPath

and JSONPath. This makes it a convenient tool for transformations that rely on one of those languages.

In the previous section, the generic XML to JSON conversion wrapped the JSON document inside the `add` property derived from the XML root element, resulting in a format that the backend does not understand. Instead of relying on a generic transformation, you can use a template to create exactly the structure that the client or backend expects.

The following example uses `setBody` together with XPath to transform the XML request into the JSON structure required by the Fruitshop API:

```
api:
  port: 2000
  flow:
    - request:
        - setBody:
            language: xpath
            contentType: application/json
            value: |
              {
                "name": ${/add/name},
                "price": ${/add/price}
              }
  target:
    url: https://api.predic8.de
```

The XML from the previous example is now converted into valid JSON for the **Fruitshop API**:

```
{
  "name": "Citron",
  "price": 0.39
}
```

Notice that `"Citron"` appears with quotes in the resulting JSON, even though the template does not contain quotes around the XPath expression. This is because `setBody` serializes values according to the specified `contentType`. When `application/json` is set, string values are automatically quoted and properly escaped.

Sidenote: Serialization
Serialization is format aware. A numeric XPath result such as `0.39` becomes a JSON number, not a string. This prevents subtle type errors in downstream services.

Support for multiple expression languages makes `setBody` a practical tool for dynamic message creation. However, it does not support loops or conditional logic. For more complex transformations, such as iterating over collections or applying conditions, a full-featured template engine is required. We will explore those options in the next sections.

Security Hint: Escaping and serializing external input
If external values are inserted into a template, they must be properly serialized or escaped, to prevent injection attacks. `setBody` does that automatically for JSON and XML when the correct content type is provided.

More about escaping and injection risks can be found in Chapter *7.6* Security Considerations for Templating.

Transforming a GET into a POST

A more powerful alternative to `setBody` is the `template` interceptor. It is based on **Groovy's template engine** and supports looping and conditional rendering. It is, however, limited to Groovy as the expression language.

We will use the `template` interceptor to convert a GET request with query parameters into a POST request with a JSON payload.

```
http://localhost:2000/add?name=Lemon&price=0.30
```

In the following API, the `template` interceptor constructs a JSON payload from the query parameters `name` and `price`.

```
api:
  port: 2000
  flow:
    - request:
        - template:
            contentType: application/json
            src: |
              {
                "name": ${param.name[0]},
                "price": ${param.price[0].toFloat()}
              }
  target:
    method: POST
    url: https://api.predic8.de/shop/v2/products
```

HTTP allows multiple query parameters with the same name. Therefore, they are exposed through the implicit variable `param`, which maps each parameter name to a list of values. Since we expect a single product name and a single price, we access the first element using the subscript operator `[0]`:

```
"name": ${param.name[0]},
```

In templates with content type `application/json`, values are serialized according to JSON rules. Because query parameters are strings, `price` must be converted to a numeric type first:

```
"price": ${param.price[0].toFloat()}
```

In the `target` section, the HTTP method is explicitly set to `POST`, overriding the original method.

This API allows clients to use a simple GET request while the gateway adapts it to the POST-based backend API. The pattern is useful when integrating legacy APIs or when implementing webhook style endpoints where parameters are passed through the URL.

In SOAP, for example, every call uses POST, even read only operations such as `getCity`. Using this pattern, such operations could be exposed as REST-style resources and invoked as:

```
GET /cities/{id}
```

💡**Semantics of GET and POST**
GET is defined as safe and idempotent. Clients and intermediaries may therefore retry GET requests automatically if something goes wrong.

In contrast, POST is not safe and not idempotent. A POST request typically creates or modifies resources. Repeating the same POST may result in duplicate side effects, such as creating multiple resources.

Converting a GET into a POST at the gateway violates HTTP semantics. From the client's perspective, the operation appears safe and idempotent, while the backend performs a state-changing operation. This **can mislead clients into retrying** a request that is not safe to repeat.

For this reason, this pattern is discouraged because it undermines proper API design. It should only be used for legacy compatibility or in constrained environments where no better alternative is possible.

Secure Templating: Escaping External Input

Templates that incorporate user input are susceptible to injection attacks. Escaping dynamic content protects against manipulation by malicious clients or even compromised backends. If escaping is disabled or misconfigured, templates can become an injection point.

Below is an unsafe variant of the previous example. Escaping is not enabled because `contentType` is not set and therefore defaults to `text/plain`. Since no automatic JSON escaping is applied, the quotes around `name` must be added manually:

```
- template:
    src: |
      {
        "name": "${param.name[0]}",
        "price": ${param.price[0].toFloat()}
      }
- setHeader:
    name: Content-Type
    value: application/json
```

At first glance, the API behaves as before. However, if a client sends a manipulated query string such as:

```
?price=0.3&name=Lemon%22%2C%22promotion%22%3A%22yes
```

an additional field is injected. The resulting JSON becomes:

```
{
  "name": "Lemon","promotion":"yes",
  "price": 0.3
}
```

The attacker terminates the original string and injects an additional property into the object. Because escaping is disabled, the template does not protect the JSON structure.

The `template` and `setBody` interceptors automatically apply content-type–aware serialization when `contentType` is set correctly.

💡Security Hint: Escaping external input
Always treat external input as untrusted. Apply proper escaping and validation, as discussed in the previous section and in Chapter *7.6* Security Considerations for Templating.

Transforming a POST into a GET

In the following example, the API constructs a GET request using the `id` property of a JSON body:

```
api:
  port: 2000
  target:
    method: GET
    url: https://api.predic8.de/shop/v2/products/${$.id}
    language: jsonpath
```

The expression `${$.id}` is embedded in the URL and evaluated using JSONPath against the incoming request body.

When the API receives the following request:

```
POST /
Host: localhost:2000
Content-Type: application/json

{
  "id": 13
}
```

the gateway requests this URL:

```
GET https://api.predic8.de/shop/v2/products/13
```

Although concise, this approach introduces injection risks if the extracted value is not properly validated or URL-encoded.

Escaping Values in URL Templates

Because the value is derived from user input, the same rule applies: never trust external data.

If a client sends:

```
{
  "id": "13&foo=1"
}
```

the raw value would be `13&foo=1`. If inserted without encoding, this could append an unintended query parameter to the request.

Instead of copying the value directly into the URL, Membrane URL encodes characters that have special meaning. The rendered path becomes:

```
/shop/v2/products/13%26foo%3D1
```

The characters `&` and `=` are percent-encoded, preventing them from being interpreted as query delimiters. The backend receives a single path segment containing the encoded value, `13&foo=1`. No additional query parameter is added. The URL structure remains intact.

In most cases, such a value will not match a valid resource and will result in an error. From a security perspective, the key point is that the structure of the request cannot be altered by injecting reserved characters.

When expressions are evaluated inside URL templates in the `target` configuration, Membrane applies URL encoding to the resulting value before inserting it into the URL. This prevents query parameter injection and preserves the structural integrity of the generated request.

Processing Lists and Map Structures

The `template` plugin can do more than just fill in placeholders. It can generate complex structures such as objects, lists, and tables.

Many API Gateways support established templating engines such as Mustache or Velocity. Membrane uses the Groovy Template Engine, which integrates naturally with the Java platform and provides full scripting capabilities.

The following example generates a JSON document dynamically from the incoming HTTP header fields. The headers are available through the map variable `header`, which contains all header names and their corresponding values:

```
api:
  port: 2000
  flow:
    - request:
        - template:
            contentType: application/json
            pretty: true
            src: |
              {
                <% header.eachWithIndex { e, i -> %>
                  <% if (i > 0) { %>,<% } %>
                  <%= e.key %>: <%= e.value %>
                <% } %>
              }
        - return:
            status: 200
```

To generate the list of JSON properties, the template iterates over all header entries and renders each name and value as a JSON field.

The conditional statement:

```
<% if (i > 0) { %>,<% } %>
```

ensures that commas are placed correctly between entries. The variable `i` represents the index of the current element in the iteration.

To test the configuration, send a request to:

```
http://localhost:2000
```

There are alternative ways to render a list or map.

One option is to output the header map directly. With `contentType` set to JSON, it will be escaped automatically:

```
<%= header %>
```

Another option is to use the Java Streams API or Groovy collection methods to construct a JSON structure programmatically:

```
{
    <%= header.entrySet().collect { e -> [
        e.key: e.value
    ]} %>
}
```

These exsamples show that templates are not limited to simple substitutions. They can combine iteration, conditional logic, and data structures to generate dynamic transformations in a compact and expressive way.

Resources

Groovy documentation on template engines
https://docs.groovy-lang.org/next/html/documentation/template-engines.html

30 Control Flow

Most message flows in an API Gateway are executed sequentially, with each interceptor processing the message in turn. However, gateways are not limited to simple linear pipelines. Many gateways support control flow constructs such as conditions and loops, enabling more sophisticated processing logic.

Conditions

The `if` plugin allows steps to be executed conditionally, based on a test expression. It can be used in both the request and response flows.

As an example, consider a lightweight form of API protection. In some scenarios, implementing a full authentication system may be unnecessary. This is often true for internal APIs, prototypes, or temporary endpoints.

The configuration below protects an endpoint with a basic API key check:

```
api:
  port: 2000
  flow:
    - request:
        - if:
            test: header['X-Api-Key'] != 'sesame'
            flow:
              - return:
                  status: 401
        - return:
            status: 200
```

If the client sends the `X-Api-Key` header with the value `sesame`, the request proceeds and the API returns `200`. Otherwise, the gateway immediately returns a `401 Unauthorized` response.

You can test this behavior with:

```
GET http://localhost:2000
X-Api-Key: sesame
```

or from the command line:

```
curl -H "X-Api-Key: sesame" localhost:2000 -v
```

💡**Sidenote**: **Lightweight protection only**
Hardcoding secrets directly into configuration files is discouraged. While this approach can work for demos or internal use, it does not scale and makes secret rotation difficult. For production systems, use proper authentication mechanisms such as managed API keys or token-based authentication, which are covered later in this book.

Choose and Case

Backends often return error messages in different formats and styles. To streamline API design, responses can be translated into standardized **Problem Details** messages. This is a good use case for the `choose` construct.

The following example shows how to use `choose` to generate structured error responses based on the status code returned by a backend service:

```
api:
  port: 2000
  flow:
    - response:
        - choose:
            - case:
                test: statusCode == 401
                flow:
                  - static:
                      contentType: application/problem+json
                      src: |
                        {
                          "type": "/problem/auth",
                          "title": "Authentication Error"
                        }
            - case:
                test: statusCode >= 400 and statusCode < 500
                flow:
                  - static:
                      contentType: application/problem+json
                      src: |
                        {
                          "type": "/problem/client",
                          "title": "Client Error"
                        }
            - otherwise:
                - log:
                    message: No error
  target:
    url: http://localhost:3000
```

The `choose` block evaluates each `case` in the order defined and executes the flow of the first matching case. If no `case` matches, the `otherwise` block is executed.

Unlike `switch` statements in languages such as C, there is no fall-through behavior. Once a matching case is found, evaluation stops.

Beyond basic conditional logic, some gateways also provide loop constructs. These make it possible to iterate over collections or repeat processing steps dynamically. We will look at these capabilities in the next section about orchestration.

Resources

RFC 9457: ***Problem Details for HTTP APIs***
https://datatracker.ietf.org/doc/html/rfc9457

31 API Orchestration

API orchestration combines multiple backend services into a single API. This can simplify client logic, reduce API traffic, and hide internal complexity. It is helpful in scenarios like authentication flows, microservice aggregation, application integration, and mobile clients where minimizing round trips is critical.

This chapter walks through three practical orchestration scenarios:

- Aggregating data from multiple backend APIs
- Handling backend authentication transparently for the client
- Processing and enriching RESTful resources during request handling

Whether you want to simplify external interfaces or bridge the gap between legacy systems and modern consumers, orchestration is a powerful tool to have in your API Gateway toolbox.

31.1 Aggregating Backend APIs

The first orchestration use case shows how a gateway can combine multiple backend calls into a single, clean API. This hides internal complexity from the client, simplifies frontend logic, and cuts down on the number of network round trips.

Let's walk through an example using the **Open Library API**. Suppose we want an endpoint that returns both the title and the author of a book. The Open Library project provides public APIs for this data, but it cannot be retrieved in a single call.

A call to:

```
GET https://openlibrary.org/books/OL29474405M.json
```

returns:

```
{
  "title": "So Long, and Thanks for All the Fish",
  "authors": [
    { "key": "/authors/OL272947A" }
  ]
}
```

(Additional fields omitted.)

This gives us the book's title and a reference to the author, but not the author's name. To retrieve that, we need a second call:

`https://openlibrary.org/authors/`**`OL272947A.json`**

which unsurprisingly returns:

```
{
  "name": "Douglas Adams"
}
```

(Additional fields omitted.)

To combine both steps into a single API call, we can orchestrate both backend requests in the gateway.

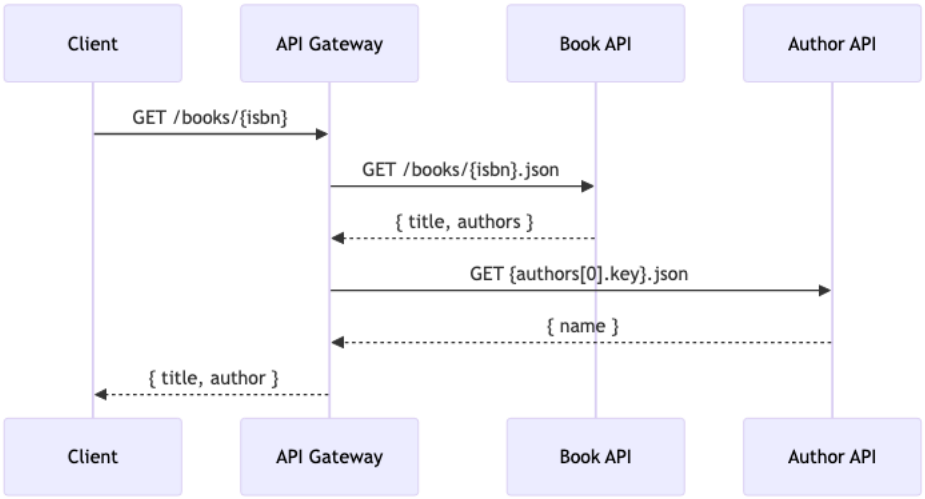


Image: Orchestrating two backend calls into a single API (rendered by Mermaid)

The diagram illustrates how the API Gateway:

- receives a single request from the client
- makes two backend calls: one for the book, one for the author
- then merges both responses into a unified result

Composing Two Backend Calls

The configuration below combines the two backend APIs by first fetching the book data and then retrieving the corresponding author name:

```
api:
  port: 2000
  path:
    uri: /books/{id}
  flow:
    - request:
        - call:
            url: https://openlibrary.org/books/${pathParam.id}.json
        - setProperty:
            name: title
            value: ${$.title}
            language: jsonpath
        - call:
            url: https://openlibrary.org${$.authors[0].key}.json
            language: jsonpath
        - template:
            contentType: application/json
            src: |
              {
                "title": ${property.title},
                "author": ${jsonPath('$.name')}
              }
    - return:
        status: 200
```

(To avoid typing, download the samples from `https://membrane-api.io/gw.txt`)

A client can now make simple requests like:

```
curl http://localhost:2000/books/OL29474405M
curl http://localhost:2000/books/OL26333978M
```

The gateway responds with a single JSON document that combines the title from the first call and the author name from the second:

```
{
  "title": "Foucault's pendulum",
  "author": "Umberto Eco"
}
```

Even though the configuration is short, there is a lot going on, so let’s walk through it step by step.

First, the gateway calls the book endpoint. The URL is built from the `{id}` path parameter:

```
- call:
    url: https://openlibrary.org/books/${pathParam.id}.json
```

From the book response, the title is extracted using JSONPath and stored in an exchange property named `title`:

```
- setProperty:
    name: title
    value: ${$.title}
    language: jsonpath
```

Next, the gateway calls the author endpoint. This time the author URL is taken from the first response using JSONPath and inserted into the URL template:

```
- call:
    url: https://openlibrary.org${$.authors[0].key}.json
    language: jsonpath
```

Finally, the template renders the response. Membrane provides additional scripting functions that can be used in expressions and templates. The function call `jsonPath('$.name')` extracts the author's name from the current message body:

```
- template:
    contentType: application/json
    src: |
      {
        "title": ${property.title},
        "author": ${jsonPath('$.name')}
      }
```

Hint: Creating Orchestrations
Build orchestrations step by step. Start with the first call and log the response. Then extract one value, log it, and continue. You can place the `log` plugin anywhere in the flow to inspect the current request or response and to verify that each step behaves as expected.

31.2 Authentication for Backend Access

Imagine a backend API that requires a session cookie for authentication. You may not want to expose that complexity to API consumers. In some cases, you might even want to offer a completely different authentication mechanism externally than the one required by the backend.

A gateway can perform the necessary login steps in the background. It authenticates against the backend system, obtains the required credentials, and forwards them with the actual request. From the client's perspective, the API remains simple and consistent, while the gateway takes care of the protocol details behind the scenes.

In this scenario, three components are involved:

1. **Protected target API**
   Requires a session cookie
2. **Authentication service**
   Issues the cookie after successful login
3. **Orchestration API**
   Logs in, retrieves the cookie, and forwards it to the protected API.

The diagram illustrates how the gateway sits between client and backend, performing authentication transparently and shielding the client from backend-specific login procedures.

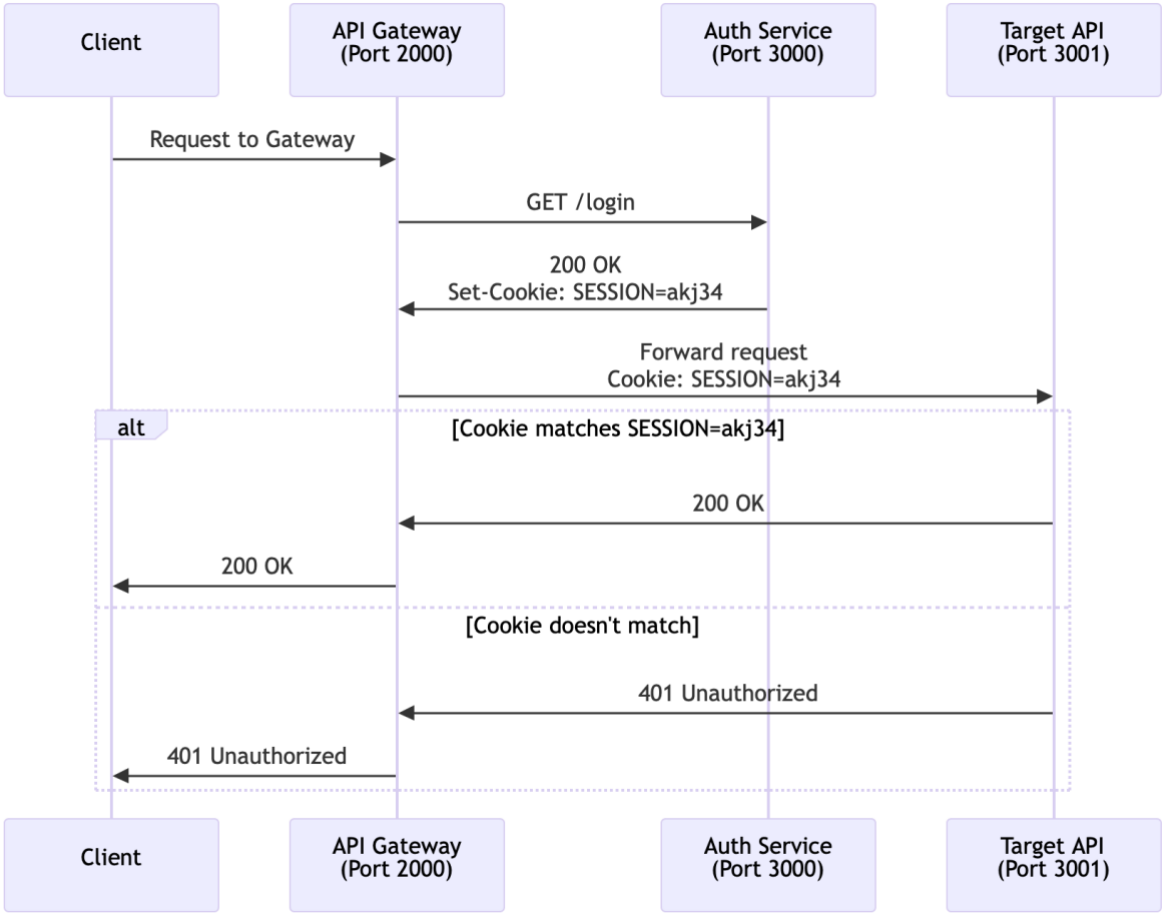


Image: API Gateway authenticates on behalf of the client

Setting it Up

All three components can be simulated by the API Gateway. Let's walk through the configuration of each part and see how they work together.

1. Target API (Backend Simulation at port 3001)

This API checks whether a valid session cookie is present. If it is, the API returns `Success!`. Otherwise, it responds with `401 Unauthorized`.

```
api:
  port: 3001
  name: Target API
  flow:
    - request:
        - log:
            message: "Session: ${cookie.SESSION}"
        - if:
            test: cookie.SESSION != 'akj34'
            flow:
              - return:
                  status: 401
        - static:
            src: Success!
        - return:
            status: 200
```

2. Authentication Service (port 3000)

This API simulates a login endpoint that sets the required session cookie:

```
api:
  port: 3000
  path:
    uri: /login
  flow:
    - response:
        - setCookies:
            - name: SESSION
              value: akj34
        - log:
            message: Login successful. Issue cookie.
    - return:
        status: 200
```

3. Orchestration API (port 2000)

The orchestration API acts as a facade. It first logs into the authentication service, extracts the cookie, and then forwards the request to the protected backend together with the cookie:

```
api:
  port: 2000
  flow:
    - request:
        - log:
            message: Authenticating via login API
        - call:
            url: http://localhost:3000/login
        - log:
            message: "Got Cookie: ${header['Set-Cookie']}"
        - setHeader:
            name: Cookie
            value: ${header['Set-Cookie'].split(';')[0]}
        - headerFilter:
            - exclude: Set-Cookie
        - log:
            message: "Forwarding Cookie: ${header['Cookie']}"
  target:
    url: http://localhost:3001
```

After the `call` to the login endpoint, the current message context is replaced with the login response, including its headers. The orchestration logic extracts the session cookie from that response and sets it in the `Cookie` header for the backend at port 3001. Only the cookie name and value are forwarded; attributes such as `Path` and `SameSite` are cut off in this expression:

```
${header['Set-Cookie'].split(';')[0]}
```

A request to `http://localhost:2000` triggers the flow and produces the following log:

```
{api=Gateway} Authenticating via login API
{api=Login API} Login successful. Issue cookie.
{api=Gateway} Got Cookie: SESSION=akj34; Path=/; SameSite=LAX
{api=Gateway} Forwarding Cookie: SESSION=akj34
{api=Target API} Session: akj34
```

The client is unaware of the intermediate login call. It simply receives the final response from the protected API.

Securing the Orchestration API

In this simplified setup, the orchestration API itself is not protected. In real-world use cases, you would typically secure this external endpoint using **API keys**, **JWTs**, or **OAuth2**. This allows you to present a modern, secure interface to clients while isolating legacy or session-based authentication mechanisms behind the gateway.

31.3 Processing RESTful List Resources

This final orchestration demonstrates how an API Gateway can **navigate** related resources referenced by hypermedia links and combine multiple responses into a single result.

RESTful APIs often expose **list resource**s, for example:

```
https://api.predic8.de/shop/v2/products
```

The endpoint returns a list of entries containing only basic information such as product names and links:

```
{
  "products": [
    {
      "name": "Bananas",
      "self_link": "/shop/v2/products/2"
    },
    {
      "name": "Figs",
      "self_link": "/shop/v2/products/7"
    },
    {
      "name": "Rambutan",
      "self_link": "/shop/v2/products/12"
    }
  ]
}
```

To retrieve additional details such as prices, each item must be fetched individually:

```
https://api.predic8.de/shop/v2/products/12
```

The detail endpoint returns:

```
{
  "name": "Rambutan",
  "price": 5.60
}
```

Now assume an application needs to display a list including prices:

```
Bananas: 1.99
Figs: 2.40
Rambutan: 5.60
```

Without orchestration, the client would first request the list and then issue an additional request for each product to retrieve its price. This requires extra client-side logic and multiple network round trips.

An API Gateway can orchestrate the entire process on the server side. It retrieves the list, requests each referenced product resource, and merges the responses into a consolidated result. The diagram below illustrates a call sequence for this orchestration.

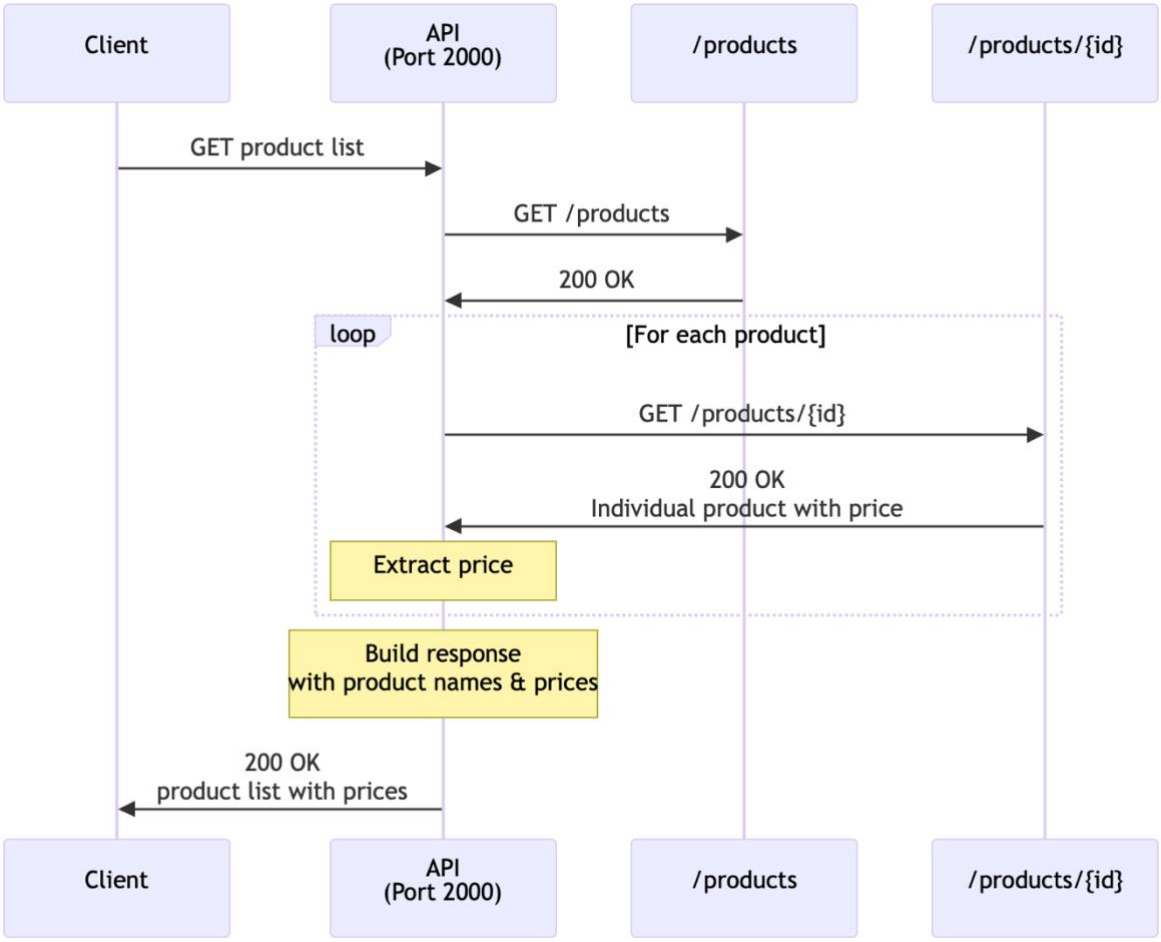


Image: Sequence of calls to get products and prices in a single API

Implementation

The following API configuration implements the orchestration. It fetches the product list, retrieves details for each product, collects the name and price, and finally renders the result as JSON.

```
api:
  port: 2000
  flow:
    - request:
        - call:
            url: https://api.predic8.de/shop/v2/products?limit=5
        - for:
            in: $.products
            language: jsonpath
            flow:
              - log:
                  message: 'Getting price for: ${it["name"]}'
              - call:
                  url: https://api.predic8.de${it['self_link']}
              - groovy:
                  src: |
                    property.res = property.res ?: []
                    property.res.add(
                      [ "name": it['name'],
                        "price": jsonPath('$.price')
                      ])
        - template:
            contentType: application/json
            pretty: true
            src: |
              {
                  "products": <%= property.res %>
              }
    - return:
        status: 200
```

Step-by-Step Explanation:

1.) Fetching the Product List:

This call retrieves five products in a single request. The `limit` keeps the example fast. You can increase or remove it to see how the orchestration behaves with larger lists.

```
- call:
    url: https://api.predic8.de/shop/v2/products?limit=5
```

The response contains the list of products. Each entry includes the product name and a link to the corresponding single-resource endpoint.

2.) Looping Over the Product List

The list now serves as input for the loop. The JSONPath `$.products` selects the `products` array from the message body. The `for` interceptor executes its subflow once for each entry.

```
- for:
    in: $.products
    language: jsonpath
    flow:
      - call:
          url: https://api.predic8.de${it['self_link']}
```

Inside the loop, the implicit variable `it` refers to the current list element. The backend URL is built by appending the product's `self_link` to the base URL, allowing the gateway to fetch details for each product individually.

3.) Creating the Result List

Inside the loop, a Groovy script builds the result list. The script initializes `property.res` if it does not exist and then adds an entry containing the product name and the price extracted from the backend response.

```
- groovy:
    src: |
      property.res = property.res ?: []
      property.res.add(
        [ "name": it.name,
          "price": jsonPath('$.price')
        ])
```

By storing intermediate results in an exchange property, the aggregated data remains available after the loop completes and can be used to generate the final response.

4.) Generating the JSON Response:

Finally, a Groovy template renders the aggregated data as a JSON document. The template outputs the list stored in `property.res`, which gets serialized automatically.

```
- template:
    contentType: application/json
    pretty: true
    src: |
      { "products": <%= property.res %> }
```

This produces a single response containing the products and their prices, while the client only makes one request.

32 Secure Data in Transit with TLS

This section explains how to configure **Transport Layer Security (TLS)** in the API Gateway for encryption and certificate-based authentication. The section includes practical examples for securing traffic on both sides of the gateway.

We start with the TLS connection from the gateway to the backend. After that, we look at how to secure the connection from the client to the gateway.

32.1 Reaching Backends over TLS

When an API Gateway connects to a backend over TLS, it acts as a **TLS client**. In this role, the gateway initiates a secure connection, verifies the backend's certificate, and establishes an encrypted communication channel. This protects data in transit, ensuring confidentiality.

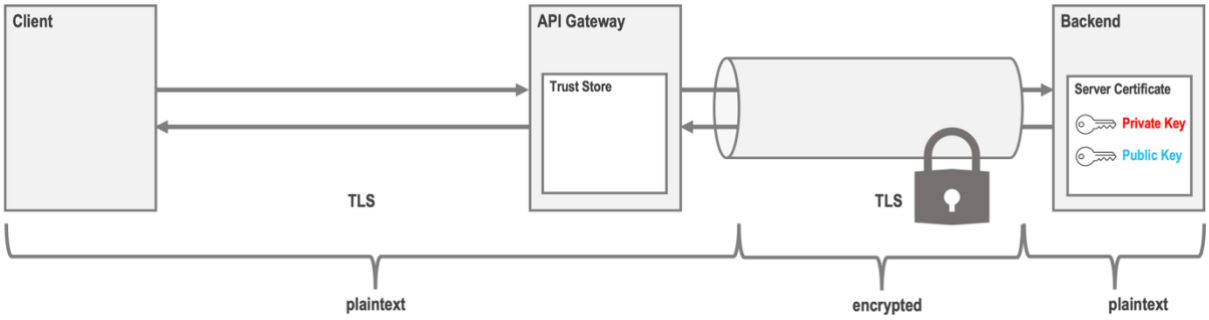


Image: TLS-secured connection between gateway and backend

By securing the connection between gateway and backend, sensitive data can be transmitted safely, even across untrusted networks.

TLS to a Backend with Public Certificate

Setting up TLS to a backend that uses a **public certificate** is straightforward. You have already seen this setup earlier in the book:

```
api:
  port: 2000
  target:
    url: https://api.predic8.de
```

Using `https://` in the backend URL is enough to instruct the gateway to establish a TLS-secured connection.

Because the backend certificate is signed by a public certificate authority (CA), the gateway can validate it using its trusted root certificates. No additional configuration is required.

Membrane uses the Java platform's truststore by default. It is typically located at:

```
<JAVA_HOME>/lib/security/cacerts
```

Although the exact location may vary depending on the Java distribution.

You can add additional trusted certificates to this truststore using `keytool`. Once added, Membrane will automatically trust these certificates for outgoing TLS connections. Alternatively, you can configure a custom truststore, as described in the following sections.

💡**Side Note: What is a truststore?**
A truststore contains a **collection of trusted root and intermediate certificates**. When either a client or a server presents its certificate during the TLS handshake, the other party uses the certificates in the truststore to verify that the presented certificate was issued by a trusted authority and that the certificate chain is valid.

32.2 Termination of TLS Connections

At some point, an encrypted connection must be decrypted so that the data can be processed. The process of decrypting an incoming connection is called TLS termination.

Most gateway features require access to the unencrypted payload. Without termination, the gateway would only see encrypted bytes and could not apply routing, message validation, or transformation.

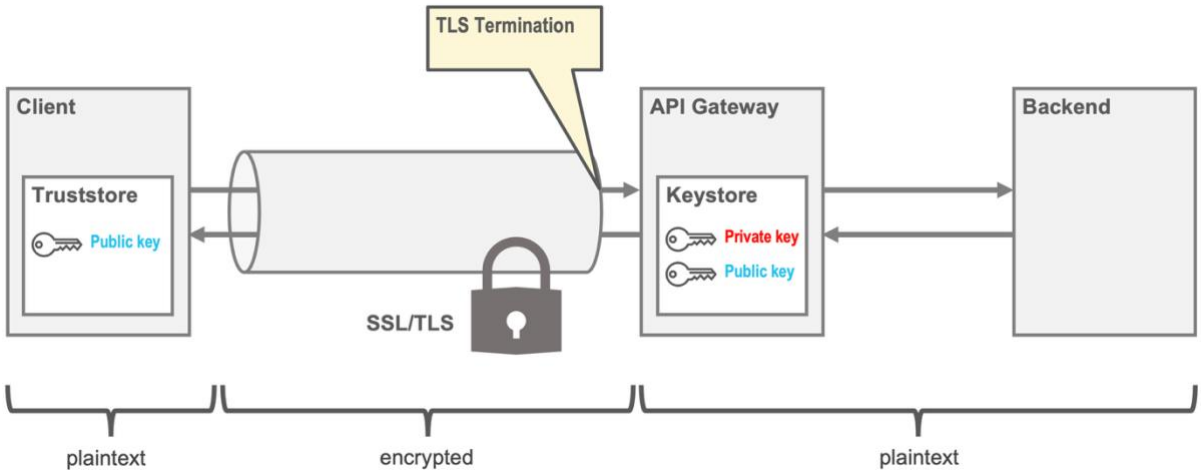


Image: Termination of a TLS-secured connection at the gateway

Handling TLS at the gateway offers several advantages:

- **Enabling gateway features**
  Logging, transformation, and security checks require access to unencrypted content.
- **Simplified backend configuration**
  When TLS is terminated at the gateway, backend services do not need to manage certificates or expose HTTPS themselves. This can be useful for legacy systems or internal services running inside private networks.

TLS termination also comes with responsibilities. The gateway must securely manage private keys and certificates. TLS introduces a small performance overhead, although with modern hardware this is rarely a bottleneck.

💡Sidenote: TLS passthrough?
In TLS passthrough mode, the gateway does not terminate TLS. Instead, it forwards the encrypted connection to the backend. This can improve performance and simplify security in some setups, but it limits what the gateway can do since it can't inspect or modify the payload.

Setting Up TLS Termination

The more involved part of setting up TLS termination is obtaining or generating the required certificates and private keys. Once these are available, configuring the gateway is relatively straightforward.

Membrane ships with **sample certificates and keys for testing** purposes. You can find them in the following directories:

- `examples/security/tls-ssl/`
- `conf/`

Typical sample files include:

- `membrane-key.pem`: for the private key
- `membrane.pem`: for the corresponding certificate

⚠ Important
These sample certificates are intended for development and testing only. Never use them in production environments.

Examining Certificates

Before configuring the gateway, you can inspect a certificate using the `openssl` tool, which is available on most platforms:

```
openssl x509 -in membrane.pem -text
```

This command outputs details such as:

- **Signature Algorithm**: Used to sign the certificate
- **Issuer & Subject**: Who issued the certificate and who it is for
- **Validity Period**: Start and expiry dates
- **Public Key Information**: Key length and algorithm

An abbreviated excerpt might look like this:

```
Certificate:
  Data:
    Serial Number: 2004060569
    Signature Algorithm: sha256WithRSAEncryption
    Issuer: CN=membrane
    Validity
      Not Before: Aug  5 10:41:09 2015 GMT
      Not After : Aug  2 10:41:09 2025 GMT
    Subject: CN=membrane
    Subject Public Key Info:
      Public Key Algorithm: rsaEncryption
        Modulus:
          00:a9:2b:33:c3:16:51:…
        Exponent: 65537 (0x10001)
    Signature Value: 25:69:1a:46:d1:23:62
```

Note that this is a self-signed certificate, as the issuer and subject are identical.

Configuring TLS Termination

Once the key and certificate are available, TLS termination can be configured as follows:

```
api:
  port: 443
  ssl:
    key:
      private:
        location: membrane-key.pem
      certificates:
        - location: membrane.pem
  flow:
    - request:
        - log: {}
  target:
    url: https://api.predic8.de
```

This configuration terminates encrypted TLS connections using the provided certificate and private key. It then logs the message in plain text and forwards the request to the backend over a new encrypted TLS connection.

The sample certificates are available in the `tutorials/security` folder of the Membrane distribution if you want to try this setup.

Testing the TLS Setup

The `curl` tool is useful for debugging TLS connections. You can test gateway and server configurations with the command:

```
curl -v -k https://localhost:443
```

Explanation of the options:

- `-v` shows detailed information
- `-k` disables certificate validation, which is necessary when testing with self-signed certificates.

The output provides insight into the TLS handshake, including certificate details:

```
* Connected to localhost (::1) port 443
* (304) (OUT), TLS handshake, Client hello (1):
* (304) (IN), TLS handshake, Server hello (2):
* (304) (IN), TLS handshake, Unknown (8):
* (304) (IN), TLS handshake, Certificate (11):
* (304) (IN), TLS handshake, CERT verify (15):
* (304) (IN), TLS handshake, Finished (20):
* (304) (OUT), TLS handshake, Finished (20):
* SSL connection using TLSv1.3 / AEAD-CHACHA20/ UNDEF
* Server certificate:
*  subject: CN=membrane
*  start date: Aug  5 10:41:09 2015 GMT
*  expire date: Aug  2 10:41:09 2025 GMT
*  issuer: CN=membrane
*  SSL certificate verify result: self signed certificate
(18), continuing anyway.
```

This confirms that the TLS handshake completed successfully and that the gateway presented the configured certificate.

In this example, TLS termination is configured for a single API. To apply TLS settings centrally across multiple APIs or connectors, refer to `tutorial/security/20-Central-SSL-Config.yaml` in the Membrane distribution.

⚠ **Security Warning!**
Always replace sample certificates with valid production certificates in live environments.

Forwarding TLS Connections Without Decryption

In some scenarios, traffic is so sensitive that even the API Gateway should not inspect it. In such cases, a Membrane `sslProxy` can forward TLS connections without decrypting them:

```
sslProxy:
  port: 443
  host: secret.example.com
  target:
    host: secret.example.com
    port: 443
```

Membrane uses the SNI (Server Name Indication) value from the client's TLS handshake to determine where to forward the connection. If the SNI matches the configured host, the encrypted stream is passed through unchanged. There is no TLS termination, no payload inspection, and no logging at the HTTP level.

In this mode, Membrane operates as a Layer 4 (TCP) proxy. Since it does not terminate TLS, it does not require a certificate or private key. Decryption happens later at the backend server.

💡Tip: Sharing TLS ports
Port 443 can be shared with other proxies, such as `api` configurations handling different hostnames. Routing decisions are then based on SNI, allowing multiple TLS-based services to coexist on the same port.

32.3 Debugging TLS Connectivity

The moment you expose port 443 to the public internet, inbound connections will start to arrive. Some are harmless. Some are curious. Some are… less friendly. Typical traffic you may encounter includes:

- **Statistical scanners** collecting data for research or monitoring
- **Security researchers** probing supported TLS versions, possibly for academic work
- **Automated bots** scanning for vulnerabilities
- **Exploit attempts** targeting known vulnerabilities

All of this creates noise. A lot of it. Logging every failed TLS handshake would quickly flood the logs. For that reason, Membrane suppresses detailed TLS error logging by default.

During initial setup, certificate rollouts, or when troubleshooting connectivity issues, it can be useful to temporarily enable verbose logging:

```
api:
  port: 443
  ssl:
    showSSLExceptions: true
```

(Additional fields omitted.)

With this setting enabled, Membrane logs detailed TLS exceptions and handshake errors. This helps to understand what is happening under the hood:

- You can identify misconfigured clients attempting to connect with outdated TLS protocol versions.
- You can spot malformed handshakes that may indicate scanning or probing activity.
- You can distinguish harmless background noise from issues that require attention.

Once the setup is stable, it's usually a good idea to disable this option to keep log volume under control.

33 Network-Level Access Control

Access Control Lists (ACLs) provide a simple way to protect APIs by restricting access based on IP addresses or address ranges. They define which clients are allowed to interact with an API and which are blocked. ACLs are particularly useful in internal or otherwise controlled network environments.

```
global:
  - accessControl:
      - deny: 192.168.3.8
      - deny: 192.168.10.0/24
      - allow: 192.168.0.0/16
      - allow: '.+\.predic8\.de$'
```

ACLs are evaluated **from top to bottom**, and the first matching rule is applied. If no rule matches, access is denied.

This configuration:

- Allows access from the `192.168.0.0/16` network, but blocks the IP `192.168.3.8` and the subnet `192.168.10.0/24` explicitly.
- Allows access from hosts whose resolved names match the regular expression.
- Rejects traffic from all other sources.

ACLs can also be applied to individual APIs. The following example restricts access to the Admin Console so that it is only reachable from the local machine:

```
api:
  port: 9000
  flow:
    - accessControl:
        - allow: 127.0.0.0/8
        - allow: localhost
    - adminConsole: {}
```

When to Use ACLs

Although IP addresses can be spoofed, ACLs remain a useful first line of defense in controlled network environments. They are particularly effective for:

- Restricting access to internal services
- Limiting exposure of administrative endpoints
- Enforcing network-level trust boundaries

For stronger security, ACLs should be combined with other mechanisms such as OAuth2 or JWT-based authentication. Together, these mechanisms create a layered security approach that significantly improves API security.

Resources

ACL Configuration Example
`MEMBRANE_HOME/tutorial/security`

34 Content Protection

Attackers can exploit subtle quirks or dangerous features in data formats such as JSON, XML, or GraphQL to trigger unexpected behavior, denial of service, or even remote code execution. Many of these attacks rely on malformed or deliberately crafted payloads that exploit parser behavior rather than application logic.

Content protection aims to block such potentially harmful input **before** it reaches backend services. The goal is not to interpret the meaning of a request, but to enforce structural safety rules and eliminate known attack vectors. By doing so, gateways make exploitation more difficult.

Because the basic requirements for content protection are well understood, most API Gateways offer similar capabilities. Differences are usually limited to configuration style and naming conventions rather than underlying functionality.

34.1 JSON Protection

In Membrane, JSON protection can be enabled by adding the `jsonProtection` interceptor either to a specific API or globally for all APIs in a configuration. The following example activates protection for the API listening at port 2000:

```
api:
  port: 2000
  flow:
    - jsonProtection: {}
    - return:
        status: 200
```

If a client sends a request like this:

```
POST http://localhost:2000
Content-Type: application/json

{
  "price": 10,
  "price": -1
}
```

the gateway detects the duplicate `price` field and rejects the request:

```
HTTP/1.1 400 Bad Request
Content-Length: 474
Content-Type: application/problem+json

{
  "title": "JSON Protection Violation",
  "type": "https://membrane-api.io/problems/user",
  "detail": "Duplicate field 'price' at line: 3"
}
```

This prevents ambiguous or malicious payloads from reaching backend systems, where inconsistent parser behavior could otherwise lead to unpredictable results or even successful exploitation.

Customizing JSON Protection

The default settings provide basic protection for most APIs. However, depending on the use case, you may need to relax limits for large but valid payloads, or tighten them further for sensitive endpoints.

All limits can be configured individually:

```
- jsonProtection:
    maxDepth: 3
    maxKeyLength: 5
    maxObjectSize: 3
    maxTokens: 10
    maxStringLength: 10
    maxArraySize: 2
    reportError: true
```

(Reference: https://www.membrane-api.io/docs/current/jsonProtection.html)

By tuning these limits, you can balance between robustness and flexibility, protecting APIs without unnecessarily rejecting legitimate requests.

Resources

JSON Protection Examples and Configuration
Check the JSON Protection sample in the `tutorial/security` directory.

jsonProtection Reference
https://www.membrane-api.io/docs/current/jsonProtection.html

34.2 XML Protection

Like JSON, XML payloads can be crafted to overwhelm parsers, bypass validation logic, or trigger dangerous features such as entity expansion or external resource access. To mitigate these risks, Membrane provides XML protection.

To activate XML protection for an API, add the `xmlProtection` interceptor:

```
global:
  - xmlProtection: {}
```

This enables a set of default protection rules that block or neutralize common XML-based attacks, such as excessive attributes, deeply nested elements, or malicious entity declarations.

If an incoming XML request contains a `DOCTYPE` declaration, for example:

```
POST /
Host: localhost:2000
Content-Type: text/xml

<?xml version="1.0"?>
<!DOCTYPE n [<!ELEMENT n (#PCDATA)>]>
<n>Hello</n>
```

the protection mechanism automatically removes the `DOCTYPE` before the message is processed further:

```
HTTP/1.1 200 OK
Content-Type: text/xml
Content-Length: 33

<?xml version="1.0"?>
<n>Hello</n>
```

This helps prevent XML External Entity (XXE) attacks and related parser exploits that rely on DTD processing.

Other violations of XML protection are rejected outright. In those cases, the gateway blocks the request and returns an error, ensuring that malformed or dangerous payloads never reach the backend.

You can fine-tune XML protection rules to match the requirements of your APIs:

```
- xmlProtection:
    maxAttributeCount: 20
    maxElementNameLength: 100
    removeDTD: true
```

By applying XML protection at the gateway, you enforce strict structural safety guarantees and reduce the risk posed by malformed or malicious XML payloads.

Resources

xmlProtection Reference
https://www.membrane-api.io/docs/current/xmlProtection.html

34.3 GraphQL Protection

GraphQL offers a high degree of flexibility and allows clients to shape responses precisely to their needs. While powerful, this flexibility can be abused if queries are not properly constrained. Deeply nested queries, recursive structures, or large batches of mutations can quickly overwhelm backend systems.

To address these risks, Membrane provides built-in GraphQL protection that validates incoming queries and mutations against the GraphQL specification and enforces structural limits.

GraphQL protection can be enabled by adding the `graphQLProtection` interceptor to an API or the global chain:

```
global:
  - graphQLProtection:
      maxRecursion: 1
      maxMutations: 10
      maxDepth: 3
      disallow:
        - mutation: updateCategory
        - introspection: {}
```

This configuration enforces several safety measures:

- Limits the maximum depth of queries
- Prevents recursive query execution
- Restricts the number of mutations per request
- Disables GraphQL introspection queries
- Explicitly blocks a selected mutation

Together, these rules significantly reduce the risk of denial of service attacks, resource exhaustion, and overly expensive query execution.

Resources

GraphQL Validation Example
See the `examples/security/graphql-validation` folder of the Membrane distribution.

graphQLProtection Reference
https://www.membrane-api.io/docs/current/graphQLProtection.html

35 Basic Authentication

Basic authentication is almost as old as the web itself. It is simple, widely supported, and easy to configure, but it has clear limitations. Credentials are sent in a Base64-encoded string, which means that without encryption, anyone intercepting the traffic could read the username and password effortlessly.

When **combined with TLS**, however, basic authentication becomes a lightweight and practical solution. While it lacks advanced features such as token expiration, scopes, or delegation, it is still useful for internal APIs, administrative endpoints, and simple scenarios where more complex mechanisms would be unnecessary overhead.

How it Works

Basic authentication transmits credentials using the `Authorization` header together with the keyword `Basic`:

```
GET http://localhost:2000
Authorization: Basic ZnJlZG86YWJjMTIz
```

The value after `Basic` is a Base64-encoded string containing `username:password`.

Curious what's inside? You can decode it locally (the command may vary by operating system):

```
echo ZnJlZG86YWJjMTIz | base64 -d
```

which results in something like:

```
fredo:abc123
```

This demonstrates an important point: **Base64 is not encryption**. It is merely encoding. Think of it as putting your credentials in a paper envelope rather than locking them in a safe. For this reason, **TLS is mandatory** when using basic authentication over an untrusted network.

If you are experimenting, you can also use an online decoder:

`https://emn178.github.io/online-tools/base64_decode.html`

Image: Decoding a base64-encoded basic auth string online

⚠ Security Warning
Never decode production credentials using online tools. Doing so is equivalent to handing your house keys to strangers.

Once the request reaches the server or API Gateway, the `Authorization` header is decoded, and the credentials are verified against a configured user store or authentication provider.

Because basic authentication is stateless, the client must include the same `Authorization` header with every request. The server does not maintain session state; each request must carry the credentials independently.

For anything beyond basic use cases, consider stronger mechanisms such as API keys, OAuth2, or JWT-based authentication, which are covered in later sections.

Setting up Basic Authentication

The following API configuration shows how to protect an endpoint using basic authentication:

```
api:
  port: 2000
  flow:
    - basicAuthentication:
        users:
          - username: alice
            password: "qwertz"
          - username: carol
            password: "$6$k3jia$NlCL.JTXitlwZ4i672In7c8M..."
    - static:
        src: You're in!
    - return:
        status: 200
```

Alice's password is stored in plaintext and can be read by anyone who has access to the configuration file. For Carol, a password hash is stored instead. A password hash cannot be directly reversed to recover the original password, although weak passwords may be vulnerable to offline dictionary or brute-force attacks.

You can test this setup using `curl`:

```
curl -u alice:qwertz http://localhost:2000
curl -u carol:abc123 http://localhost:2000
```

There are several tools available to create password hashes, such as `mkpasswd` or `openssl`. For example:

```
openssl passwd -6 -salt k3jia "yourPassword"
```

This produces a hash like:

```
$6$k3jia$iCAgbWXSHNaVLlrDVRyar/NtbWfvBaBKiu4fAGqLx2mN5Oux5ZcGS
pMyaBC.3uTNIJySq1LOSasBH3fPH15v9/
```

Security Tip: Store password hashes

In production environments, always use hashed passwords and store them securely. Plaintext credentials should never appear in production configurations. Membrane's basic authentication implementation supports common password hash formats, making it easy to improve security without changing client behavior.

Managing Users in Files and Databases

Defining users directly in the configuration file is the easiest way to get started with basic authentication. But this approach has limitations:

- **Configuration reload**
  Every time a user is added or removed, the gateway configuration needs to be reloaded.
- **Scalability**
  Managing large numbers of users in a single configuration file becomes unwieldy.
- **Integration limitations**
  Storing credentials in the config makes it hard to integrate with external identity systems or user management tools.

To address these limitations, **Membrane** supports alternative user stores, including:

- Standard `.htpasswd` files
- Integration with **SQL databases**
- Custom Java implementations

This allows you to manage users more flexibly, automate updates, and better align with existing authentication infrastructure.

Using Files as a Password Store

Instead of defining users directly in the main configuration file, you can reference an external password file. This keeps credentials separate from the gateway configuration and makes user management more practical, especially when multiple people or systems need to update the user list.

A minimal setup using a file-based store looks like this:

```
flow:
  - basicAuthentication:
      htpasswdFileProvider:
        location: ./.htpasswd
```

The `htpasswdFileProvider` expects a file in the standard `.htpasswd` format. Each line contains a username and a hashed password. Membrane reads the file on startup and validates credentials against it.

💡Sidenote: .htpasswd file
This mechanism behaves similarly to Apache's Basic Auth, so if you have existing `.htpasswd` files from other systems, you can often reuse them directly.

How to Generate a `.htpasswd` File

`.htpasswd` files are a long-standing convention for storing usernames with hashed passwords, typically used by Apache HTTP Server. The format is widely supported, simple, and works well for small to medium user sets.

The file can be generated using the `htpasswd` command-line tool.

On most systems, the tool is included in the Apache HTTP Server utilities package:

- **Linux (Debian/Ubuntu)**
  `sudo apt install apache2-utils`
- **Linux (RHEL/Fedora/CentOS)**
  `sudo dnf install httpd-tools`
- **macOS (Homebrew)**
  `brew install httpd`

After installation the `htpasswd` command becomes available.

To create a new file and add the first user:

```
htpasswd -c .htpasswd alice
```

The `-c` flag creates the file. Running this command again with the same flag would overwrite the file, so only use `-c` for initial creation.

To add another user later:

```
htpasswd .htpasswd bob
```

A resulting `.htpasswd` file may look like this:

```
alice:$apr1$zGXp9.Px$uLogsgPwJoMVIWtA0Uv76.
bob:$apr1$ciDjPlh3$S2qaMyAl43SVswoCnscNz/
```

Each entry contains a username and a hash. Many hash formats are supported, including MD5-based (`$apr1$`), SHA variants, and bcrypt, depending on the `htpasswd` version.

Security hint: Hash algorithms
Although Membrane supports weaker hash types for compatibility, you should always choose the strongest available algorithm supported by your toolchain.

Resources
Membrane includes ready-to-run examples:

1. `examples/security/basic-auth/simple` – using `.htpasswd`
2. `examples/security/basic-auth/database` – using a database-backed user store

36 API Keys

API keys can protect endpoints with minimal setup effort. Many gateways support API keys and provide infrastructure that scales to large numbers of clients , with features such as key rotation, analytics, and quotas.

In the following example, API key validation is configured globally for all APIs.

```
global:
  - apiKey:
      stores:
        - simple:
            - value: aed8bcc4-7c83-44d5-8789-21e4024ac873
            - value: 61e5a2d8-21aa-4d0b-bdae-76e60ee52803
      extractors:
        - header: X-Api-Key
```

With this configuration, all APIs exposed by the gateway are protected. If you only want to secure a single endpoint, you can add the `apiKey` interceptor directly to that API's flow.

To authenticate, clients must include a valid API key in each request. In this setup, the key is expected in the `X-Api-Key` HTTP header:

```
GET http://localhost:2000
X-Api-Key: 61e5a2d8-21aa-4d0b-bdae-76e60ee52803
```

If the provided API key does not match any of the configured values, the request is rejected with a **403 Forbidden** response.

API Key Extraction

API keys can be passed through different parts of a request, such as HTTP headers, query parameters, or even the message body. Membrane supports flexible extractors that allow API keys to be retrieved from any location.

It is common to allow more than one extraction method at the same time. For example, using headers avoids exposing the key in the logs, while query parameters can provide a convenient way to call an API from a browser or simple tooling without specialized REST or HTTP clients.

The configuration below allows API keys to be extracted from:

- an HTTP header,
- a query parameter, and
- the request payload using JSONPath.

Using SpEL or Groovy expressions, keys can be extracted from virtually any part of the request.

```
extractors:
  - header: X-API-KEY
  - query: api-key
  - expressionExtractor:
      expression: $.api-key
      language: jsonpath
```

With this configuration, clients can authenticate in the following ways:

1. Using HTTP header

```
GET / HTTP/1.1
Host: localhost:2000
X-API-KEY: aed8bcc4-7c83-44d5-8789-21e4024ac873
```

2. Using query string

```
GET /products?api-key=aed8bcc4-7c83-44d5-8789-21e4024ac873
Host: localhost:2000
```

3. Using JSON payload

```
POST / HTTP/1.1
Host: localhost:2000
Content-Type: application/json

{
  "api-key": "61e5a2d8-21aa-4d0b-bdae-76e60ee52803"
}
```

While allowing multiple extraction methods can improve usability, it should be done with care.

⚠ Security Hint: Avoid API keys in query strings
Passing sensitive data in query parameters is generally discouraged. Query strings are frequently logged by browsers, backend services, and even the gateway, which increases the risk of accidental exposure.

⚠ Security Hint: API keys need TLS
API keys may look cryptic, but they are neither encrypted nor inherently protected. They are transmitted in plain text. Regardless of where the API key is extracted from, always use TLS to protect it in transit.

36.1 Storing API Keys

Storing a small number of API keys directly in the gateway configuration is convenient and acceptable in simple setups. However, managing API keys in configuration files does not scale and introduces security risks. For example, API keys may inadvertently be committed to version control together with the configuration.

For this reason, API keys should be stored and managed externally rather than alongside source code or gateway configuration. Suitable options include dedicated files, databases, or specialized key management systems.

Externalizing key storage improves security, simplifies key rotation, and allows independent lifecycle management without requiring changes to the configuration or gateway restarts.

Storing API Keys in a File

In a Membrane API key file, each line contains one API key. Optionally, scopes can be assigned by appending them after a colon, separated by commas:

```
3141
key_321_abc: admin
7a26cae9-ed29-40b3-bc99-5b1914bb8498: read, write
```

Scopes will be used at the end of this chapter for **role-based access control**.

Configuring a file-based API key store is straightforward:

```
components:
  api-key-store:
    apiKeyFileStore:
      location: keys.txt
```

A file store can easily handle hundreds or thousands of API keys. However, as soon as you need key rotation, revocation, or automated provisioning, moving API key storage to a database is usually the better choice.

Storing API Keys in a Database

Using a relational database such as PostgreSQL or MariaDB decouples authentication data from the gateway configuration. This makes automation easier and allows API keys to be managed dynamically without redeploying or restarting the gateway.

The example below shows how to configure Membrane to use PostgreSQL as an API key store:

```
components:
  dataSource:
    bean:
      class: org.apache.commons.dbcp2.BasicDataSource
      properties:
        - name: driverClassName
          value: org.postgresql.Driver
        - name: url
          value: jdbc:postgresql://localhost:5432/postgres
        - name: username
          value: postgres
        - name: password
          value: secret
---
global:
  - apiKey:
      stores:
        - databaseApiKeyStore:
            datasource: '#/components/dataSource'
            keyTable: key
            scopeTable: scope
```

This configuration connects Membrane to a local PostgreSQL database and uses two tables:

- **key** to store the API keys
- **scope** to store the scopes associated with each key

If these tables do not exist yet, Membrane creates them automatically on startup.

Any JDBC-compatible database can be used as an API key store. You must add the appropriate JDBC driver to the `lib` directory of the Membrane distribution.

A complete example, including database setup and Docker scripts, can be found in the `examples/security/api-key/jdbc-api-key-store` folder of the Membrane distribution.

MongoDB API Key Store

In addition to relational databases, Membrane also supports storing API keys in MongoDB. This is particularly useful when supporting large numbers of clients or operating multiple distributed gateway instances.

The configuration below shows how to use a MongoDB collection as the backing store for API keys:

```
api:
  port: 2000
  flow:
    - apiKey:
        stores:
          - mongoDBApiKeyStore:
              connection: mongodb://localhost:27017/
              database: apiKeyDB
              collection: apikey
  target:
    url: https://api.predic8.de
```

To populate the collection with API keys and associated scopes, you can use `mongosh`:

```
mongosh --eval "use('apiKeyDB'); db.apikey.insertMany([
{ id: '345%FSe3', scopes: ['read', 'write'] },
{ id: '3c7f6c34', scopes: ['read'] },
{ id: '343265FA', scopes: ['read', 'admin'] },
{ id: 'flower2025', scopes: ['read', 'write'] }]);"
```

Each document represents a single key. The `scopes` field enables role-based access control and can be evaluated by authorization logic at the gateway.

A complete working example can be found in the `examples/security/api-key/mongodb-api-key-store` folder of the Membrane distribution.

36.2 Role-based Access Control (RBAC)

API keys are not limited to authentication alone. When scopes are assigned to keys, they can also support role-based access control. A scope represents a role or permission such as `read`, `write`, or `admin`. How scopes are stored was shown earlier in the section about API key stores.

If an API key store provides scopes, they are automatically attached to the exchange as it passes through the gateway. The following configuration extracts an API key from the request, resolves its scopes, and forwards them to the backend using an HTTP header:

```
api:
  port: 2000
  flow:
    - apiKey:
        stores:
          - apiKeyFileStore:
              location: api-keys.txt
        extractors:
          - header: X-Api-Key
    - request:
        - setHeader:
            name: X-Scopes
            value: ${scopes()}
  target:
    url: http://localhost:3000
```

At the backend, the scopes can be evaluated to make authorization decisions. The example below protects the `admin` endpoint and only allows access to clients with the `admin` scope:

```
api:
  port: 3000
  path:
    uri: /admin
  flow:
    - request:
        - if:
            test: not header['X-Scopes'].contains('admin')
            flow:
              - static:
                  src: Only admins!
              - return:
                  status: 403
        - static:
            src: You're in!
        - return:
            status: 200
```

This approach keeps authorization logic simple while allowing the gateway to handle authentication and scope resolution centrally.

🔒**Security Hint: Protecting the backend**
If you forward credentials to a backend service, ensure that the backend is not directly accessible from the outside. Use **firewall** rules, trusted IP **allowlists**, or **mutual TLS** to ensure that only the gateway can reach it. Otherwise, clients could bypass the gateway and access the backend directly.

37 JSON Web Tokens

JSON Web Tokens (JWTs) are widely used in modern API security, and Membrane API Gateway supports them on both sides of the authentication flow. Membrane can act in two roles:

- a **JWT issuer**, generating tokens for clients after successful authentication
- a **JWT verifier**, validating incoming tokens before forwarding requests to backend services

The next sections walk through a complete bearer-token flow (as introduced in Chapter *15.2* How (Bearer) Tokens Work*)*, showing how Membrane handles both roles. You can use Membrane for either role or for both at the same time.

37.1 Issuing JWTs

Membrane can function as a JWT issuer, producing signed tokens for clients.

We start with a simple example before building a more complete flow in the next section.

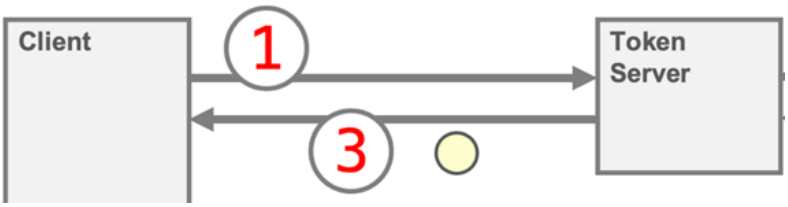


Image: The first steps of the bearer process

To generate a JWT, Membrane needs a signing key. Because JWTs rely on digital signatures, the security of your setup depends on the secrecy and quality of this key. For production environments, an asymmetric RSA key pair with at least **3072 bits** is currently recommended. Review your cryptographic choices periodically, especially as the year 2031 approaches, when stronger algorithms may become advisable.

Step 1: Generate a Private Key

Membrane includes a utility to generate RSA keys in JWK format:

- On Windows:

```
membrane.cmd generate-jwk -o demo.jwk
```

- On Linux:

```
membrane.sh generate-jwk -o demo.jwk
```

This command creates a file named `demo.jwk` containing the private key. **Keep this file secret.** Anyone who obtains it can create tokens that appear legitimate.

Step 2: Configure Membrane to Issue Tokens

Place the `demo.jwk` file in the same directory as `apis.yaml`, then configure a simple token endpoint:

```
api:
  port: 2000
  name: Token Server
  path:
    uri: /token
  flow:
    - request:
        - template:
            contentType: application/json
            src: |
              {
                "sub": "user@predic8.de",
                "aud": "order"
              }
        - jwtSign:
            jwk:
              location: demo.jwk
    - return: {}
```

Calling:

```
curl http://localhost:2000/token
```

returns a freshly issued JWT. Example:

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtpZCI6IjhubHFiNXJ1dGJyYT
k5NzV1NWYwbzM3NTJoIn0.eyJzdWIiOiJ1c2VyQHByZWRpYzguZGUiLCJhdWQi
OiJvcmRlciIsImlhdCI6MTc1MDkzMDUyOSwiZXhwIjoxNzUwOTMwODI5LCJuYm
YiOjE3NTA5MzA0MDl9.WO9ZnO83yjBsefzTAkdLUDxW4pRXNNHFZog6JZNVFYl
y6zUmMn1dYIMy79sBdFBPQ1KA5q6Vy3iVyFAYWSXo6Emb9MGwl5DGL2WlCifrg
UGRJWhrPhtoImYXkj10HWOScEBAZWICC1esJbCTbxGgQr2X1MZk4h0as700Ou0
WSNo-Cyxi2798V69oYwF0i1ALJsVEtTXYOw3k3PU3sMa_g3i3JaUT7I9lnNj5D
p1Wc7B5fpilstVGP1Tt8eTlHhxsafCAArMjBIdhXYUf2KEQp5eKLla-51hKsPr
C3zBnybofqnxtJuOQDqemKYX2aDmf8RUyxledJnnAyWQa90ZGmg
```

Congratulations, your Membrane instance is now issuing JWTs.

When decoded, the token looks like this:

```
{"alg":"RS256","kid":"membrane"}
.
{
  "sub":"user@predic8.de",
  "aud":"order",
  "iat":1750930529,
  "exp":1750930829,
  "nbf":1750930409
}
.
<signature>
```

The payload combines the static claims from the template (`sub` and `aud`) with dynamically generated timestamps (`iat`, `exp`, and `nbf`). Membrane automatically adds `iat`, `nbf`, and `exp` based on default validity settings. Because timestamps change every second, each request produces a slightly different token. The signature also changes, since it depends on both the header and the payload.

Sidenote: The signature
The signature is a binary value produced by the signing algorithm. Before it is added to the JWT, it is Base64URL-encoded so it can be transmitted safely in HTTP headers and URLs.

37.2 Protecting the Token Generation Process

Allowing anyone to request a token is obviously not acceptable. To secure the token generation process, add an authentication layer before Membrane issues a JWT. A common and simple approach is to require an API key, as described in Chapter *36* API Keys.

Static payloads such as:

```
{
  "sub": "user@predic8.de",
  "aud": "order"
}
```

are fine for early testing, but real systems typically derive claims from the authenticated caller. As a simple improvement, we add a `scope` claim based on the API key used.

As an alternative, you could also use API Orchestration to call a remote Identity and Access Management (IAM) API, fetch user attributes and inject them into the template. That approach is more complex but also more flexible.

Requiring an API Key Before Issuing a Token

The following configuration ensures that Membrane only issues a JWT if the caller presents a valid API key:

```
api:
  port: 2000
  name: Token Server
  path:
    uri: /token
  flow:
    - apiKey:
        stores:
          - apiKeyFileStore:
              location: demo-keys.txt
        extractors:
          - header: X-Api-Key
    - request:
        - template:
            contentType: application/json
            src: |
              {
                "sub": "api-user",
                "aud": "order",
                "scope": ${scopes()}
              }
        - jwtSign:
            jwk:
              location: demo.jwk
    - return:
        status: 200
```

The function `scopes()` returns the list of scopes associated with the validated API key.

Create a file called `demo-keys.txt` with the following content:

```
key_321_abc: admin
3141: finance
123456789: finance, accounting
7a26cae9-ed29-40b3-bc99-5b1914bb8498: read, write
```

This setup guarantees:

- Only callers with a **valid API key** (from `demo-keys.txt`) will receive a token.
- The token's payload includes a `scope` **claim**, such as:

  `"scope": ["read", "write"]`

  or

  `"scope": ["admin", "finance"]`

  depending on which API key was used.

A resulting token might look like this:

```
{
  "sub":"api-user",
  "scope": ["read", "write"],
  "aud":"order",
  "iat":1750930529,
  "exp":1750930829,
  "nbf":1750930409
}
```

This setup already gives you a lightweight token server suitable for many API use cases.

💡**Sidenote**: **Scaling up**
Membrane can also delegate token issuance to a dedicated identity providers such as **Keycloak, Microsoft Entra ID**, or **AWS Cognito**.

37.3 Verifying JWTs

Membrane can also act as a **JWT verifier** between clients and backend APIs.

Validating tokens at the API Gateway centralizes authentication, keeps backend services simpler, and ensures consistent enforcement of security policies.

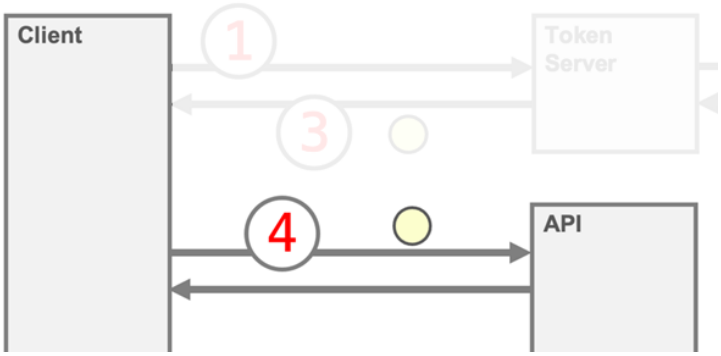


Image: The last steps from the bearer process

Step 1: Set Up an Example Backend API

Let's start with a simple backend service that returns protected content:

```
api:
  port: 3000
  name: Example Backend API
  flow:
    - template:
        src: |
          {
            "content": "Protected content!"
          }
    - return:
        status: 200
```

This backend is intentionally minimal. In a real system, it would implement your business logic, while Membrane would handle authentication before requests reach it.

Step 2: Protect the API with Membrane

Next, we place Membrane in front of the backend and configure it to verify incoming JWTs:

```
api:
  port: 4000
  name: Secured Access to Demo API
  flow:
    - jwtAuth:
        expectedAud: order
        jwks:
          - location: demo.jwk
  target:
    url: http://localhost:3000
```

With this configuration:

- Only requests containing a **valid JWT** are forwarded.
- The token must be signed with the **key corresponding to `demo.jwk`.**
- The token must be intended for the `order` audience, as indicated by its `aud` claim. This ensures that tokens issued for other services cannot be reused here.

If any check fails, Membrane rejects the request and returns an error.

⚠ **Security Tip:**
In production, ensure the backend service (the one on port 3000) is not directly reachable from the outside. All traffic should go through the gateway.

Step 3: Test the Setup

Send a request with a valid token:

```
curl -H "Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI
1NiIsImtpZCI6IjhubHFiNXJ1dGJyYTk5NzV1NWYwbzM3NTJoIn0.eyJzdWIiO
iJ1c2VyQHByZWRpYzguZGUiLCJhdWQiOiJvcmRlciIsImlhdCI6MTc1MDkzMDU
yOSwiZXhwIjoxNzUwOTMwODI5LCJuYmYiOjE3NTA5MzA0MDl9.WO9ZnO83yjBs
efzTAkdLUDxW4pRXNNHFZog6JZNVFYly6zUmMn1dYIMy79sBdFBPQ1KA5q6Vy3
iVyFAYWSXo6Emb9MGwl5DGL2WlCifrgUGRJWhrPhtoImYXkj10HWOScEBAZWIC
C1esJbCTbxGgQr2X1MZk4h0as700Ou0WSNo-Cyxi2798V69oYwF0i1ALJsVEtT
XYOw3k3PU3sMa_g3i3JaUT7I9lnNj5Dp1Wc7B5fpilstVGP1Tt8eTlHhxsafCA
ArMjBIdhXYUf2KEQp5eKLla-51hKsPrC3zBnybofqnxtJuOQDqemKYX2aDmf8R
uyxledJnnAyWQa90ZGmg" localhost:4000
```

If the token is valid, you'll receive:

```
{
  "content": "Protected content!"
}
```

Five minutes later, sending the same request will likely fail:

```
{
  "title": "Security error.",
  "type": "https://membrane-api.io/problems/security",
  "detail": "JWT validation failed.",
  …
}
```

The token has expired. You can either request a new one or adjust the token's validity period, although short lifetimes are generally safer.

37.4 JWT Best Practices

The aud Claim

The JWT issuer and verifier must agree on the audience. On the issuer side, the template contained:

```
"aud": "order"
```

On the verifier side, Membrane was configured with:

```
- jwtAuth:
    expectedAud: order
```

If these values do not match, Membrane rejects the token during verification.

This tight coupling might seem trivial in a small setup with one issuer and one verifier, but it becomes essential in larger architectures. When a single token issuer serves many services that verify tokens, consistent use of the `aud` claim ensures that:

- Tokens are only accepted by the services they were intended for
- Token misuse is minimized, especially when multiple APIs coexist
- You avoid accidental exposure of sensitive endpoints to callers holding tokens meant for other systems

A clear audience strategy prevents entire classes of cross-API access mistakes.

Do not Share the Private Key

In earlier examples (Sections *37.1* Issuing JWTs and *37.3* Verifying JWTs), both the issuer and the verifier used the same `demo.jwk` file. This is acceptable for demonstration purposes but is not appropriate for real systems.

In practice:

- **The private key must remain on the issuer's machine only.**
- **Verifiers should receive only the public key**, typically distributed in JWK format.

Membrane uses the public key to validate signatures, and that's all it needs. Even if a verifier's configuration is exposed, no signing material is leaked because it contains only public information.

Maintaining this separation is a fundamental security requirement when working with asymmetric cryptography.

Using JWT Claims as HTTP Headers

Many backend services are not JWT-aware. They still need caller information without parsing tokens themselves. Membrane can extract claims and forward them as HTTP headers.

Example:

```
- jwtAuth:
    expectedAud: order
    jwks:
      - location: demo.jwk
- setHeader:
    name: X-Sub
    value: ${property.jwt.sub}
```

This:

1. **Verifies the JWT**, checking both the signature and the `aud` claim.
2. **Extracts the `sub` claim** and forwards it as an HTTP header named `X-Sub`.

Example log output:

```
X-Sub: user@predic8.de
```

⚠ **Security Tip: Header spoofing**
Ensure that clients cannot forge headers like `X-Sub`. You should remove or overwrite incoming headers with the same name before adding trusted values.

38 OAuth2 and OpenID Connect

APIs are everywhere, and so are the systems and users trying to access them. Whether the caller is a mobile app, a browser, or a backend service, each one needs a reliable way to prove who they are and what they are allowed to do. OAuth2 and OpenID Connect (OIDC) provide exactly that.

These protocols standardize authentication and authorization across distributed systems. Instead of implementing custom login and access logic for every service, you can rely on well-established and widely supported frameworks that handle identity securely and consistently.

For an API Gateway, OAuth2 and OIDC are essential because the gateway fulfills two important roles:

1. **Token Verification**
   The gateway checks access tokens on incoming API requests before forwarding them to backend services. Strictly speaking, token format and verification happen outside the core OAuth2 and OIDC specifications, but centralizing this logic at the gateway is common and recommended.

2. **Token Acquisition (Web Context)**
   In browser-based scenarios, the gateway can perform the OAuth2 flow on behalf of the user. It redirects the user to the Authorization Server, retrieves the token, and attaches it to downstream requests. This reduces token exposure in frontend code and improves security.

In both cases, backend services receive requests that already contain a validated, trustworthy token. They can use the token's claims to enforce authentication and authorization without needing to speak OAuth2 or OIDC themselves.

You will configure both use cases in the upcoming chapters.

38.1 Token Verification

An API Gateway can **verify** incoming OAuth2 or OIDC **access tokens** before forwarding a request to a backend API.

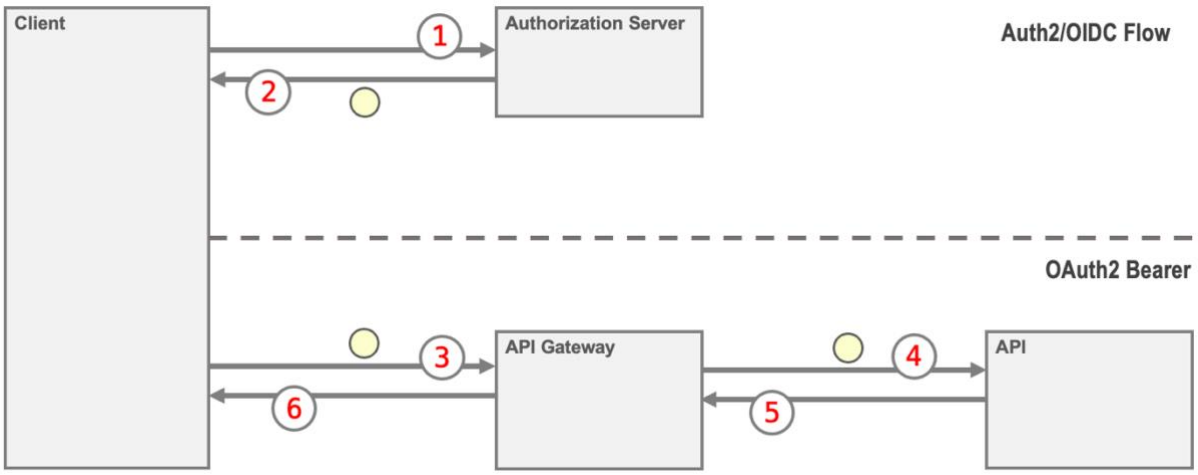


Image: Membrane API Gateway verifying OAuth2 access tokens

The simplest setup is to configure the Authorization Server to issue JWTs (JSON Web Tokens) as access tokens. In that case, verification at the gateway works much like the process described in Section *37.3* Verifying JWTs. With an OIDC-compliant provider, the configuration can be even simpler because key discovery happens automatically.

Membrane ships with a configuration example:

```
examples/security/oauth2/azure-ad-with-jwts
```

The README explains how to use **Microsoft Entra ID** as the Authorization Server.

A typical configuration snippet for JWT verification looks like this:

```
- jwtAuth:
    expectedAud: api://2axxxx16-xxxx-xxxx-faxxxxxxxxf0
    jwks:
      jwksUris: https://login.microsoftonline.com/
                                common/discovery/keys
```

Without **OIDC Discovery**, gateways must be configured manually with the Authorization Server's public keys. With discovery, the gateway fetches the Authorization Server's public keys automatically.

The `expectedAud` value corresponds to the Application ID URI of the *App Registration* in the authentication server (e.g., `api://<application-client-id>`).

Microsoft publishes its JWKS at:

```
https://login.microsoftonline.com/common/discovery/keys
```

Your tenant may use a different URL depending on its settings.

The audience value:

```
api://2axxxx16-xxxx-xxxx-faxxxxxxxxf0
```

refers to the *Application ID* of the *App Registration* created by your Azure administrator.

Flow Overview

Step 1: Token Acquisition
The client obtains an access token from Microsoft.

Step 2: API Request
The client calls the API Gateway and attaches the token:

```
Authorization: Bearer ...token...
```

Membrane forwards the request only if:

- the token is **signed by Microsoft**, and
- the token contains an `aud` (audience) claim that matches the configured value.

Not just Entra ID and not just JWKS

Membrane's JWT verification works with any standards-compliant OAuth2 or OIDC provider. JWKS and Discovery are convenient but optional. You can also:

- configure public keys manually
- use providers without Discovery support
- or integrate with identity systems such as **Auth0**, **Keycloak**, or custom enterprise solutions.

The verification principles remain the same.

38.2 Authorization Code Flow

When operating in a web context, Membrane API Gateway can **obtain an access token on behalf of the user**. In this setup, Membrane identifies the user and their browser through a session cookie. Membrane maintains these sessions in its built-in session store unless configured otherwise.

Session cookies are not part of the OAuth2 specification, but they are a practical and widely used way to track login state in web applications.

Flow Overview (Authorization Code Flow)

Step 1: Token acquisition
When the user's browser accesses the API Gateway without an existing login session, the gateway detects this and starts an *OAuth2 Authorization Code Flow*. After the user authenticates with the Authorization Server, Membrane receives an authorization code, exchanges it for an access token, and stores the resulting access token in the user's session, marking the session as authenticated.

The session cookie is then associated with this access token. Session cookies are not part of OAuth2 itself. They are a gateway-level mechanism used to maintain login state between browser requests.

Step 2: API request
On subsequent requests, the browser sends the session cookie to the gateway. Membrane looks up the session, retrieves the associated access token, and attaches it to the outgoing request before forwarding it to the backend API.

This approach keeps access tokens out of browser JavaScript and reduces the risk of token leakage through XSS attacks. It also centralizes token handling at the gateway, ensuring consistent behavior across all requests and avoiding duplicated OAuth2 logic in multiple frontend applications.

💡**Sidenote: Why use the Authorization Code Flow?**
The Authorization Code Flow is designed for applications running in a browser. It keeps user credentials away from the client application and performs the token exchange on the server. When combined with **Proof Key for Code Exchange (PKCE)**, which Membrane enables by default, it is the most secure OAuth2 flow for public clients such as single-page applications (SPAs) and mobile apps.

39 Legacy Integration of SOAP Web Services

Software does not come with an expiration date, and some systems outlive the era they were designed for. Legacy protocols, especially **XML**-based technologies like **SOAP** and **Web Services**, are still deeply rooted in many enterprise environments.

The challenge is often not getting rid of them but making old and new systems work together.

As long as a legacy system communicates over HTTP, it can be integrated through an API Gateway. This allows you to apply gateway features such as routing, authentication, and transformation, even if the backend feels like it came straight from 2004.

Some API Gateways, including Membrane, go beyond simple HTTP forwarding and provide dedicated support for XML and SOAP services. Typical capabilities include:

- XML processing and transformation using **XSLT**
- Handling SOAP requests and responses
- **XML-to-JSON** and **JSON-to-XML** transformations
- **Validation** of messages against **XSD schemas and WSDL** documents
- Routing based on XML wrapper elements, SOAPAction headers, or WS-Addressing

SOAP-to-REST capabilities allow you to expose a modern, JSON-friendly REST API to clients while continuing to use an XML-based backend interface. This gives you the best of both worlds: a stable backend and a clean, modern API facade.

In short, an API Gateway can serve as a modernization bridge. Instead of replacing legacy systems, you wrap them with a modern API layer.

39.1 Sample Web Services Mock

For **testing** and development, it's helpful to have a simple Web Service available. Membrane includes the `sampleSoapService` plugin that provides a basic SOAP service mock.

By deploying the following configuration:

```
api:
  port: 2000
  path:
    uri: /city-service
  flow:
    - sampleSoapService: {}
```

you can access a WSDL document by visiting:

```
http://localhost:2000/city-service?wsdl
```

This WSDL allows you to use tools like **SoapUI** to generate and send requests to the service.

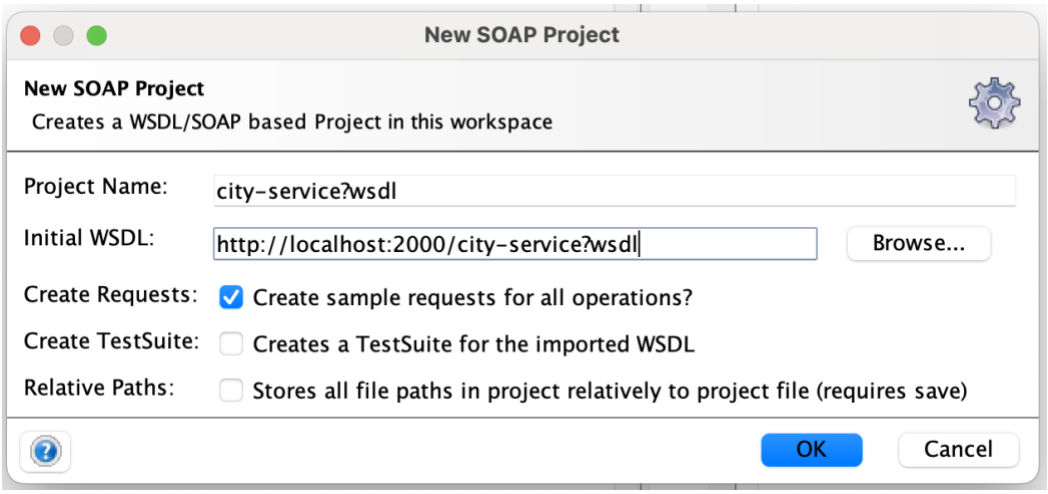


Image: Creating a project from a WSDL document in SoapUI

You do not need specialized tools to try out the service. You can use **curl**, or the **REST client extension in Visual Studio Code**. Here's an example request:

```
POST /city-service
Host: localhost:2000
Content-Type: text/xml

<s11:Envelope
xmlns:s11="http://schemas.xmlsoap.org/soap/envelope/">
  <s11:Body>
    <getCity xmlns="https://predic8.de/cities">
      <name>Bonn</name>
    </getCity>
  </s11:Body>
</s11:Envelope>
```

The `sampleSoapService` plugin responds with a SOAP message:

```
HTTP/1.1 200 OK
Content-Type: text/xml

<s:Envelope
  xmlns:s="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:cs="https://predic8.de/cities">
  <s:Body>
    <cs:getCityResponse>
      <country>Germany</country>
      <population>327000</population>
    </cs:getCityResponse>
  </s:Body>
</s:Envelope>
```

Even if the gateway does not offer SOAP support, you can mock a SOAP Web Service using a template. That's exactly what the next section will cover.

39.2 Mocking a Web Service

You can simulate a Web Service by returning a static SOAP response directly from the gateway. This is a simple technique that works with almost any API Gateway.

The example below shows an API that responds with a SOAP envelope when it is called:

```
api:
  port: 2000
  flow:
    - response:
      - static:
          pretty: true
          contentType: text/xml
          src: |
            <s11:Envelope
             xmlns:s11="http://schemas.xmlsoap.org/soap/envelope/">
              <s11:Body>
                <getCityResponse xmlns="https://predic8.de/cities">
                  <country>England</country>
                  <population>8980000</population>
                </getCityResponse>
              </s11:Body>
            </s11:Envelope>
    - return:
        status: 200
```

(To avoid typing, download the samples from `https://membrane-api.io/gw.txt`)

It does not matter how the API is called. It accepts all requests on port 2000, regardless of whether the request is a valid SOAP message.

This approach is helpful for testing and development. It allows you to simulate legacy services and test client integrations without setting up or maintaining a full SOAP backend.

Using the SoapBody Template

Membrane provides a dedicated `soapBody` template for Web Services that makes crafting SOAP messages easier. Instead of manually writing the SOAP envelope, you only define the payload. The gateway automatically wraps it with the correct SOAP envelope and body element.

The example from the previous section can be simplified like this:

```
api:
  port: 2000
  flow:
    - response:
        - soapBody:
            version: '1.2'
            pretty: true
            src: |
              <getCityResponse xmlns="https://predic8.de/cities">
                <country>England</country>
                <population>8980000</population>
              </getCityResponse>
    - return:
        status: 200
```

Membrane selects the correct SOAP namespace and sets the appropriate `Content-Type` header based on the configured SOAP version:

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml

<s12:Envelope
     xmlns:s12="http://www.w3.org/2003/05/soap-envelope">
  <s12:Body>
    <getCityResponse xmlns="https://predic8.de/cities">
      <country>England</country>
      <population>8980000</population>
    </getCityResponse>
  </s12:Body>
</s12:Envelope>
```

For SOAP 1.1, the `Content-Type` would be `text/xml`, and the namespace would change accordingly.

39.3 Exposing SOAP Web Services as REST APIs

Old SOAP-based Web Services can get a second life by exposing them as modern, REST-style JSON APIs. A gateway can act as a translation layer that:

- Transforms messages between JSON and SOAP
- Maps API endpoints and REST resources to SOAP operations
- Translates HTTP methods and sets the appropriate HTTP headers

All of this happens without requiring changes to the backend service.

REST-to-SOAP integration is particularly useful when modernizing systems incrementally or when the Web Services backend is owned by a third party and cannot be modified. The gateway provides a modern API facade while keeping the existing service untouched.

REST GET to SOAP

The diagram shows the flow from an incoming `GET /cities/{city}` request (top left) to a JSON response (top right). In between, the gateway builds a SOAP request, switches the method from GET to POST, forwards the request to the SOAP service, then converts the response back into JSON.

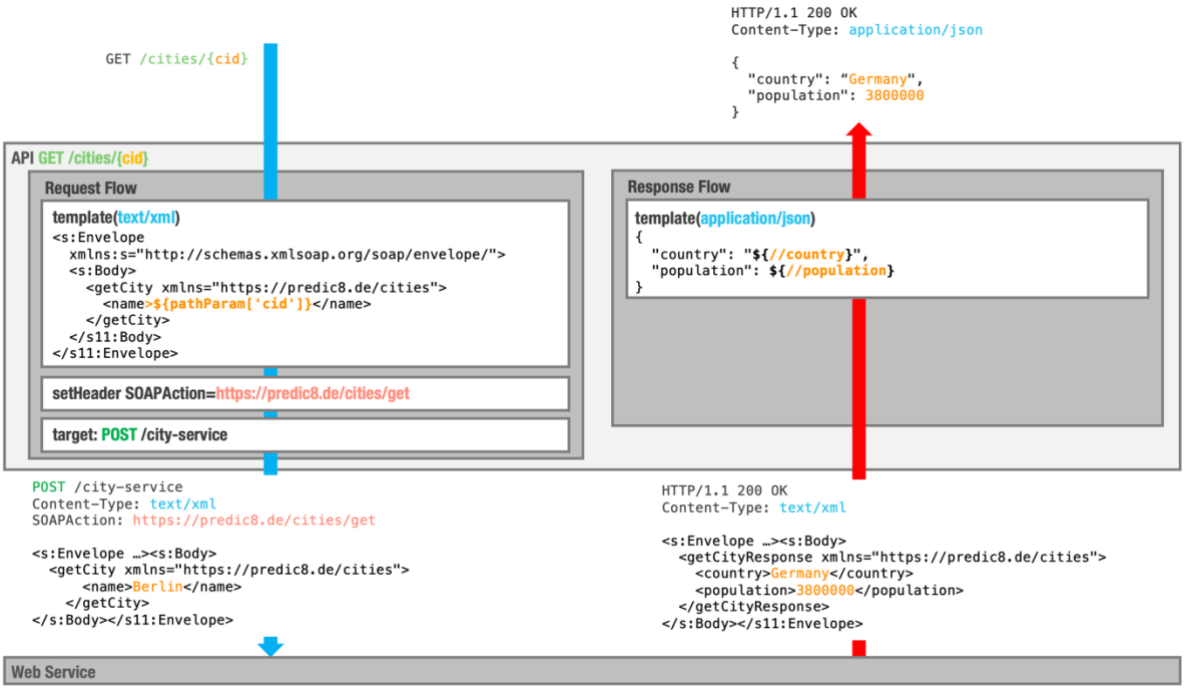


Image: Creating a RESTful GET resource for a SOAP service

An implementation for that setup might look like this:

```
api:
  port: 2000
  method: GET
  path:
    uri: /cities/{city}
  flow:
    - request:
        - soapBody:
            src: |
              <getCity xmlns="https://predic8.de/cities">
                  <name>${pathParam.city}</name>
              </getCity>
        - setHeader:
            name: SOAPAction
            value: https://predic8.de/cities/get
    - response:
        - setBody:
            language: xpath
            contentType: application/json
            value: |
              {
                "country": ${//country},
                "population": ${//population}
              }
  target:
    method: POST
    url: https://www.predic8.de/city-service
```

This setup performs the following steps:

- Accepts a REST-style `GET` request at `/cities/{city}`
- Inserts the `city` path parameter into a SOAP request
- Sets the `SOAPAction` header
- Sends the SOAP message to the Web Service
- Extracts `country` and `population` from the SOAP response
- Returns a JSON message to the client

Some SOAP implementations rely on the `SOAPAction` HTTP header to route the request to the correct operation. Ensure that the correct `SOAPAction` value is specified.

```
SOAPAction: https://predic8.de/cities/get
```

You can typically find it in the WSDL binding section, for example:

```
<wsdl:binding name="CSBinding" type="cs:CSPort">
  <s:binding style="document"
             transport="http://schemas.xmlsoap.org/soap/http">
  </s:binding>
  <wsdl:operation name="getCity">
    <s:operation soapAction="https://predic8.de/cities/get"/>
  </wsdl:operation>
</wsdl:binding>
```

Transforming JSON Lists into XML

Templates can also transform complex structures such as lists or nested objects between JSON and XML.

This JSON document contains a list with fruits:

```
{
  "fruits": [
    { "name": "Apricot","price": 3.87 },
    { "name": "Date", "price": 2.35 }
  ]
}
```

A template can turn it into a SOAP message like this:

```
<s11:Envelope xmlns:s11="http://schemas.xmlsoap.org/soap/envelope/">
  <s11:Body>
    <f:getFruitsResponse xmlns:f="https://predic8.de/fruits">
      <fruit>
        <name>Apricot</name>
        <price>3.87</price>
      </fruit>
      <fruit>
        <name>Date</name>
        <price>2.35</price>
      </fruit>
    </f:getFruitsResponse>
  </s11:Body>
</s11:Envelope>
```

The template reads the list from the JSON body using the JSONPath expression `$.fruits` and then iterates over that list to render one `<fruit>` element per entry:

```
- soapBody:
    pretty: true
    src: |
      <f:getFruitsResponse xmlns:f="https://predic8.de/fruits">
        <% jsonPath('$.fruits').each { it -> %>
          <fruit>
            <name>${it.name}</name>
            <price>${it.price}</price>
          </fruit>
        <% } %>
      </f:getFruitsResponse>
```

`soapBody` automatically generates the SOAP envelope around the provided body content.

The expression language used inside the template is Groovy. That means you can use Groovy functions for conditions or number formatting.

Transforming XML Lists to JSON

This section covers the reverse direction: transforming a list from XML to JSON. In this example, a SOAP message is converted back into the original JSON structure.

The input XML contains namespaces, which makes XPath expressions slightly more involved:

```
<f:getFruitsResponse xmlns:f="https://predic8.de/fruits">
  <f:fruit>
    <f:name>Apricot</f:name>
    <f:price>3.87</f:price>
  </f:fruit>
  <f:fruit>
    <f:name>Date</f:name>
    <f:price>2.35</f:price>
  </f:fruit>
</f:getFruitsResponse>
```

To work with namespaced elements, a global `xmlConfig` component maps the prefix `f` to its namespace URI:

```
components:
  ns:
    xmlConfig:
      namespaces:
        - prefix: f
          uri: https://predic8.de/fruits
```

The configured prefix can then be used in XPath expressions inside the template:

```
- template:
    pretty: true
    contentType: application/json
    src: |
      <%
        def fruits = xpath('//f:fruit')
        def total = fruits.getLength()
      %>
      { "fruits": [
          <% for (int i = 0; i < total; i++) {
             def fruit = fruits.item(i)
          %>
              {
                "name": <%=xpath('string(./f:name)', fruit)%>,
                "price": <%=xpath('number(./f:price)',fruit)%>
              }<%= (i < total - 1) ? "," : "" %>
          <% } %>
        ]
      }
```

The XPath `//f:fruit` searches the entire XML document for `fruit` elements in the namespace `https://predic8.de/fruits`. The result is a `NodeList`, which the template iterates over using an indexed loop.

Inside the loop, the variable `fruit` references the current node:

```
def fruit = fruits.item(i)
```

For the first list element, `fruit` contains:

```
<f:fruit>
  <f:name>Apricot</f:name>
  <f:price>3.87</f:price>
</f:fruit>
```

To extract values, the helper function `xpath` executes XPath relative to the current node.

```
<%= xpath('number(./f:price)', fruit) %>
```

This reads the `<f:price>` element and converts its value into a JSON number.

The result is a JSON document with a `fruits` array, matching the structure from the previous section. With both transformations in place, you can build a complete conversion cycle between REST and SOAP in either direction, enabling gradual modernization without replacing the backend.

Using Groovy for Message Transformation

Nothing beats a scripting language when it comes to message transformation, in particular for complex data mappings and manipulations. This is notably true for **Groovy**, which offers expressive and concise support for both **JSON and XML**.

Below is the same transformation shown earlier, implemented using Groovy:

```
- groovy:
    src: |
      import groovy.xml.XmlSlurper
      import groovy.json.JsonOutput

      def xml = body.toString()

      def root = new XmlSlurper(false, true).parseText(xml)
      root.declareNamespace(
        s11: "http://schemas.xmlsoap.org/soap/envelope/",
        f:   "https://predic8.de/fruits"
      )

      def fruits = root.'s11:Body'
                       .'f:getFruitsResponse'
                       .'f:fruit'.collect { n ->
        [
          name : n.'f:name'.text(),
          price: new BigDecimal(n.'f:price'.text())
        ]
      }

      def payload = JsonOutput.toJson([fruits: fruits])
      message.setBodyContent(payload.getBytes())
```

`XmlSlurper` allows you to navigate XML documents as if they were object graphs, making path-style access intuitive and concise. Namespaces can be declared explicitly and then referenced in element navigation.

`JsonOutput` converts data structures such as maps and lists directly into JSON. Together, these tools make Groovy a powerful option for complex transformations involving XML, JSON, and calculations.

For us, writing and later maintaining a transformation in Groovy is easier than working with XSLT or large templates. The syntax is familiar, debugging is possible, and the logic can be structured like regular code.

By moving scripts into external files, API configurations become much cleaner and easier to read. As a useful side effect, this also enables automated and isolated testing of the transformation logic without running the gateway itself.

In the following setup, the request and response transformations are fully delegated to Groovy scripts stored in separate files. Different scripts handle normal SOAP responses and SOAP faults, keeping each transformation focused and easy to maintain.

```
api:
  port: 2000
  path:
    uri: /products
  flow:
    - request:
        - groovy:
            location: json-to-soap-request.groovy
    - response:
        - if:
            test: //*[local-name()='Fault']
            language: xpath
            flow:
              - groovy:
                  location: soap-fault-to-json.groovy
            else:
              - groovy:
                  location: soap-response-to-json.groovy
  target:
    url: http://localhost:2000/product-service
```

You can find the complete example, including all referenced Groovy scripts, in the `tutorial/transformation` folder of the Membrane distribution.

REST Create with POST to SOAP

Various REST-to-SOAP mappings can be implemented using the building blocks introduced so far. A common scenario is exposing an existing SOAP service as a REST-style POST endpoint.

In this setup, the gateway accepts a JSON payload from the client, renders a corresponding SOAP request using a template, and forwards it to the Web Service.

On the response path, the gateway processes the SOAP response, extracts the relevant values, and converts them into a JSON representation. It can also adapt HTTP semantics to REST conventions, for example by returning status code `201 Created` and setting a `Location` header.

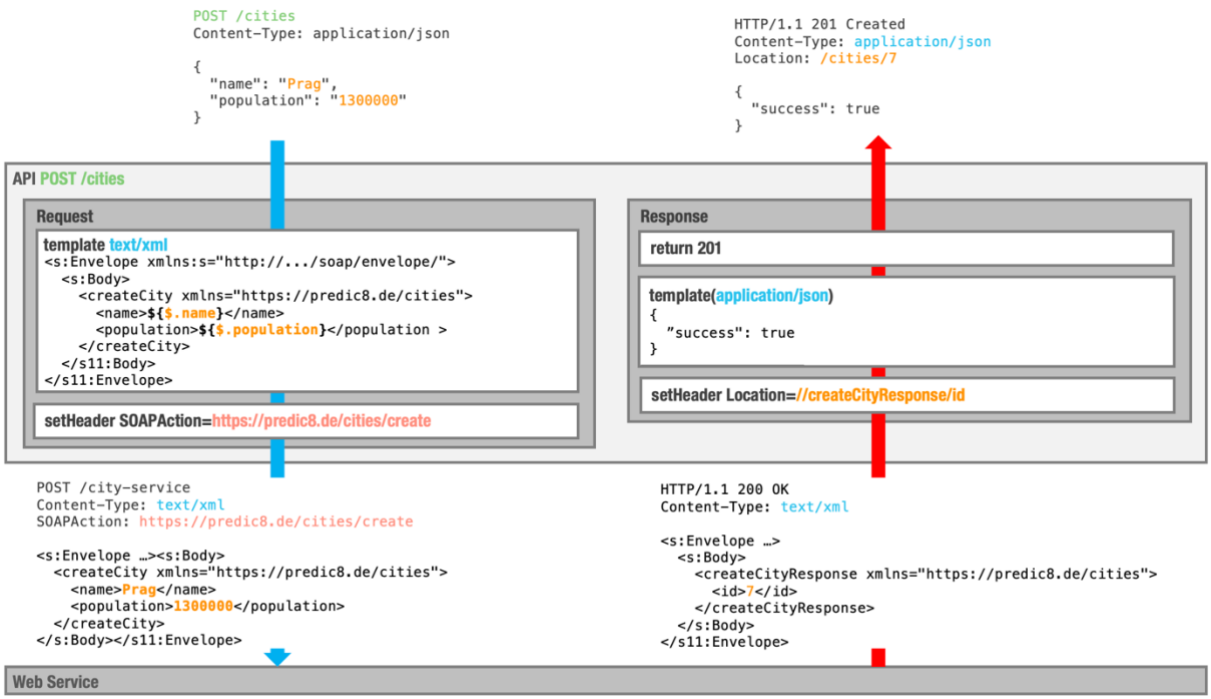


Image: Mapping a SOAP operation to a RESTful POST endpoint

Routing SOAP Requests

As OpenAPI shows, REST API endpoints are identified by the combination of an HTTP method and a path.

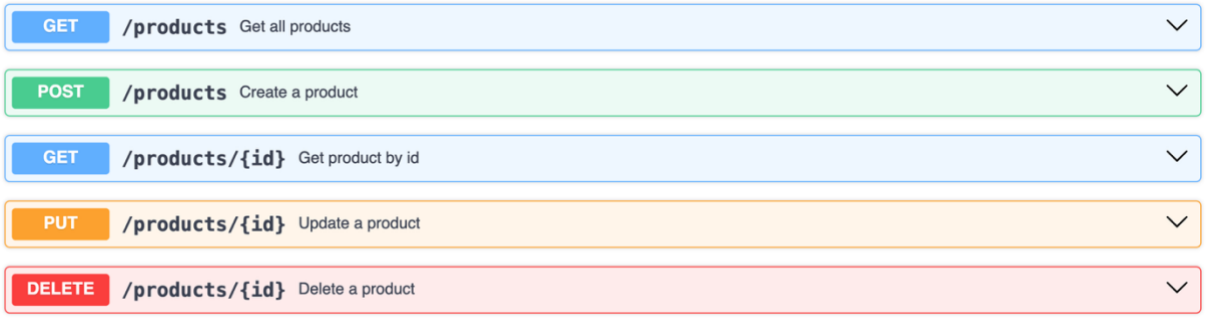


Image: Endpoints in OpenAPI

In REST, each resource has its own address, which makes routing straightforward. Throughout this book, we've used configurations like this:

```
api:
  port: 2000
  method: POST
  path:
    uri: /products
```

With SOAP, routing is different. A single Web Service typically exposes multiple operations, but all of them share the same endpoint. For example, a product service might expose all its operations at:

```
POST /product-service
```

Since all requests are sent to the same address, the gateway cannot distinguish operations based on method or path alone. Instead, it must inspect the message payload to determine the operation. In SOAP, the operation is identified by the name of the XML element inside the SOAP body.

```
<s11:Envelope xmlns:s11="http://schemas.xmlsoap.org/soap/envelope/">
    <s11:Body>
        <ps:createProduct xmlns:ps="https://predic8.de/products">
            <name>Sapodilla</name>
            <price>1.80</price>
        </ps:createProduct>
    </s11:Body>
</s11:Envelope>
```

Here, the element `<ps:createProduct>` identifies the SOAP operation. Routing is typically implemented by using the name of this element to select the correct API.

Membrane allows routing decisions based on message content. Instead of matching on method or path, you can define a test expression. If the test evaluates to true, the API is selected.

XPath is a natural fit for SOAP routing because it allows precise matching against XML structures inside the SOAP body.

First, define the XML namespace mapping:

```
components:
  ns:
    xmlConfig:
      namespaces:
        - prefix: ps
          uri: https://predic8.de/products
```

Then define APIs that route based on the SOAP operation:

```
api:
  port: 2000
  test: //ps:getProducts
  language: xpath
  ...

---
api:
  port: 2000
  test: //ps:createProduct
  language: xpath
  ...
```

Each API is selected based on the presence of the corresponding operation element in the SOAP body.

If you are using a gateway that does not support XPath-based routing directly, you can achieve similar behavior by using a scripting interceptor or implementing a custom plugin to inspect the message content.

SOAPAction Routing

Some SOAP stacks determine the operation to execute based on the `SOAPAction` HTTP header. The advantage of using a header instead of inspecting the message body is that routing decisions can be made without parsing the XML payload. This can improve performance and allows routing decisions earlier in the request processing pipeline.

At the gateway level, the value of the `SOAPAction` header can be evaluated using a simple expression. This makes it possible to route SOAP messages based on the intended operation even at networking equipment that lacks SOAP or XML awareness.

```
api:
  port: 2000
  test: header.SOAPAction == 'https://www.predic8.de/get-city'
  ...

---
api:
  port: 2000
  test: header.SOAPAction == 'https://www.predic8.de/create-city'
  ...
```

39.4 Automatic SOAP to REST Transformation

The transformation between SOAP and REST is tricky. There is no general algorithm to follow, and building a tool that does this reliably is hard. The main reason is that the two styles encode meaning in different ways.

SOAP models **Remote Procedure Calls (RPC)**, where interactions are expressed as named operations. REST models **resources** and uses HTTP methods to express actions on those resources. When you try to translate one into the other, you run into naming and semantics questions that a tool cannot answer without real-world domain knowledge.

In practice, automation can assist with the mechanical parts, such as generating stubs, or producing example requests. However, the core design decisions require someone who understands the business domain, whether that is a developer or a well-guided LLM.

Take a SOAP style operation like:

```
cancelOrder(orderId)
```

When we asked an AI to map this RPC call to REST, three different solutions emerged:

```
POST /orders/{orderId}/cancel
PATCH /orders/{orderId}
DELETE /orders/{orderId}
```

All three could make sense depending on what *cancel* means in the business domain. Is the order being deleted, or is its status changing to canceled? Is cancel an idempotent action? Does the system allow partial updates?

Now try the reverse direction. How would you map the REST endpoint:

```
DELETE /orders/{orderId}
```

back to an RPC style call? Is it `deleteOrder(orderId)` or `cancelOrder(orderId)`? Without knowing whether the resource is truly removed or only marked as canceled, both names are plausible, and they imply different behavior.

39.5 Best Practices

Here are several practical recommendations when doing legacy integrations.

One API for One Endpoint

Web Services often expose multiple operations. It may seem logical to handle all of them within a single API definition at the gateway. In practice, this quickly leads to large and hard-to-maintain configurations.

A proven approach is to map each SOAP operation to its own API definition. If a WSDL defines 12 operations, create 12 separate API configurations at the gateway. This keeps each flow focused, easier to read, and simpler to test and evolve.

From the client's perspective, these APIs can still appear as a single logical API by sharing a common base path. They can also be described together in one OpenAPI definition.

Be Explicit About Contracts

When mapping between SOAP and REST-style APIs, work with clearly defined contracts. Use WSDL for the Web Service and OpenAPI for the REST API.

With explicit contracts in place, you can focus on the mapping between the two interfaces. Without them, discussions about field names, data types, and edge cases will consume your time and energy.

Clear contracts reduce ambiguity, align expectations early, and turn integration work into a technical task instead of an ongoing negotiation about the business domain.

Use AI to Generate Transformations

A fully automated REST to SOAP mapping is probably still hard to get right end to end. But AI can be a huge help when you create transformations, especially for templates and scripts. In our experience, as of early 2026, the first output is often close but rarely usable without adjustments. The sweet spot is iterative work: prompt, run it, collect errors, adjust the prompt, and feed back the failure details. After a few rounds you typically end up with something usable much faster than starting from scratch.

If you want to try this yourself, here is a prompt you can use to generate a simple SOAP to REST integration with XML to JSON mapping.

Prompt: For a SOAP to REST Transformation

```
Create a Membrane API Gateway configuration that exposes the
REST endpoint POST /shop/v2/products as SOAP Web Service. Map SOAP
version 1.1 request bodies to JSON and map JSON responses back
to SOAP, including status codes and Content-Type headers. Create
a lean solution. Do not include security, logging, health
checks, or other unneeded features.

Deliverables:
- A Membrane YAML configuration (v7) implementing the mapping.
Request and response templates in Groovy for the mapping.
- A WSDL with the SOAP operation, including XSD, messages,
portType, binding and service.
- Example curl commands and SOAP requests for testing.
- Document the mapping in comments in the YAML and WSDL.
```

- Error handling: Map REST status code to SOAP Faults.
- Guide explaining how to set up the solution

Assumptions:
- REST request contains name and price.
- If any of these assumptions are incorrect, infer the most reasonable mapping.

40 Proxying SOAP

In the previous chapter, we explored how an API Gateway can modernize SOAP services by transforming messages and exposing REST-style interfaces. In many situations, however, no transformation is required. Clients and backend systems both speak SOAP, and the gateway simply needs to proxy the service while adding capabilities such as routing, validation, or security.

For this purpose, Membrane provides a specialized configuration shortcut called **soapProxy**. Instead of manually configuring SOAP-specific interceptors and WSDL handling, the proxy can derive much of its configuration directly from the service description. This allows you to expose or protect SOAP services with minimal setup while still benefiting from gateway features.

A Proxy for SOAP Services

The `soapProxy` simplifies routing and provides SOAP-specific capabilities out of the box:

- WSDL publishing and URL rewriting
- Request and response validation based on the WSDL
- A built-in Web Services explorer

This makes it ideal for exposing or proxying legacy SOAP services with minimal configuration.

Behind the scenes, a `soapProxy` is translated into a regular `api` configuration that is preconfigured for Web Services. It is essentially syntactic sugar that saves you from repeating common SOAP-related configuration steps.

The following example configures a SOAP proxy using a WSDL hosted on a remote server:

```
soapProxy:
  port: 2000
  wsdl: https://www.predic8.de/city-service?wsdl
```

On startup, Membrane retrieves the WSDL, analyzes it, and automatically configures routing, rewriting, and validation.

The service address, namespaces, and other metadata from the WSDL are used to configure the proxy automatically. This highlights one of the core strengths of service descriptions such as WSDL and OpenAPI: their metadata can be used at runtime for validation, routing, and configuration.

After startup, the WSDL is available at:

```
http://localhost:2000/city-service?wsdl
```

Clients can access the service through the gateway without changing their configuration, while the gateway handles routing and validation transparently.

Rewriting of WSDL URLs

The WSDL document of a service is not simply passed through by the gateway. It is dynamically rewritten before being returned to the client. In particular, the `<s:address>` and `<xsd:import>` elements are updated to point to the gateway rather than the original backend.

When you retrieve the WSDL from the gateway at the following URL:

```
http://localhost:2000/city-service?wsdl
```

Membrane derives the service address from the request URL and replaces the original address in the `service` section of the WSDL:

```
<wsdl:service name="CityService">
  <wsdl:port name="CityPort" binding="cs:CityBinding">
    <s:address location="http://localhost:2000/city-service">
    </s:address>
  </wsdl:port>
</wsdl:service>
```

By default, Membrane derives the **protocol**, **host**, and **port** from the client request and uses those values to rewrite the address dynamically. This works well in simple setups where the gateway is directly accessible.

In real-world deployments, however, the gateway often runs **behind a firewall**, a reverse proxy, or within a **containerized environment**. In such cases, the address seen by external clients differs from the gateway's internal address. To handle this, you can configure a fixed public address using the `wsdlRewriter` interceptor:

```
soapProxy:
  path:
    uri: /my-service
  port: 2000
  wsdl: https://www.predic8.de/city-service?wsdl
  flow:
    - wsdlRewriter:
        host: my.host.example.com
        protocol: https
        port: 443
        path: /my-service
```

With this configuration, the gateway rewrites the WSDL so that it points to the configured external address. This ensures that clients importing the WSDL, for example into SoapUI or when generating client code, are directed to the correct public-facing endpoint exposed by the gateway.

Web Service Explorer

For convenience, Membrane includes a built-in Web Service explorer. You can access it by opening the base service URL in a browser:

```
http://localhost:2000/my-service
```

The explorer is only served when Membrane recognizes a browser based on the `accept` HTTP header. If the request does not appear to come from a browser, a SOAP error is returned instead.

The explorer provides a simple web interface that summarizes the proxied Web Service. It shows the target namespace, available operations, input and output messages, and the rewritten WSDL served by the gateway.

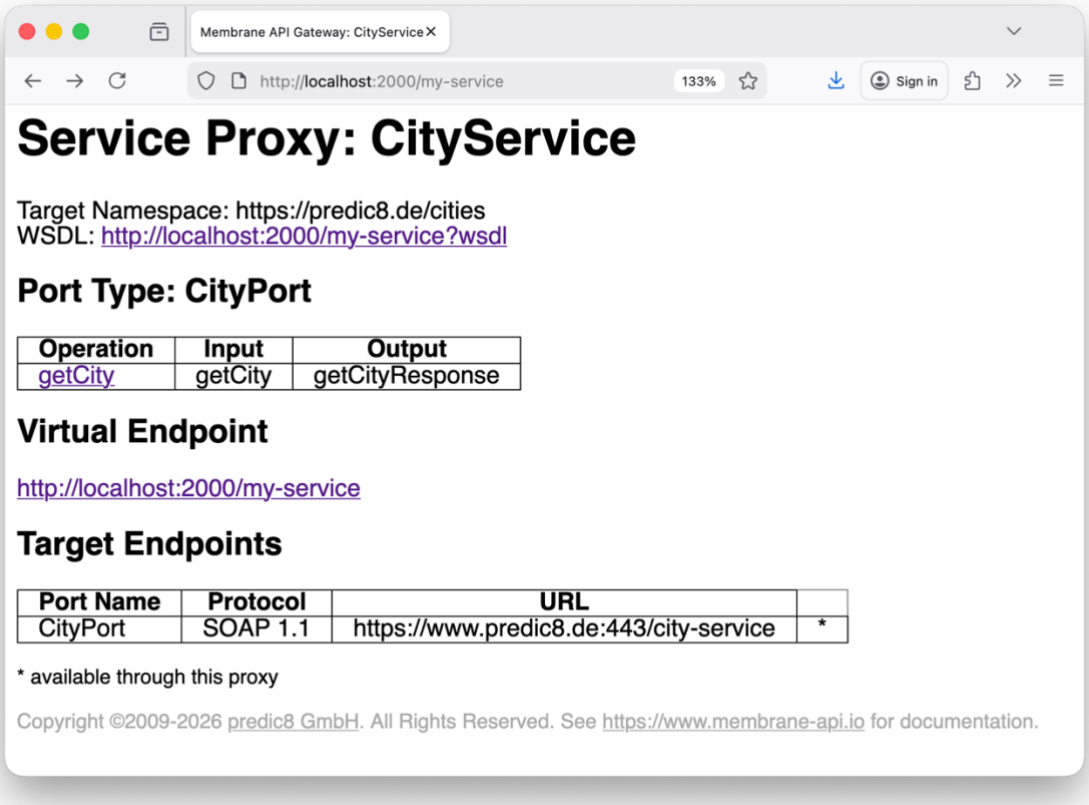


Image: Web Services Explorer

With a click on an operation such as **getCity**, Membrane displays a sample SOAP request generated automatically from the WSDL. This makes it easy to understand the expected message structure and is useful for testing and debugging.

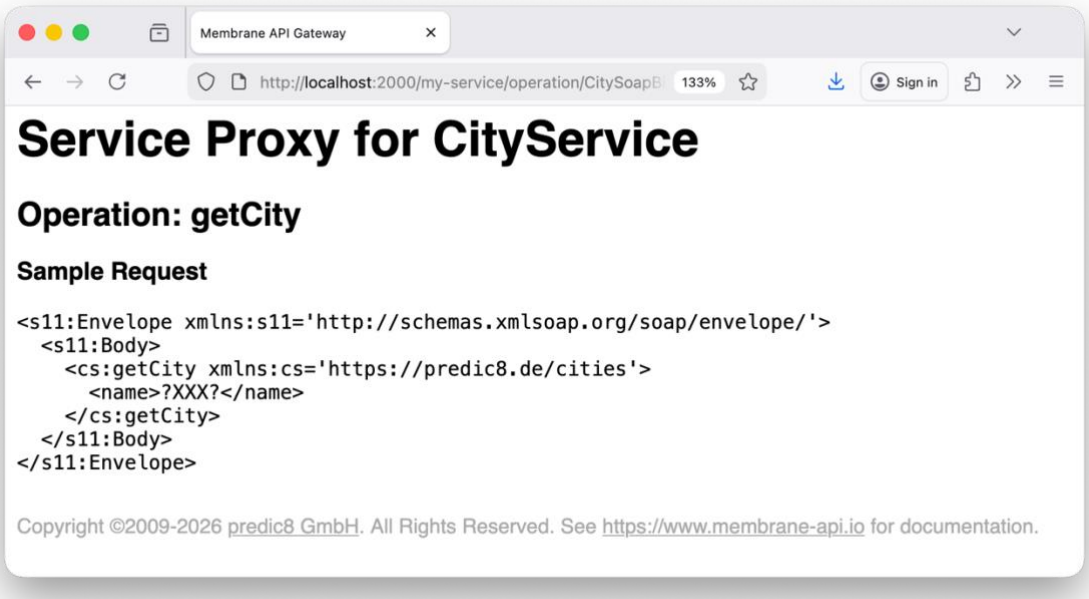


Image: SOAP message template

WSDL Validation

The `soapProxy` can validate SOAP messages against the definitions in the WSDL. When enabled, both incoming requests and outgoing responses are checked against the WSDL structure and the referenced XSD schema types.

```
soapProxy:
  port: 2000
  wsdl: https://www.predic8.de/city-service?wsdl
  flow:
    - validator: {}
```

With validation enabled, the gateway ensures that messages conform to the service contract before they reach the backend or the client.

If a request does not match the expected format, the gateway immediately returns a SOAP fault with detailed validation errors. For example, the following invalid SOAP request:

```
<s11:Envelope ..>
   <s11:Body>
      <cit:getCity>
         <foo/>
      </cit:getCity>
   </s11:Body>
</s11:Envelope>
```

results in this response:

```
HTTP/1.1 400 Bad Request
Content-Type: text/xml;charset=UTF-8
X-Validation-Error-Source: REQUEST

<soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <Fault>
      <faultcode>Client</faultcode>
      <faultstring>WSDL message validation failed</faultstring>
      <detail>
        <validation>
          <item>
            <message>cvc-complex-type.2.4.a: Invalid content was
found starting with element 'foo'. One of '{name}' is
expected.</message>
            <line>5</line>
            <column>16</column>
          </item>
        </validation>
      </detail>
    </Fault>
  </soap:Body>
</soap:Envelope>
```

This early feedback helps detect incorrect client requests immediately and enforces strict contract-based communication. It also improves the security of exposed services and is valuable in enterprise environments, where SOAP services often rely on strict message structures and strong schema guarantees.

How a soapProxy works

A `soapProxy` configures itself automatically from a WSDL description and assembles the required interceptors under the hood.

The diagram visualizes a `soapProxy` with WSDL rewriting and validation enabled. Incoming requests are accepted by the listener and then pass through a chain of SOAP-specific interceptors before being forwarded to the target Web Service.

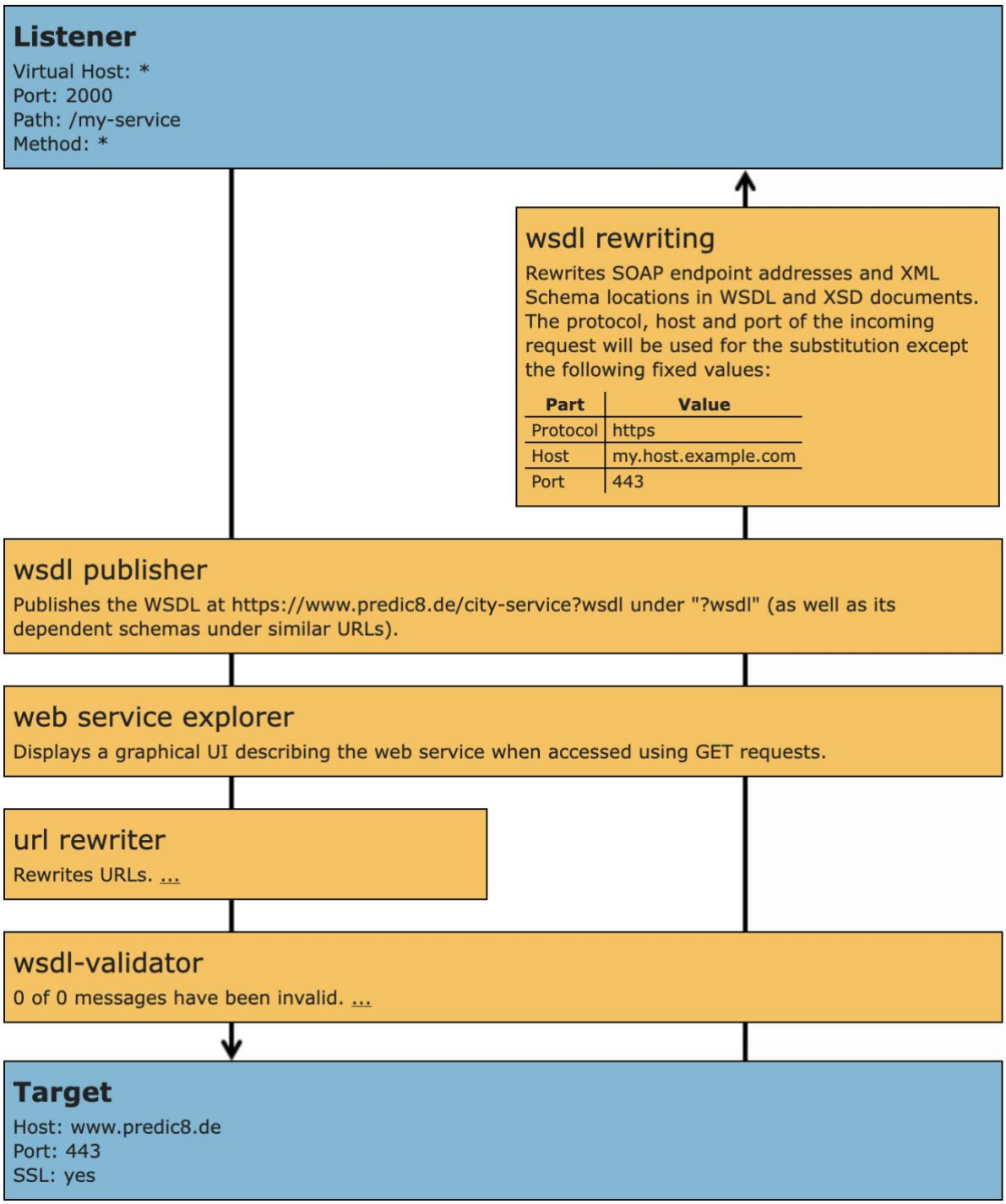


Image: Visualization of a soapProxy in the admin console

Instead of using a `soapProxy`, you can configure a regular `api` and add all required interceptors manually. The `tutorials/soap` folder in the Membrane distribution contains examples that demonstrate this step by step. Building the flow manually gives you full control and allows configurations that go beyond the defaults provided by `soapProxy`.

Despite its convenience, the `soapProxy` is not a black box. It can be extended like any other API by adding additional interceptors to its flow:

```
soapProxy:
  port: 2000
  wsdl: https://www.predic8.de/city-service?wsdl
  flow:
    - log:
        message: "Path: ${path}"
    - rateLimiter:
        requestLimit: 1000
        requestLimitDuration: PT1H
```

41 Operation

Running an API Gateway is not a one-time task. It requires continuous operation, monitoring, and maintenance to keep APIs reliable and secure. Whether you are exposing a handful of APIs or operating a high-traffic platform, day-to-day operations make the difference.

This chapter focuses on the operational side of an API Gateway. It covers monitoring, observability, and message tracing. Everything you need to stay in control once the gateway is live.

Gateways provide a set of features to monitor, debug, and observe API traffic, both in real-time and retrospectively. While these tools are helpful during development and testing, they become essential in production environments.

41.1 Admin Console

In enterprise environments, centralized monitoring platforms such as Grafana, Splunk, or Elastic Stack are commonly used to consolidate operational data from many different systems. As a result, product-specific dashboards are no longer as essential as they once were.

Even if not strictly required, a gateway-specific admin console can provide capabilities that complement centralized monitoring. It can offer configuration visibility, quick diagnostics, and targeted troubleshooting that are difficult to achieve through aggregated monitoring tools alone.

Activating the Console

Membrane, like many other gateways, can run without a user interface. This keeps resource consumption low and minimizes the attack surface.

To use the admin console, you expose it through a regular API configuration. The console itself is implemented as a plugin, similar to other interceptors.

```
api:
  port: 9000
  flow:
    - adminConsole: {}
```

Once this configuration is active, point a browser to:

http://localhost:9000

You will be greeted with an overview of the deployed APIs and some basic statistics. Separate tabs provide visualizations of API configurations, runtime information, a list of clients, and the state of load balancer nodes.

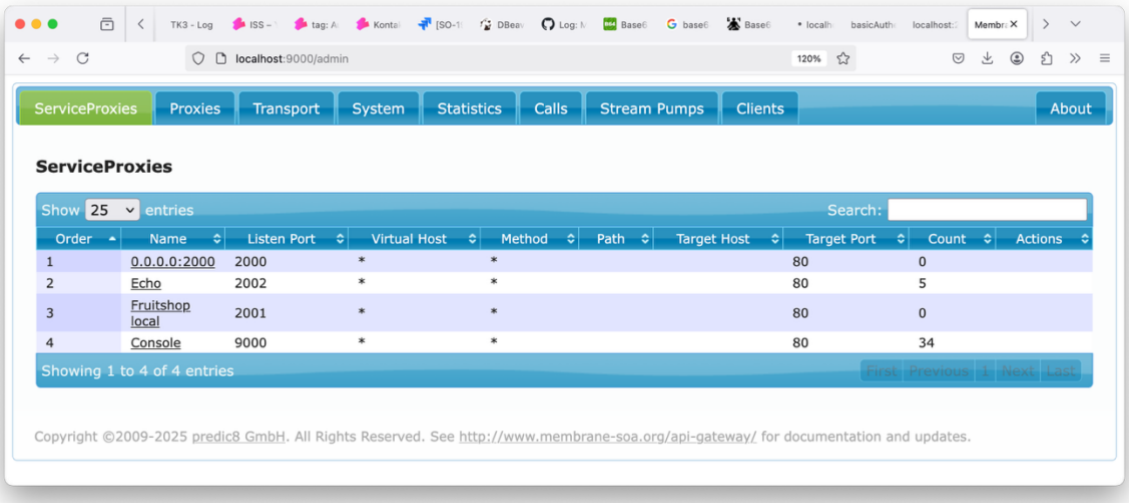


Image: List of APIs in the admin console

The **Calls** tab provides information about recent traffic. Membrane stores a configurable number of request and response messages in memory so they can be inspected later. You can control how much memory is allocated for this feature. See the section on **MessageExchangeStores** below for details.

Show 25 entries

Time	Status Code	Proxy	Protocol	Method	Path
2026-02-01 17:41:43.980	304	gtm.predic8.de SSL	1.1	GET	/_/service_worker/61k0/sw.js?origin=https%3A%2F%2Fwww.predic8.de
2026-02-01 17:41:31.884		Fruitshop V2 HTTPS	1.1	GET	/shop/v2/products/13
2026-02-01 17:41:23.941	404	www.predic8.de HTTPS	1.1	GET	/icons/Octicons-mark-github.svg
2026-02-01 17:41:23.036	200	www.predic8.de HTTPS	1.1	GET	/
2026-02-01 17:41:17.461	200	membrane-soa.org	1.1	GET	/api-gateway-doc/5.3/configuration/reference/customStatementJdbcUserDataProvider.htm
2026-02-01 17:41:16.889	200	Fruitshop V2 HTTPS	1.1	GET	/shop/v2/products/13
2026-02-01 17:41:15.896	200	predic8.de	1.1	GET	/activemq-hornetq-rabbitmq-apollo-qpid-vergleich.htm
2026-02-01 17:41:06.743	200	www.predic8.de HTTPS	1.1	HEAD	/
2026-02-01 17:41:06.723	301	predic8.de HTTPS	1.1	HEAD	/

Image: Requests and responses that passed the gateway recently

Clicking a timestamp reveals the full details of a request and the corresponding response, including HTTP headers and the message body.

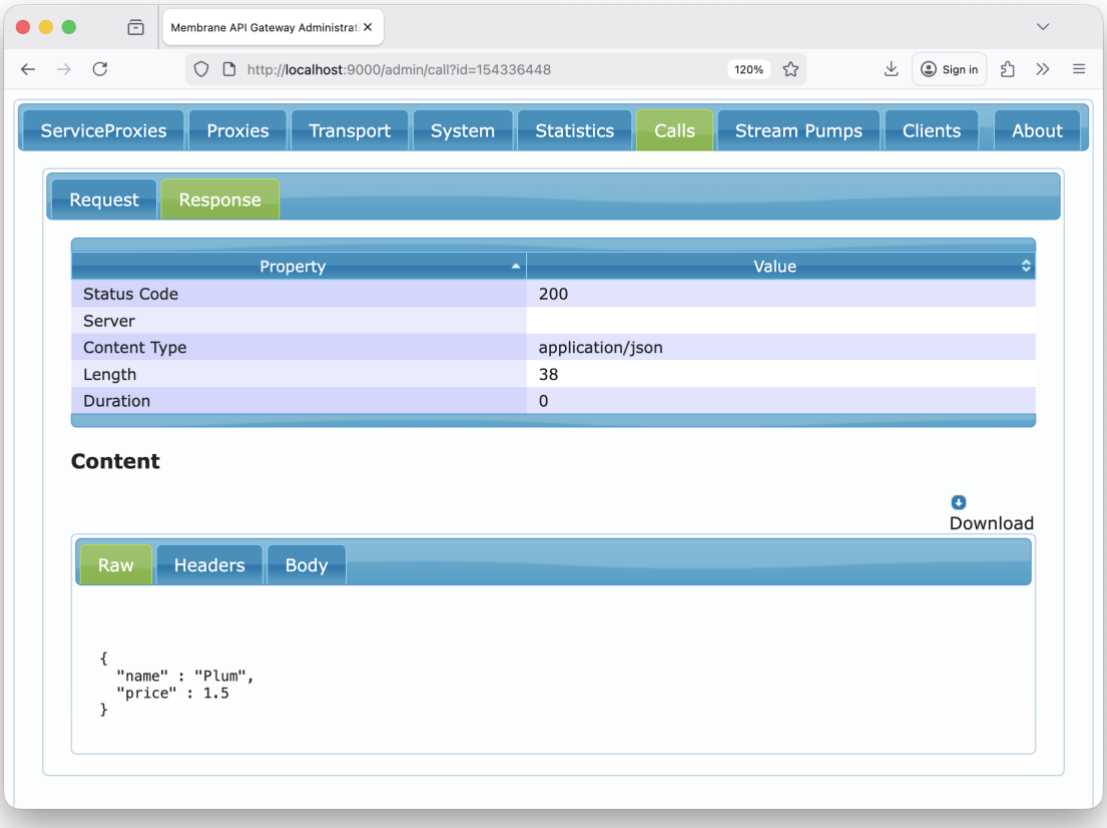


Image: Response message body intercepted from a backend response

Securing the Console

The admin console behaves like any other API endpoint and can be secured in the same way. You can apply mechanisms such as access control lists, API keys, or JWT-based authentication, depending on your security requirements.

The example below shows how to protect the console with basic authentication and configure it as read-only to prevent unauthorized changes.

```
api:
  port: 9000
  flow:
    - basicAuthentication:
        users:
          - username: admin
            password: ba9b-fcd63
    - adminConsole:
        readOnly: true
```

With this configuration, only authenticated users can access the console, and no configuration changes can be made through the UI.

For additional options, such as role-based access control or avoiding plaintext credentials in configuration files, see the security chapter earlier in this book.

41.2 Monitoring with Prometheus and Grafana

In enterprise environments, administrators usually prefer a centralized monitoring solution that covers multiple systems, rather than relying on the individual dashboards provided by each product.

Membrane offers basic metrics through its admin console. However, if you need deeper insight, long-term storage, or correlation with metrics from other systems, it's better to export gateway metrics to a dedicated monitoring platform such as the Elastic Stack or Prometheus with Grafana.

API Gateway Metrics

An API Gateway produces a wide range of operational metrics that are valuable for monitoring and alerting. Typical metrics include:

- Number of requests per second
- Volume of incoming and outgoing traffic
- Distribution of HTTP status codes (e.g., 200, 401, 403, 500)
- Response time statistics (minimum, maximum, and average)
- Error rates and failure patterns

These metrics can be collected, stored, and analyzed by systems such as Prometheus.

Prometheus

Prometheus is a popular open source monitoring system for collecting, storing, and querying time-series metrics. It works by periodically scraping HTTP endpoints exposed by monitored applications.

Membrane can expose a metrics endpoint directly. The configuration below defines an API that publishes gateway-wide metrics in Prometheus format:

```
api:
  port: 2000
  path:
    uri: /metrics
  flow:
    - prometheus: {}
```

The endpoint is exposed through a dedicated API that contains only the Prometheus interceptor. You can test the endpoint manually:

```
curl localhost:2000/metrics
```

The response is a plain-text document optimized for efficient scraping. The following excerpt shows metrics that track the HTTP status code distribution:

```
membrane_count{rule="shop",code="200"} 5149
membrane_count{rule="shop",code="401"} 6637
membrane_count{rule="shop",code="500"} 7999
```

By default, Prometheus scrapes metrics endpoints every 15 seconds. In practice, scrape intervals typically range from a few seconds to several minutes, depending on system load and monitoring requirements.

Once ingested into Prometheus, these metrics can be visualized in Grafana dashboards, enabling real-time monitoring, alerting, and historical analysis of API traffic and behavior.

Grafana Dashboard

While Prometheus is responsible for collecting and storing metrics, visualization is typically handled by Grafana.

Grafana is an open source visualization platform that supports multiple data sources, including Prometheus. It allows you to build interactive dashboards that display key metrics such as request rates, traffic volume, error distributions, and response times in real time.

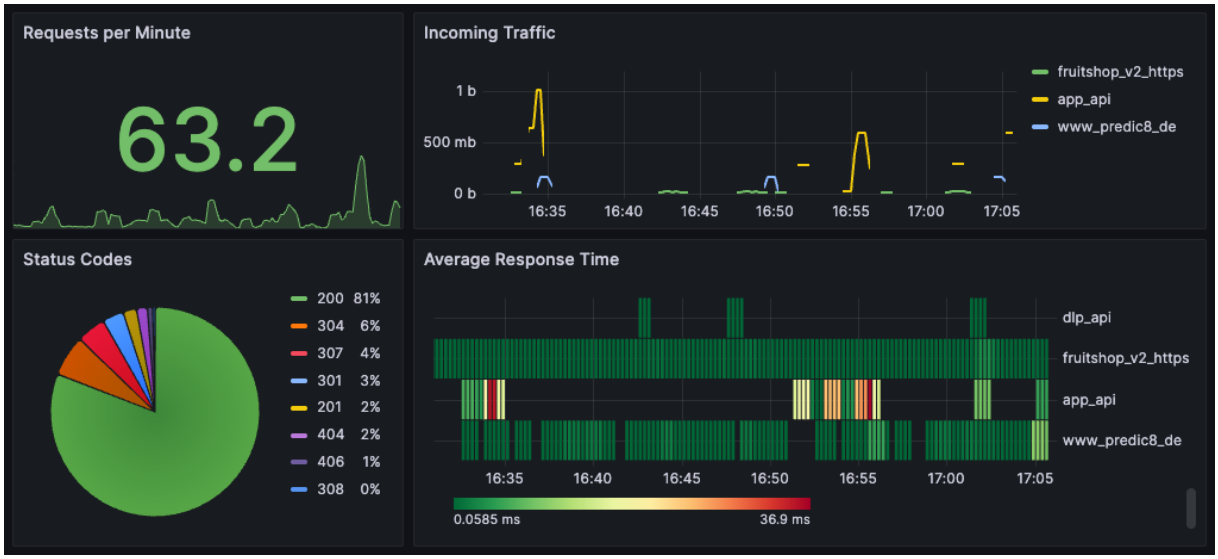


Image: Grafana dashboard from the Membrane examples

The Membrane distribution includes a ready-to-run example in the folder:

`examples/monitoring-tracing/prometheus`

that provides a **Docker Compose** setup integrating **Membrane, Prometheus, and Grafana**. The setup comes with a predefined Grafana dashboard, allowing you to explore gateway metrics immediately without additional configuration.

41.3 Access Log

If you operate an API, it's important to understand who is using it and when. Just like web servers, an API Gateway records each request with details such as the timestamp, client IP address, requested path, and response status. For REST APIs, this effectively acts as an audit log, allowing you to trace `GET`, `POST`, and other HTTP operations.

The Common Log Format

The **Common Log Format (CLF)** is a widely adopted standard used by web servers, proxies, and load balancers. Each request is written as a single line containing a fixed set of fields:

- **host**: The IP address of the client
- **ident**: Historically used for client identity, usually - in modern systems
- **authuser**: The authenticated user, if available
- **date**: Timestamp of the request
- **request**: HTTP method, path, and protocol version
- **status**: HTTP response status code
- **bytes**: Size of the response payload in bytes

A typical access log entry looks like this:

```
192.168.2.81 - - [11/03/2025:20:25:13 +0100] "GET
/shop/v2/orders/ HTTP/1.1" 200 519
```

Access logs are invaluable for troubleshooting, security audits, and usage analysis. They also form the basis for traffic analytics and can be forwarded to centralized logging systems such as **Elasticsearch**, **Splunk**, or **Loki** for long-term storage and correlation with other operational data.

Membrane's Access Log

Membrane provides flexible access logging based on the **Log4j** framework. This allows you to customize log entries and include almost any piece of information that is available during request processing, such as specific HTTP headers or even values extracted from a JSON payload.

Because logging is handled through Apache Log4j, you can take advantage of its mature feature set. This includes log rotation, compression, and integration with external logging systems.

Resources

Common Log Format, Wikipedia
https://en.wikipedia.org/wiki/Common_Log_Format

Access Log Example
examples/logging/access

41.4 API Tracing with OpenTelemetry

As API landscapes grow more complex, with increasing numbers of services and microservices, it becomes essential to know who is calling whom. As up-to-date architecture diagrams are often missing, tracing becomes invaluable.

Modern tracing technology allows you to follow a single API request as it propagates through multiple systems, forming a call graph that may branch into several downstream services.

OpenTelemetry is an open standard for collecting and exporting telemetry data, including traces. It ensures that all participating systems use a common protocol to report call information to a central collector. Whether you use Java, .NET, Python, or another mainstream platform, agents and libraries are available to instrument applications and send trace data.

Imagine a trace that starts at the API Gateway and continues through several downstream services. With distributed tracing, you can see the full call graph, including timing, dependencies, and potential bottlenecks.

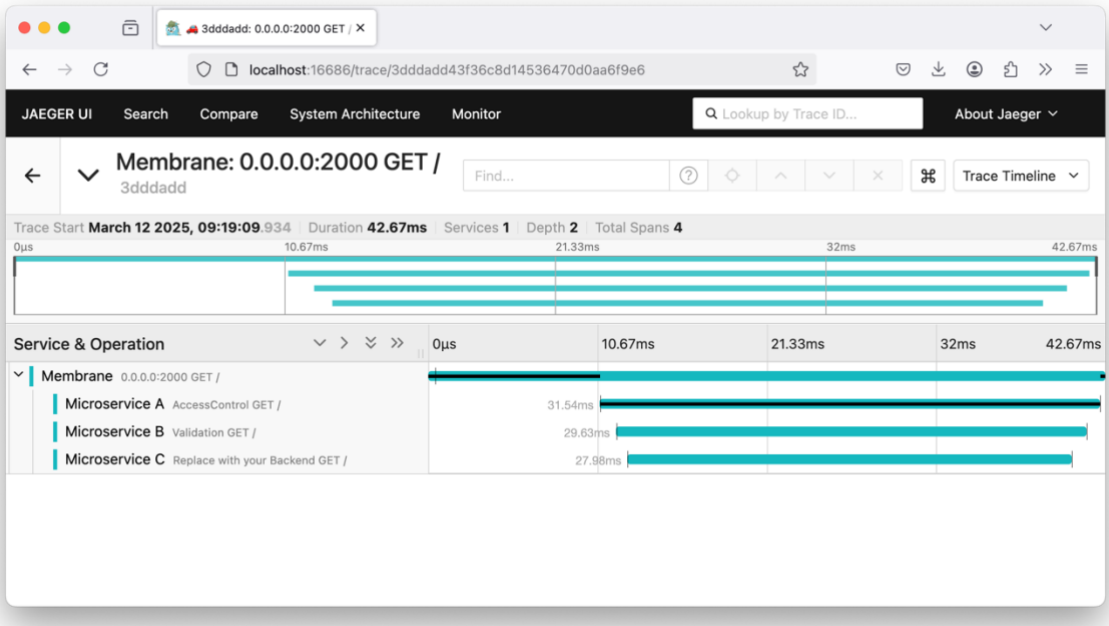


Image: Callgraph spanning an API Gateway and 3 microservices

Setting up OpenTelemetry

Configuring Membrane to export traces to an OpenTelemetry collector such as **Jaeger** is straightforward. You can enable OpenTelemetry either per API or globally for all deployed APIs. The following example enables tracing globally:

```
global:
  - openTelemetry:
      sampleRate: 1.0
      otlpExporter:
        host: localhost
        transport: grpc
```

With this configuration in place, Membrane automatically emits tracing data for all APIs and sends it to the configured OpenTelemetry collector. The `sampleRate` parameter controls how many requests are traced. A value of `1.0` means that every request is traced, while lower values trace only a fraction of the traffic.

Resources

OpenTelemetry @ Cloud Native
https://opentelemetry.io/

Jaeger: open source, distributed tracing platform
https://www.jaegertracing.io/

OpenTelemetry Example
examples/monitoring-tracing/opentelemetry

41.5 Message Logging

Whether you are debugging APIs or operating them in production, it is useful to inspect the actual messages flowing through the gateway. Membrane provides several options to log traffic at different levels of detail.

Writing Messages to the Log

The simplest way to inspect traffic is to write message data directly to the log. This can be done by placing a `log` interceptor into the flow:

```
api:
  port: 2000
  flow:
    - log: {}
```

With this configuration, requests and responses passing through the API are written to the log.

Logging What Matters

Instead of logging entire payloads, you can extract and log only the information that is relevant. This keeps log files smaller and easier to read:

```
- log:
    message: |
      Got: ${body}
      Header: ${header}
      Trace: ${header['Trace-Id']}
```

To extract specific fields from a JSON message body, use **JSONPath**:

```
- log:
    message: "Fruit: ${$.name}"
    language: jsonpath
```

For XML payloads, XPath works the same way:

```
- log:
    message: "Fruit: ${//name}"
    language: xpath
```

Selective logging is useful when logs are forwarded to centralized systems such as Elasticsearch, Grafana, or other log aggregation and visualization tools. This reduces noise while still providing the data needed for monitoring and troubleshooting.

41.6 Message Stores

For longer term storage or more advanced inspection, Membrane supports **exchange stores** that persist full requests and responses, including the message body. Supported backends are **in-memory**, **file-based**, and **MongoDB** storage.

In-Memory Store

Membrane includes a lightweight in-memory message store for development and for short-term inspection in production environments. It allows you to trace requests and responses in real time without relying on external storage systems.

Messages are kept entirely in memory and can be inspected via the **Calls** tab in the Admin Console. The configured limit defines the maximum amount of memory in bytes, that may be used to store exchanges.

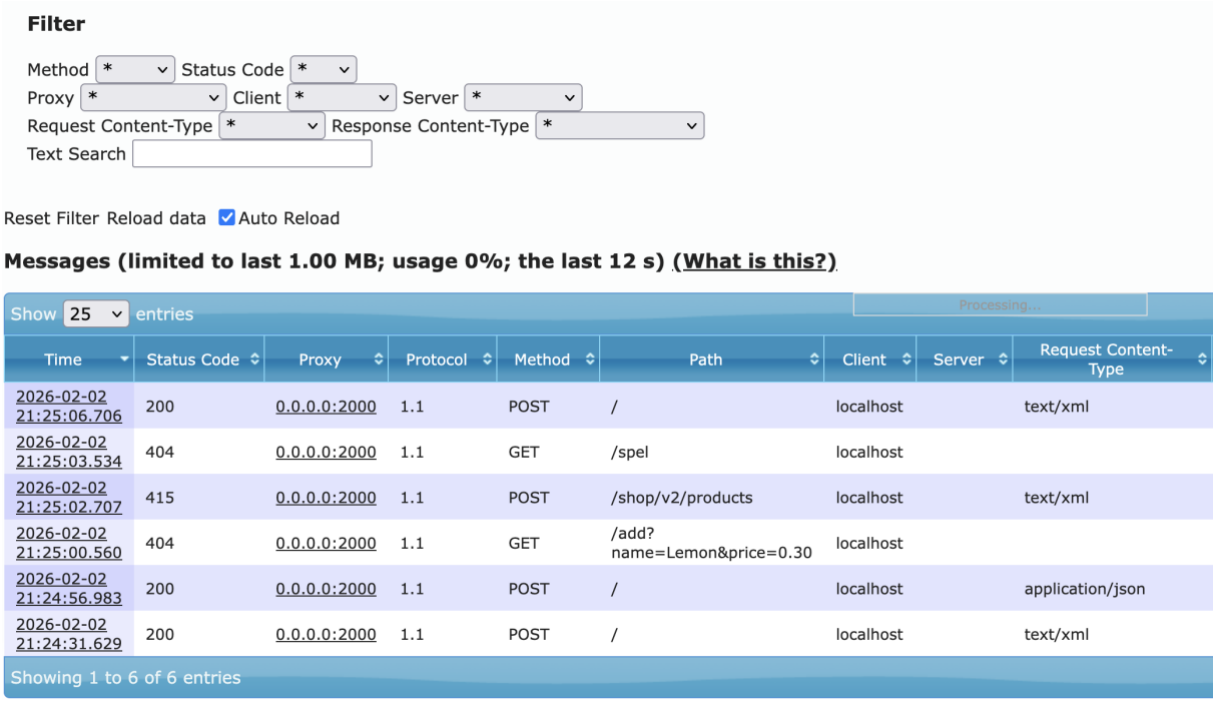


Image: Calls table of the admin console

The table provides filtering options to search for specific messages, such as:

- All calls that returned a 401 status code
- Calls from a specific client or targeting a specific backend
- All requests using a certain HTTP method, such as POST

When the memory limit is reached, the oldest exchanges are automatically discarded to make room for new ones. By default, Membrane allocates **1 MB** to the store to keep the memory footprint low. If you want to reduce resource usage even further, you can replace it with the `ForgetfulMemoryExchangeStore`, which does not retain any messages.

You can override the default configuration by defining your own store:

```
components:
  store:
    limitedMemoryExchangeStore:
      maxSize: 222222222
```

Choose the size limit based on traffic volume and how long messages should remain available for inspection. In many setups, keeping a few hours of traffic is sufficient for tracing and debugging purposes.

File Message Store

To persist messages beyond gateway restarts, you can use the `fileExchangeStore`. With this store, each request/response exchange is written to disk as a separate file:

```
components:
  my-store:
    fileExchangeStore:
      dir: exchanges
      maxDays: 7
```

Each exchange is stored as an individual file in the specified directory. The optional `maxDays` setting defines how long exchanges are retained before they are automatically deleted.

MongoDB Message Store

If you need centralized, persistent, and queryable message storage, Membrane also supports MongoDB as a backend for exchange storage:

```
components:
  my-store:
    mongoDBExchangeStore:
      connection: mongodb://localhost:27017/
      database: membrane
      collection: exchanges
```

Compared to file-based storage, MongoDB allows you to query stored exchanges directly.

Resources

File ExchangeStore Example
`examples/extending-membrane/file-exchangestore`

MongoDB ExchangeStore Example
`examples/extending-membrane/mongodb-exchange-store`

42 API Gateway Performance

Membrane API Gateway is designed with performance in mind. Its architecture includes several built-in optimizations that keep latency low and throughput high.

HTTP Streaming

Membrane streams data as early and as far as possible. That means it can start forwarding requests and responses before the full message has even arrived at the gateway. No need to wait for the last byte, Membrane gets moving as soon as enough data from the HTTP header shows up.

This streaming behavior minimizes buffering and avoids holding large payloads in memory, making Membrane fast and memory-efficient.

💡Sidenote: Plugins and streaming
Features such as message transformation and content filtering require the full message body to be loaded and parsed first. This prevents the gateway from streaming the message directly and may introduce additional latency and memory usage.

If performance is critical, pay attention to which interceptors or plugins you enable and whether they require buffering instead of streaming.

That said, avoid premature optimization. Only tune performance once you actually observe a bottleneck. Even with validation and transformation plugins enabled, Membrane is fast enough for most deployments without special optimization.

Keep-Alive

Membrane uses persistent TCP connections, also known as **HTTP keep-alive**, to reduce the overhead of establishing new connections. It maintains a pool of open TCP connections to backend services and reuses them whenever possible.

This avoids the time-consuming process of setting up a new connection for every single request. Especially in high-latency networks, reusing connections can save precious milliseconds.

Interestingly, some users have reported better performance with Membrane than without it. In setups where HTTP clients did not implement connection pooling correctly, Membrane maintained persistent connections to the backend. In these cases, the gateway actually improved performance.

Thanks For Reading

Thanks for reading, or at least skimming through, **The API Gateway Handbook**. We hope you gained useful insights, practical patterns, and a clear understanding of how gateways help operate and secure APIs.

This book is a living project and will continue to evolve. For updates and errata, visit:

https://www.membrane-api.io/api-gateway-ebook.html

We appreciate your feedback. If you have ideas, questions, or suggestions related to API Gateways, get in touch with us.

If you find Membrane useful, consider starring the repository on **GitHub**, sharing a post on **LinkedIn**, or linking to the project:

```
https://github.com/membrane/api-gateway
```

Cheers,
Thomas & Tobias

bayer@predic8.de
polley@predic8.de