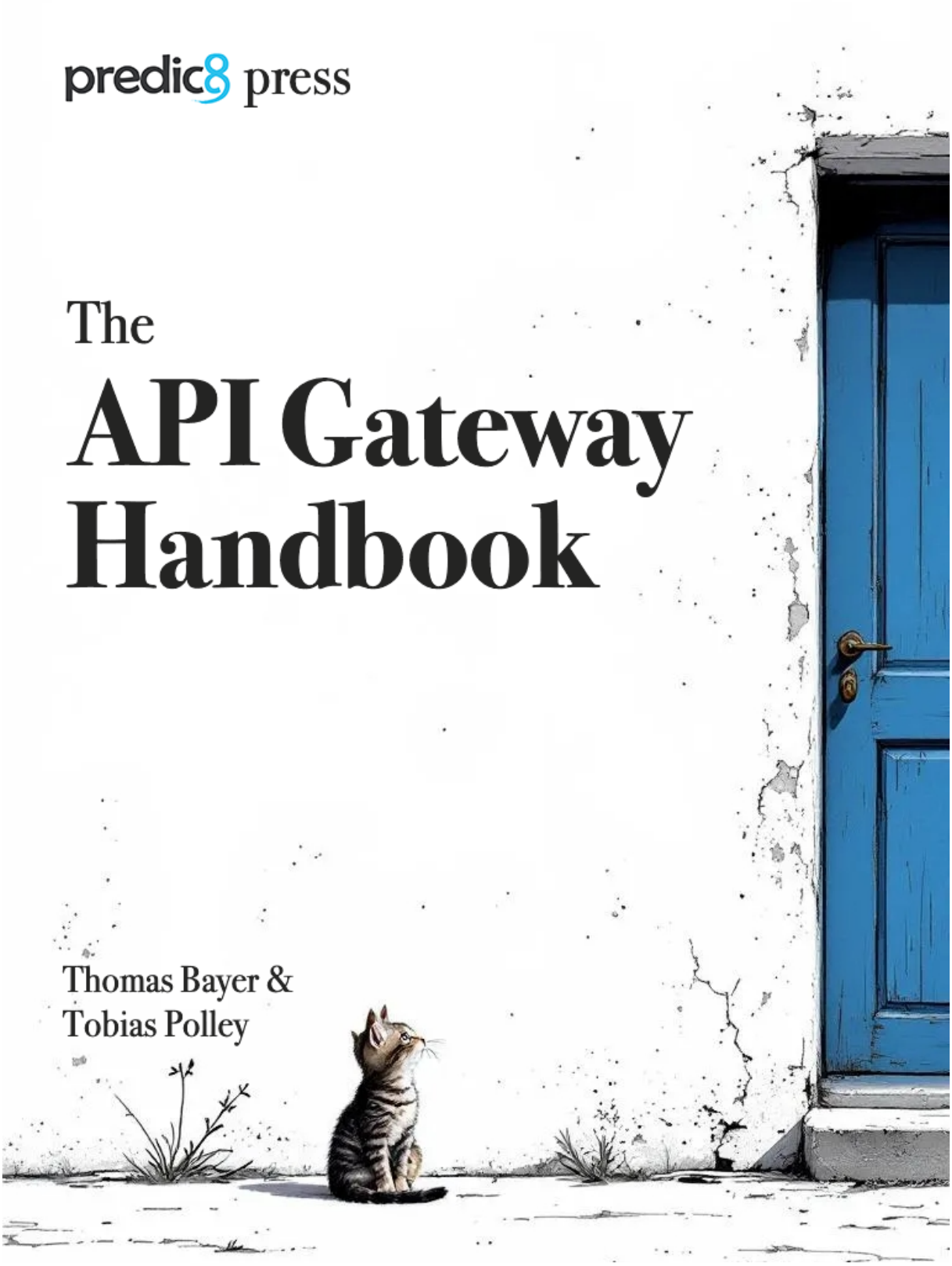
predic8 press

The API Gateway Handbook



Thomas Bayer &
Tobias Polley

The API Handbook

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 2025/09/28

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

Disclaimer

While the author and publisher have made every effort to ensure the accuracy and completeness of the content, no responsibility is assumed for errors, inaccuracies, omissions, or any inconsistencies herein.

The examples, tools, and techniques discussed are illustrative in nature. Readers are advised to adapt and validate any solutions provided according to their specific requirements, technology stack, and organizational practices.

The author and publisher shall not be held liable for any damages, including but not limited to direct, indirect, incidental, or consequential damages arising from the use or reliance on the content of this book.

Any references to specific products, tools, or brands are for illustrative purposes only and do not imply endorsement or recommendation by the author or publisher.

Table of Contents

Preface

Part I

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 world. APIs have become the universal language of 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 all become harder to manage. 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.

We wrote this book for architects, developers, and platform teams working with APIs in any form. Our goal was to keep it concise, hands-on, and vendor-neutral, no marketing talk, just real-world use cases, trade-offs, and design decisions.

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

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

Think of this book as both a guidebook and a toolbox for working effectively with API Gateways.

0.2 Why You Should Read This Book?

This book is for anyone working with APIs in an organizational setting, whether you're on a platform team, in operations, or focused on API development and architecture. It offers practical guidance from foundational ideas to advanced configurations and 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 running an API Gateway
- You're building with microservices, working in a cloud-native stack, or integrating across hybrid systems

No deep prior knowledge is required. We'll walk you step by step through key concepts from HTTP fundamentals to JSON Web Tokens and OAuth2.

Part I is especially useful for API designers, product owners, and project managers who need a clear, high-level view of what API gateways are and how they fit into modern architecture.

Part II is tailored to developers, operations teams, and API specialists who want to see how everything works in practice, with real examples and configuration details.

Ultimately, you should read this book if you're aiming to build secure, maintainable, and scalable API infrastructure, and want a practical guide to help you get there.

0.3 How to Read This Book

If you're new to API Gateways, begin with **Part I**. It lays the foundation by introducing core concepts in a logical, easy-to-follow progression.

Part II goes deeper, presenting practical examples (using the Membrane API Gateway) to show how specific problems can be solved. Even if you work with 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 this book cover to cover, or simply dip into the chapters that are most relevant to your work or current interests.

0.4 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).

0.5 How We Wrote This Book

Writing this book was both a technical challenge and a creative process. The ideas, structure, examples, and insights came from years of hands-on experience with real API gateway deployments, the development of our open-source API Gateway, and countless conversations with the community. But we didn’t write it alone.

AI was our patient assistant. Always ready to rephrase, polish, or fix clunky English, and never once complaining about late-night edits. The ideas, concepts, and experience are 100% human, but AI helped us express them more clearly (and spared you from our terrible grammar).

0.6 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.

After a pre-release, this is the first edition of the book. Version 1.0.0. Like any software, especially a 1.0.0, a book will have bugs. If you spot mistakes, have suggestions for improvements, or want to share general feedback, we would love to hear from you. With your help, we plan to release 1.1 in a couple of months.

Just send us an email at:

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

Thanks for helping us make this book better.

0.7 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 API wasn'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 (channel predic8 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 1
API Gateways Fundamentals

This part lays the groundwork for understanding API Gateways. We start by revisiting the fundamentals, APIs, HTTP, and practical tools like curl and Postman, to make sure everyone's on the same page. From there, we move on to a deeper look at API Gateways: what they are, the problems they solve, and how they help improve security, scalability, and API operations.

Whether you're just getting started or want to strengthen your understanding, Part I provides the essential context you need to make sound architectural and operational decisions.

1 Foundation

First, let’s establish a solid foundation by covering the essential technical concepts you'll need throughout this book. If you're already familiar with APIs, HTTP, and common API tooling, feel free to skip ahead directly to the dedicated section about API Gateways in chapter 2.

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.

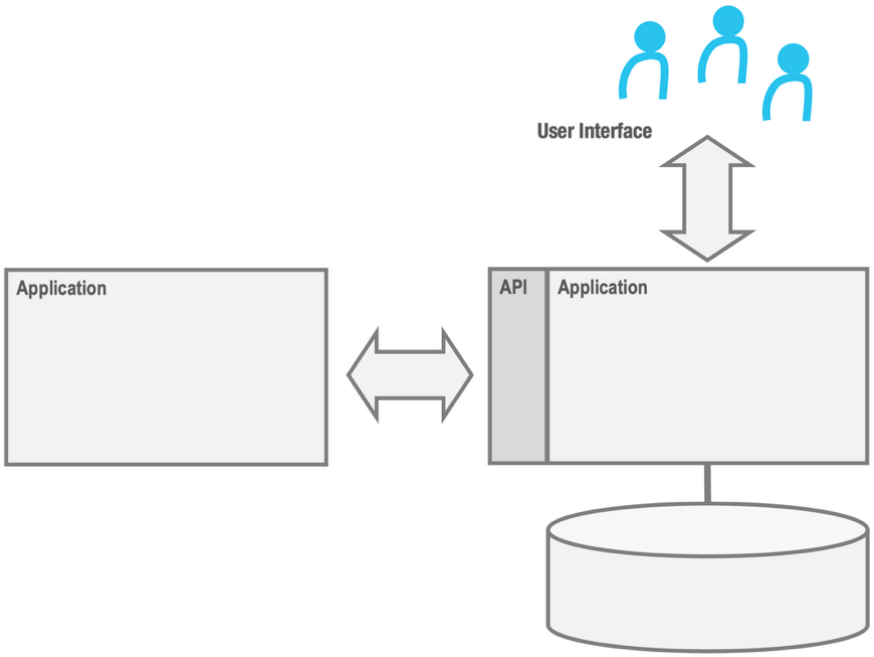


Image: User Interface and API

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)**, which serves as the backbone of communication on the web. Originally invented over 30 years ago, HTTP is the protocol that web browsers use to access web pages, making it fundamental to how we interact with the internet.

HTTP is known for its simplicity, which contributes 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 provide the foundation needed for the chapters to come.

HTTP operates on 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 find the internet address (IP address) of the host `api.predic8.de`
2. **Connection Establishment**
   Once the IP address is resolved the browser opens a connection to the web server.
3. **Sending the Request**
   Then, the browser sends an HTTP request to the server asking for a 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 the 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`: The format of the message body below.

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 OK` 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. But for general API design and compatibility, **HTTP/1.1 is the most widely supported choice**, especially when interoperability is a priority.

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"
  }
}
```

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

Resources

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

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.

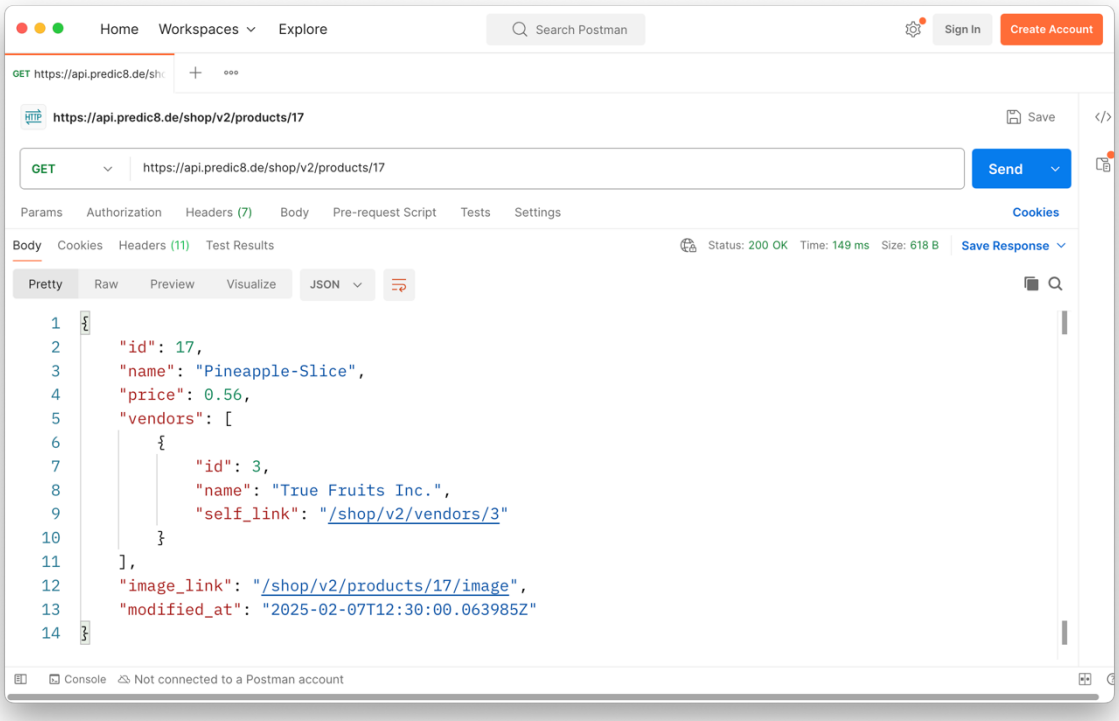


Image: HTTP client in Postman

Resources

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 also support features such as autocompletion, which makes writing HTTP requests quick and comfortable.

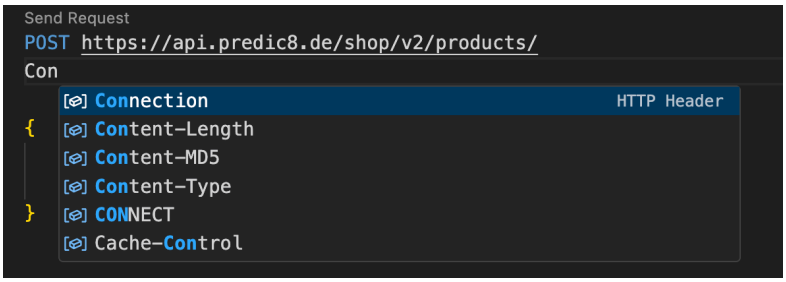


Image: HTTP Autocompletion with the REST Plugin

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

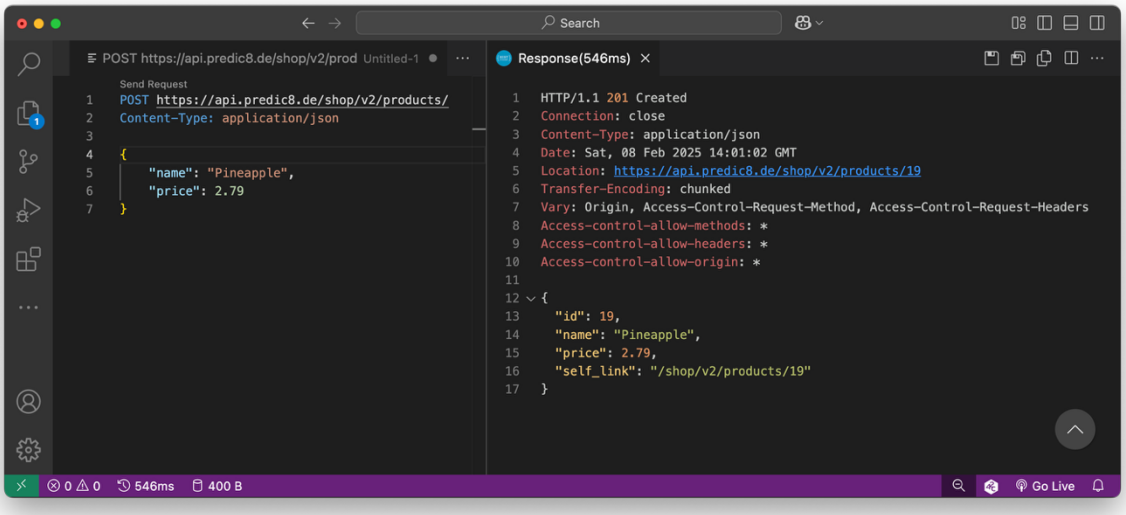


Image: Request and Response in the REST Client Plugin

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.

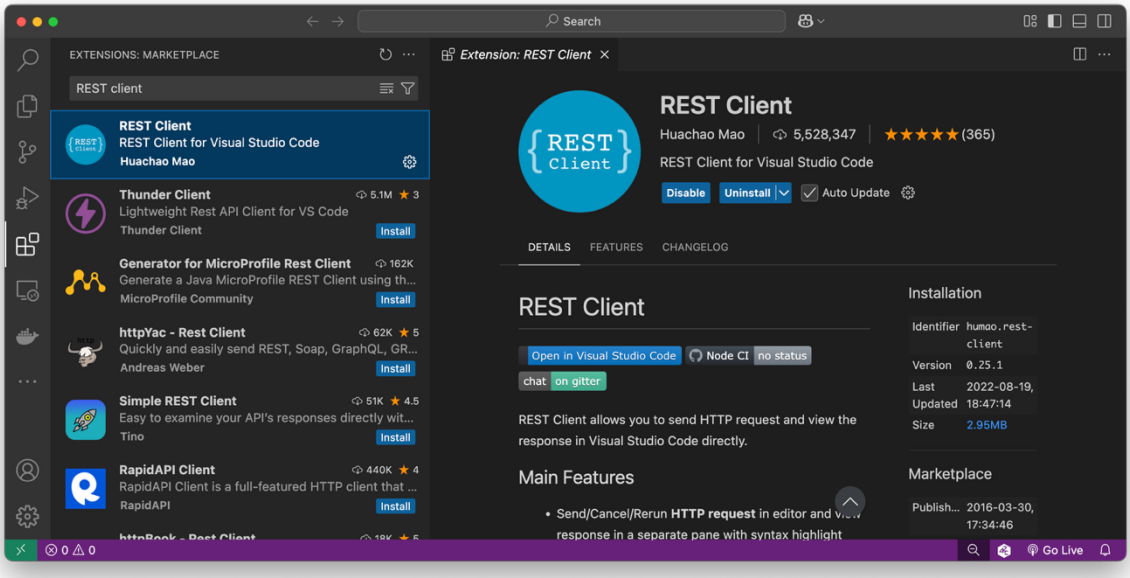


Image: REST Client Plugin in Visual Studio Code

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

Resources

REST Client, Microsoftt 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.

API Gateways are essentially specialized reverse proxies. They not only route requests but also manage, monitor, and optimize communication between clients and backend services. To understand what makes them different, and why they matter, it helps to first look at the difference between a traditional proxy and a reverse proxy.

Proxies (Forward Proxies):
A traditional proxy, also called a forward proxy, sits on the client side of a connection, between the client and the public internet.

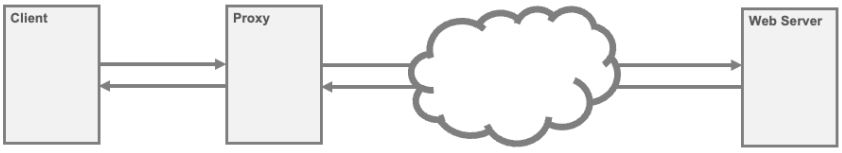


Image: Forward proxy between client and Internet

Its primary purposes include:

1. **Caching**: Speeds up communication by storing 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 server side, right in front of one or more backend servers. From the client's perspective, it looks like they're communicating directly with the target server. But in reality, the reverse proxy receives the request and forwards it to the appropriate backend server.

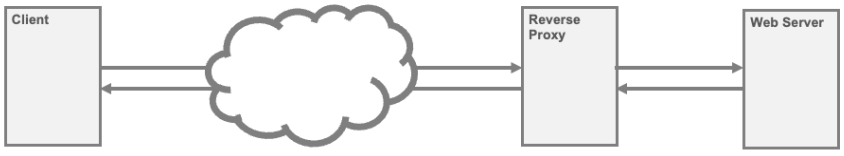


Image: Reverse proxy between Internet and server

A reverse proxy:

1. Accepts client requests
2. Forwards them to the appropriate server
3. Returns the server's responses to the client

Reverse proxies can provide **additional value**, such as:

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

In the following sections, we'll 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, rate limiting, 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's sneaking in anything suspicious.

2.1 Responsibilities of API Gateways

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

Routing
API Gateways forward incoming requests to the appropriate backend services. The client interacts only with the gateway and doesn’t need to know the internal network structure or backend addresses. This abstraction simplifies the client and enables backend flexibility.

Security
Gateways provide critical security features, including **authentication, authorization** and **content inspection.**

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 transform messages between different formats, such as converting XML payloads into JSON or adapting data to meet client requirements. This functionality ensures compatibility between clients and backend services.

Orchestration
In more complex scenarios, the gateway can combine responses from multiple backend services into a single API response. For instance, a Gateway might aggregate data from several microservices to deliver 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 with advanced governance and policy features.

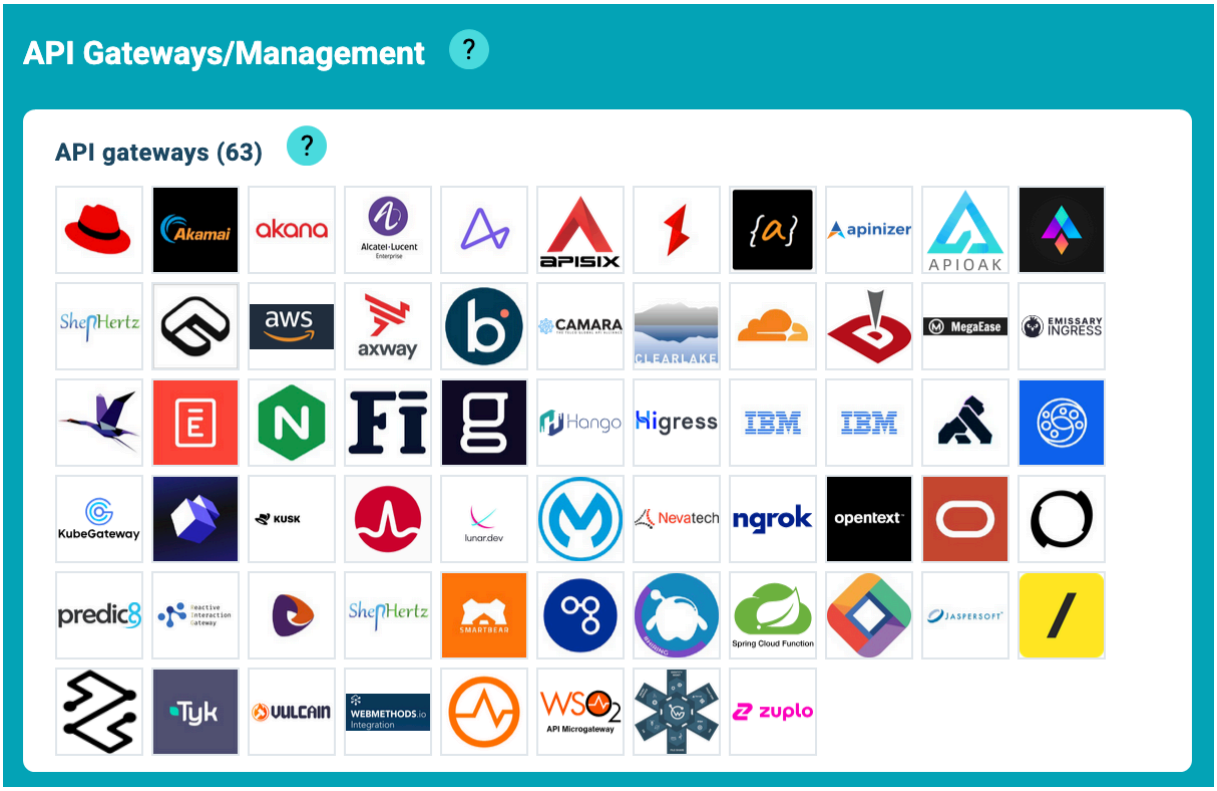


Image: API Gateway products @ API Landscape

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

Edge Gateway

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 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 Gateway

Major clouds like Amazon AWS and Microsoft Azure offer their own API Gateways that integrate seamlessly into their respective cloud platforms. These cloud-native gateways benefit from built-in scalability, security, 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 allow you to embed API Gateway functionality directly within your applications, eliminating the need for a separate gateway deployment. By using these libraries, you gain wide control over the gateway's behavior and can customize its features according to your application's specific needs.

Prominent examples on the Java platform include **Spring Cloud Gateway**, which provides comprehensive routing, security, and filtering features directly within **Spring Framework**-based applications.

Kubernetes API Gateways

Kubernetes is an open-source container orchestration platform that enables the management of containerized workloads and services, both on-premise 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 valuable insights on how API gateways integrate with, and enhance your Kubernetes environment.

Kubernetes Ingress Controller

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

- **Service Discovery:**
  An ingress controller can leverage the Kubernetes Service Discovery to automatically discover which pods are serving as backends for an API. This dynamic discovery ensures that traffic is always directed to the correct and currently available targets.
- **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 most basic configuration of these gateways is deeply integrated with and native to Kubernetes. Realizing advanced features however which are not standardized (like path rewriting) is a bit cumbersome.

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 a Kubernetes Special Interest Group (SIG Network), aiming 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 (e.g. 90%-10%-traffic splitting and traffic mirroring) 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 traditional API Gateways or even traditional Kubernetes Ingress controllers entirely. 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 existing 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 be used to hide network complexity 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 the 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 50 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 other gateways built on top of Envoy, such as Istio, Consul, or Ambassador, due to Envoy’s extremely efficient footprint—typically around just 10 MB.

Artifical 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.

2.3 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 features specifically designed to manage and optimize API communication. While you can technically use a reverse proxy like nginx to forward API traffic, it lacks key API-centric capabilities. For example, a traditional reverse proxy doesn't understand **API keys**, the **OpenAPI specification**, or other API-specific features.

To do its job, the API Gateway sits between API clients and backend services, just like a reverse proxy. The diagram below shows how it fits into the architecture: the API Gateway serves as the single entry point, exposing backend services under a unified domain such as `api.predic8.de`.

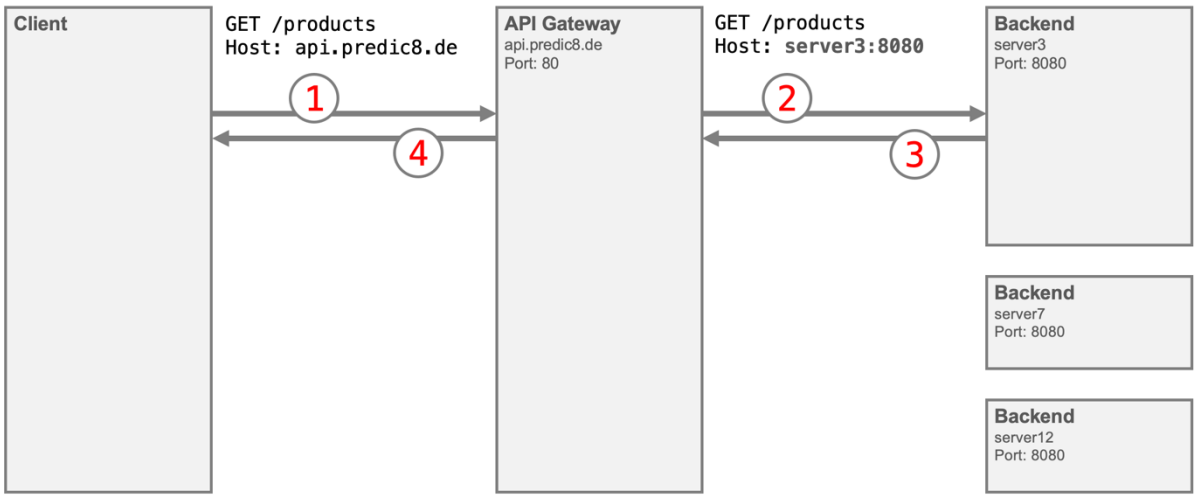


Image: Location of an API Gateway in the message flow

Here's how it works:

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 point of view, it's talking directly to the API. Behind the scenes, though, the Gateway is acting as a smart proxy, routing traffic, applying rules, and managing the flow.

At this stage, the behavior of an API Gateway closely resembles that of a **reverse proxy**. However, while reverse proxies are designed for general HTTP request handling, API Gateways provide additional, API-specific functionalities that cater to the unique requirements of API communication. To provide the API centric functionality most gateways are reverse proxies equipped with plugins adding API specific functionality.

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, along with many other use cases. The exact terminology varies by product: some call them *plugins*, others *policies* or *filters*, but the idea is the same.

Gateways like **APISIX**, **Kong**, and **APIcast** are built on top of fast, efficient reverse proxies and come preloaded with a wide selection of plugins. These extensions are often grouped into categories such as:

- **Transformation** – modify headers, rewrite URLs, change payload formats
- **Authentication** – enforce API key, JWT, OAuth2, or custom auth flows
- **Security** – protect against threats like SQL injection or XML bombs
- **Observability** – provide logging, metrics, tracing, and monitoring
- **Traffic Control** – manage rate limits, quotas, retries, and circuit breakers

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

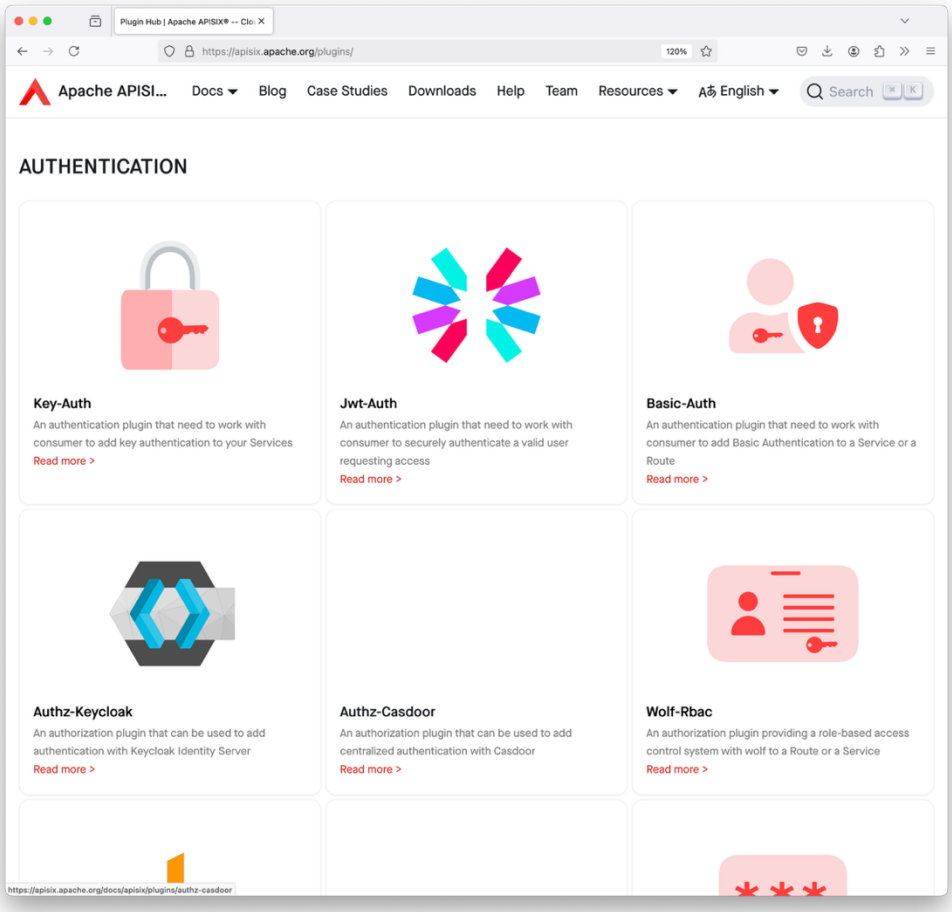


Image: Plugins at APISIX plugin hub

3.2 Message Flow

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

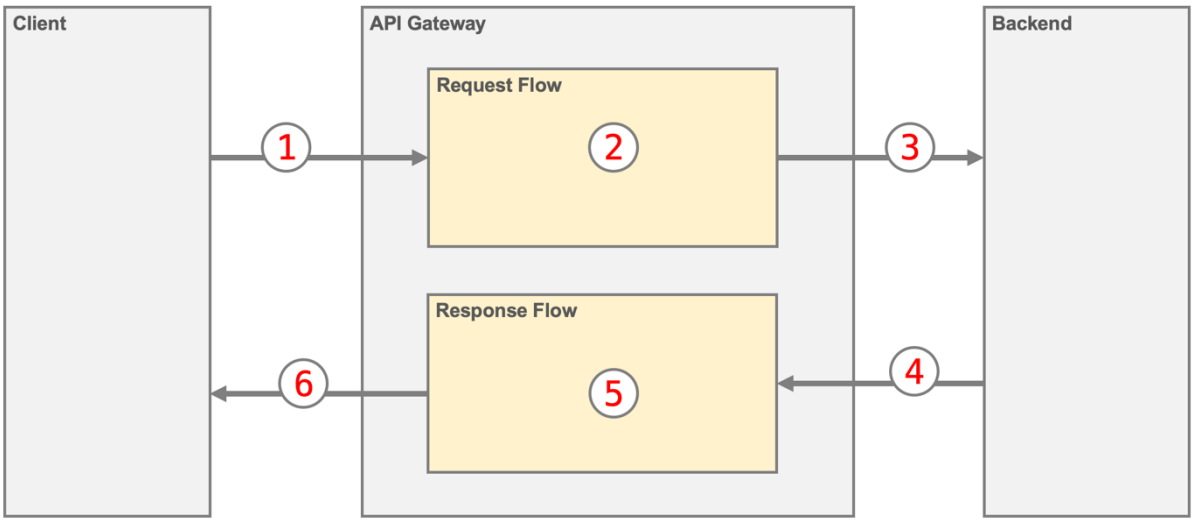


Image: Message flow between client, API Gateway and backend

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 act like blocking requests with invalid credentials or logging payloads to files.

3.2.1 Plugin Placement

API Gateways provide functionality through **plugins**. For a plugin to be effective, it must be integrated "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 these universal requirements, most API Gateways offer a **global flow** (or global pipeline) through which **all messages** pass. By placing a plugin in the global flow, you ensure 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.

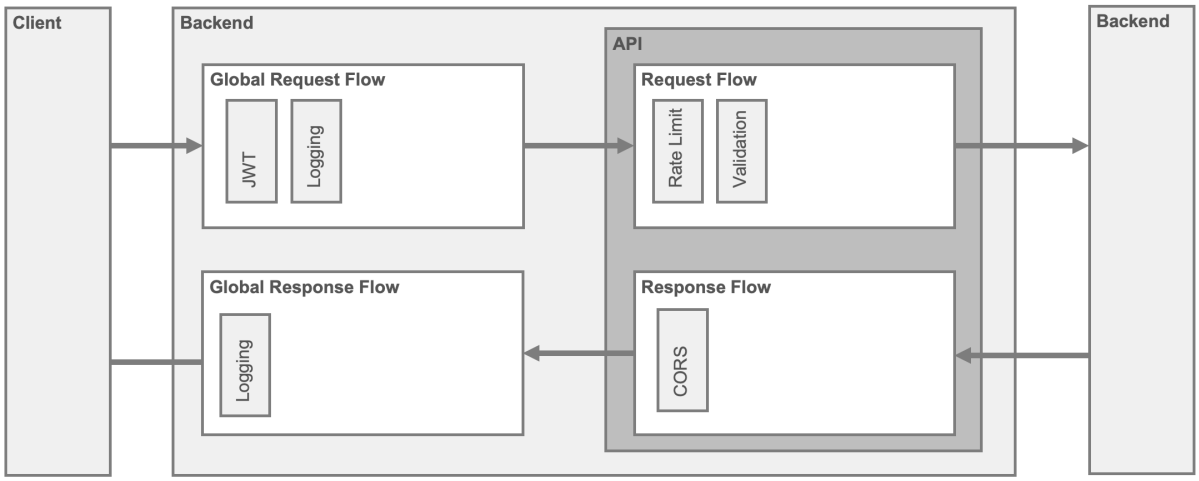


Image: API Gateway with global and API local flows

Other plugins might only be relevant for a particular API or endpoint. For instance, you may want a **schema validation** plugin to check the request format only for a certain API.

Gateways typically allow each API to have its own request and response flows. By placing a plugin locally, you **limit its scope to a specific route** or service.

This flexibility, **global vs. local** flow and **request vs. response placement**, enables you to control precisely where and how your gateway applies its functionality. By carefully planning plugin placement, you ensure that each API and every request/response is handled according to its unique requirements.

Some plugins need to be engaged in the request and response flow at the same time. In the illustration below the OpenAPI plugin is part of both flows. This is necessary cause it has to validate requests and responses.

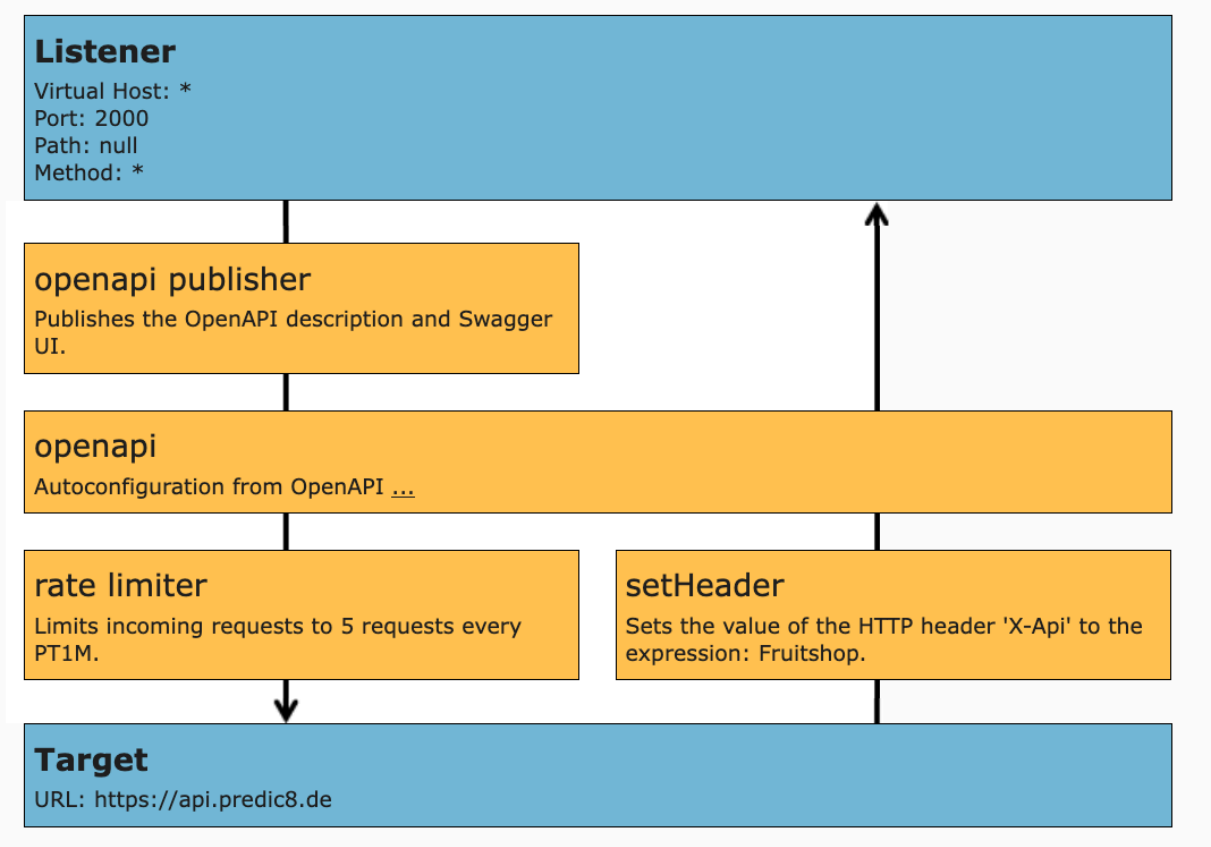


Image: Visualization of an API in Membrane's admin console

Let's look at a different example. Suppose you have a backend service that can handle only ten concurrent requests at a time. To protect it from overload, you'll need a plugin that maintains a counter:

- The counter is **incremented** when a request enters the flow.
- It's **decremented** when the corresponding response is returned.

If the counter reaches ten, any new incoming requests must be **blocked or denied** until the number of active requests drops below the threshold.

To accomplish this, the plugin must be engaged in **both the request and response flow** simultaneously. It needs to track each call through its entire lifecycle to ensure the counter accurately reflects the number of **in-flight** requests.

This kind of flow-aware logic requires **stateful coordination** inside the gateway, something beyond what simple request filtering can achieve.

3.2.2 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 in the same language—or at least a compatible 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 enable the use of plugins written in a language different from that of the gateway core, some API gateways support a feature called a **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.

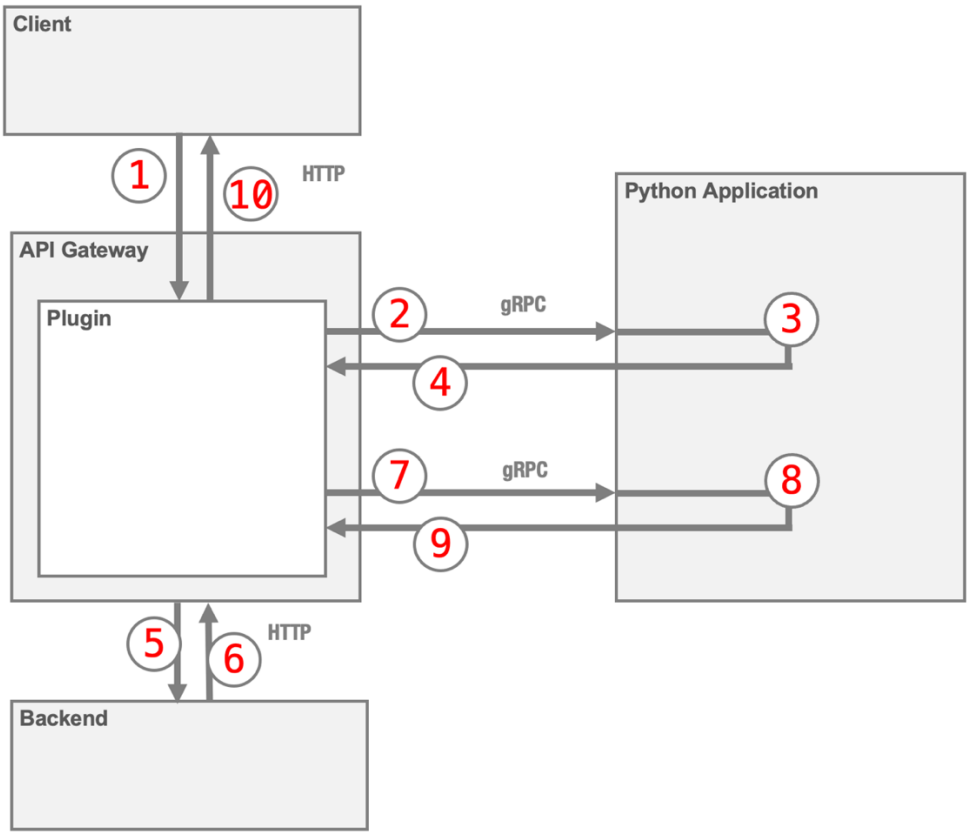


Image: Extending an API with Python code via a plugin runner

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.

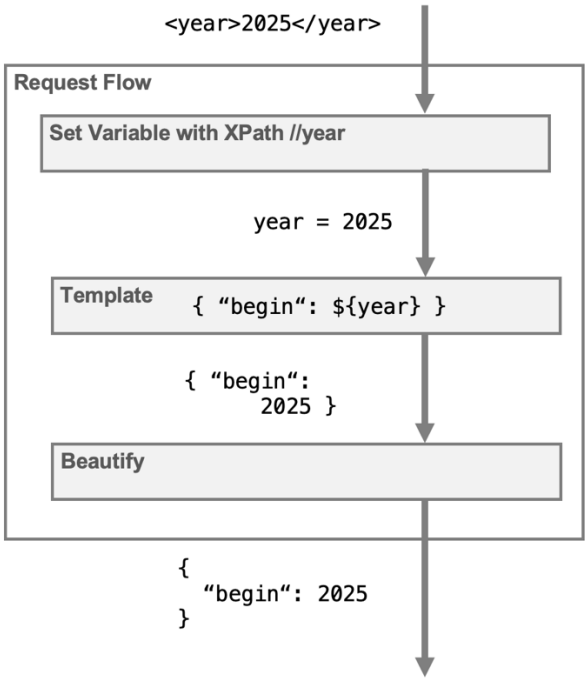


Image: Multiple plugins working together

The glue that binds plugins together are expression languages. They provide the magic that makes collaboration between plugins possible. In the next section, we'll take a closer look at how they work and why they matter.

3.3 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 it uses, 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 choices. Before you go all-in on one, check that your gateway supports it.

SpEL (Spring Expression Language)

SpEL is part of the Spring Framework and offers more advanced features than the Jakarta Expression Language. It's a powerful alternative to OGNL and MVEL, and it's widely used in Spring-based applications.

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 JVM, that means they can read files, open sockets, or execute external commands, **capabilities that make security officers nervous**. For this reason, some gateways run

Groovy in restricted or sandboxed environments or allow administrators to disable scripting entirely in production.

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 especially handy for transforming JSON payloads. Consider this transformation example:

```
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})
    )
})
```

Jsonpath

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

This expression:

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

returns all `name` fields under any `article` object, regardless of nesting depth.

XPath

If you're working with **XML** payloads, XPath is the go-to choice. Often called **SQL for XML** XPath provides concise, powerful expressions to navigate and extract values from XML documents. It has a relatively shallow learning curve, yet it provides advanced features to handle even complex querying tasks.

Take this expression, for example:

```
//article[@id=3]/name
```

it returns the `<name>` element of the `<article>` with an `id` attribute equal to `3`.

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.

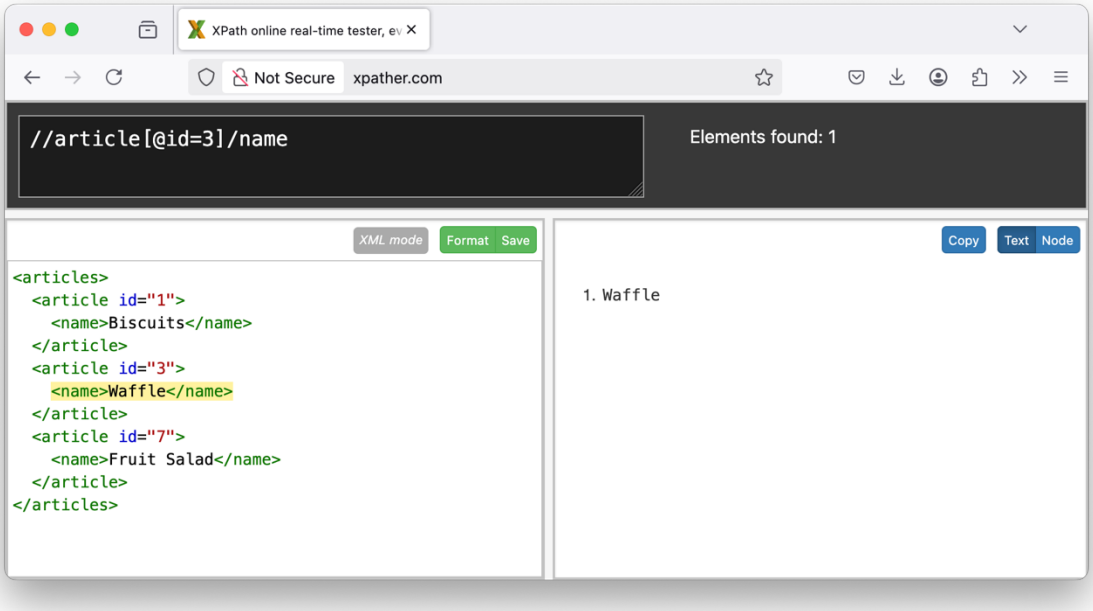


Image: XPath expression tester with document view and result output

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

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

Here are a few more useful XPath examples:

Expression	Description
`/articles/article/name`	Retrieves the text content of all articles
`//article[1]`	First article element
`//article[last()]`	Picks the last article element
`//article[1]/@id`	Gets the id attribute of the first article.

3.4 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 your API Gateway. For example, Nginx-based gateways often use **OpenResty** with **Lua**, gateways written in **Go** (like Ambassador) use Go, and Java-based gateways (like Gravitee or WSO2) are typically extended with **Java**.

4 Deployment

This chapter introduces the core components of API Gateways and their deployment models. It explains how to integrate an API Gateway into your organizational infrastructure, including how to position it within your **network architecture**, work with **firewalls**, and operate securely within a **Demilitarized Zone (DMZ)**.

You'll learn what to consider when placing gateways at the edge, in internal segments, or across hybrid environments and how these choices affect security, performance, and maintainability.

4.1 Gateway Components

API Gateways come in a variety of configurations. Some operate as **standalone applications**, while others require additional components like **databases**, **cache servers**, or **monitoring tools**.

Standalone Gateway (Stateless)

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

The advantage? A restart resets the gateway to a clean, known state like rebooting your computer to fix a glitch. This kind of setup is simple, robust, and easy to manage.

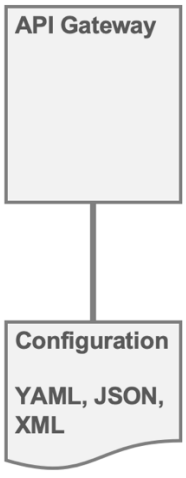


Image: Minimalistic standalone API Gateway

Gateway with Database-Backed

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.

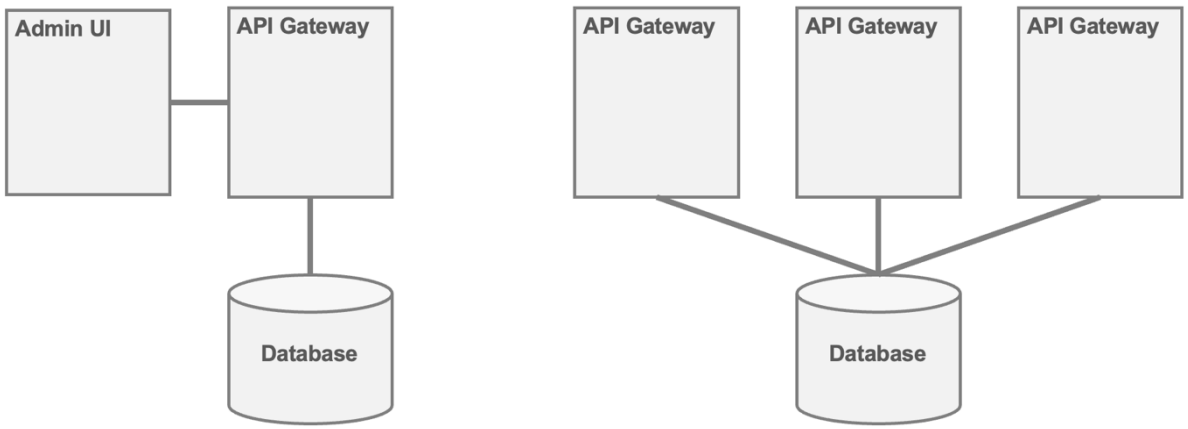


Image: API Gateway and cluster with shared database for configuration and metrics

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 scale as needed.

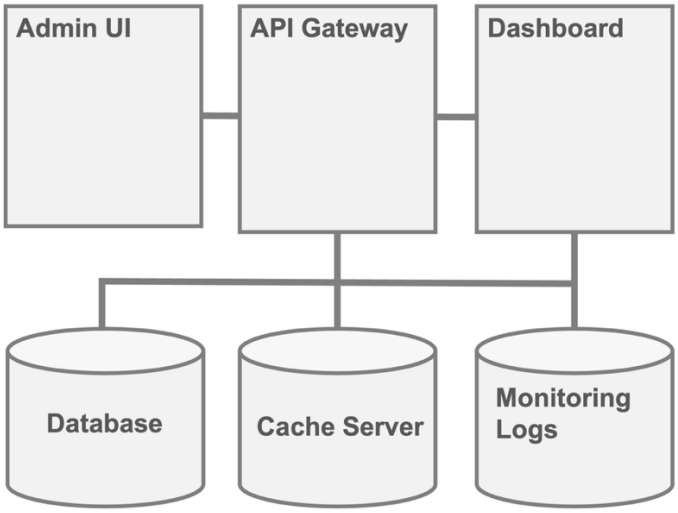


Image: Modular API Gateway setup with optional integrations

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.

4.2 Gateway Positioning

An API gateway connects two distinct areas. Common scenarios include positioning gateways between the public internet and internal company networks or bridging on-premise systems with cloud-based infrastructure.

4.2.1 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 calls for 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 main 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.

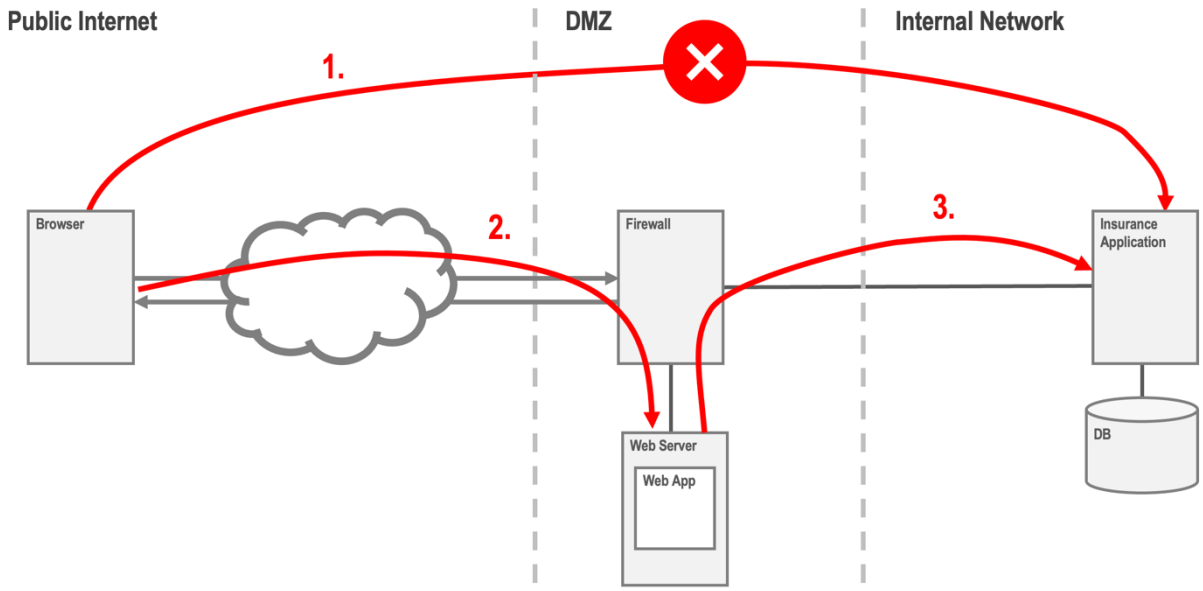


Image: Demilitarized zone between Internet and internal network

A DMZ has these key characteristics:

1. No direct routing between Internet and internal networks.
2. Inbound web traffic is directed to Web servers located within the DMZ.
3. DMZ hosts can initiate connections to internal resources.

A secure self-service portal can be setup as follows:

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

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.

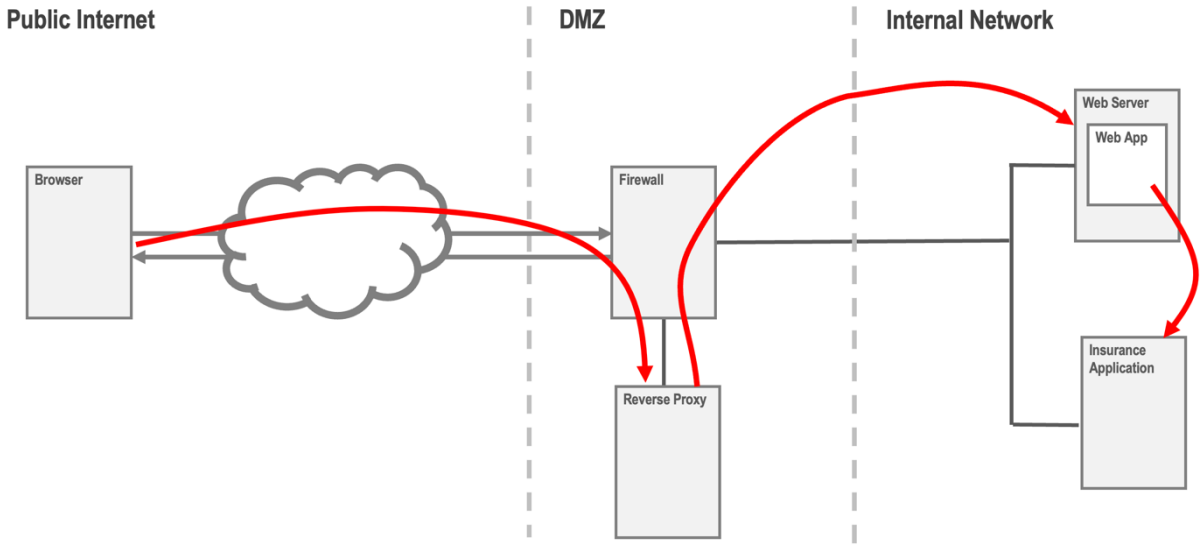


Image: DMZ with reverse proxy

Benefits of a reverse proxy:

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

4.2.2 Backend for Frontend (BFF) Pattern

Modern web applications, particularly **Single Page Applications (SPAs)**, run in the browser as JavaScript apps. Unlike traditional web apps that render HTML on the server, SPAs talk to backend services via APIs to fetch data and invoke functionality.

A **BFF** is a dedicated backend component located in secure zones like the DMZ, acting as a tailored bridge between browser-based frontends and internal APIs.

Key characteristics of BFF:

- **Dedicated per frontend**
  Each frontend typically has its own dedicated BFF.
- **Tailored requests and responses:**
  The BFF handles API requests and ensures the frontend receives **only** the exact data it needs.
- **Request validation**: Ensures frontend requests adhere strictly to formats and rules before they reach sensitive internal services.
- **Authentication and authorization**
  Manages security tokens, sessions, and access control.

The BFF pattern helps create a secure, maintainable, and frontend-optimized architecture especially in environments where security boundaries like DMZs are in play.

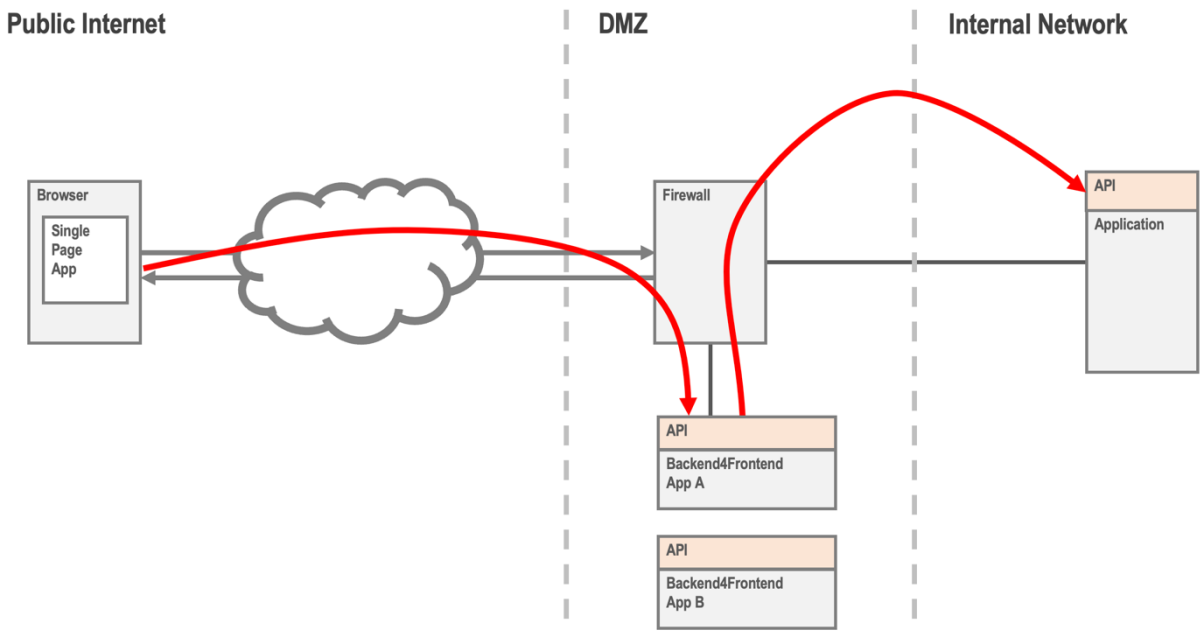


Image: Backend For Frontends (BFF) in the DMZ

Challenges with the BFF Pattern

While the Backend for Frontend pattern improves security and is well-suited for modern clients like Single Page Applications (SPAs), it comes with notable trade-offs:

1. **Increased development effort and slower time to market**
   Building and maintaining a dedicated BFF for each frontend requires substantial engineering effort. This additional layer can slow down product launches.

2. **Maintenance overhead**
   Whenever internal APIs evolve, corresponding BFFs need to be updated as well. This tight coupling increases coordination overhead across teams and adds friction to making changes.

Replacing BFF with API Gateways

While the BFF pattern serves a purpose, it often creates more complexity than necessary. API Gateways offer a simpler, faster alternative with several key advantages:

1. **No custom code**
   Unlike custom BFF applications, API Gateways require no additional coding. APIs can be exposed by configuration, significantly speeding up the process.
2. **Faster deployment**
   Adding or updating an API on a gateway takes minutes. Compare that to the days or weeks needed to roll out a custom BFF.

3. **Scalability**
   API Gateways are designed to handle scale. It's common to run hundreds of APIs on a single gateway instance, without needing to spin up new services.
4. **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.
5. **Out-of-the-Box features:**
   API Gateways offer production-grade features right out of the box, such as rate limiting, validation and authorization.

These capabilities turn an 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.**

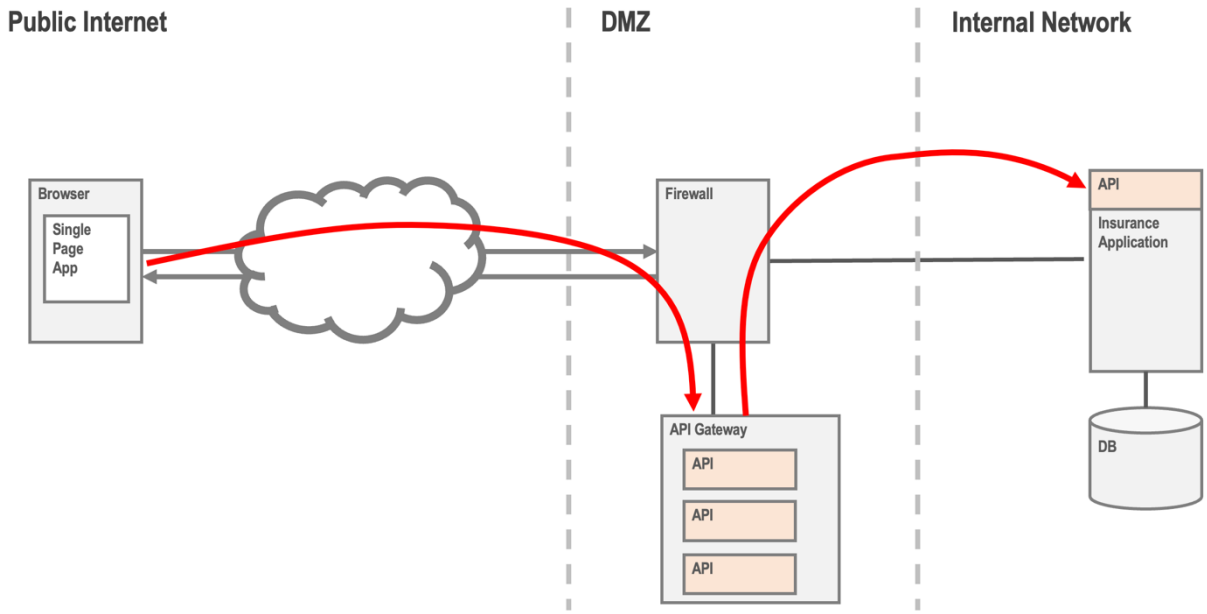


Image: Single API Gateway replacing multiple BFFs

4.2.3 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.

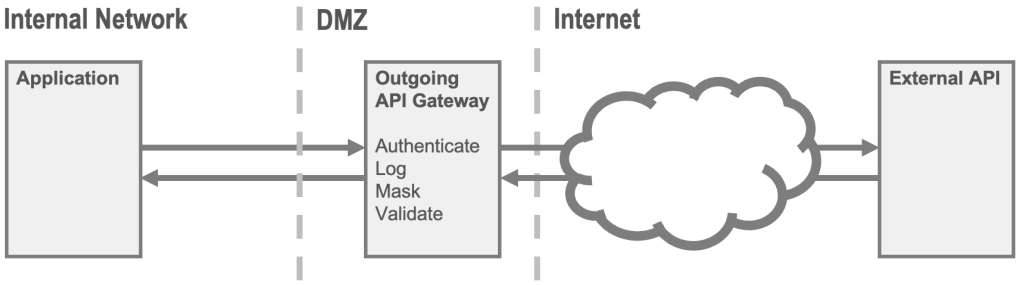


Image: Outgoing gateway routing API requests to external services

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 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: outbound.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

{ "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.

4.2.4 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, rate-limiting, 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 monolithic **Enterprise Service Buses (ESBs)**. 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 or C++ tend to perform particularly well in these scenarios.

The table below shows memory footprints of selected gateways based on simple runtime measurements:

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: Microgateways

Note: Memory usage depends on the specific version and setup. These values were obtained through practical 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 use depending on the scenario.

4.3 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.

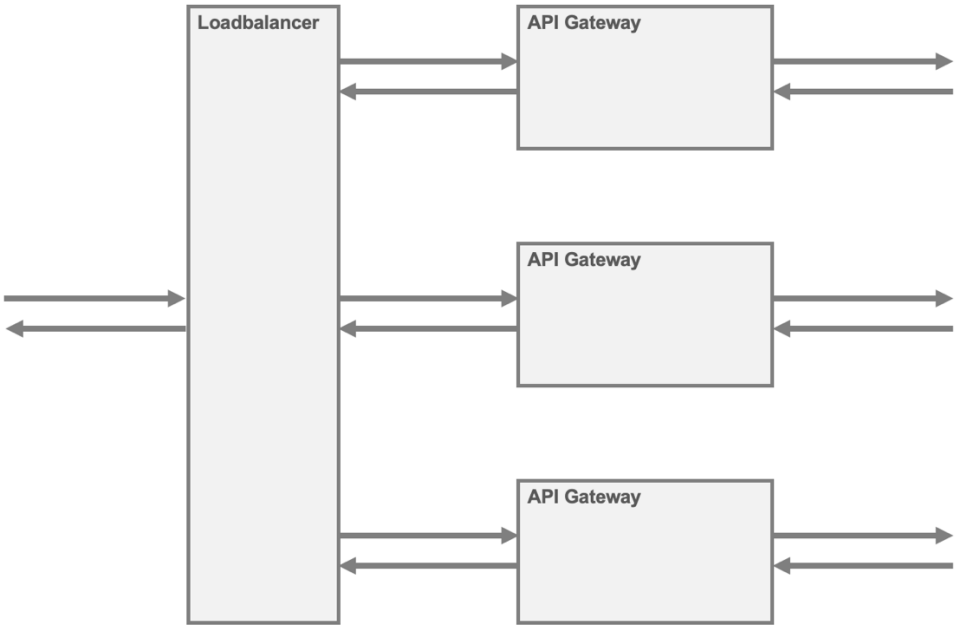


Image: Cluster of API Gateways behind a load balancer.

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.

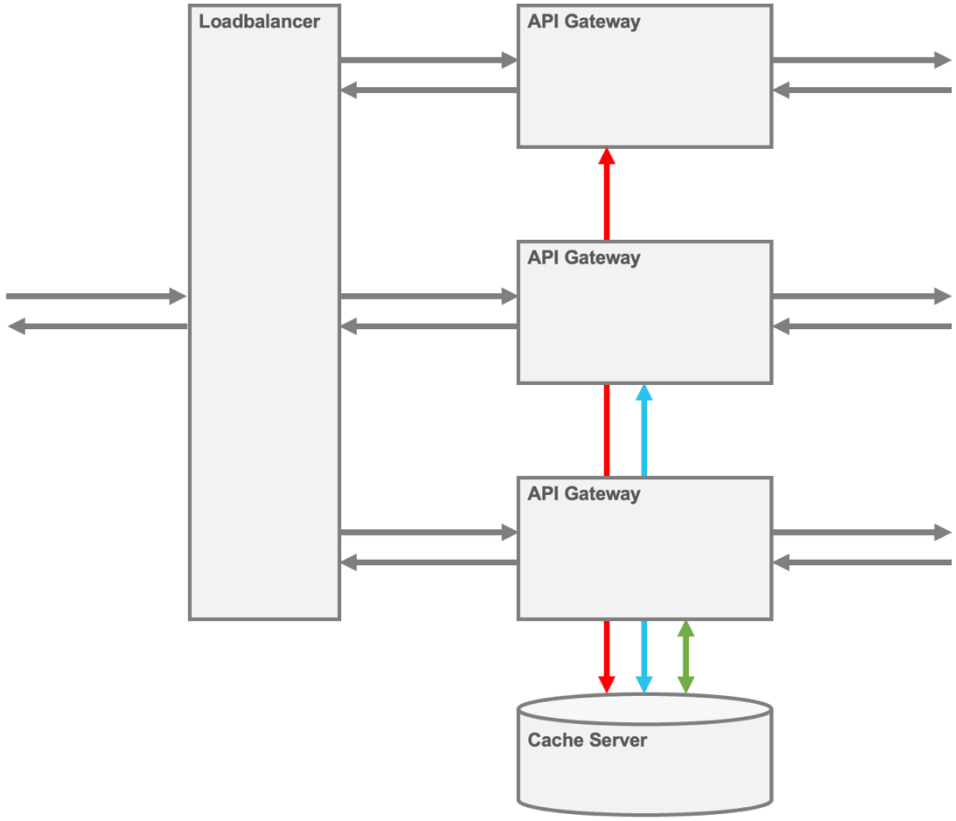


Image: Gateways sharing a cache server like Redis to manage a unified state

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.

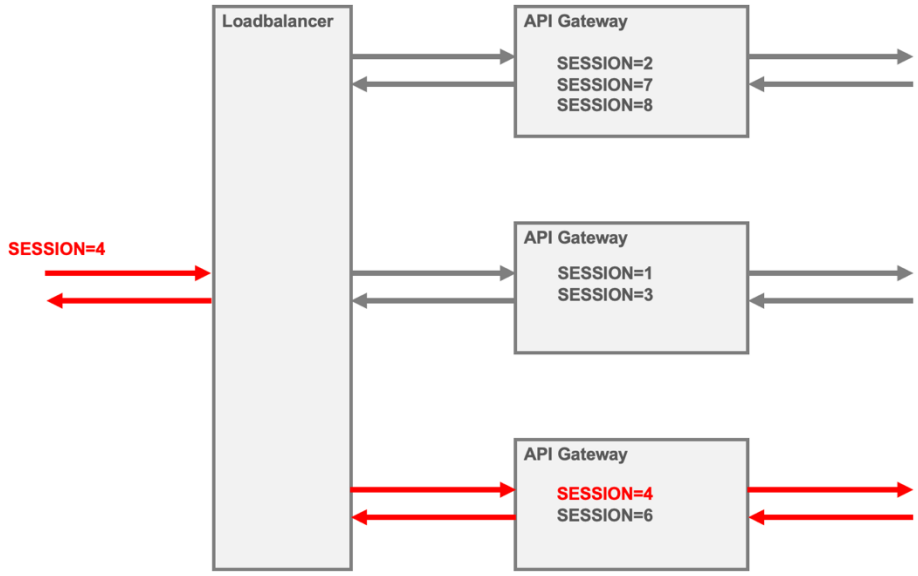


Image: API Gateway instance selection based on session ID

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.

4.4 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.

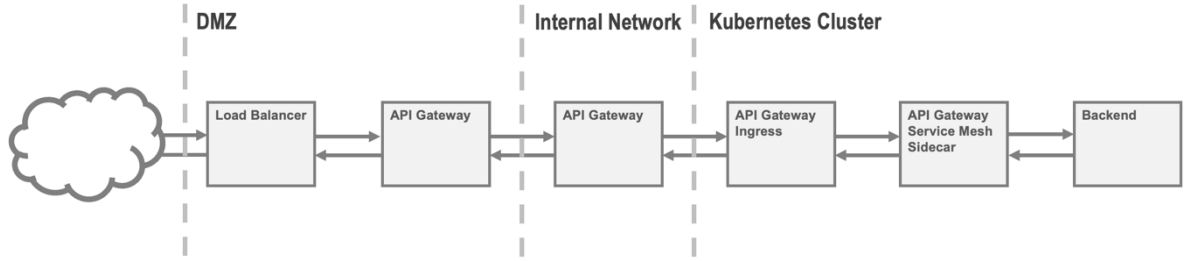


Image: Chain of multiple gateways from DMZ to backend

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 some **performance overhead**. Every hop adds latency and processing time. The challenge is to balance security, observability, and reliability against acceptable performance.

Still, the overhead might be lower than you think.

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**, expecting 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.

4.4.1 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 **steppingstone** 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.

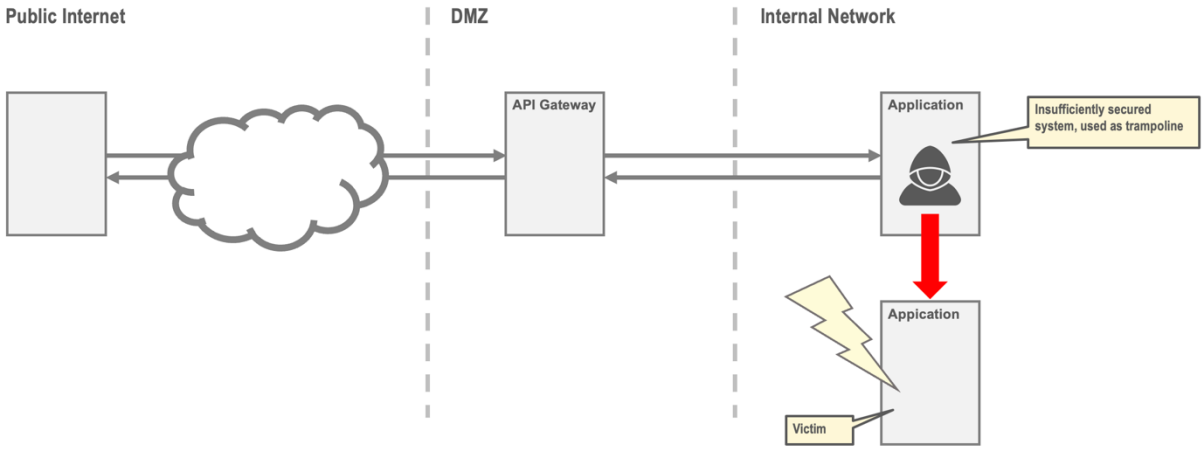


Image: Attacker jumping from compromised intranet system to internal targets

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.

But more zones bring more complexity.

Finer-grained segmentation makes **API routing** between services trickier. Internal systems that previously talked directly to each other now need to cross multiple boundaries—each enforcing 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, well-formed, and validated traffic can pass from one zone to another.

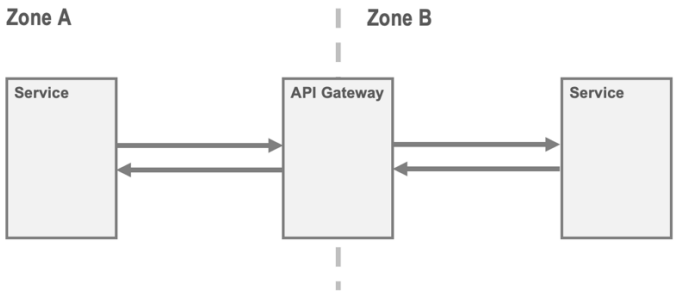


Image: Gateway routing API traffic between internal network zones.

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

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

In the past, setting up an API gateway often meant using a graphical user interface or managing it as part of a larger, sometimes heavyweight, **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 becomes 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 the 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.

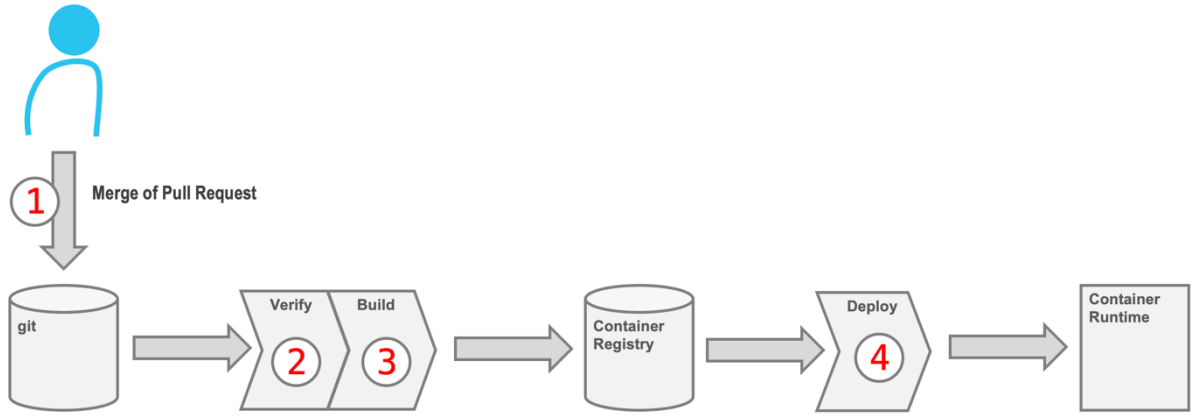


Image: DevOps-based API deployment process

This automated, GitOps-style workflow ensures:

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

APIOps doesn't just improve efficiency it also increases confidence in your 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 can often 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 both documentation and deployment. It simplifies onboarding, reduces human error, and enables repeatable, automated rollout 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.

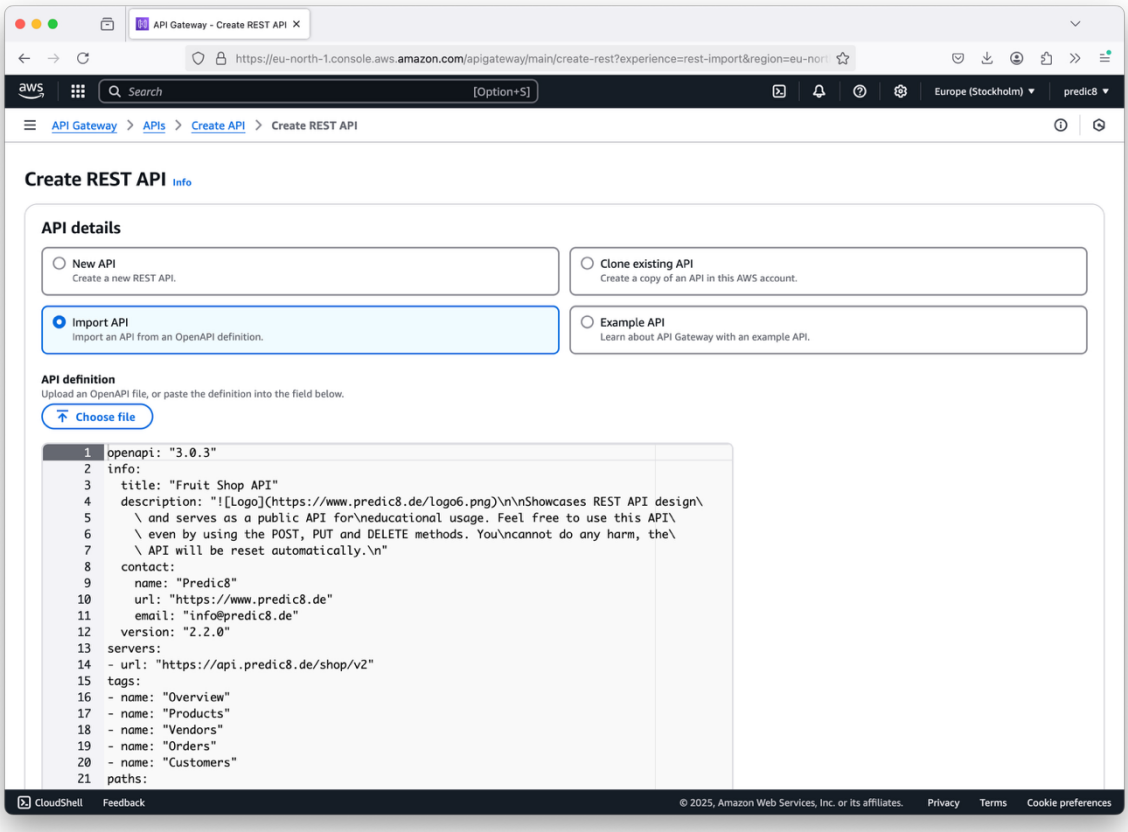


Image: Creating an API from an OpenAPI document in the AWS Console

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:

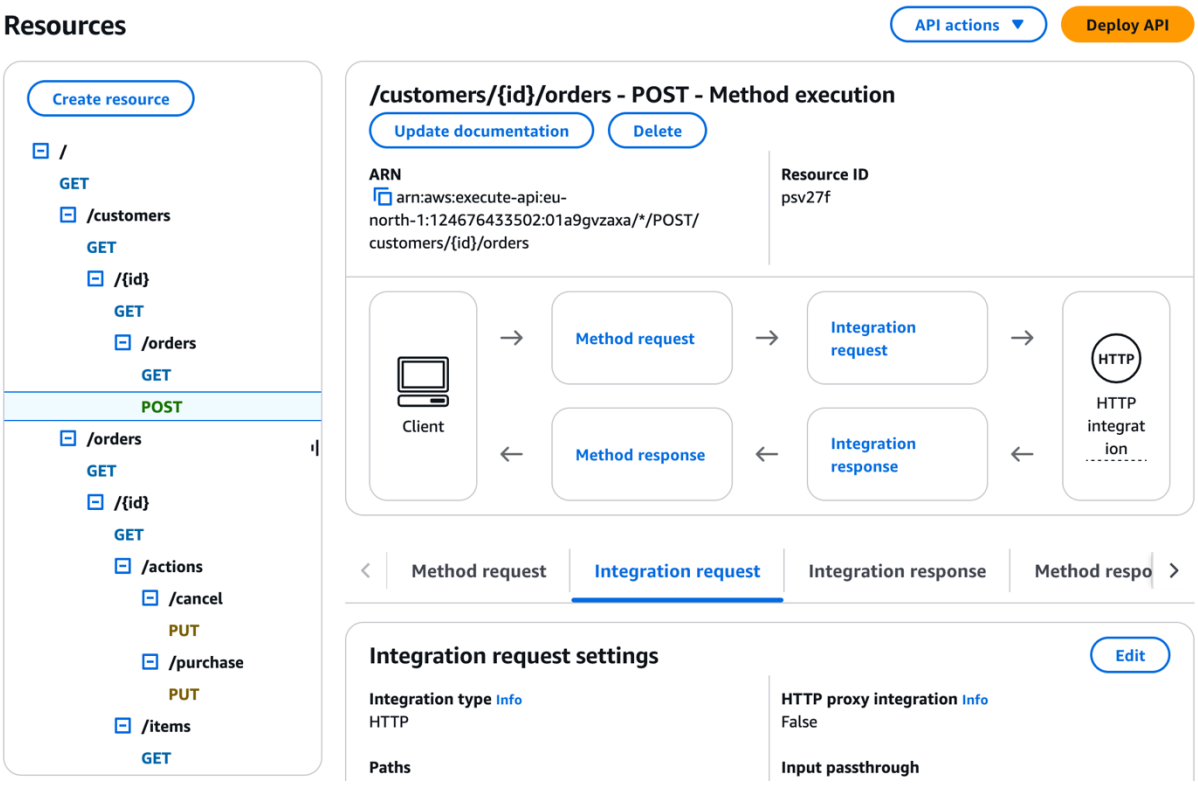


Image: API created from an OpenAPI document in the AWS console

This approach isn't just convenient, it fits perfectly with automated workflows. Since OpenAPI files are structured and versioned, they can live in your git repository 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.

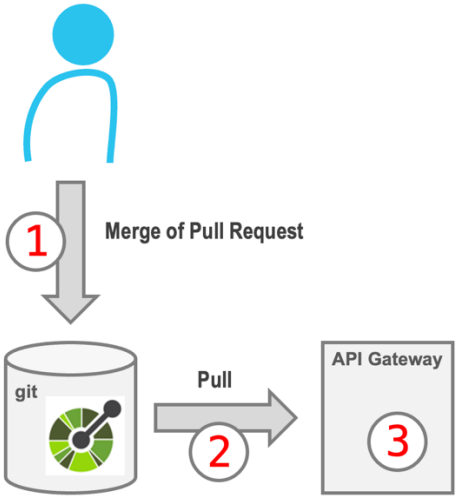


Image: Automated deployment pipeline triggered by OpenAPI changes

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 from another team.

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 security rules for the **Top 10 OWASP API Risks**.

💡 **Sidenote: What is Spectral?**
Spectral is a customizable linter for OpenAPI that helps teams enforce consistency, quality, and security across their 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, compliant, and production-ready.

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. It enables a shared source of truth that supports automation and fosters collaboration.

💡 **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 the API**. This is what's known as **service discovery**.

Below is an example snippet from an OpenAPI file that lists two environments:

```
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've got an API Gateway in front of the backend. A developer **downloads the OpenAPI** file from the gateway that simply forwards the original document from the backend, unchanged. The developer generates a client based on that OpenAPI, and the client ends up talking directly to the backend, **bypassing the gateway.**

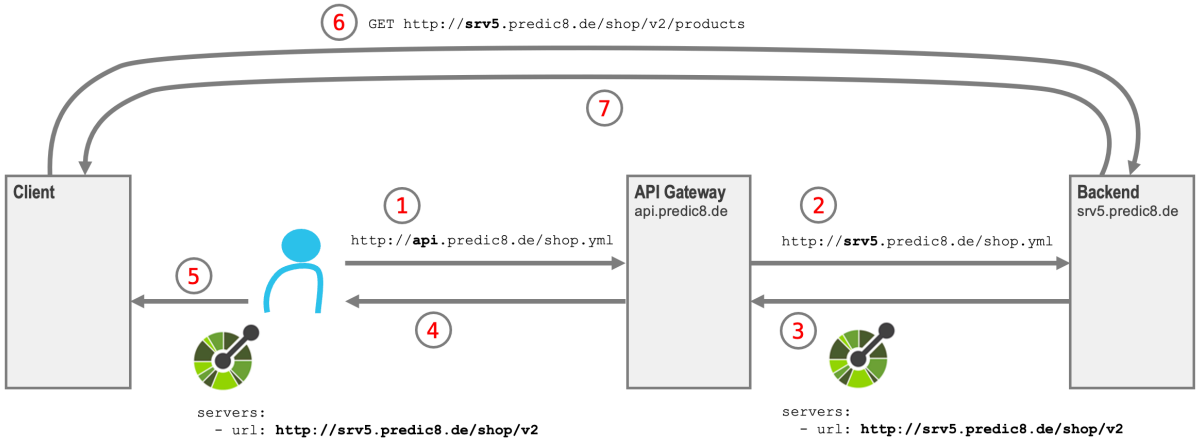


Image: Bypassing the API Gateway caused by an unmodified OpenAPI document

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 address of the gateway.

Sure, you could manually adjust the OpenAPI and serve that modified version instead of fetching it from the backend. But that quickly becomes a maintenance headache. Whenever

the backend API changes, say, a new endpoint is added or a parameter changes, you'd have to update your copy by hand. That's easy to forget, especially in larger teams or automated pipelines.

Letting the gateway dynamically rewrite the document avoids that problem. It passes through the original structure and definitions from the backend, but swaps in the correct URL so that clients always talk to the gateway, not the backend directly.

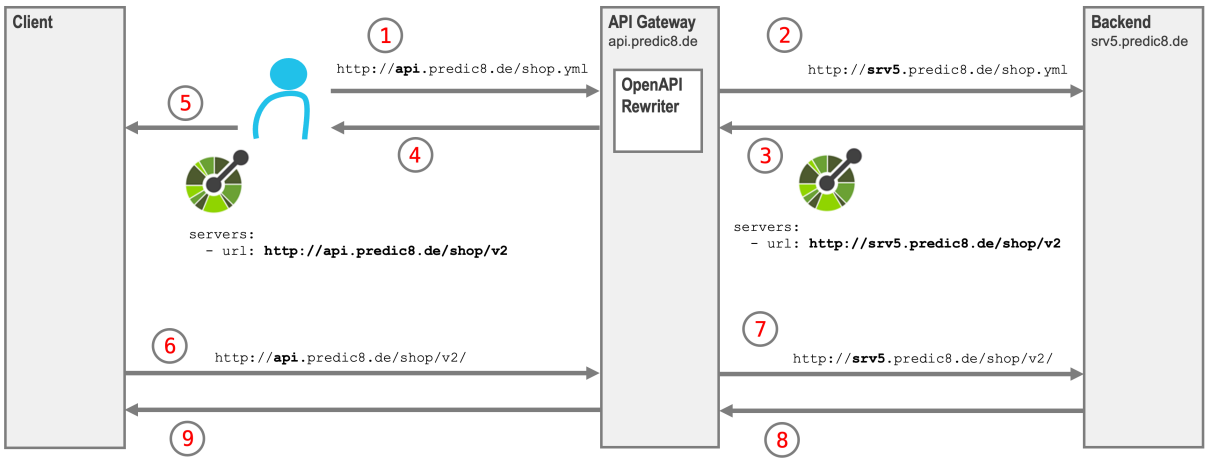


Image: API Gateway rewriting URLs in OpenAPI descriptions

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.

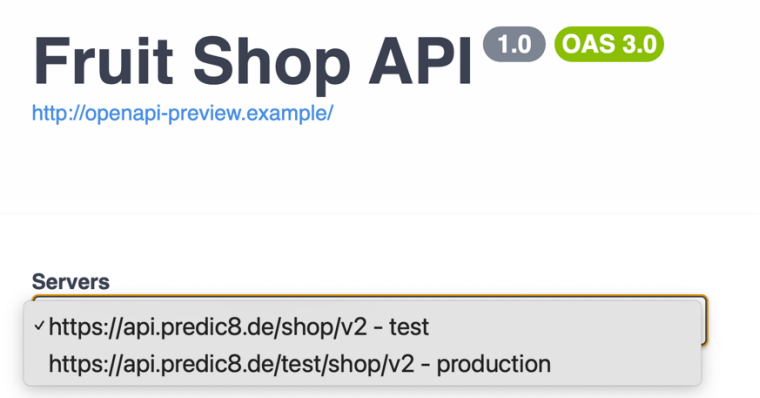


Image: Two rewritten server URLs in the Swagger UI

6.3 Message Validation with OpenAPI

OpenAPI describes an API in detail, which makes it ideal for validating messages.

Many API gateways and testing tools can ensure that requests and responses follow the rules laid out in the OpenAPI description. 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**
  Confirming that authentication and authorization rules.
- **Status Codes**
  Making sure the response code matches the operation's definition.

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 charachters!"
        }
      ]
    }
  }
}
```

🔎 **Looking for Hands-On Examples?**
If you want to see how these OpenAPI features work in practice, head over to chapter 27. There, you'll find real-world configurations and use cases like URL rewriting and request validation using the Membrane API Gateway.

7 API Orchestration

A single API can aggregate or coordinate multiple underlying APIs to fulfill a request. 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.

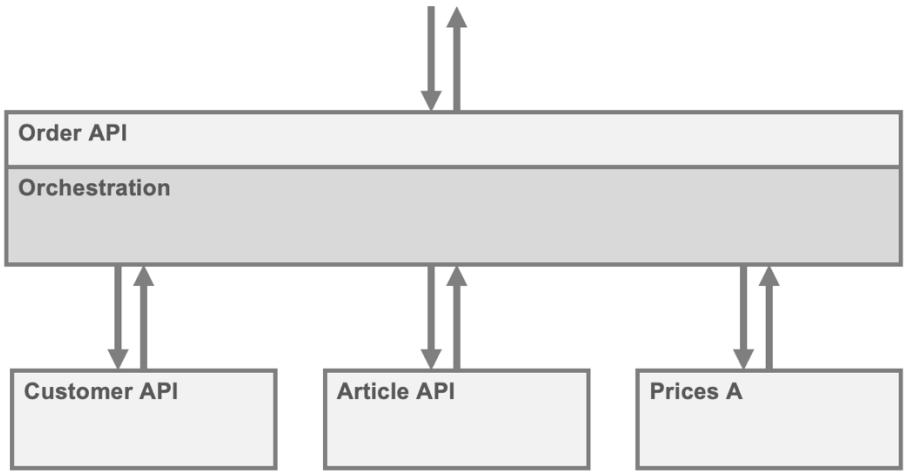


Image: Orchestration of fine-grained APIs

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 support built-in orchestration features, such as:

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

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** or **function runtimes** (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 typically 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, 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 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.

8 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 your organization's ability to respond to evolving threats.

An API Gateway can:

- **Handle transport encryption** using SSL/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—**integrity**, **confidentiality**, **authentication**, and **authorization**—which form the foundation for understanding how secure API communication works in practice.

8.1 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 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.

8.2 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. Encrypting messages ensures that even if someone captures the traffic in transit, they can't make sense of it without the appropriate keys.

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

1. **Encrypting the message itself**
   This ensures that even if the message is intercepted, only someone with the correct key can read its contents. This approach is often used for **data at rest** or in very sensitive API scenarios.
2. **Establishing a secure communication channel**
   More commonly in APIs, confidentiality is achieved by creating a secure, encrypted connection between the client and server. This is done using **Transport Layer Security (TLS)**, the successor to SSL. With TLS, the message content remains private while in transit, protecting it from eavesdroppers. We'll discuss TLS in the next chapter.

💡 **Sidenote: TLS vs. Message Encryption**
TLS encrypts the entire communication channel—headers, payloads, and all—but only while the data is in transit. Once it reaches the endpoint and is decrypted, the protection ends. **Message-level encryption** (such as with JSON Web Encryption) secures the message itself, so it stays protected even after transmission. This is especially useful when messages pass through intermediaries or are stored for later processing.

8.3 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 don't yet know what it is allowed to do.

Authorization, on the other hand, 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/

9 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.

9.1 The Man in the Middle

API communication involves two parties: the client and the server. To protect the integrity and confidentiality during the exchange, 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.**

9.2 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 out of habit to TLS as SSL.

Two Benefits of TLS

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.

9.3 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**

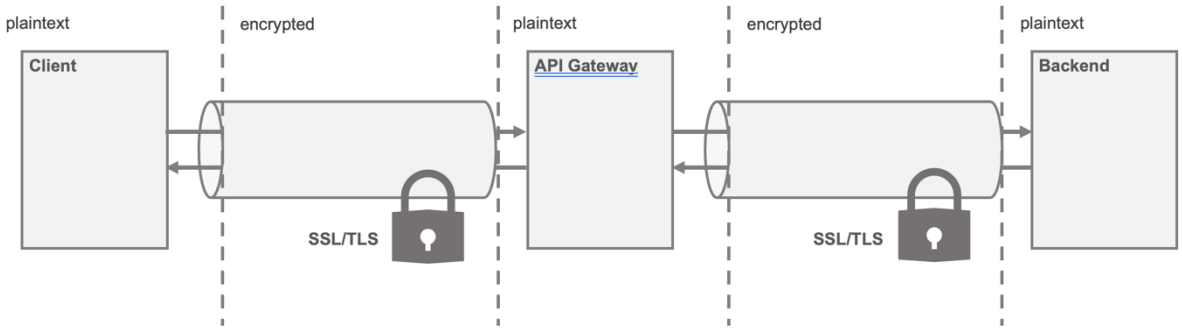


Image: TLS communication between client, gateway and backend

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.

10 Content Protection

A well-worn adage in security 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**.

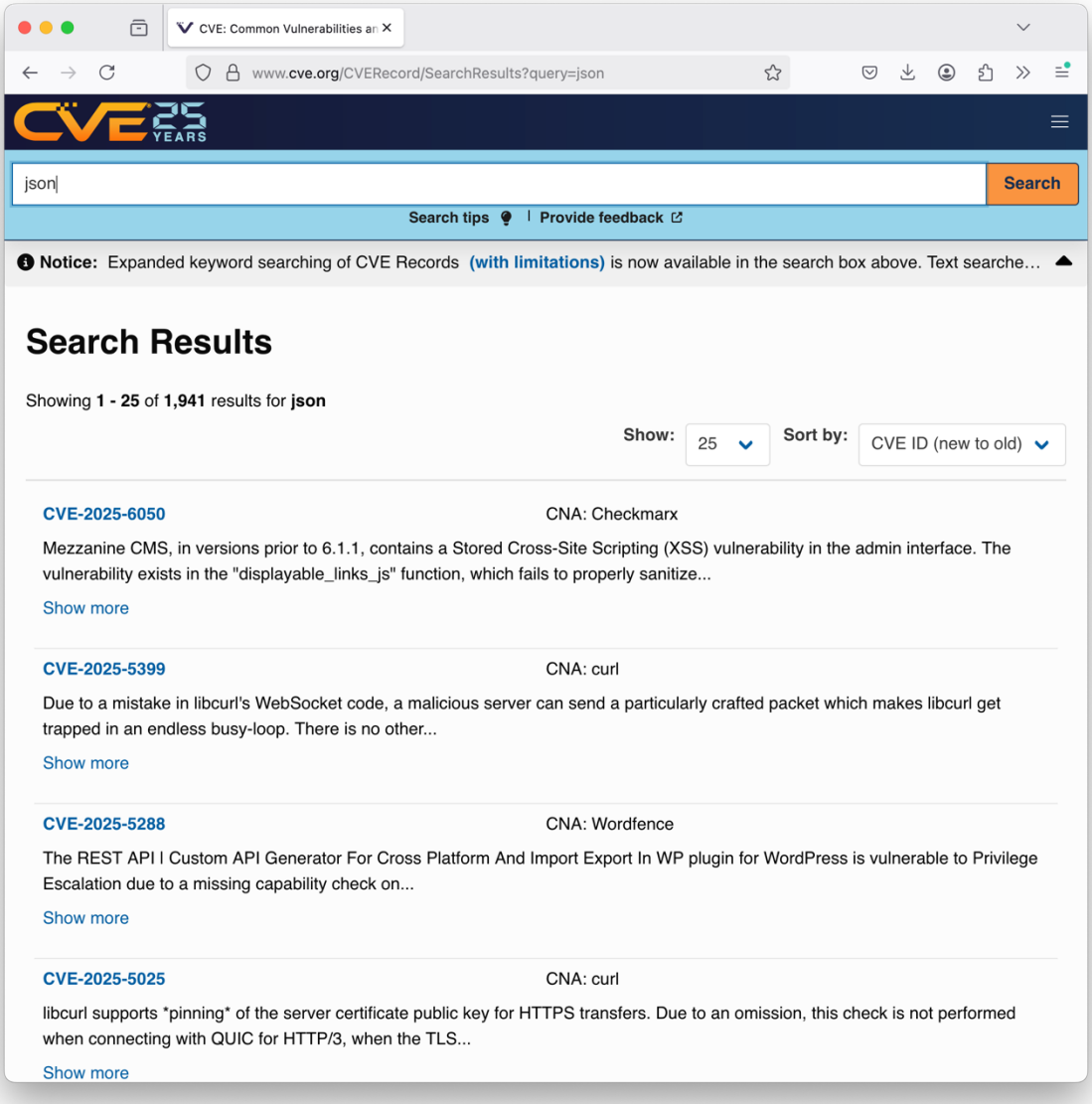


Image: Search result for JSON vulnerabilities on cve.org

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

10.1 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 aren't supposed to contain duplicate keys. But if they do, parsers handle them differently, some use the first occurrence, others use 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 consuming disproportionate resources.

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

10.2 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 a 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>
```

💡 **Sidenote: Why this example looks different**
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` in this case. 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.

10.3 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 conditions, or even the bypassing of 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 in size. 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 (DoS)** 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 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 can include even hidden or 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 (remote procedure calls).

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 is the door for possible denial of service attacks
- Bypass rate limits and perform brute-force attacks

10.4 Content Protection

To mitigate content-based attacks, you can add **content protection rules** directly into your API gateway configuration. Once enabled, the gateway scans and 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 MIME Types

Vendor-specific MIME types like `application/vnd.predic8.product+json` use the `+json` suffix to signal that the content is JSON-encoded, even though the full media type is custom. This convention allows generic JSON parsers to process the payload as long as they recognize the suffix, so it's essential that your API Gateway or security tools don't just check for `application/json` but also any type ending in `+json`.

Resources

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

10.5 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`. 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's defined in the API spec. If not, the request is rejected.

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

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

💡 **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.

11 Injection Attacks

Injection attacks happen when attackers insert malicious input into a system, causing it to execute unintended commands or queries. Common types include SQL, command, code, XPath, LDAP, and XML External Entity (XXE) injections. The impact can vary from leaking or altering sensitive data to gaining unauthorized access or even full system compromise.

11.1 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--
Host: localhost:3000
```

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 (JWT) can be a vehicle for injection attacks.

11.2 Input Validation with OpenAPI

Input validation is one effective ways to reduce the risk of injection attacks. It can block many malicious inputs outright or at least make exploitation more difficult. Take the search parameter `q` from the previous example. If it were 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 shuts out many injection attempts. The SQL injection from the earlier example, for instance, used 85 characters, far beyond the 20-character limit enforced here.

Tip: Input validation isn't just about correctness. It reduces the attack surface at the edge.

To make APIs more secure, define every parameter and field 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 your OpenAPI definitions. The OWASP API Security ruleset already includes checks for the OWASP API Security Top 10, but 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 a maxLength.'
    given: '$.paths[*][*].parameters[?(@.in=="query")]'
    then:
      field: schema.maxLength
      function: defined
```

That said, input validation alone won't eliminate all injection threats. Blocking every suspicious word like `union`, `drop`, and `or` would mean rejecting even legitimate user input in some cases. 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.

11.3 Why Validation Alone Isn't Enough

Input validation is often cited as an effective defense against injection attacks. But consider this email:

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

At first glance, it doesn't appear valid, yet common email validators accept it without complaint:

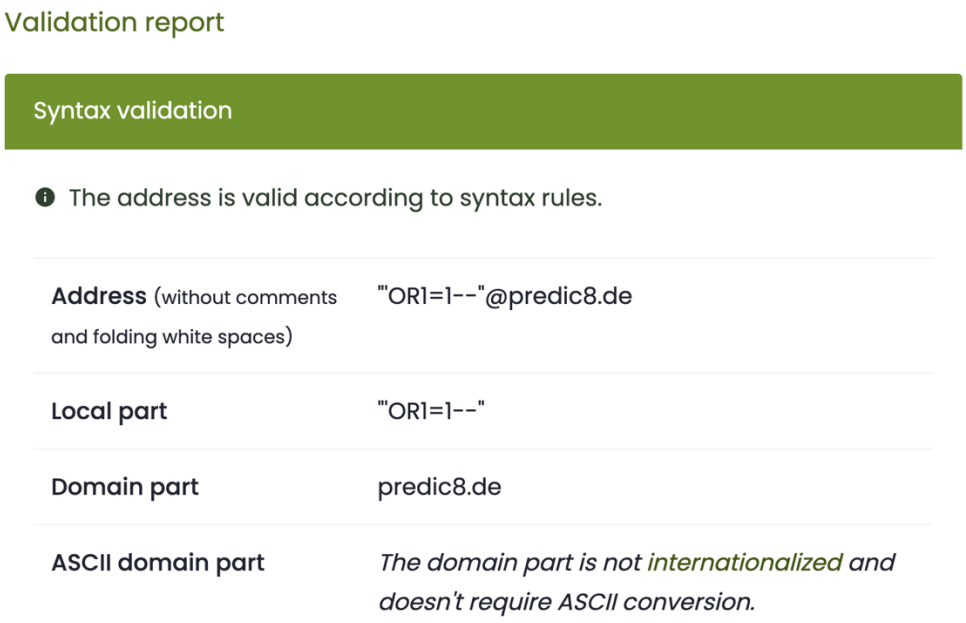


Image: Excerpt of validation result at https://verifalia.com/validate-email

Even though this string passes email validation rules, it contains an SQL injection that can bypass authentication. Using this as a username, an attacker can potentially log in without a valid password, usually as the first user listed, often the administrator.

While validation at the gateway or service is valuable and raises the bar for attackers, it is insufficient on its own. Genuine protection against SQL injection requires secure coding practices at the backend, primarily through parameterized queries (prepared statements) rather than dynamic SQL queries.

Unfortunately, budget or legacy constraints often make secure coding impractical. In such cases, external protection measures via firewalls or API gateways become necessary.

11.4 Effective Injection Protection

As mentioned, secure backend coding and robust architecture are the most effective defenses. However, several tools and strategies can offer additional protection when placed in front of a backend service:

Injection Signatures

- Injection scanners analyze incoming requests for specific patterns indicative of potential attacks. While effective, they may also produce false positives. For example, scanning for the `'` character could incorrectly block valid names like O'Reilly.
- Curated rule sets, such as those provided by Snort, offer constantly updated detection signatures.

Machine Learning and Artificial Intelligence

- AI-driven tools can detect anomalies and suspicious patterns more flexibly than fixed signature-based rules. Many platforms and plugins are now leveraging AI for advanced injection detection.

11.5 API Gateway vs. Web Application Firewall (WAF)

Both API gateways and Web Application Firewalls (WAFs) can protect against injection attacks. However, their roles differ slightly:

- **WAFs** typically excel at generic injection detection, and they're usually already integrated into enterprise infrastructure.
- **API Gateways** are ideal for input validation specific to APIs. Using JSON or XML schema definitions (XSD), gateways can validate requests precisely per API endpoint. If this validation is offloaded to the WAF, configuration updates must accompany every API change, increasing management complexity.

Many enterprises adopt a layered approach, leveraging both gateways and WAFs: the WAF handles generalized injection scanning, while the gateway provides schema-specific validation.

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

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

12 Message Validation

Message validation is a core defense mechanism in API security. Since APIs often operate at the boundary of systems or organizations, they must carefully inspect incoming data before processing it.

To validate effectively, the system needs a clear understanding of what a valid request looks like. These expectations should come from the business side. Only they can define what values, formats, and rules make sense. Once defined, these rules can be formalized and handed over to the API Gateway or security tools.

API Gateways can enforce these definitions using structured formats such as JSON Schema, XML Schema (XSD), OpenAPI, or WSDL. With precise schemas in place, gateways can reject invalid input before it reaches backend systems.

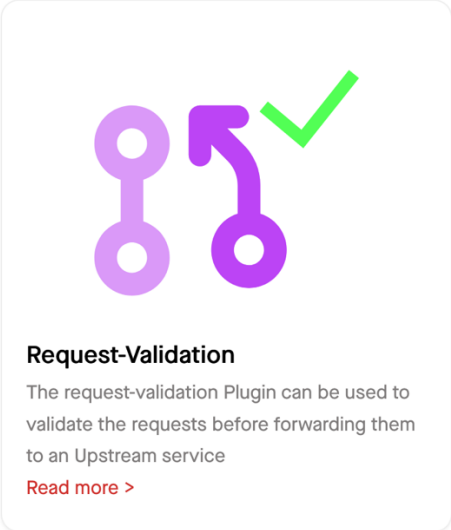


Image: The Request Validation Plugin for the APISIX Gateway

12.1 Response Validation

While much attention is given to input validation, output validation is often an afterthought. It’s easy to assume that since the backend generates the output, it's safe by default. But overlooking what an API sends out can result in unintentional **information disclosure**.

Why Is Response Validation Important?

Consider the following API 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.springframework.http.converter.json.HttpMessageConverter.read(Mes
      sageConverter.java:343)
At
      org.springframework.web.servlet.mvc.method.annotation.RequestProcesso
      r.resolveArgument
at
      org.springframework.web.servlet.DispatcherServlet.doService(Dispatche
      rServlet.java:964)
at
      org.springframework.web.servlet.FrameworkServlet.processRequest(Frame
      workServlet.java:1006)
at
      org.springframework.web.servlet.FrameworkServlet.doPost(FrameworkServ
      let.java:909)
at javax.servlet.http.HttpServlet.service(HttpServlet.java:696)
at
      org.springframework.web.servlet.FrameworkServlet.service(FrameworkSer
      vlet.java:883)
at javax.servlet.http.HttpServlet.service(HttpServlet.java:779)
at
      org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(Appl
      icationFilterChain.java:227)
... 51 more",
}
```

This stack trace unintentionally reveals detailed information about the internal workings of the server:

Technology Stack

- The application uses the **Spring Framework** 5.3.x and **Jackson** for JSON deserialization 2.13.x.
- It runs on a **Tomcat/Catalina** server 8.5.x or 9.0.x.
- Line numbers in the trace helped to estimate specific library versions.

Specific Methods and Libraries

- Method names like `MessageConverter.readJavaType` and classes like `DispatcherServlet` reveal how requests are handled internally.
- Attackers can craft payloads targeting known vulnerabilities or quirks in these methods.

Armed with specifics like these, an attacker could search the **Common Vulnerabilities and Exposures (CVE)** database for known exploits affecting the versions of **Jackson**, **Spring**, or **Tomcat** in use. Attackers like verbose and detailed error messages. Once they know what technologies are in play, they can iterate with malformed requests to provoke different error behaviors and harvest more clues.

This is why **output validation** is just as important as input validation. By filtering sensitive information like stack traces or exception details, you deny attackers the information they need for targeted, informed attacks.

The screenshot below shows know vulnerabilities of an old Jackson library that can serve a hacker for his attack.

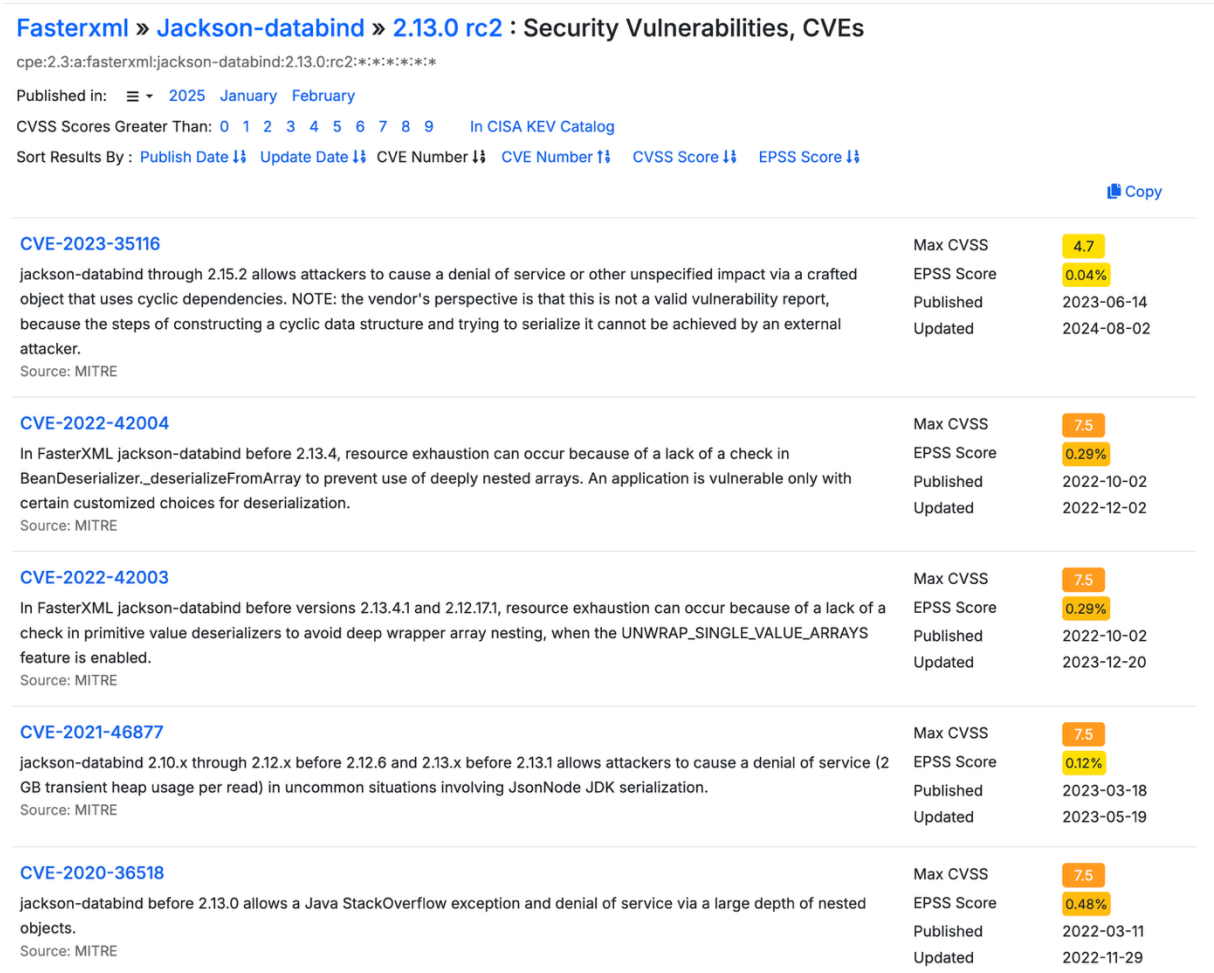


Image: Known vulnerabilities in an outdated Jackson library

Don’t forget output validation. For example, the Apache APISIX gateway includes a plugin for validating incoming requests, but it doesn’t offer a built-in plugin for validating responses. This missing piece can result in sensitive or confidential data unintentionally leaking in the response, especially if backend services return overly detailed error messages or internal fields.

> **Tip: Block Stack Traces in Responses**
> Always block or sanitize 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 middleware to catch and rewrite such responses before they reach the client.

Resources

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

The Problem with Response Validation

Activated response validation can lead to awkward validation errors where the problem is something completly different.

The image below illustrates this case. An error occurs during processing in the backend at step 3. The backend returns an error message step 4. The response validation in the gateway expects a valid response that is described in a JSON Schema, XML Schema or Open API document. Instead it gets a generic error message from the backend that is not described. As a result the validaiton will fail and the client gets a validation error message instead of the database failure in the example. Because the error the client gets does not reflect the root cause the search for the problem can be hard.

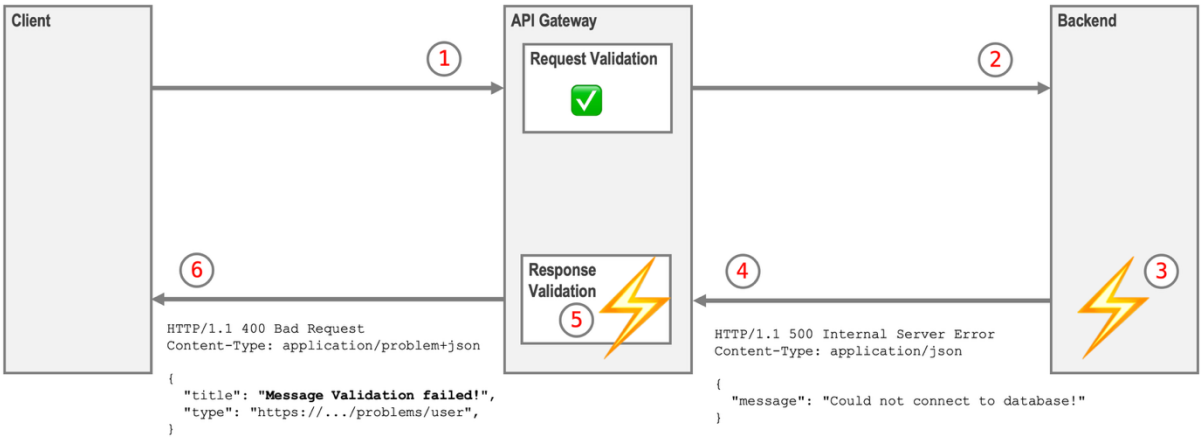


Image: Failed Response Validation after Backend Error

Some gateways support to turn off validation for error messages. But this simple solution is not satisfieing as technical details from error messages can slip through the gateway to the client.

It would be ideal if the backend is returning only error messages that are described in the schema used at the gateway to validate the message. Unfortunately this isn't often the case in real world. There are old or off the shelf backends that return their own error message.

There is a solution to this dilemma. Let the gateway transform the backends error messages to a format that is described by the schema and the validator can handle.

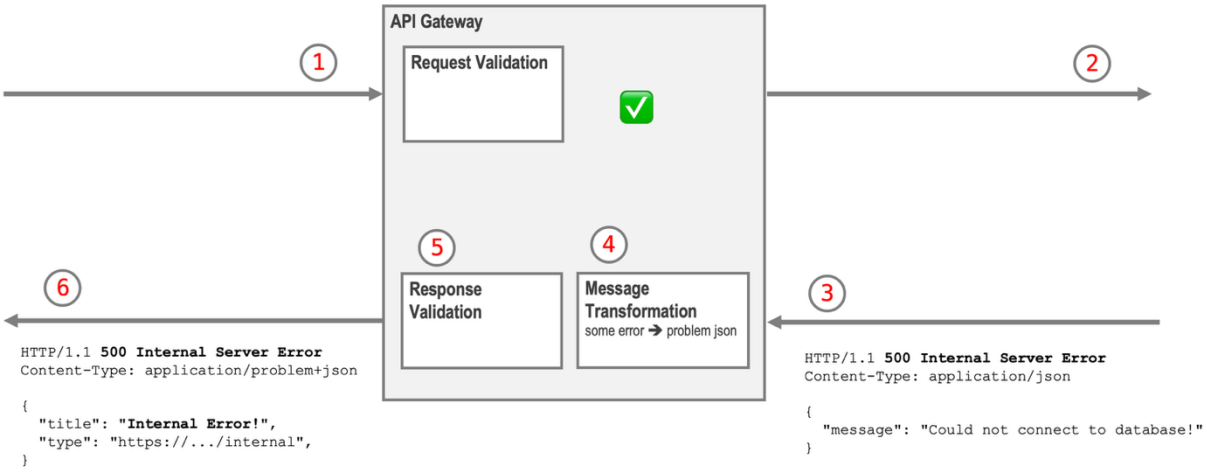


Image: Transforming error custom messages from the backend before validation

For validation to work, error messages must be defined in a JSON Schema, OpenAPI spec, or XML Schema. We'll look at how to do that in the next section.

12.2 Describing Error Messages

To properly validate responses, you need not only a schema for successful messages but also for error messages. Otherwise, you risk running into an awkward situation: the backend returns a helpful error, but the user just sees a vague "response does not match schema" from the gateway. Not exactly helpful.

Problem Details (RFC 7807)

RFC 7807, **Problem Details for HTTP APIs**, defines a standardized format for error messages in HTTP-based APIs. It's designed to be both human-readable and machine-parseable, making it easier for clients to understand and process errors in a consistent way.

Here's a typical example:

```
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"
}
```

The `title` and `type` fields are required according to the problem details specification,. The `uri` field shown here is an optional, custom extension added by the API. That's the beauty of Problem Details, you can start with a standard structure and easily extend it to fit your needs.

Validating Error Responses with OpenAPI

OpenAPI allows you to define not only successful responses but also structured error responses. This lets an API Gateway validate error messages just as it would validate success payloads, helping to prevent unintended data leaks or malformed error responses.

Here's an example OpenAPI snippet for an endpoint that returns both a successful and a **404** error response:

```
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 how the `404` response explicitly uses the `application/problem+json` media type and references a shared `Problem` schema.

Here's how the schema might be defined:

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

The `Problem` object includes the mandatory `type` and `title` fields from **RFC 7807**, plus a custom `uri` field specific to this API. The line `additionalProperties: false` is key. It ensures that unexpected fields in error messages are caught by the OpenAPI validation at the gateway. This helps prevent sensitive internal details from slipping through in verbose error responses.

💡 **Security Tip**: **Error Validation with OpenAPI**
Use the same level of precision for your error schemas as for your success responses. OpenAPI validation doesn't just improve documentation, it helps enforce clean, consistent, and secure API behavior.

Status Code Wildcards in OpenAPI

Listing out every possible HTTP error code in your OpenAPI spec can get tedious. Fortunately, OpenAPI supports wildcards to help you 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 your API definitions more compact while still covering a wide range of potential responses. It's especially useful when combined with shared response schemas like Problem JSON.

💡 **Tip**: While wildcards are helpful for reducing redundancy, you should still define specific responses when you want to return custom messages for certain status codes.

Resources

Problem Details for HTTP APIs
https://datatracker.ietf.org/doc/html/rfc7807

12.3 JSON Validation

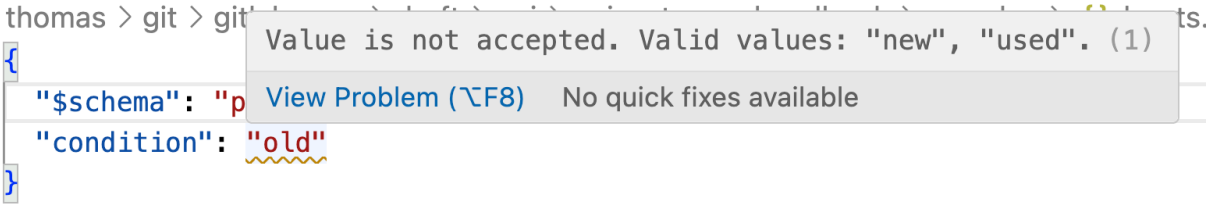
In OpenAPI documents, message bodies are described using **JSON Schema**—even if the OpenAPI file itself is written in YAML (yes, that's totally valid). When an API Gateway validates a message against an OpenAPI definition, it's essentially performing JSON Schema validation under the hood.

However, OpenAPI validation doesn't just stop at the message body, it also covers the **path and HTTP method**. JSON Schema validation, by itself, only applies to the content of the message body. If you only have a standalone JSON Schema, you can still set up validation against it.

Let's look at a quick example. Here's a simple JSON Schema:

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

This defines an object with one field, `condition`, which may only contain either `"new"` or `"used"`. Now consider this incoming message:



The validation will fail, `"old"` is not part of the allowed `enum`. The gateway can reject this request immediately, preventing bad or unexpected data from ever reaching your backend.

To enforce this kind of control, many gateways support JSON Schema validation via plugins or built-in policy engines.

> 💡 **Sidenote: JSON Schema Basics**
> JSON Schema is a widely adopted format for describing JSON data. It lets you define types, required fields, formats, and constraints such as min/max values, regex patterns, or enumerated values.

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

12.4 XML Validation

Just like JSON uses JSON Schema, XML has **XML Schema Definitions (XSDs)** for validation. These schemas define the expected structure of an XML document, its elements, attributes, data types, and the required order.

An API Gateway can use XSDs to verify whether a message is **valid**, meaning it adheres to the rules defined in the schema.

This kind of validation is especially useful for **legacy systems** or **B2B integrations**, where XML remains a widely used format.

12.5 OpenAPI Validation

Since OpenAPI typically includes schema definitions for both requests and responses, it's well suited for validating API messages.

One key advantage of OpenAPI over standalone JSON Schema validation is its tight integration with HTTP paths and methods. This allows the API Gateway to automatically determine which schema applies to a request, based on the endpoint being called. With plain JSON Schema, you'd need to define separate schema files for each request and response or fall back to overly generic definitions that offer less protection.

Some API Gateways support OpenAPI validation natively, while others provide tools that convert OpenAPI documents into validation policies or configuration files for the gateway.

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

13 API Keys

API keys offer a **simple** way to secure your API without the need for complex protocols. They allow you to control who can access your API, make basic authorization decisions, and even track usage patterns. This straightforward approach makes API keys especially appealing when you need rapid integration and minimal overhead, even if API keys don't provide the fine-grained control or dynamic capabilities of more advanced mechanisms like JSON Web Tokens (JWT).

13.1 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 a header, `X-Api-Key`, with the API key as value. The API or gateway verifies the key by matching it against a list or database. If the key is found, the request is authenticated; if not, the API returns an `HTTP 401 Unauthorized` status code.

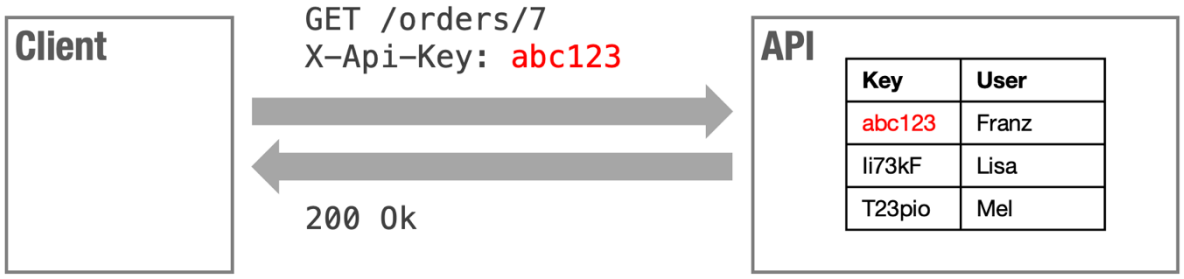


Image: API Key HTTP header

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 extensive because all clients using the shared key would be affected. Therefore, it is best practice to assign each client its own unique API key, ensuring that a breach only impacts a single client rather than the entire system.

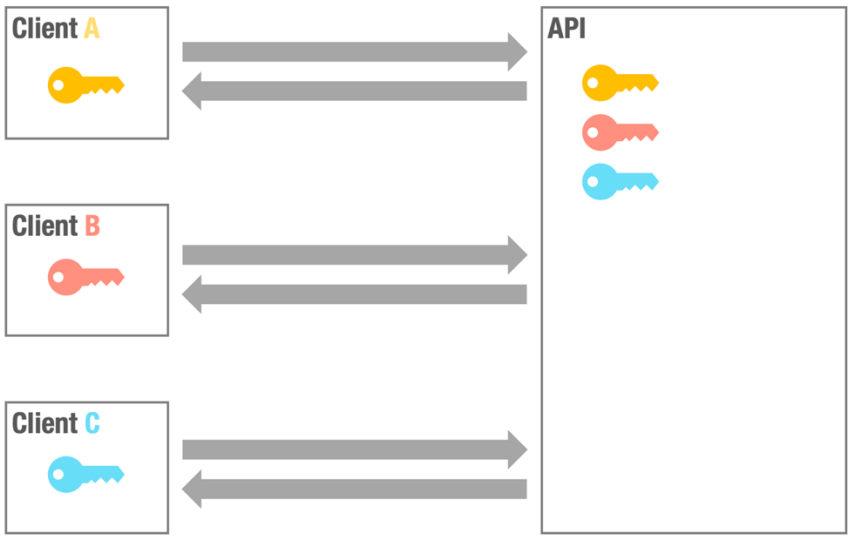


Image: Multiple clients with individual API Keys

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 API key,** there's no session, cookie, token renewal, or persistent connection involved.

At first glance, this may seem inefficient. Why include and validate the key every time? Isn't that an 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. 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 **fully stateless**, which brings significant architectural benefits:

- Easier horizontal scaling
- Improved reliability (no session storage needed)
- Better fault tolerance across distributed systems

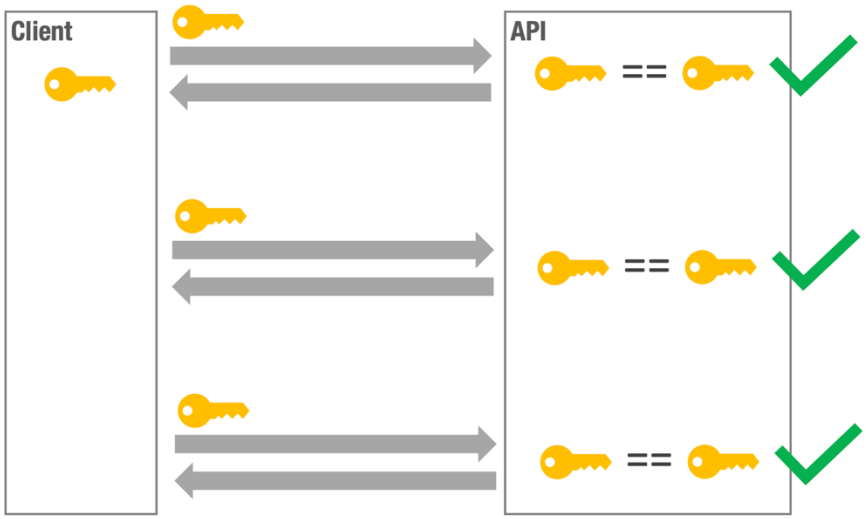


Image: Stateless Security, each request must provide the API Key

API keys and the Basic Authentication mechanism, known from HTTP 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 do not only authenticate requests but also enable additional functionality such as analytics, rate limiting, and more granular security policies based on API Keys.

14 Tokens and API Security

Using a combination of **username and password** for authentication has several well-known limitations:

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 for APIs: they act as **temporary, scoped credentials** that are easier to manage, revoke, and control without exposing the underlying username-password combination.

14.1 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 food truck

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.

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 or the client's identity
2 Can be scoped to specific actions (e.g., “read-only access”)
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**.

14.2 How (Bearer) Tokens Work

Most API tokens today 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:

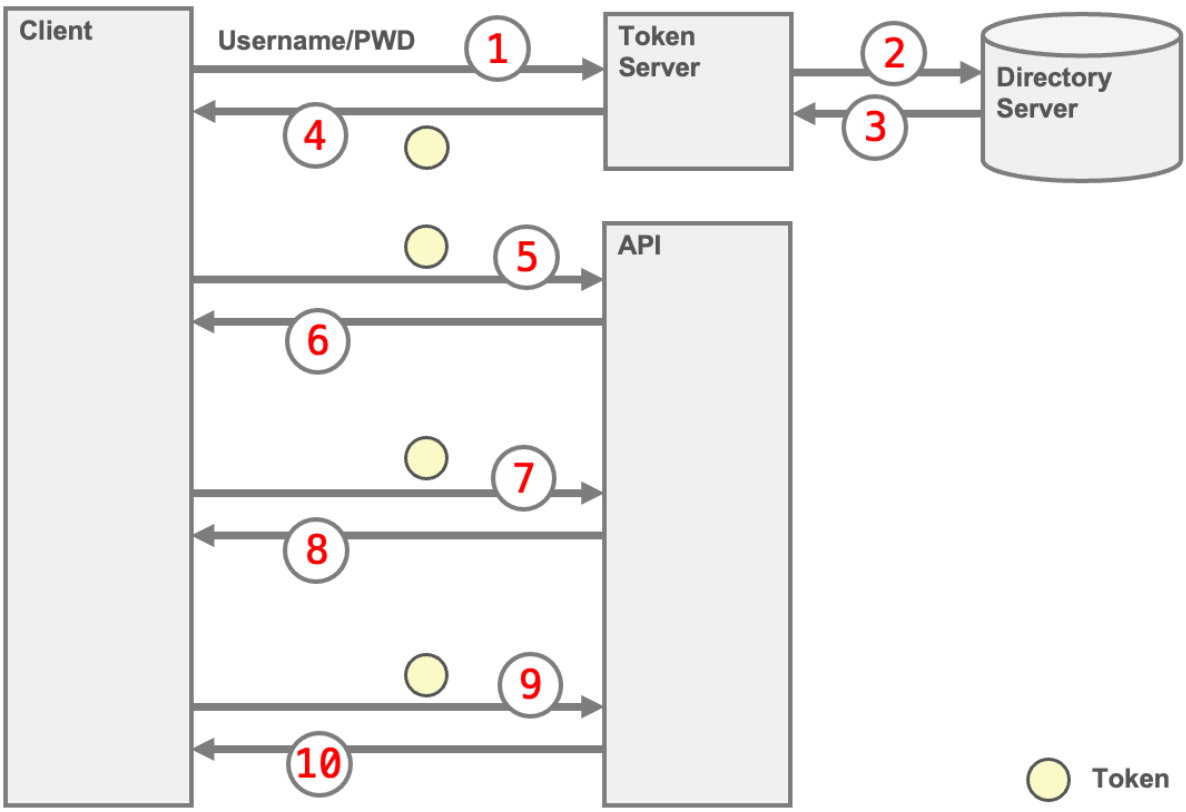


Image: Issuing and presenting a bearer token

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 info, 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 toke*n*** 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're simple and flexible. But they do rely on keeping the token out of the wrong hands.

14.3 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 encrypts a value using the client's public key. The client must then decrypt that value using its private key and return the result. 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 service.

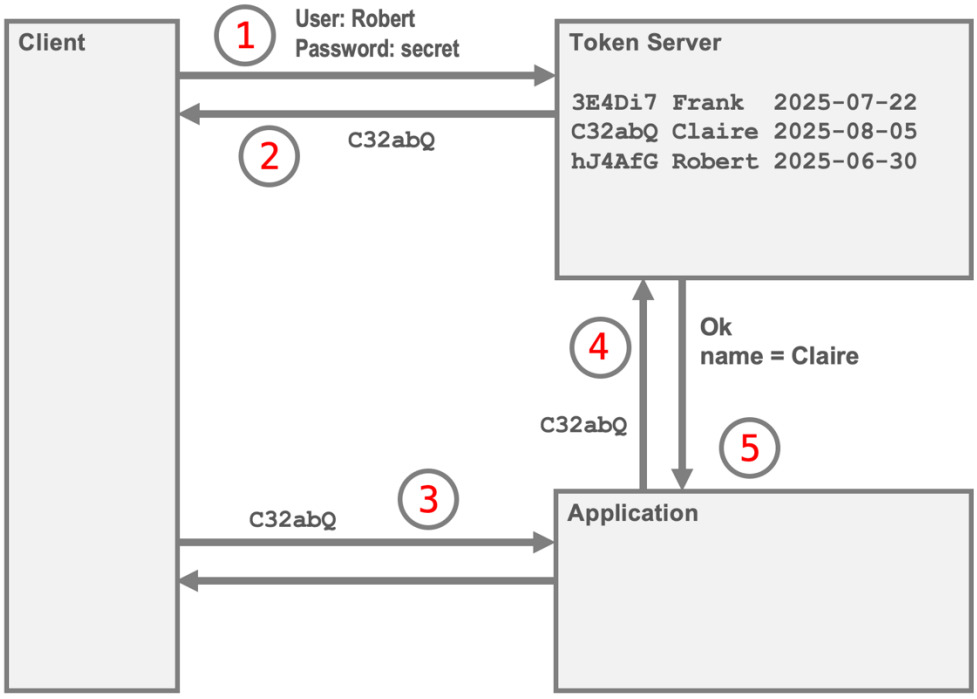


Image: verification of an opaque token by the token server

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.

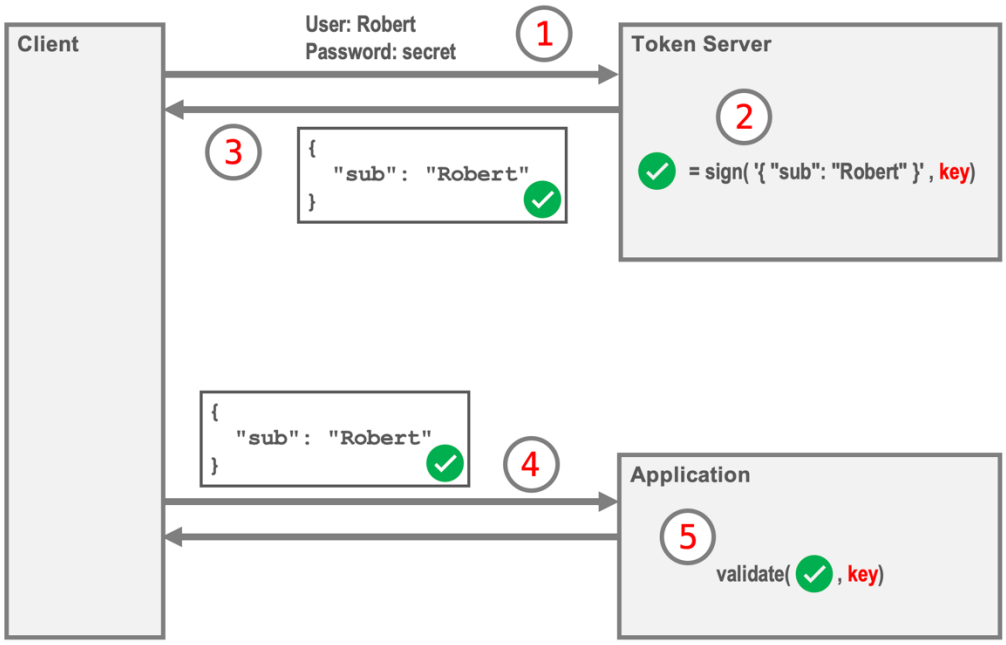


Image: Verification of a structured token by the receiver without consulting a token server

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** can be verified locally without a network call, making them highly efficient. But once issued, they're hard to revoke. That's why expiration becomes even more important.
- **Opaque tokens** require server-side verified, which adds overhead but allows for centralized revocation and better control.

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.

15 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 common 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..
```

Although this is discouraged for security reasons.

Let's dive deeper into these simple yet powerful tokens.

15.1 What is a JSON Web Token?

JSON Web Tokens (JWT)**,** pronounced "jot", are compact, 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 HTTP request includes a JWT in the `Authorization` header using the Bearer token scheme:

```
GET /customers/ HTTP/1.1
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJodHRwczovL2FjY291bnRzLnByZWRpYzguZGUiLCJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IlRvYmlhcyBQb2xsZXkiLCJpYXQiOjE3NTAxNzA5OTZ9.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 web server might choose to allow the HTTP 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)

15.1.1 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 contents, but cannot alter them 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 typically is **not considered highly 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.

15.2 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 tokens 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.

15.2.1 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, while the payload (and e.g. the username, for that matter) might contain special characters like äöü:+/, the JWT's serialization does not.

15.2.2 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.

Header

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.

Payload

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
```

Decoded, it looks like this:

```
{
  "iss": "https://accounts.predic8.de",
  "sub": "1234567890",
  "name": "Tobias Polley",
  "iat": 1750170996
}
```

We will discuss the claims shown here in section 15.2.4.

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.

15.2.3 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 consists 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 — this is the part you can't 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.

15.2.4 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 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 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 so called **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.

15.3 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 you should 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**.

15.3.1 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.

   You might want to allow a little wiggle room (say, up to 30 seconds) to account for network latency. 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!

15.3.2 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 strictly require tokens valid for more than one API, you might consider issuing tokens with "aud": ["finance", "lunch-menu"]. However, be aware of the security implications in this case! If you have more APIs, e.g., "wiki.predic8.de", "tickets.predic8.de", and "crm.predic8.de", using separate tokens still provides more security, of course.

16 OAuth2 and OpenID Connect

OAuth2 and OpenID Connect are the backbone of modern authentication and authorization on the internet. They're the reason you can log into a new app using your Google or Microsoft account without creating yet another password.

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.

16.1 OAuth2

Back in chapter 14.2, we talked about using tokens at a festival to buy food. Let's stretch that analogy just a little further. It still holds up.

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 credit card?

This is exactly what OAuth2 standardizes: **the process of acquiring a token**.

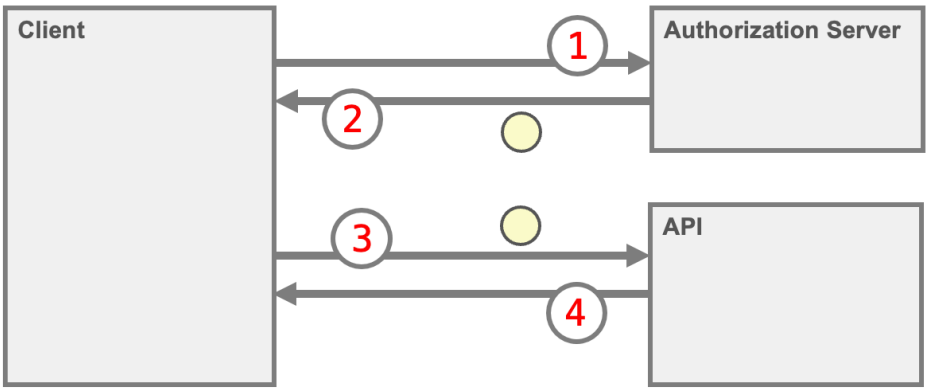


Image: Acquiring a token using OAuth2 and using it for API access

OAuth2 focuses on how the token gets back to the client (step 2). But it doesn't fully specify what the client must send to the authorization server to authenticate itself (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?

16.2 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 GUI (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).

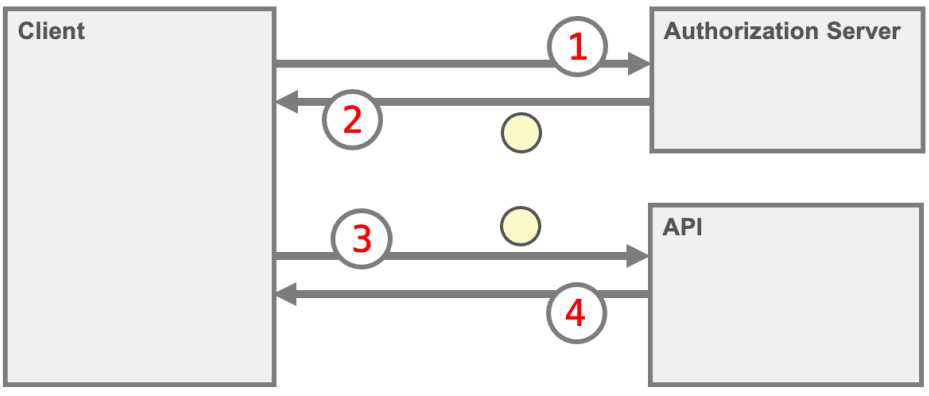


Image: API Access *on behalf of the client*

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 the 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.

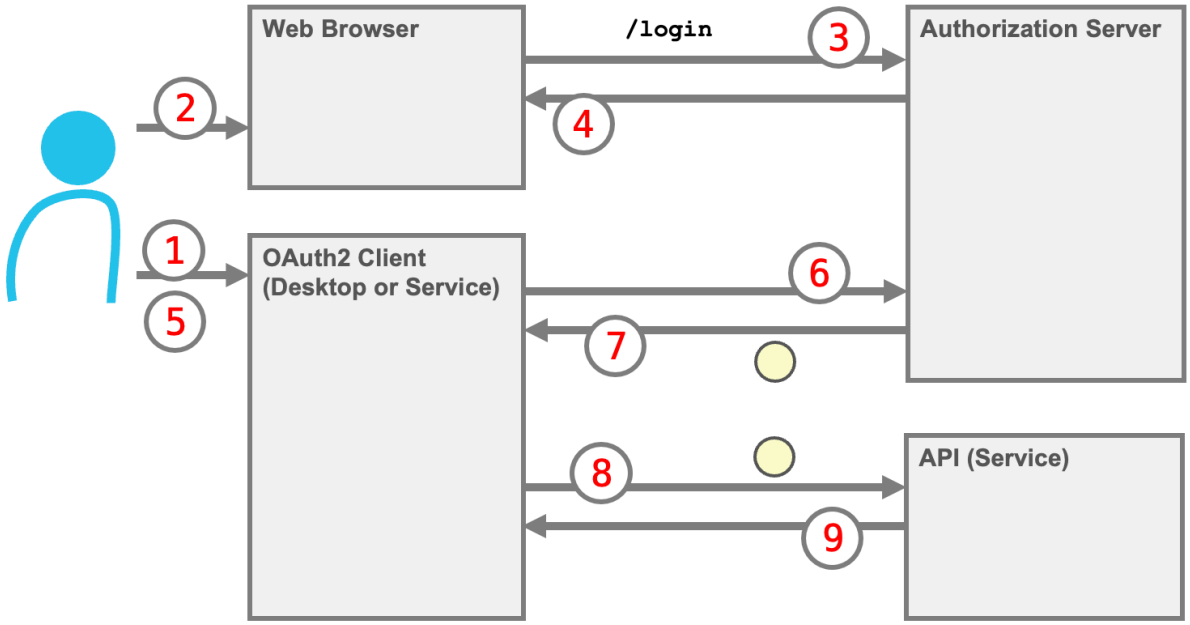


Image: A user authenticates through a desktop app, which then accesses the API on their behalf.

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**.

16.3 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 it?
- 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.

16.4 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 really complicated, OIDC makes the configuration of API Gateways really simple!

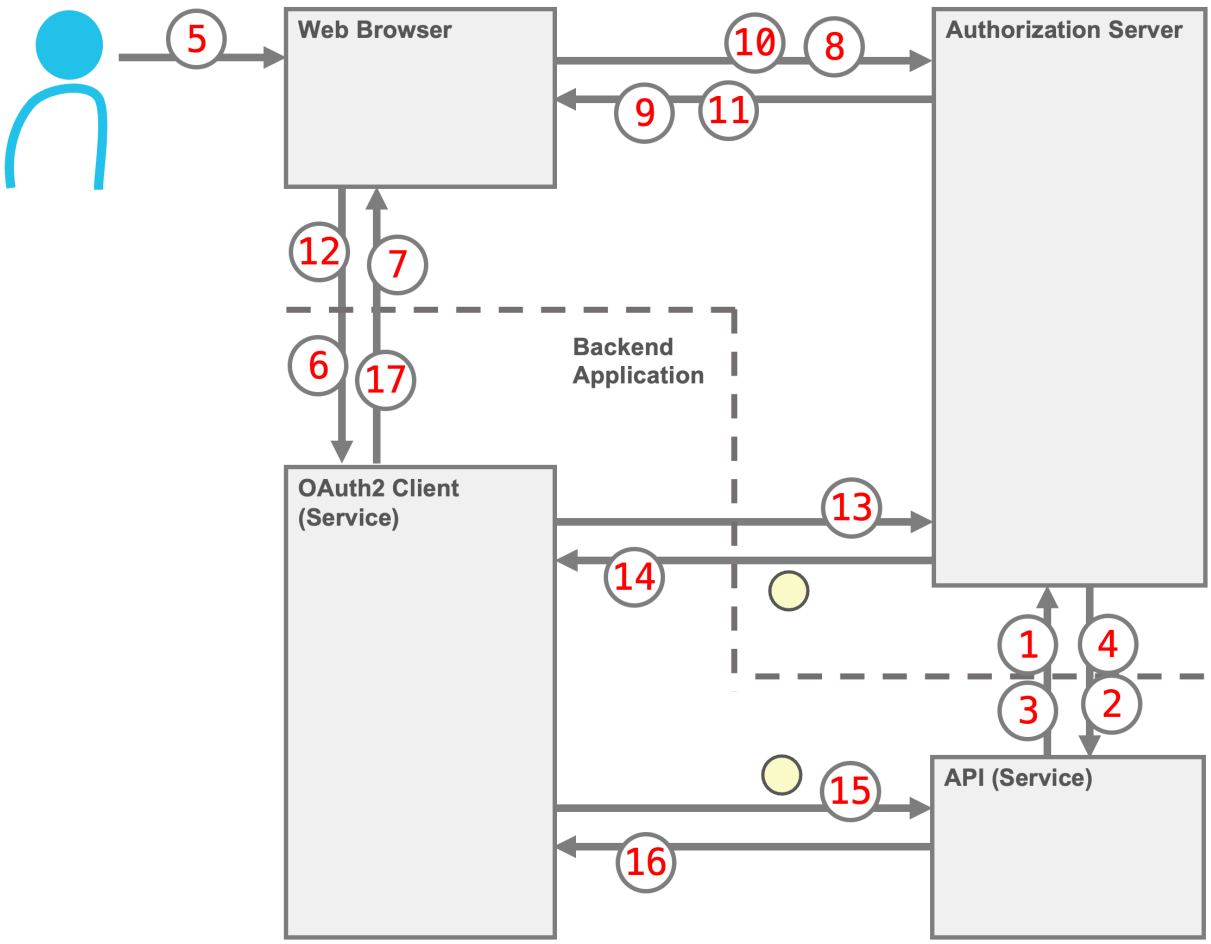


Image: A Demo Case of OAuth2, OpenID Connect (Core and Discovery) and Best Practice

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 his 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 it 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 0.	✅ 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.

16.5 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.

16.6 Setting up a JWT Verifier with OIDC

OAuth2 and OpenID Connect (OIDC) 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.

All you need to configure is the **base URL of your Authorization Server**. That's it. The gateway or verifier takes care of the rest: OIDC Discovery defines that appending `/.well-known/openid-configuration` to that base URL to be the URL of the discovery document.

The illustration below shows the discovery flow:

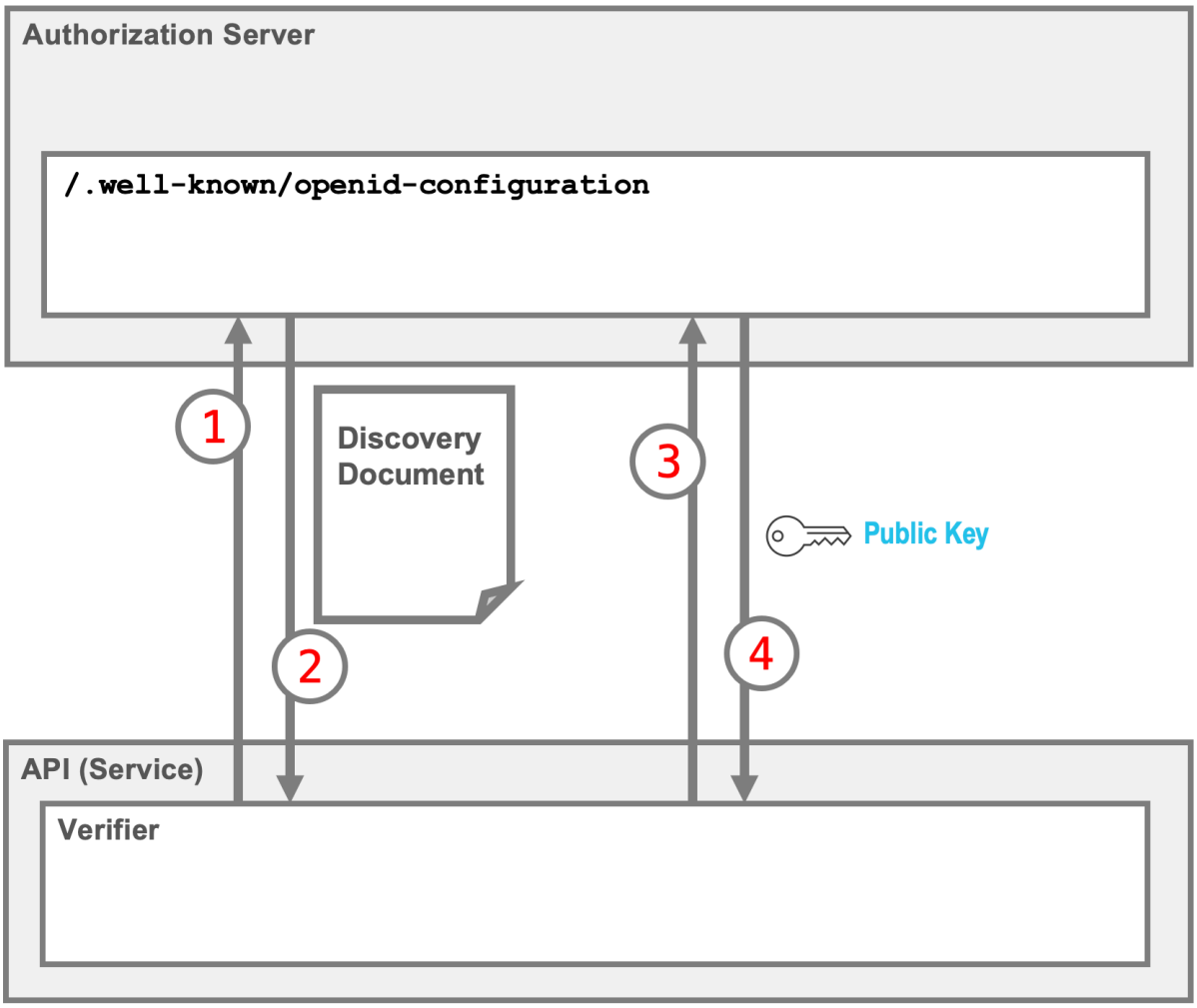


Image: How an API retrieves public keys and configuration from an OpenID Auth Server

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 this process happens over HTTPS and the identity provider is authenticated via TLS, the verifier can safely trust the keys it receives actually belong to the Authorization Server.

This dynamic setup is a best practice when working with modern identity providers like Keycloak, Auth0, Azure AD, or Google Identity and it saves you from manual key management headaches.

⚠️ **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.

16.7 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 API or API Gateway receives the JWKS via a *trusted outbound* TLS connection from the Authorization Server. Thereby, trust in them is established.

16.8 Verification of JWT Signature and Claims

The API or API Gateway receives the JWT on the other hand 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's checks must include:

- `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.

17 Rate Limiting

API calls happen fast, so fast that you hardly notice how many are zipping through until it's too late. Take this example Python script. It creates a million products in the Fruitshop:

```
import requests

for i in range(1000000):
    requests.post("https://api.predic8.de/shop/v2/products/",
                  json={"name": f"Fruit-{i}", "price": 1.99}
                 )
```

Tiny scripts like this can quickly overwhelm an API. You're welcome to run this script against the Fruitshop API and see what happens.

Why would anyone 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 use.
- **Resellers**
  Users who build their business around your APIs.
- **Developer tests:**
  A harmless coworker or student trying out your 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

So, what happens when you push an API a little too hard and cross its rate limit? The server enforces its rules. Here's the response a client received after sending more than 500 requests in just one hour:

```
HTTP/1.1 429
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 are especially useful because they give the client clear guidance on what to do next. Whether to 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)

💡 **Sidenote: What about X-RateLimit?**
Many APIs still use older, non-standard headers such as:

```
X-RateLimit-Limit: 5
X-RateLimit-Remaining: 0
X-RateLimit-Reset: 48
```

Supporting both standard `RateLimit-*` and legacy `X-RateLimit-*` headers improves compatibility, especially when integrating with older APIs.

How does Rate Limiting work?

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's imperfect: IPs 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 only on IP addresses. Many API Gateways even let you configure custom expressions with JSONPath, XPath, or similar languages.

Here are a few examples:

Expression	Descrption
`header['Authentication']`	Count requests by authenticated user (HTTP header).
`jwt.claims['sub']`	Count by JWT subject (user ID).
`request.path`	Count by request path. User or id doesn't matter.

`$.product.id`	Count by product ID in the JSON payload using JSONPath.

This flexibility allows you to apply rate limits 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 more flexibility: users can interact with different endpoints without being unfairly throttled, while still protecting your system 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.

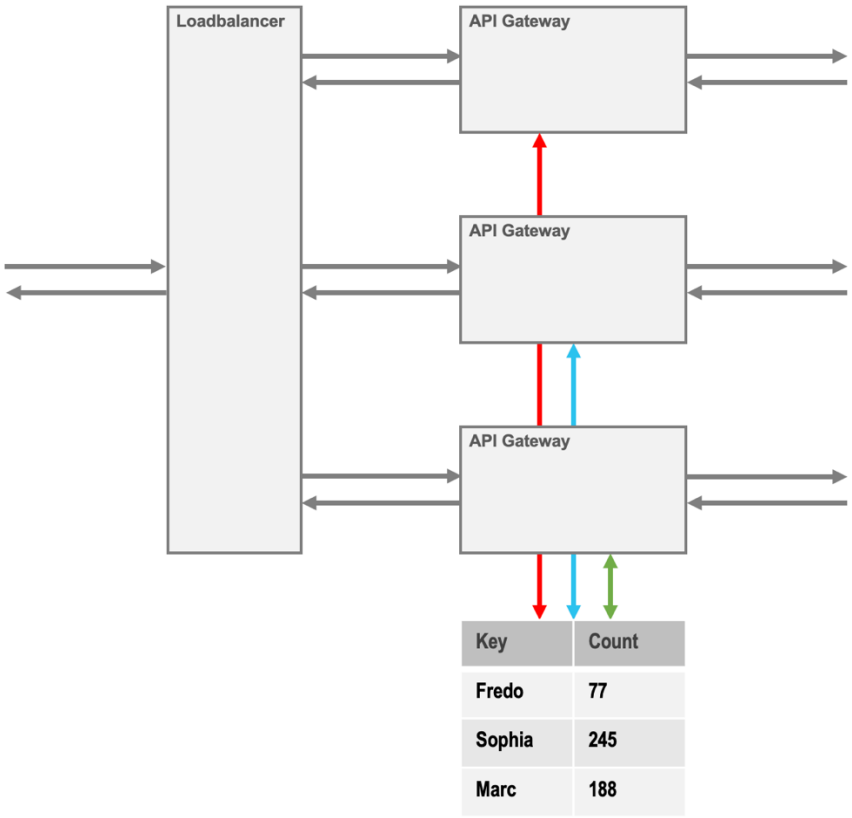


Image: Shared counters for rate limiting

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.

Trick: 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/

18 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, and that can introduce privacy risks.

Logging is a common example. 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, this ID counts as personal data. Just like names, addresses, or phone numbers, and therefore 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 - - [07/05/2025:12:00] "GET /employees/XXXXX
```

This keeps the log useful for operations and debugging while protecting the individual's identity.

Masking can be applied not just to path parameters, but also to query strings, headers, and even request or response bodies.

19 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 XPath injection.

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.

19.1 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. Similar to how OpenAPI describes REST APIs, WSDL outlines 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 Definitions (XSDs)**, 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 are, the more powerful validation becomes, not only for message structure but also as a layer of defense that improves security.

20 Cross Origin Resource Sharing (CORS)

JavaScript-based API clients are often part of modern web pages. **Single-page applications (SPAs)**, in particular, load data dynamically and frequently call headless backend systems using APIs. But browsers enforce a strict security mechanism called the **Same-Origin Policy**. This policy blocks scripts from one origin from accessing resources hosted on another. It's a core protection against cross-site scripting attacks. And yes, it applies to APIs too.

Cross-Origin Resource Sharing (CORS) 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 a **web page running inside a browser**. Server-to-server communication and native mobile apps are not affected.

20.1 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).

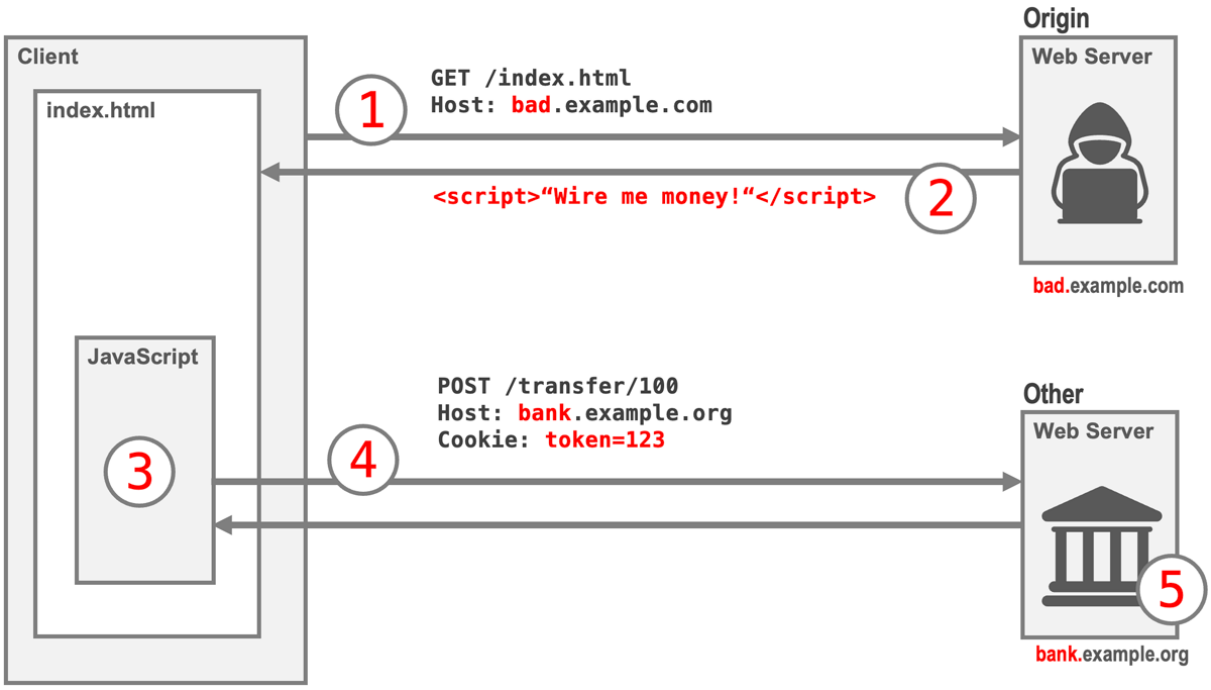


Image: CSRF attack against an API

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 security by enforcing the **Same-Origin Policy**. These protections prevent cookies from being sent in cross-origin requests unless explicitly allowed, significantly reducing the risk of CSRF in modern web apps.

> 💡 **Sidenote**: **CSRF and bearer tokens**
> Cross-Site Request Forgery (CSRF) mainly affects session-based authentication 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.

20.2 How the Same-Origin Policy prohibits API Calls?

Modern browsers protect users from CSRF attacks by enforcing the **Same-Origin Policy**. This policy restricts scripts running in the browser from making HTTP requests to a different origin than the one that served the page.

An *origin* is defined by the combination of **protocol**, **hostname**, and **port**.

While this is great for security, it can get in the way of legitimate use cases, like calling APIs from single-page applications (SPAs).

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 sees this as a cross-origin request and **blocks the POST request**.

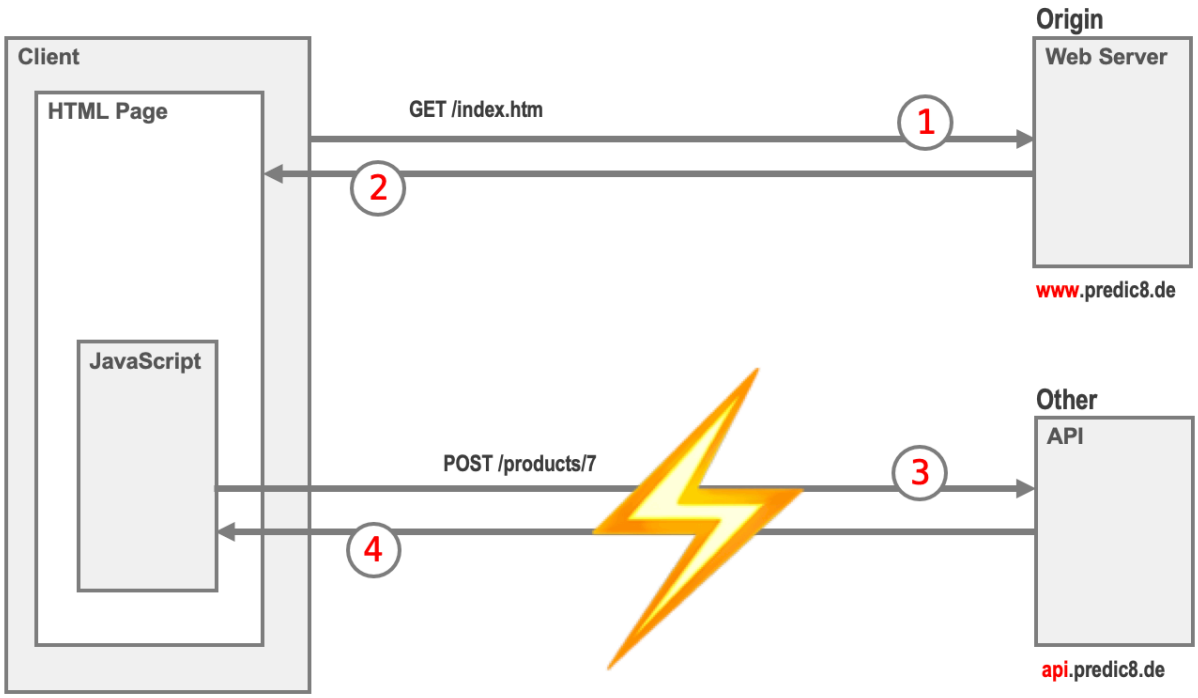


Image: Browser enforcing **Same-Origin policy** on a POST request to a different server.

That's where Cross-Origin Resource Sharing comes in. CORS provides a way for servers to **declare which cross-origin requests are allowed**, and under what conditions. 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.

20.3 How does CORS work?

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 in the request and in the response.

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. If the server allows that origin, it responds with:

```
Access-Control-Allow-Origin: https://www.predic8.de
```

This tells the browser: “You can allow a call from code that comes 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 the browser’s built-in security mechanism that blocks cross-origin requests by default.
CORS is a **way for servers to opt-in** and tell the browser, “This origin is allowed.”
Important: CORS is **enforced by the browser**, not by the server. If a backend forgets to include CORS headers, the browser will block the request, even if the backend would’ve responded.

20.4 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.

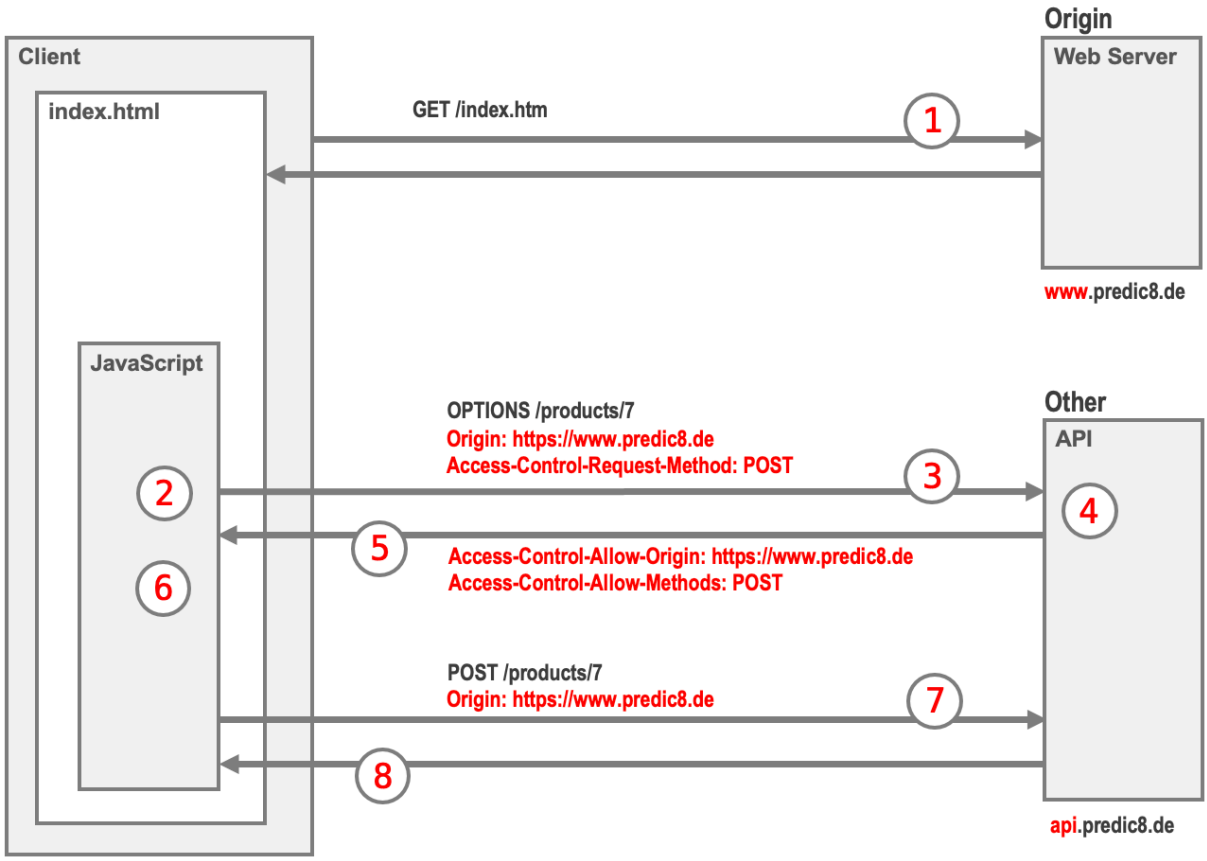


Image: How a browser performs a CORS preflight before a POST request

Here's how it works, step by step (illustrated in the image above):

1. A page with JavaScript is loaded from the origin.
2. 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 API call, it first sends an **`OPTIONS`** request to ask if 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
```

4. The server receives the `OPTIONS` request and evaluates it 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
Access-Control-Max-Age: 1800
```

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.

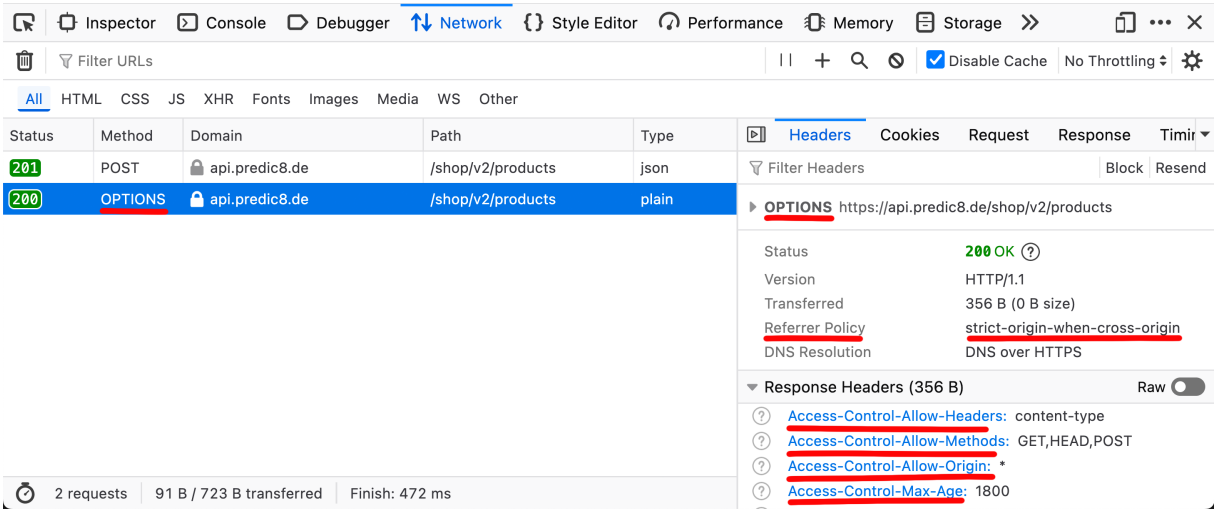


Image: Preflight request shown in the browser's developer tools

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 sometimes solve the issue at the gateway level as described in the next section.

20.5 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 sees and communicates with the gateway.

From the browser's point of view, everything comes from a single origin, so **no CORS restrictions apply**.

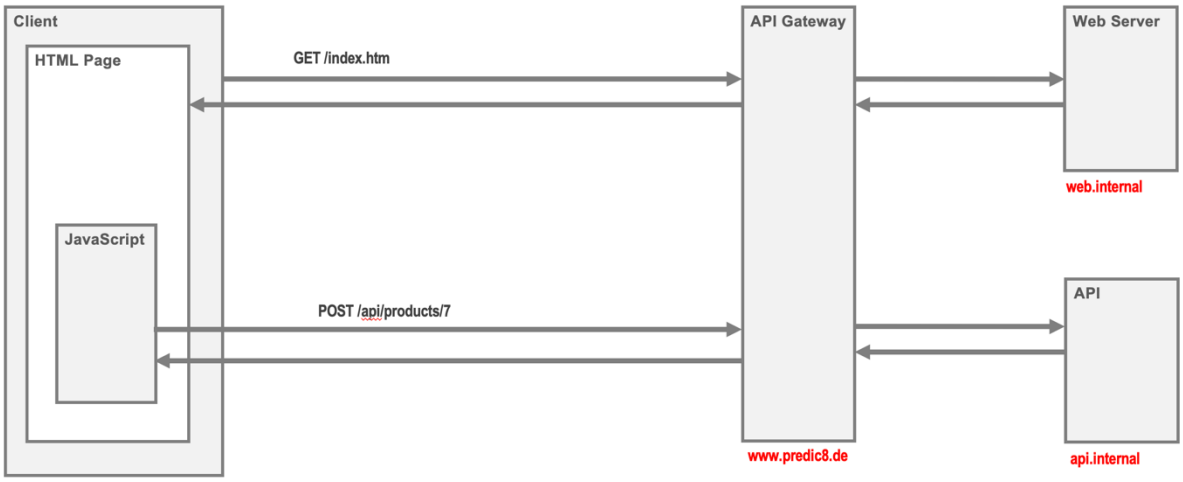


Image: Gateway setup that avoids CORS by hiding web server and API behind one address

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 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 offer built-in support for CORS, usually in the form of a **plugin** or **policy**.

These components 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 that provides the API. All you need to do is configure the gateway with the rules you want to allow, such as permitted origins, methods, or headers.

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
    }
  }
}
```

💡 **Sidenote: Backend stays clean**
Because the gateway handles all the CORS logic, your backend services can stay focused on core functionality without worrying about browser quirks or cross-origin headers.

Resources

Cross-Origin Resource Sharing (CORS), @mozilla.org
https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CORS

21 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.

21.1 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).

21.2 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 more intelligently and adapt to changing conditions, though at the cost of added complexity.

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.

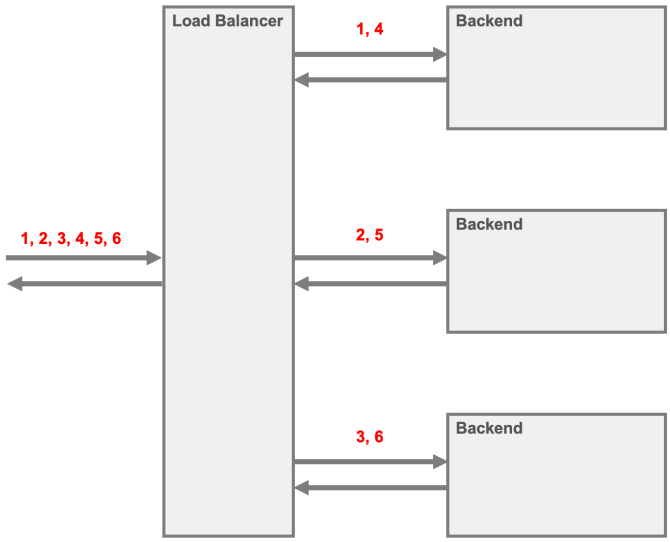


Image: Round Robin distribution of requests across servers

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 used without being overloaded.

Although Round Robin feels almost trivial, it does involve keeping track of a counter that identifies the next node. That counter must be shared between requests. Even though it's only a counter, sharing and synchronizing access to it comes at a cost. In high-performance environments, this overhead is small but worth noting. To avoid it, some systems use a simpler approach: Random Robin.

Random Robin

Random Robin is even simpler than Round Robin. Instead of keeping a counter, it rolls the dice with a random number to pick the next server. Since it doesn't maintain any state, it's lightweight and robust.

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.

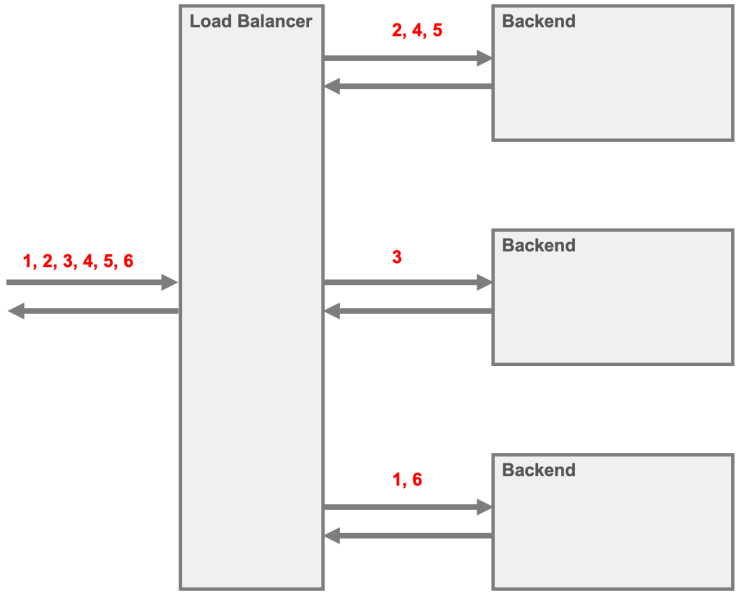


Image: Random distribution of requests

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 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.

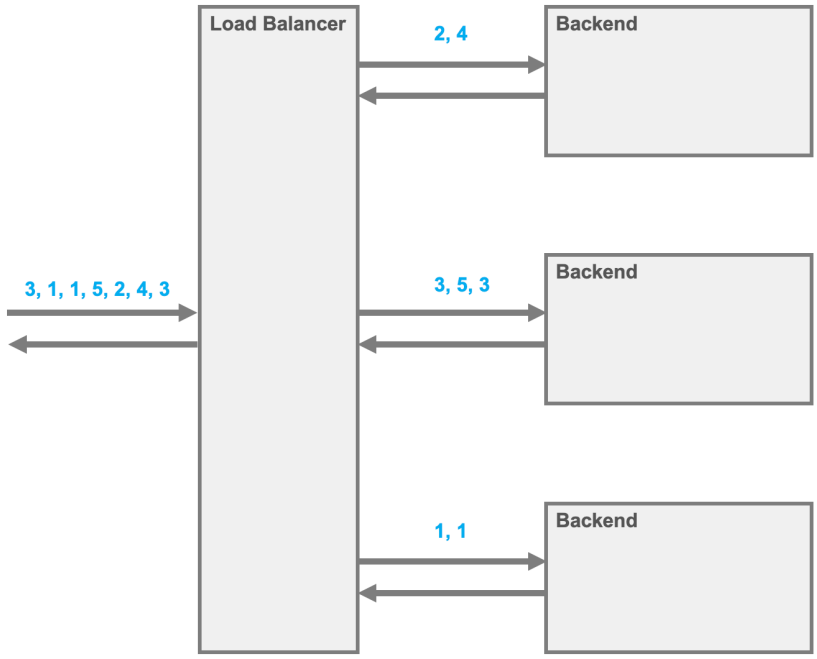


Image: Sticky sessions based on IDs

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.

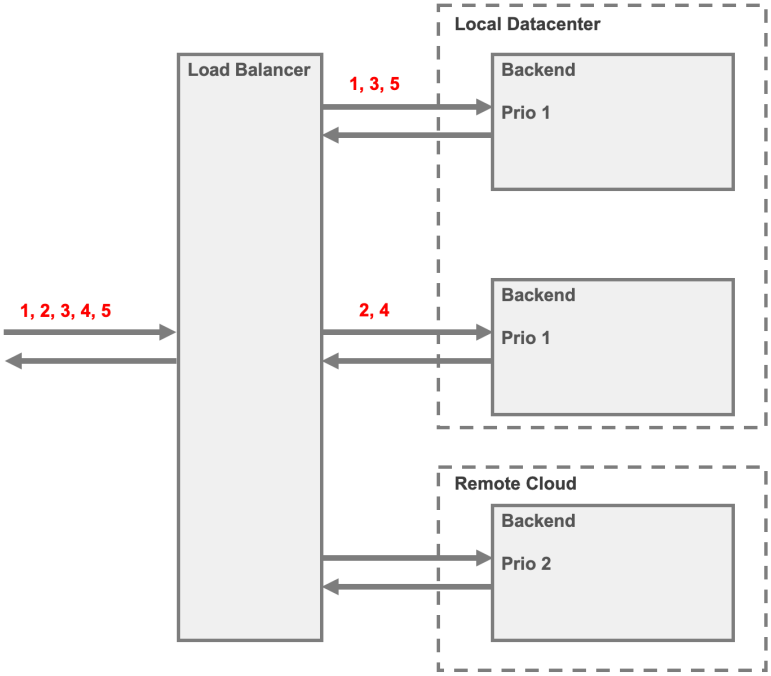


Image: Using priorities to route requests to backends in a local data center.

If those local servers go down, the balancer simply switches to the next available priority. In the second illustration, all local servers have problems, so the requests are rerouted to the priority 2 backends in the remote cloud.

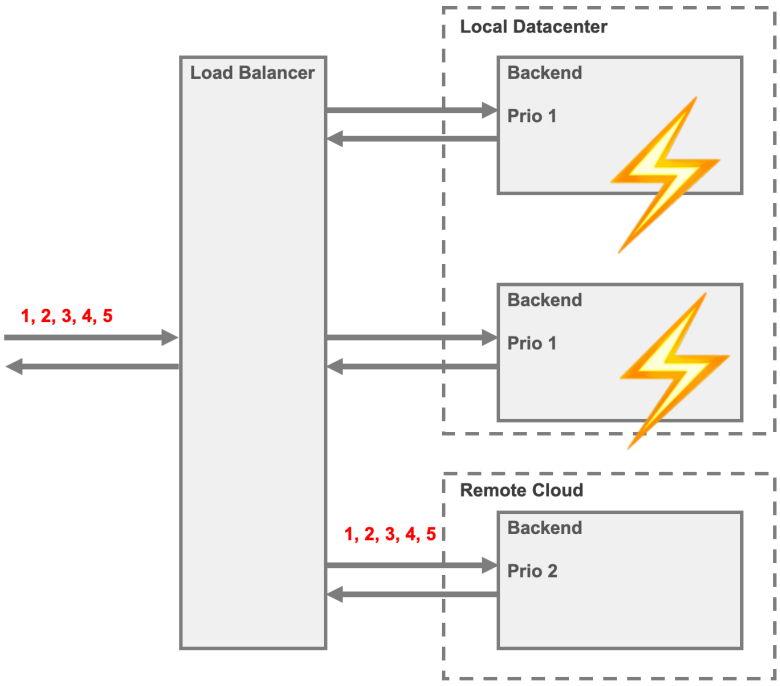


Image: Requests rerouted to remote cloud with lower priority

21.3 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.

Of course, dynamic balancers come with added complexity, and 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 build 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're especially valuable if downtime must be detected and avoided. For expensive resources (like AI models), dynamic balancing can help squeeze the most out of each server by optimizing performance and utilization, though at the cost of added complexity.

21.4 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 distribution *before* client calls hit them. But health checks themselves generate traffic. Often the volume of health checks exceeds the actual client request volume.

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.

No additional traffic is needed. The statistics come “for free” from normal exchanges. 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.

21.5 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 at least 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 and responding, 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 some retry logic, and even your browser quietly retries certain requests without you 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.

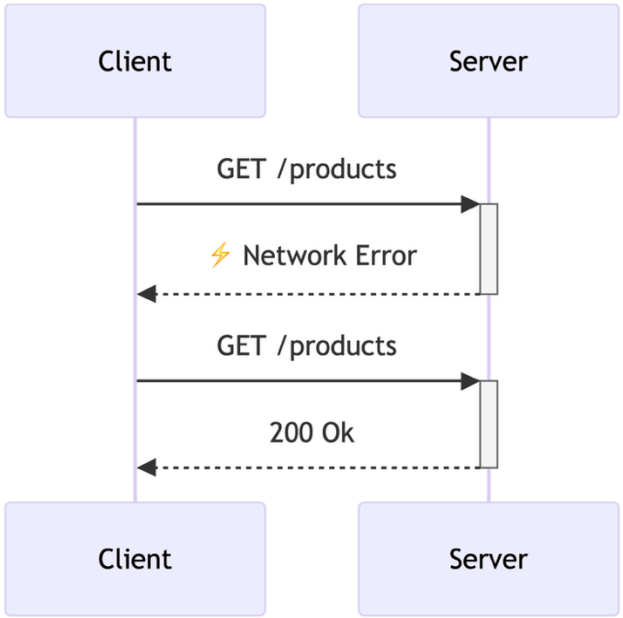


Diagram: Client retrying a request after a network error

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 retry automatically 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.

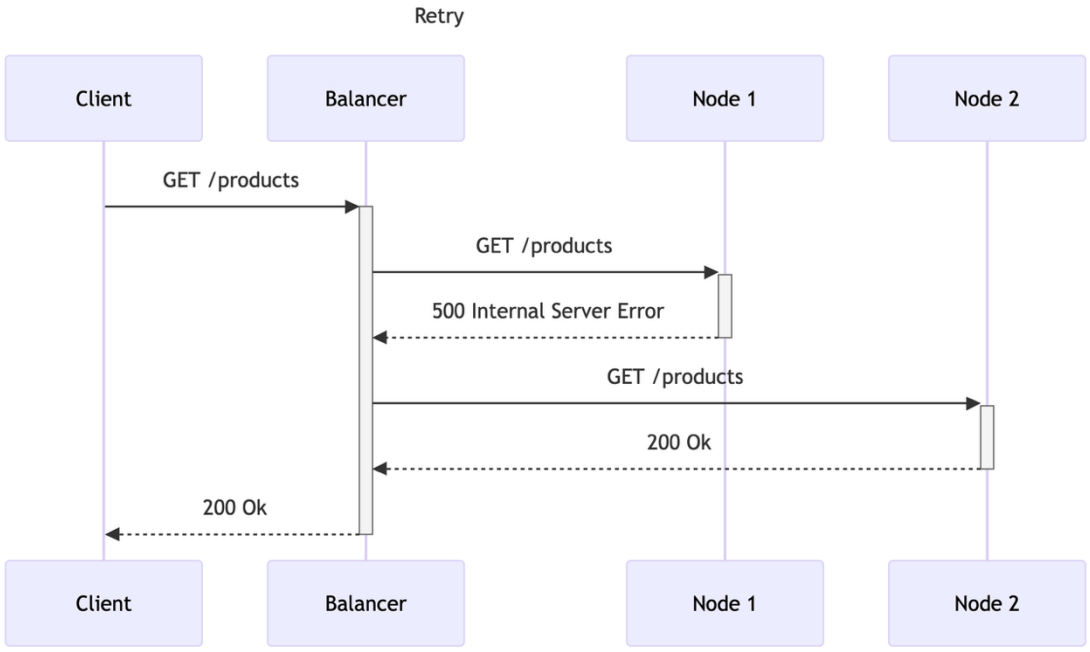


Image: Balancer retries failed request on second node

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 state of the server is no longer the same as before the call. But if the exact same request is repeated, the outcome does not change further. In other words, the repetition has no additional effect.

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 does not introduce new side effects. 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:

```
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 almost never worth retrying, because they indicate user or client errors. A *404 Not Found* or *405 Method Not Allowed* will return the same result no matter how many times you repeat the request.

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. But 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 server 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

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 absolute certainty. 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 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.

That is why understanding the exact meaning of network-related error codes is 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 is safe to retry even 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	Is Retry safe?
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 client.	Yes

If there is even the slightest doubt that 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 aware of common network errors and know how to react to them. In most 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.

21.6 Single Point of Failure

High availability means avoiding a single point of failure. Instead of relying on a single backend server, you usually have at least two or even more nodes. A load balancer can then distribute requests to a healthy server.

But what if the load balancer itself goes down? In that case, it becomes the weakest link. To avoid this situation, you can take different approaches.

DNS Load Balancing

The Domain Name System (DNS) can be used to achieve load balancing directly 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 simply 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. It's no surprise that many of the largest websites on the Internet, including Apple, Google, and Cloudflare rely on DNS-based balancing.

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 means DNS is central not just for big websites, but also for container orchestration at scale.

Anycast Routing

The next approach, **anycast routing**, also avoids a central balancer by sharing the same IP address by multiple servers. When a client connects, the Internet's routing infrastructure automatically directs the request to the server that is closest in network terms. Much like DNS load balancing, anycast eliminates a single point of failure and distributes traffic naturally across multiple endpoints.

The setup, however, is more complex and comes with some caveats. Connections may break if the advertised IP address shifts during an active session. Because the same IP is served by multiple nodes, TLS keys must be shared consistently across all endpoints. And since each request can end up at a different server, maintaining session stickiness or stateful authentication is more difficult.

Despite these challenges, anycast is widely used and supported directly by major cloud providers. Services such as *AWS Global Accelerator*, *Google Cloud Platform's Global Load Balancer*, *Azure Front Door*, and *Cloudflare* all offer anycast-based load balancing.

Keep the Load Balancer Simple

If DNS load balancing or anycast are not an option, you may have to accept that the load balancer itself could become a single point of failure. Generally, anything that is complex and likely to fail should be made redundant, think of backend applications that depend on databases and external services. But components that are simple, unlikely to fail, and easy to recover can sometimes be left as a single instance. Load balancers and API Gateways often fall into this category.

A balancer running in a small, stateless container without a database can usually run for a long 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 if containers are not an option.

To minimize the impact of failures, monitoring is key. Detecting problems early and reacting quickly keeps downtime short and prevents the balancer from becoming a weak spot.

22 Performance

An API gateway sits between client and backend, adding an extra hop in the communication path. Naturally, this introduces some overhead. But how much does this really matter in practice?

22.1 Latency

Latency is the time it takes for a request to travel from the client to the server and back, including all the processing that happens in between. In the context of APIs, it's typically measured from the moment a client sends a request to the moment it receives the response.

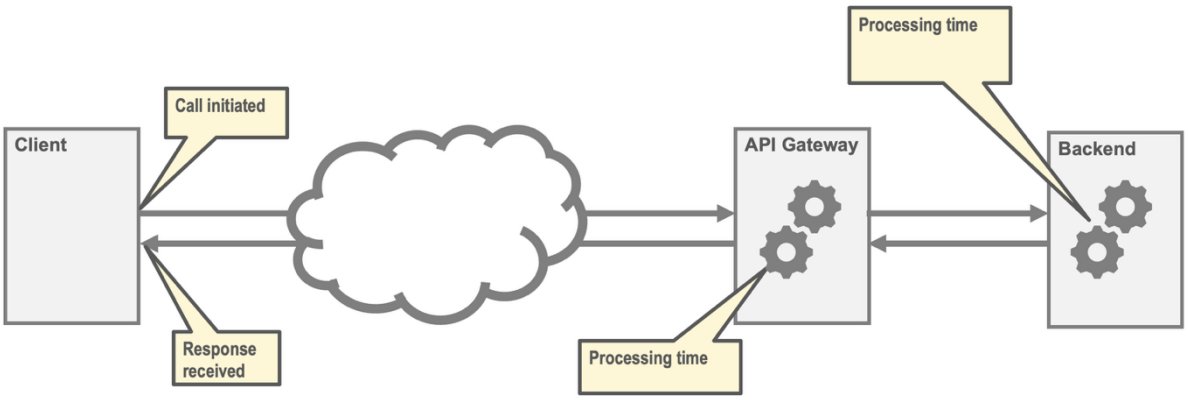


Image: Measuring the latency of an API call

Low latency is crucial for user-facing applications such as web UIs or mobile apps, where every millisecond affects perceived responsiveness.

An API Gateway introduces an additional hop between the client and the backend. Each request passes through several processing steps, each adding a small amount of delay:

1. Reading the request from the network socket
2. Decryption (if TLS is used)
3. Deserialization of the HTTP headers and payload
4. Route the request to the correct API and determine applicable plugins
5. Execution of global and route-specific plugins (e.g., validation, transformation)
6. Serialization of the response payload
7. Encryption before sending the response

These steps are performed **twice**: once for the incoming request and again for the outgoing response.

What's the Typical Overhead of an API Gateway?

For 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 typically remains in the **low-millisecond** range.

In most real-world systems, the **backend services** are the primary contributors to total response time not the gateway.

However, performance can degrade if the gateway is overloaded with **resource-intensive plugins**. Operations like deep payload inspection, full-schema validation, or complex transformations can significantly increase processing time.

Streaming Optimizations

Some gateways support **streaming**, where the request payload is piped directly to the backend without being fully deserialized bypassing steps like body parsing and serialization. This reduces latency significantly.

However, **streaming is only possible if no plugin needs access to the payload**. As soon as a plugin performs operations like JSONPath evaluation or schema validation, full message processing is required and streaming is disabled.

22.2 Bandwidth (Throughput)

How many requests can a gateway handle concurrently?

The **throughput** or number of requests per second (RPS) that a gateway can handle depends on several key factors:

- **Enabled plugins and processing tasks**
  Features like OpenAPI validation, JSON/XML transformations, or content filtering can significantly affect performance.
- **Message size and complexity**
  Small payloads under 1 KB are processed much faster than large, deeply nested XML documents hundreds of megabytes in size.
- **Applied security mechanisms**
  Token validation, rate limiting, and TLS encryption all add processing overhead.

Beyond configuration, **hardware and the gateway product itself** also play a role. Go-based gateways like **KrakenD** and **Tyk** advertise benchmarks over **80,000 RPS**. That's certainly impressive, but real-world performance can vary. For example, on a five-year-old laptop, I achieved **10,000 RPS** using a Java-based gateway. So, it's not just about the programming language.

In most cases, the engaged plugins have a far greater impact on performance than the core gateway engine itself.

22.3 Performance Tuning

Truth be told, performance tuning is rarely needed. If you really want to squeeze out a few extra RPS (requests per second) or reduce latency, here are some proven tips:

- **Disable or reduce logging.**
  Logging can add noticeable overhead, especially at high throughput.
- **Measure plugin performance.**
  Some plugins can be resource hungry. Identify the heavy hitters.
- **Use built-in metrics.**
  Most gateways expose valuable performance data.
- **Set up monitoring.**
  Tools like Prometheus and Grafana help you visualize and track performance bottlenecks.

If you're interested in getting the overall performance across multiple API Gateways and microservices, consider integrating **OpenTelemetry**. It provides distributed tracing so you can measure each segment of the communication path with precision.

Still hitting the ceiling? Then scale out. API Gateways are stateless, which makes it easy to add more instances to distribute the load evenly. (More on that in Section 0.)

Resources

KrakenD Homepage
https://www.krakend.io/

Tyk Performance Benchmarks
https://tyk.io/performance-benchmarks/

Part 2
API Gateways in Practice

This part gets hands-on. Real-world examples will bring key concepts and common scenarios to life, 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’s a lightweight, open-source tool that makes practical experimentation easy and transparent.

That said, the examples presented here aren't specific to Membrane. The patterns and configurations apply broadly and can be adapted to most modern API gateways. The goal is to understand the underlying principles—not to focus on a particular product.

23 Membrane API Gateway

Membrane API Gateway is a lightweight and flexible solution designed for simplicity and extensibility. Its clean configuration style and powerful feature set make it ideal for demonstrating a wide range of API gateway techniques.

It has been battle-tested in organizations of all sizes and across industries worldwide.

Membrane is released under the **Apache 2 open source license**, a permissive license that allows free use, modification, and distribution, even in commercial environments. With no licensing fees or usage restrictions, it's a practical choice for both learning and production deployments.

Interestingly, some commercial API gateways on the market are built on top of Membrane, underscoring its reliability and solid architectural foundation.

23.1 Installation and First Steps

This section walks you through the installation process to ensure you're ready to follow the upcoming examples. You'll learn how to install Membrane and to verify it's running properly.

Don't worry, installation usually takes less than 10 minutes.

Membrane is written in Java and offers multiple deployment options. In this section, we'll focus on two popular methods: running it with Docker and using the standalone Java distribution. For additional deployment options, see:
https://www.membrane-api.io/deployment/

23.1.1 Standalone Java Installation

If your system already has a Java runtime, or if you can install one, the standalone installation is an **excellent way to try Membrane** and explore its features. This method makes it easy to run the many examples included with the distribution. In production, you can later choose to run Membrane in a container if desired.

Step 1: Verify Your Java Installation

Ensure you have Java 21, or a newer version installed by running:

```
java -version
```

The output should resemble:

```
openjdk version "21.0.5" 2024-10-15 LTS
```

If you need to install Java, visit https://www.java.com/en/download/ and follow the provided instructions.

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 the Membrane folder and start the gateway:

```
cd membrane-api-gateway-6.1.0

./membrane.sh
```

or

```
cd membrane-api-gateway-6.1.0

membrane.cmd
```

on Windows.

After starting, Membrane displays information about the deployed APIs:

```
13:41:12,032  INFO 1 main ProxyInfo:31 {} - Started 5 APIs:
13:41:12,034  INFO 1 main ProxyInfo:33 {} -   API 0.0.0.0:2000 using
OpenAPI specifications:  - "fruitshop-v1-1" @ fruitshop-api.yml
13:41:12,034  INFO 1 main ProxyInfo:33 {} -   API 0.0.0.0:2000 /names
13:41:12,034  INFO 1 main ProxyInfo:33 {} -   API Groovy
13:41:12,034  INFO 1 main ProxyInfo:33 {} -   API 0.0.0.0:2000
13:41:12,035  INFO 1 main ProxyInfo:33 {} -   API Console
```

Step 4: Accessing an API

Open in your browser:

http://localhost:2000

You should see a JSON document returned. This data comes from the backend service at: https://api.predic8.de

In the next section, we'll take a closer look at the API configuration and explore how to customize the gateway's behavior.

💡 **Hint**: **No direct Internet Connection**
If this doesn't work, make sure your machine has direct Internet access. Some corporate networks or environments use proxies that block outbound connections.
If you're behind such a proxy or have no direct Internet access, check out the file `proxies-offline.xml` in the `conf` folder. It contains instructions on how to run Membrane offline.

23.1.2 Docker Installation

Even if Docker is part of your long-term plan, it's a good idea to begin with the full Membrane distribution running locally via Java, as described in the previous section. That makes it easier to explore the many included examples, most of which aren't containerized and are simpler to run in a local environment.

That said, if you prefer Docker, you can absolutely follow along using it. For production environments, Docker or Kubernetes is generally the preferred choice, and Membrane supports both out of the box.

To get started quickly with Docker, you can use this command:

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

However, to follow along with the examples in this book, we recommend using a setup that gives you access to and control over the `proxies.xml` configuration. Here's how to do that:

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: Starting a Membrane Docker Container

Open a terminal and navigate to the Membrane distribution directory:

```
cd membrane-api-gateway-6.1.0
```

Start the container, mounting the `proxies.xml` file from the `conf` folder. Make sure you are really in the `membrane-api-gateway-*` folder:

On macOS/Linux:

```
docker run -it -p 2000:2000 \
 -v "$(pwd)/conf/proxies.xml:/opt/membrane/conf/proxies.xml" \
  predic8/membrane
```

On Windows (PowerShell or CMD):

```
docker run -it -v -p 2000:2000
${PWD}\conf\proxies.xml:/opt/membrane/conf/proxies.xml
predic8/membrane
```

Troubleshooting

📌 **Note: PWD**
If *`${PWD}`* doesn't work in PowerShell, replace it with the full path manually, e.g.
`C:\Users\YourName\Downloads\membrane-api-gateway-6.0.1\conf\proxies.xml`

🔧 **Right Directory**
Check if `conf/proxies.xml` is reachable

Step 3: Testing the Installation

Open in the browser:

http://localhost:2000

You should see a JSON document like this:

```
{
  "apis": [
    {
      "name": "Shop API Showcase",
      "description":"API for REST exploration, test and
demonstration. Feel free to manipulate the resources using the
POST, PUT and DELETE methods. This API acts as a showcase for
REST API design.",
      "url":"/shop/v2/"
    }
  ]
}
```

💡 **Hint:** If the browser shows a connection error, make sure Docker is running and you have a working Internet connection.

24 API Configuration

Membrane's behavior is configured in the `proxies.xml` file located in the `conf` folder. This is an XML configuration file based on the **Spring Framework**, which gives Membrane much of its flexibility and extensibility.

While there is experimental support for YAML and Kubernetes Custom Resource Definitions (CRDs), we recommend sticking with the XML format, especially when working through the many ready-to-use examples included in the distribution.

You can edit the configuration with any text editor, but using tools like **IntelliJ** or **Visual Studio Code** is strongly recommended. These editors offer helpful XML features such as:

- Auto-completion (Ctrl+Space)
- Syntax highlighting
- Inline documentation based on the XML Schema declared at the top of the file

These features make editing more efficient and significantly reduce the chance of configuration errors.

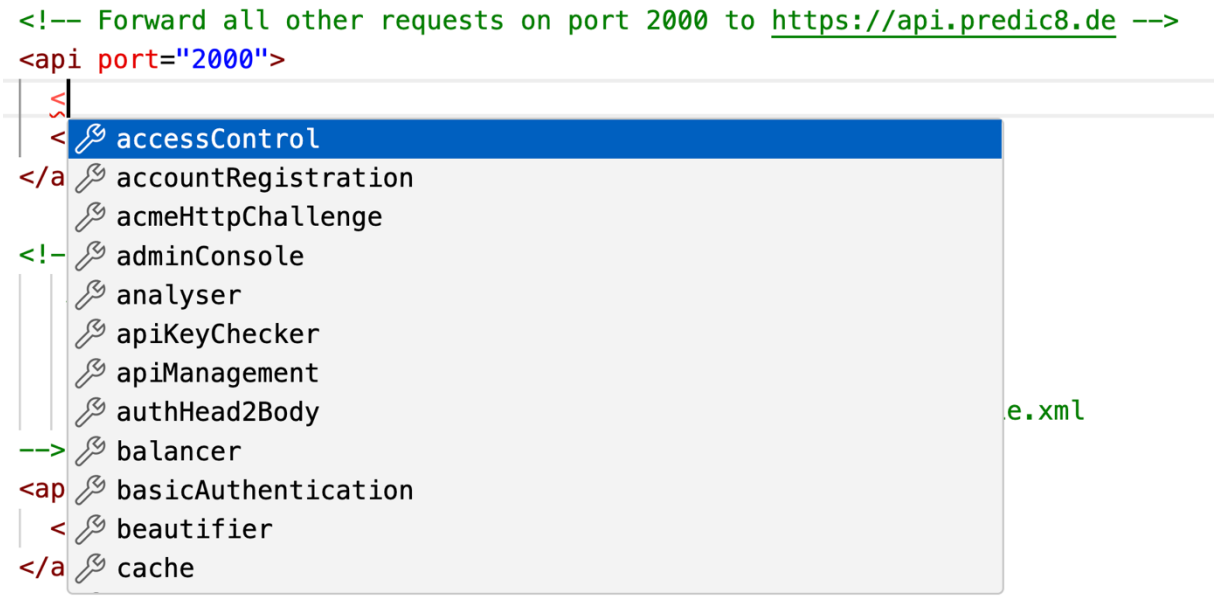


Image: Help from Autocompletion

First API Configuration

The following steps show how to extend a basic API configuration to enable logging of requests and responses.

Step 1: Open proxies.xml

Navigate to the `conf` subfolder in the Membrane distribution and open the `proxies.xml` file in your preferred editor.

Step 2: Add Logging

Locate this section in the file:

```
<api port="2000">
  <target url="https://api.predic8.de"/>
</api>
```

Now insert a `<log/>` element before the `<target>` so the configuration looks like this:

```
<api port="2000">
  <log/>
  <target url="https://api.predic8.de"/>
</api>
```

This tells Membrane to log HTTP request and response information for all traffic handled by this API.

Step 3: Save and Reload

Save the file. Membrane automatically detects changes to `proxies.xml` and reloads the configuration.

If something doesn't seem to work, check the terminal output for any log statements or error messages. These often provide helpful clues for resolving configuration issues.

Step 4: Test the Change

Open your browser and visit:

http://localhost:2000

Then check the terminal window. You should now see detailed logs for the incoming request and the corresponding response, confirming that logging is working.

Sidenote: Configuration Hot Reloading
Membrane automatically reloads the configuration whenever changes to `proxies.xml` are detected. However, the reload may be delayed while traffic is active to prevent breaking in-progress requests. If that happens, you can stop Membrane manually by pressing `Ctrl+C` in the terminal and then restart it. Your updated configuration will be applied on restart.

This feature can be turned off by setting `<router hotDeploy="false">` in the `proxies.xml` file.

24.1 Configuration Errors

If your `proxies.xml` file contains an error, Membrane will report it during startup or when the configuration is reloaded. These messages appear in the terminal or log output and can help pinpoint issues.

While Membrane aims to provide helpful diagnostics, some low-level XML parsing errors, especially those from the underlying XML parser, can be difficult to decipher.

Fortunately, most modern editors like IntelliJ IDEA or Visual Studio Code offer built-in XML support. Errors are highlighted directly in the editor, often with tooltips explaining the problem. This can save you time and frustration when editing the configuration.

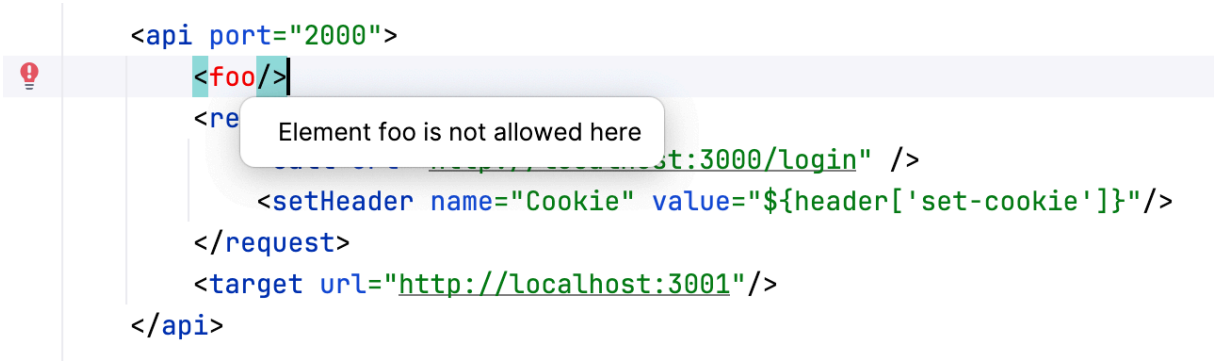


Image: XML Validation Error displayed in Editor

25 Routing Traffic

Routing is one of the core responsibilities of an API Gateway. It ensures that incoming client requests are forwarded to the correct backend service. This is typically done through API definitions that match certain criteria, such as HTTP methods or request paths.

The example below shows how to configure Membrane to route traffic. In this case, the gateway listens for `GET` requests on port `2000` that begin with the path `/shop/v2`, and forwards them to the backend host `api.predic8.de`:

```
<api port="2000" method="GET">
  <path>/shop/v2</path>
  <target url="https://api.predic8.de"/>
</api>
```

You can try this setup yourself. Once the configuration is in place, test the route using the command line:

```
curl http://localhost:2000/shop/v2/
```

Alternatively, use the **REST Client** plugin in **Visual Studio Code** to send and inspect the request.

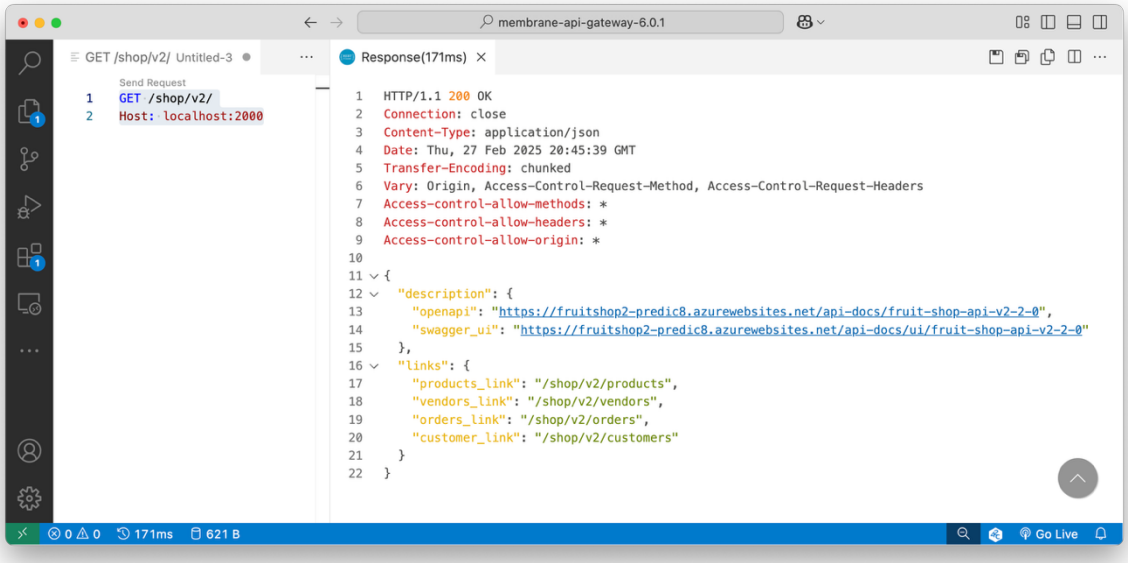


Image: Using Visual Studio Code as API client using the **REST Client** plugin

25.1 Sequence of API Matching

API Gateways often manage dozens, sometimes hundreds, of API definitions. When a request arrives, the gateway evaluates them in the order they appear. It stops at the first match and processes the request using that configuration.

Let's take a look at an example where two APIs are defined:

```
<api name="API 1" port="2000" method="GET">
  <path>/shop/v2</path>
  <log message="Method: ${method}"/>
  <target url="https://api.predic8.de"/>
</api>

<api name="API 2" port="2000">
  <path>/shop/v2</path>
  <log message="Method: ${method}"/>
  <target url="https://api.predic8.de"/>
</api>
```

Now let's test this setup step by step.

Case 1: Receiving a GET request

After this request:

```
GET /shop/v2/products
Host: localhost:2000
```

In the console output, you'll see something like this:

```
10:38:33,081  INFO 69 ... {api=API 1} - Method: GET
10:38:33,220  INFO 69 ... {api=API 1} - Method: GET
```

Note that **API 1** was selected.

Why two entries? Because the `<log>` element is executed twice, once during the request phase and once during the response phase.

Case 2: Receiving a POST request

Let's look at a POST request:

```
POST /shop/v2/products
Host: localhost:2000
Content-Type: application/json

{
  "name": "Biscuits", "price": 1.99
}
```

Here's what happens:

- **API 1** is skipped because it only allows `GET`.
- **API 2** has no method restriction, so it matches and handles the request.

The log output confirms this:

```
... {api=API 2} - Method: POST
```

Case 3: Receiving a request to an unknown path

A request with an unknown path:

```
GET /nirvana HTTP/1.1
Host: localhost:2000
```

Neither API definition matches the path, so Membrane returns: **404 Not found**.

Default API for Unmatched Requests

In some cases, it's helpful to define a fallback for requests that don't match any API definition. You can configure a default API at the end of your `proxies.xml` file to catch all unmatched requests.

Here's an example:

```
<api port="2000">
  <response>
    <static>My own not found message!</static>
  </response>
  <return statusCode="404"/>
</api>
```

With this setup, any request on port 2000 that doesn't match a more specific API will receive a `404 Not Found` response along with your custom message.

Tip: **Custom Not Found**
Place a fallback API **after** all other definitions providing a custom error message.

25.2 Routing Criteria

API Gateways can base routing decisions on almost any part of a client's request, not just the path, method, or port. Common routing criteria include:

Criteria	Description
Port	Port on which the request was received.
IP	IP address of the client.
HTTP 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 header fields
Content	The content or payload of the request.

Content based Routing

Even the payload itself can influence routing. For example, an API Gateway can inspect the message body to decide how to handle a request.

Here's an example using a condition on a JSON field in the request body:

```
<api port="2000"
     test="json['name'] == 'Lolly'"
     language="SPEL">
  <path>/shop/v2</path>
  <log message="It is a lolly!"/>
  <target url="https://api.predic8.de"/>
</api>
```

Try this API with:

```
POST /shop/v2/products
Host: localhost:2000
Content-Type: application/json

{
  "name": "Lolly"
}
```

and have a look at the log output.

💡**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 about expression languages for details.

25.3 URI Templates

URI templates**,** common in OpenAPI definitions and many REST frameworks, allow flexible path definitions using placeholders that capture values directly from the request path.

For example, the following template:

`/products/{pid}`

matches a request like:

`/products/7`

The value `7` is extracted and assigned to the placeholder `pid`, which is accessible through the variable `pathParam`.

In API gateways, URI templates can be used as **routing criteria**. Here's a configuration example that routes requests based on the structure of the incoming path:

```
<api port="2000">
  <path>/products/{pid}</path>
  <log message="Product: ${pathParam.pid}"/>
  <return/>
</api>

<api port="2000">
  <path>/products</path>
  <log message="List of products"/>
  <return/>
</api>

<api port="2000">
  <path>/customers/{cid}</path>
  <log message="Customer: ${pathParam.cid}"/>
  <return/>
</api>
```

With this setup, incoming requests are handled as follows:

- **GET** http://localhost:2000/products
  Logs: "List of products."
- **GET** http://localhost:2000/products/7
  Logs "Product: 7"
- **GET** http://localhost:2000/customers/14
  Logs "Customer: 14"

25.4 Naming APIs

Clear and descriptive names make it much easier to keep track of your APIs—especially once your gateway is handling dozens or even hundreds of them.

If you don't assign a name, Membrane will generate one automatically based on routing parameters. For example:

```
0.0.0.0:2000 /products/{pid}
```

While this works, it's not exactly easy to read or remember. A better way is to use the `name` attribute to give your API a meaningful label:

```
<api port="2000" name="Fruitshop">
  <path>/shop/v2</path>
  <target url="https://api.predic8.de"/>
</api>
```

These names appear in the **logs, monitoring dashboards**, and the **Membrane web console**:

```
13:11:13 INFO 5 Log:148 {api=Fruitshop} - Path: /shop/v2
```

Having named APIs makes debugging and monitoring a whole lot easier. Instead of trying to decipher cryptic port and path combos, you'll see exactly which API handled the request.

26 Message and Exchange Objects

Most API gateways wrap an incoming request and its corresponding response into a shared structure called an **exchange**. This exchange object acts as the central hub where the gateway and its plugins read, inspect, and manipulate both the request and the response.

This design streamlines processing and enables features like authentication, rate limiting, and transformation, all using the same unified object.

When a request arrives, the gateway wraps it in an exchange and sends it along the **request flow**. At that point, the exchange only holds the request, but there's no response yet. Once the backend replies, the response is added to the same exchange, and processing continues along the **response flow**. Now, both request and response are available for logging, modification, or policy enforcement.

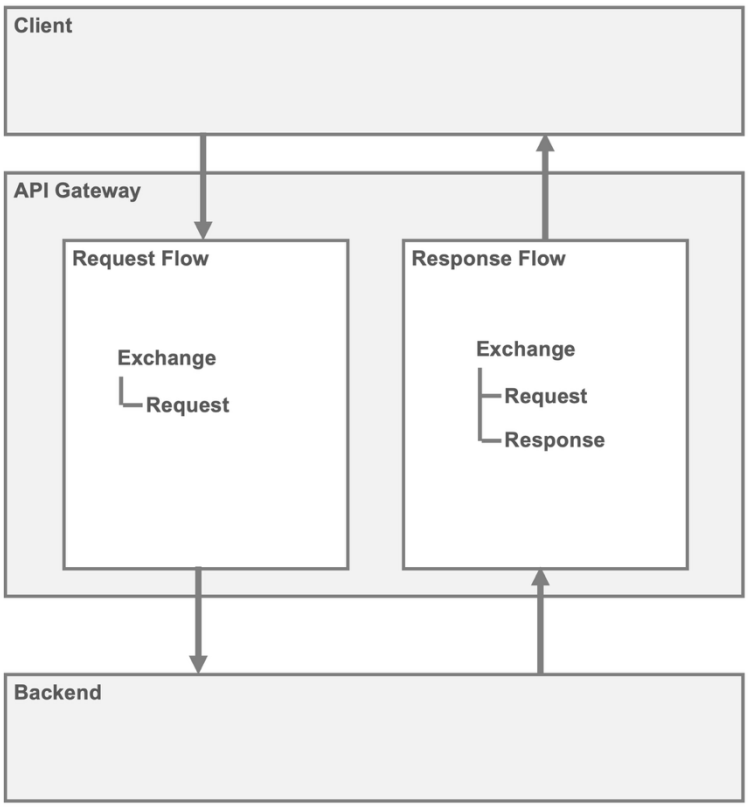


Image: Structure of an exchange object in the request and response flows

The **request** and **response** parts of the exchange look similar:

- Both contain HTTP headers
- Both can carry a body (e.g. JSON, XML, binary)

However, they also differ:

- The request includes the HTTP **method** and the **path**
- The response adds the HTTP **status code**

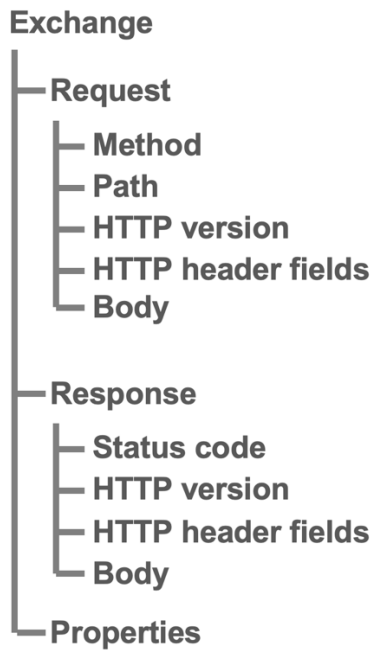


Image: Structure of the exchange object

But there's more: the 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 might store the result of an authentication check so that later plugins can use that for permission checks. Others may use properties for dynamic routing or tracking custom metrics.

In the next section, we'll explore how exchange properties enable multiple plugins to work together seamlessly and efficiently.

26.1 Message Properties

API configurations often apply multiple plugins in sequence, each performing its own task. For instance, one plugin might extract a value from the incoming request's query string, while a subsequent plugin uses that value to build a custom JSON message. This chain of operations requires a way to share and persist data across stages, which is where *message properties* (also known as *exchange properties* or *variables*) come into play.

Consider the following two APIs. The first API accepts a query string parameter named **color** and returns an HTTP header after calling a backend. The second API just serves as a simple mock backend, always returning the same static response.

```
<api port="2000">
  <request>
    <setProperty name="color" value="${param.color}"/>
  </request>
  <response>
    <setHeader name="X-Color" value="${property.color}"/>
  </response>
  <target host="localhost" port="2001"/>
</api>

<api port="2001">
  <response>
    <static>Hi!</static>
  </response>
  <return/>
</api>
```

Now call the first API:

```
GET http://localhost:2000?color=red
```

During the request flow the query parameter `color` is read and stored in an exchange property with the same name. The gateway then invokes the backend API. After receiving the response, the gateway can still access the `color` property and insert it into the `X-Color` response header. The final response to the client looks like this:

```
HTTP/1.1 200 Ok
Content-Type: text/plain
X-Color: red
```

Without storing the color value in the exchange, it wouldn't be possible to access it in the response flow. Exchange properties are what connect the request and response flows.

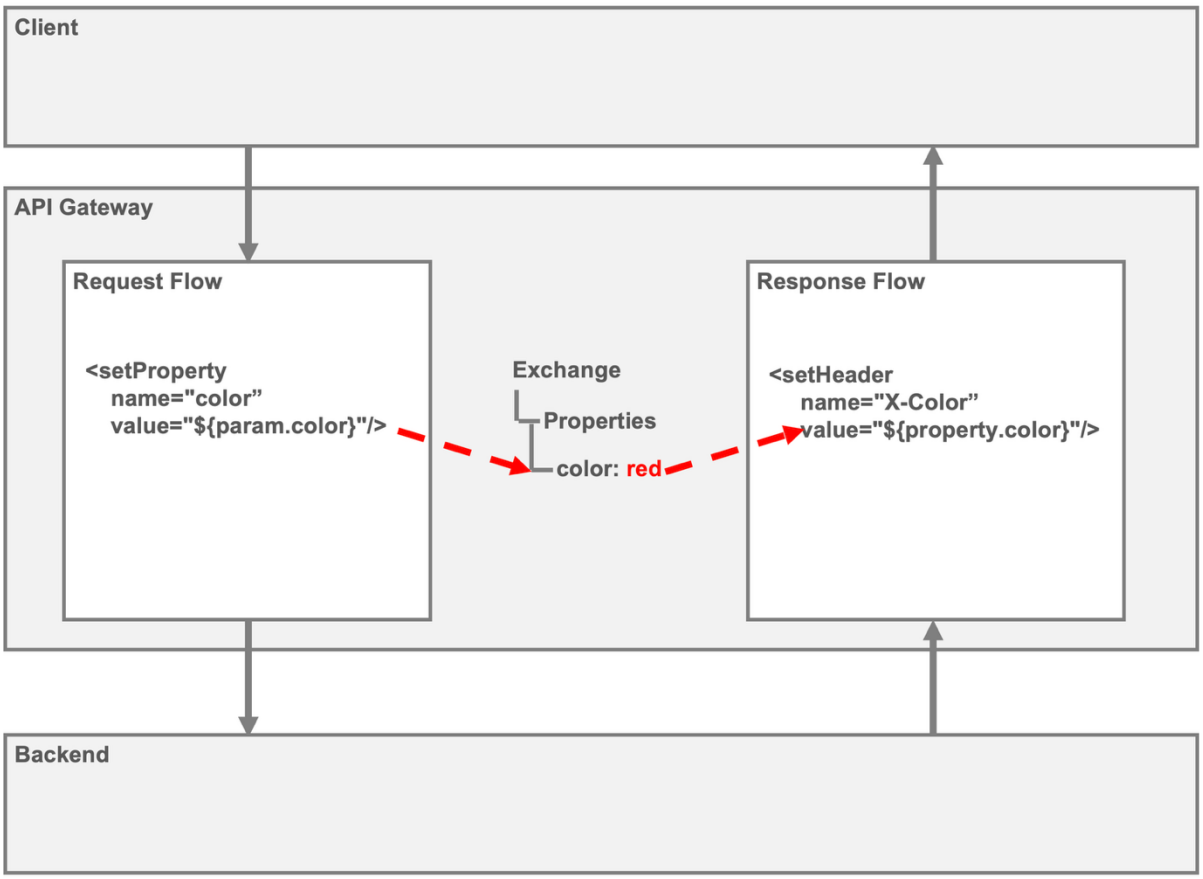


Image: Using properties to share values between request and response flow

Message properties are key-value pairs stored in the exchange context. They allow plugins to remain loosely coupled while still communicating with one another. This makes it possible to build sophisticated behavior across an API flow.

26.2 Short Circuit Responses

In some cases, you may want an API to return a response immediately, without calling a backend service. This is especially useful for testing, mocking, and error handling. You'll see this pattern frequently throughout this book.

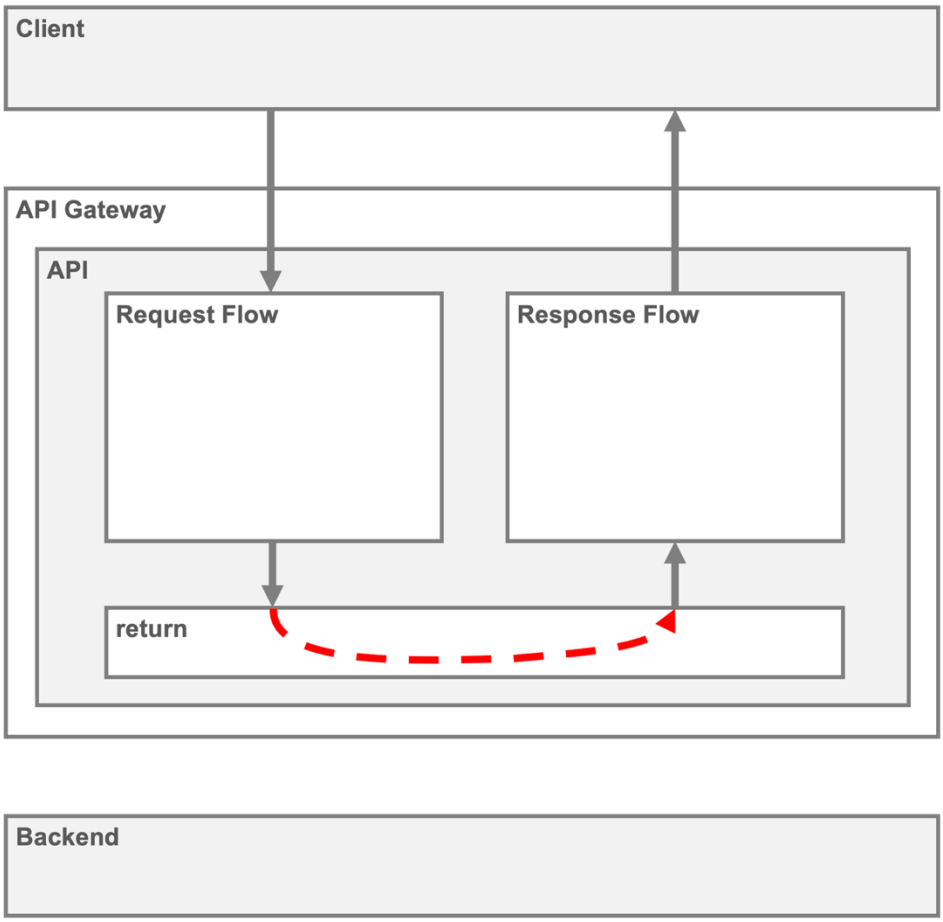


Image: API returning without routing to a backend

Best Practice: Handling 404 Not Found

A common use case for short circuiting is to define a **catch-all** API that returns a `404 Not Found` status. We've seen this pattern earlier in the routing chapter. By placing this API at the end of a configuration, you ensure it only applies when no other API matched the request.

```
<api port="2000">
  <response>
    <static>My personal not found message!</static>
  </response>
  <return statusCode="404"/>
</api>
```

Best Practice: Temporarily Blocking Access to Resources

Sometimes you need to block access to an endpoint of your API, for example, during maintenance or a rollout freeze. You can use a static response to reject specific requests without touching the backend or the routing logic.

Here's how to block all `POST` requests to `/products`:

```
<api port="2000" method="POST">
  <path>/products</path>
  <response>
    <static contentType="application/json">
      {
        "message": "Temporarily unavailable"
      }
    </static>
  </response>
  <return statusCode="503"/>
</api>
```

This approach is useful for maintenance windows or temporary lockouts without having to disable or reconfigure entire services.

27 OpenAPI

In this chapter, we'll explore how to put OpenAPI documents to practical use with your API Gateway. You'll learn how to:

- Set up APIs automatically using OpenAPI as configuration
- Let the gateway rewrite addresses inside OpenAPI documents
- Validate requests and responses against OpenAPI definitions
- Establish APIOps best practices using OpenAPI as a central source of truth

These features help streamline your workflows, improve reliability, and reduce manual effort when managing APIs at scale.

📖 **Background First?**
This chapter builds on the general concepts from Chapter 6: OpenAPI. If you're new to OpenAPI or want a vendor-neutral overview, we recommend starting there before diving into the Membrane-specific examples.

27.1 Gateway Configuration Using OpenAPI

OpenAPI documents define your API's structure, paths, methods, parameters, authentication, and more. But beyond serving as documentation, these specs can be used directly to configure API gateways. This keeps the actual runtime behavior aligned with what's declared in the spec, reducing errors and simplifying setup.

Take the following example. The gateway configuration is almost entirely driven by an OpenAPI file. All you need to add is the port:

```
<api port="2000">
  <openapi location="fruitshop-api.yml"/>
</api>
```

Once deployed, you can open a generated overview page at:

```
http://localhost:2000/api-docs
```

The gateway reads the OpenAPI file, extracts the title, version, and available paths, and presents them in a compact overview.

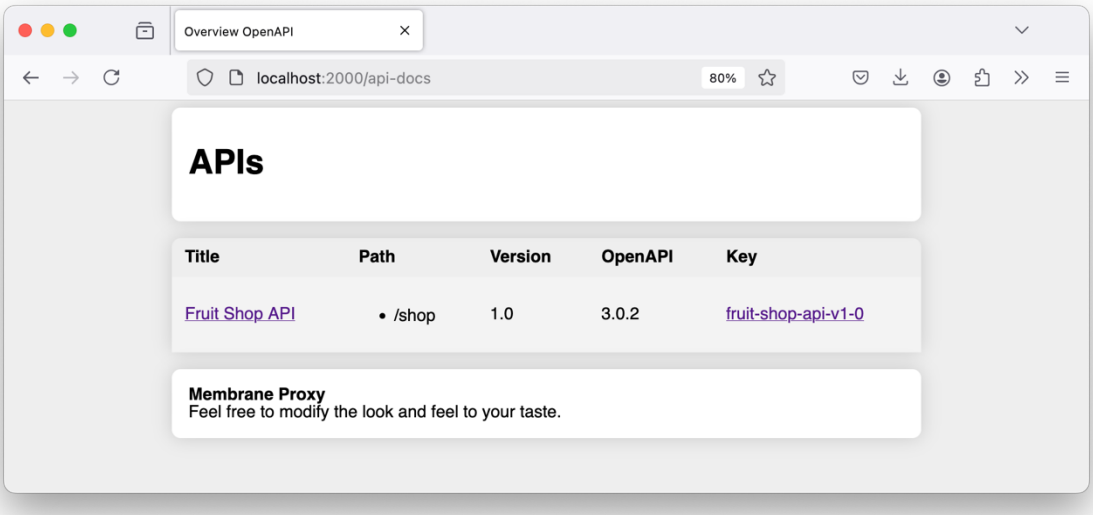


Image: Overview of APIs deployed from OpenAPI

From here, developers can either download the OpenAPI document or launch **Swagger UI** to explore and test the API interactively.

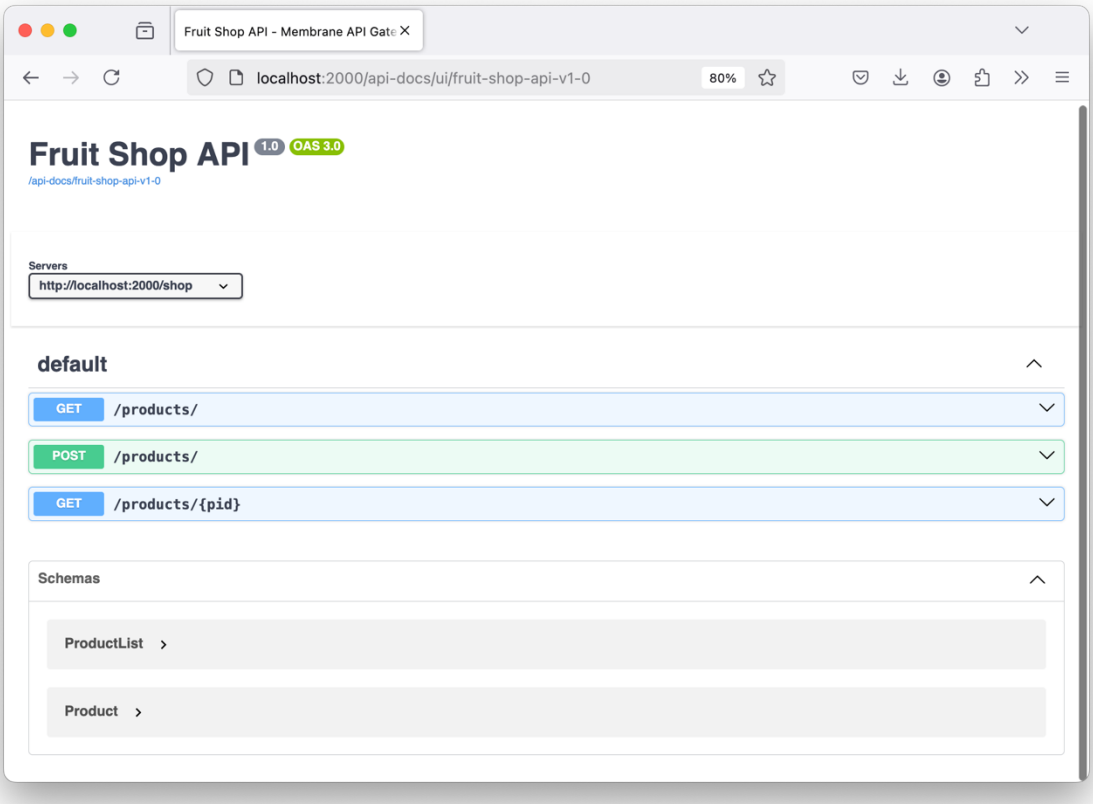


Image: Swagger UI with deployed OpenAPI

Loading Multiple APIs

You can of course configure each OpenAPI individually, but if you're exposing dozens of APIs, there's a simpler way:

```
<openapi dir="conf"/>
```

This will scan the `conf` directory and automatically load every OpenAPI file it finds.

Referencing Remote Documents

Membrane can also fetch OpenAPI specs from remote locations. Just provide a URL:

```
<openapi location="https://api.predic8.de/api-docs/fruit-shop-api-v2-2-0"/>
```

This is especially useful when storing specs in a central repository or a version-controlled web server. Once the document is updated, you can roll out the new configuration simply by restarting the gateway.

> 💡 **Sidenote: Declarative Configuration**
> Declarative configurations are a cornerstone of modern API ops. When the spec defines both your documentation *and* your runtime behavior, it's easier to ensure consistency, validate changes, and automate deployments.

27.2 Configuring OpenAPI Rewriting

As described in Part I, Chapter 6.2 about OpenAPI URL Rewriting, the `servers` field needs to reflect the public address of the API, not the internal address of the backend. Otherwise, clients generated from that OpenAPI might bypass the gateway entirely.

By default, the API Gateway rewrites the `servers` section automatically. It takes the **protocol**, **hostname**, and **port** from the incoming request and uses them to replace the values in the OpenAPI document before serving it.

Take a look at how the rewritten URLs depend on the incoming request:

Request	Rewritten URL in the OpenAPI
http://localhost:80/shop.yml	http://localhost:80/shop/v2
http://127.0.0.1/shop.yml	http://127.0.0.1/shop/v2

This behavior works well in development environments where internal and external addresses are often the same.

But things change once you move into production.

In real-world setups, gateways usually sit behind firewalls, inside private networks, or tucked away in containers. The gateway address visible to external clients is usually different from the gateway's internal address.

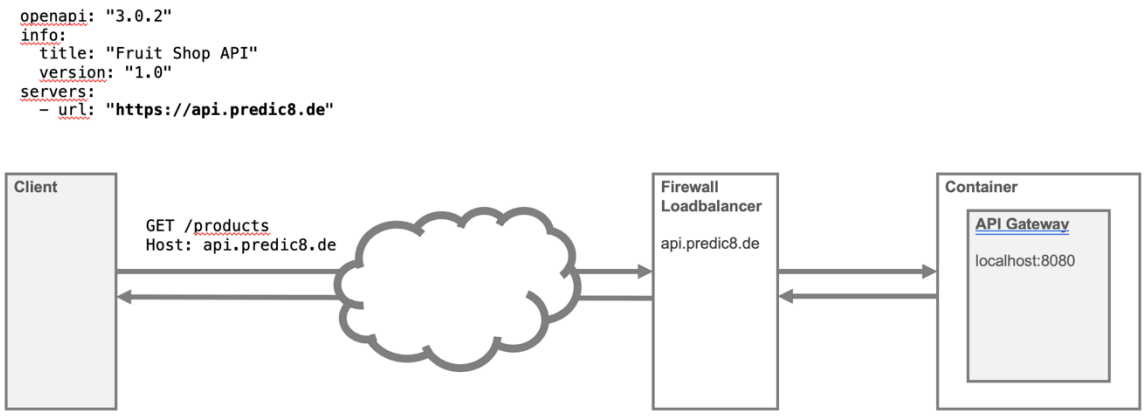


Image: External and Internal address of an API

To ensure external clients get the correct **public-facing address**, you have to override the rewrite behavior with explicit parameters:

```
<api port="2000">
  <openapi location="fruitshop-api.yml">
    <rewrite host="api.predic8.de"
             protocol="https"
             port="443"/>
  </openapi>
</api>
```

This configuration tells the gateway to serve an OpenAPI document with the correct public entry point:

```
servers:
  - url: https://api.predic8.de/shop/v2
```

💡 **Sidenote**: Why not just serve a manually updated OpenAPI?
You could, but that introduces manual maintenance. Any time the backend API changes, like when a new endpoint is added, you'd have to update both the backend and the OpenAPI stored at the gateway. With dynamic rewriting, the gateway can fetch the current OpenAPI from the backend and simply replace the address. You get accurate, up-to-date specs with minimal effort and no duplication.

27.3 OpenAPI Message Validation

Membrane supports request and response validation based on OpenAPI specifications, but it's turned off by default. You can activate request validation by adding the `validateRequests` attribute to the API definition:

```
<api port="2000">
  <openapi location="fruitshop-api.yml"
           validateRequests="yes"/>
</api>
```

Sidenote: Why OpenAPI Validation is disabled by default
Validation isn't turned off because of performance concerns. The overhead for typical requests is minimal. Instead, it's disabled by default to avoid unexpected behavior. Once request validation is active, response validation often makes sense too. But this can lead to surprises, especially when error responses from the backend don't match the OpenAPI spec and get blocked by the gateway. Keeping validation off by default ensures a smoother experience until you're ready to enable both directions intentionally.

Once enabled, requests that match the OpenAPI contract will pass through without issue. For example:

```
POST /shop/v2/products
Host: localhost:2000
Content-Type: application/json

{
  "name": "Figs",
  "price": 2.7
}
```

In the **Fruitshop's** OpenAPI document, the `price` field is defined as a non-negative number:

```
price:
  type: number
  minimum: 0
```

If someone sends a value that violates this rule, say, a negative price, the gateway rejects the request and responds with 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": {
    "method": "POST",
    "uriTemplate": "/products",
    "path": "/shop/v2/products",
    "errors": {
      "REQUEST/BODY#/price": [
        {
          "message": "-10 is smaller than the minimum of 0",
          "complexType": "Product",
          "schemaType": "number"
        }
      ]
    }
  }
}
```

Membrane uses the **Problem Details for HTTP APIs** format to return validation errors. This standardized format includes structured fields that make it easier for clients to 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'd prefer to keep error messages more generic, you can suppress the detailed output using the `validationDetails` attribute:

```
<openapi location="fruitshop-api.yml"
          validateRequests="yes"
          validationDetails="no"/>
```

Response Validation

You can also enable **response validation**, which verifies that backend responses conform to the contract as well:

```
<openapi location="fruitshop-api.yml"
         validateRequests="yes"
         validateResponses="yes"/>
```

While request validation is commonly used, response validation is often overlooked. Still, it’s just as important for catching bugs, improving client compatibility, and **preventing accidental data leaks**.

We'll dive deeper into response validation in the security-focused chapters later in the book.

27.4 APIOps with OpenAPI

APIOps brings DevOps thinking into the world of APIs. It’s all about applying automation, testing, and repeatable processes to the entire API lifecycle, from design and development to deployment and monitoring.

By adopting APIOps practices, teams can reduce manual work and deliver consistent quality across all environments. When OpenAPI specifications are integrated into the CI/CD pipelines, any change to an API can trigger automated validation, testing, and rollout to development, staging, or production.

API gateways like Membrane support this approach by allowing APIs to be deployed directly from OpenAPI descriptions and configuration files. That helps you keep implementation and documentation in sync.

Deploying Membrane with OpenAPI in Docker

Membrane can be deployed using container images and **included OpenAPI** descriptions. This allows API definitions to be version-controlled, tested, and rolled out as part of CI/CD pipelines. In this approach, a Docker image encapsulates the Membrane API Gateway along with OpenAPI documents, so that the API gateway configuration is immutable and tied to the image version.

The **Dockerfile** below builds a custom Membrane image. Containers created from this image include the API Gateway and an API configured from an OpenAPI document:

```
FROM predic8/membrane

USER root

RUN apt-get update && \
    apt-get install -y wget && \
    rm /opt/membrane/conf/*.yml

# Download OpenAPI and place it conf/
RUN wget "https://github.com/predic8/rfq-
api/releases/latest/download/rfq-api-v1.oas.yml" -O
/opt/membrane/conf/rfq.oas.yml

USER membrane

# Copy the configuration file into the container
COPY proxies.xml /opt/membrane/conf

EXPOSE 2000

ENTRYPOINT ["/opt/membrane/membrane.sh"]
```

Let's break down what this Dockerfile does:

- **Base Image:** It starts from the official `predic8/membrane` image, which contains the Membrane API Gateway runtime.
- **Switch to Root (temporarily):** It uses `USER root` to perform installation steps that require root privileges.
- **Install Tools & Cleanup:** It updates package lists and installs `wget` (used to fetch the OpenAPI spec). It also removes any default `.yml` configuration files in `/opt/membrane/conf` (to avoid deploying unintended sample APIs).
- **Download OpenAPI Spec:** It fetches the latest OpenAPI specification file (in this example, `rfq-api-v1.oas.yml`) from a GitHub release and places it into Membrane's `conf` directory. By downloading at build time, the image will always contain a specific version of the API spec.
- **Switch to Non-Root:** It switches back to the `membrane` user (a non-root user provided by the base image) for running the gateway. This is a security best practice to avoid running the server as root inside the container.
- **Add Custom Configuration:** It copies in our custom `proxies.xml` configuration file into the container's `conf` directory.
- **Expose Port:** It exposes port **2000**, which is the default port where Membrane will listen (as configured in `proxies.xml`).
- **Entrypoint:** It sets the container entrypoint to run Membrane (`membrane.sh`) on startup.

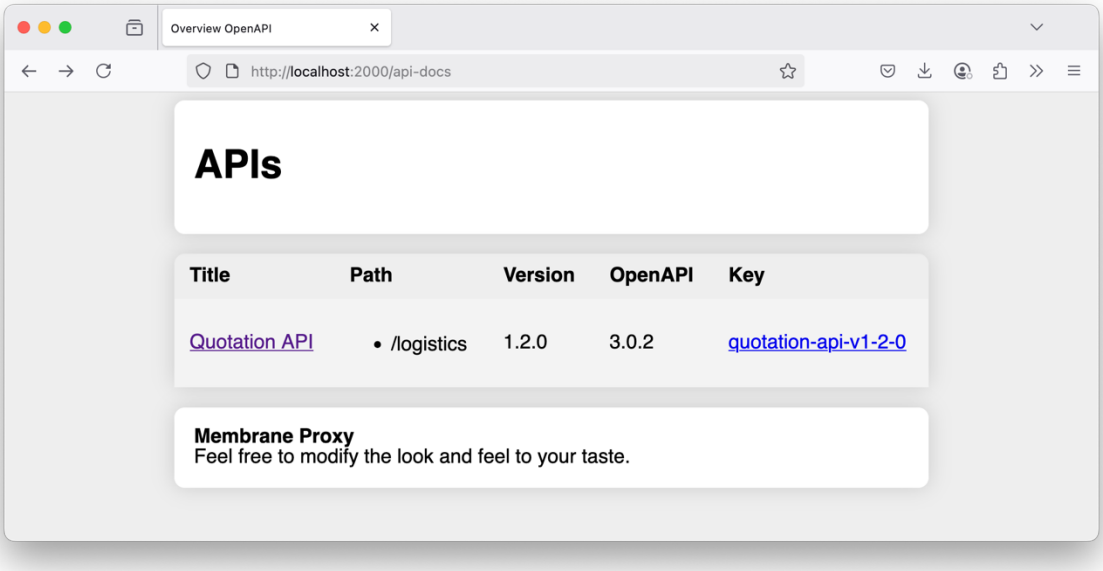
The `proxies.xml` file (injected into the image via the `COPY` command) defines Membrane's configuration. This file should be kept under version control along with the OpenAPI specification. In our simple setup, the `proxies.xml` is configured to automatically deploy all OpenAPI definitions found in the `conf` folder:

```
<spring:beans ...>
  <router>
    <api port="2000">
      <openapi dir="conf" validateRequests="yes"/>
    </api>
  </router>
</spring:beans>
```

Now you can build the image and run a container from it:

```
docker build -t membrane:1 .
docker run -it -p 2000:2000 membrane:1
```

Once started, visit **`http://localhost:2000/api-docs`** in a browser to see the deployed API documentation. Membrane provides a built-in documentation UI: a listing of all deployed APIs and an integrated Swagger UI for each.



27.5 Best Practices for Membrane OpenAPI Deployments

When containerizing Membrane with OpenAPI specs, consider the following best practices to improve reliability, security, and maintainability:

- **Keep Config and Spec in Version Control**
  Store your `proxies.xml` and OpenAPI specification files in a Git repository. This way, changes to the API contract are tracked and can undergo peer review just like code changes. It also allows automation (via CI/CD) to rebuild and deploy the gateway whenever the API spec or configuration changes, ensuring the gateway is always in sync with the intended API contract.
- **Pin Versions for Reproducibility**
  Avoid using floating versions like `latest` for base images or external downloads. In the Dockerfile example, you might replace the GitHub `latest` URL with a specific release version or tag for the OpenAPI file. Likewise, use a fixed version of the `predic8/membrane` base image. Pinning dependencies ensures that your builds are reproducible and prevents unexpected changes. (In production, you should always pin your image versions to avoid undefined behavior.)
- **Minimize the Image Footprint**
  Remove any build-time tools or caches to keep the image slim and secure. For example, after using `wget` to download the spec, you can remove `wget` and clean up apt caches in the same `RUN` layer. This reduces the attack surface and image size.
- **Run as Non-Root**
  We already follow this by switching to the `membrane` user. Running the gateway as a non-root user is a crucial security practice. If someone were to compromise the process, the damage would be limited to the container and that user's privileges, rather than granting root access. Always ensure that any volume mounts or file paths needed by Membrane (logs, etc.) are writable by the `membrane` (or chosen) user so the server runs smoothly without elevated rights.
- **Use Health Checks and Monitoring**
  When running Membrane in Docker (especially in orchestration environments like Kubernetes), set up health endpoints or use Membrane's status pages for liveness and readiness probes. For example, the `/api-doc` (or `/api-docs`) endpoint itself could serve as a simple health check.
- **Plan for API Versioning**
  Over time, you may publish new versions of your API. It's a good practice to version your OpenAPI files (e.g., `rfq-api-v2.oas.yml` for a future v2). Membrane can host multiple OpenAPI specs simultaneously, for instance, one container could include both v1 and v2 specs by placing both files in the `conf` directory (each spec with a different base path or versioned URL). This allows the gateway to serve both versions in parallel, letting clients migrate gradually. When doing so, update your `proxies.xml` to include all relevant OpenAPI files (or use the `dir="conf"` approach as shown, which picks up any number of specs in that folder).
- **Automate Testing in CI/CD**
  Incorporate tests for your deployed API as part of the pipeline. For example, after building the container, you might run it in a CI environment and execute a suite of contract tests or sample requests against the exposed endpoints. This ensures that the combination of Membrane + OpenAPI spec works as expected (e.g., all routes are functioning and validation is correct). Additionally, use OpenAPI linters (like **Spectral** or **openapi-cli**)

during CI to catch issues in your spec (such as missing field descriptions or schema errors) before they get deployed.

- **Avoid bundling Secrets into the Image**
  Never bake credentials into Docker images. Instead, inject them at runtime using environment variables, Docker secrets, or Kubernetes config maps. This helps avoid accidental exposure and keeps your images portable and secure.
- **Regularly update Dependencies**
  Keep an eye on updates for Membrane and for your API spec. Updating the `predic8/membrane` base image to the latest stable version will bring in security patches and new features (Membrane is actively maintained, so new releases may improve OpenAPI support or fix bugs). Since your setup makes the gateway deployment part of your CI/CD, rolling out a new Membrane version can be as simple as changing the base image tag and rebuilding, which should be done periodically. The same goes for the OpenAPI spec. If it evolves, coordinate updates to the spec with deployments of the gateway image through your pipeline.

By following these practices, you ensure that deploying Membrane via Docker remains **robust, secure, and easy to manage** as your APIs evolve. Your gateway becomes a part of your application delivery, benefiting from the same versioning and testing discipline as the rest of your code.

Storing OpenAPI Descriptions in Git

Managing OpenAPI documents in a source code repository unlocks powerful automation and collaboration. By treating the OpenAPI YAML/JSON as code, you can integrate it into your CI/CD pipeline seamlessly. For example, you might set up a GitHub Actions workflow or GitLab CI job that triggers whenever the OpenAPI spec or `proxies.xml` is updated on the main branch. This job could build and push a new Membrane Docker image and even deploy it to a staging environment for testing.

Using Git also means you can leverage **pull requests** to review API changes: team members can discuss and approve adjustments to the API contract before they go live, preventing accidental breaking changes. Storing OpenAPI files in Git enables collaborative design, automated deployments, and consistent rollouts across environments. Whenever a pull request is merged, your pipeline can confidently deploy the updated specification to the API Gateway, knowing that it has been reviewed and tested. This leads to **fewer surprises** in production and a faster iteration cycle for API development.

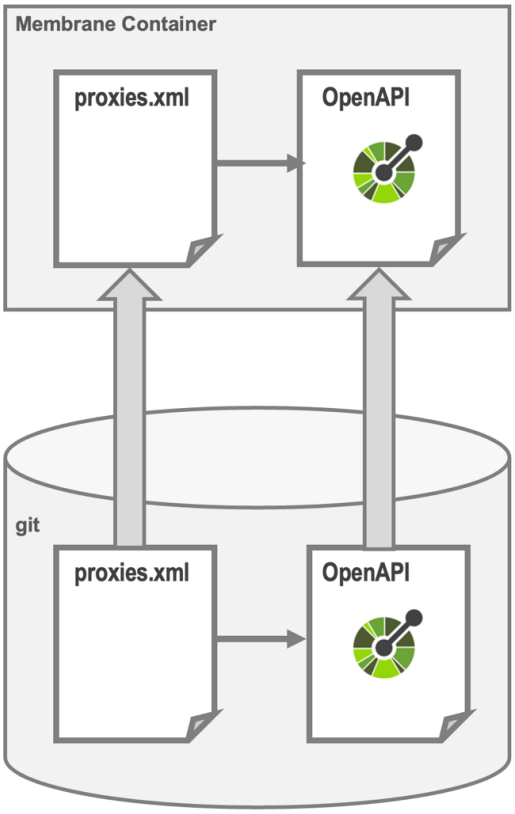


Image: Container from versioned configuration and API description

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=

28 Transformation and Message Manipulation

Message transformations in an API Gateway can vary from simple operations such as modifying HTTP headers to complex conversions of JSON or XML payloads. These transformations allow you to tailor requests and responses to meet specific application needs.

28.1 Manipulating HTTP Headers

Let's begin with a straightforward yet practical example: adding and removing HTTP header fields.

Adding a HTTP Header

API gateways often add HTTP headers to outgoing messages to enable CORS or supply credentials.

The example below demonstrates how to add a custom response header, `X-Foo`, with a value of `42`:

```
<api port="2000">
  <response>
    <setHeader name="X-Foo" value="42"/>
    <static>Ok</static>
  </response>
  <return/>
</api>
```

Since the `<setHeader>` element is placed inside the `<response>` block, the header is added to outgoing responses.

To test this configuration, run the following command in your terminal:

```
curl -v http://localhost:2000
```

The response should contain the `X-Foo` header:

```
HTTP/1.1 200 Ok
Content-Type: text/plain
X-Foo: 42

Ok
```

28.2 Passing HTTP Headers to a Backend

HTTP headers can be forwarded from a gateway to an upstream backend, a technique widely used for security purposes. For example, a token assigned to a request at the gateway can be validated by the backend. The following example simulates this setup using two APIs.

Gateway API: Adding the Token

This API listens on port 2000. It adds a confidential token `X-Token` to outgoing requests and forwards them to the backend running on port 3000:

```
<api port="2000">
  <request>
    <setHeader name="X-Token" value="abc123"/>
  </request>
  <target port="3000" host="localhost"/>
</api>
```

Backend API: Logging the Token

The second API, running on port 3000, acts as the backend service. It logs the received `X-Token` header value and returns a **204 No Content** status code:

```
<api name="backend" port="3000">
  <log message="Received: ${header['X-Token']}"/>
  <return statusCode="204"/>
</api>
```

Both APIs are deployed on the same gateway for demonstration purposes. Even though they run on the same machine, the gateway routes the request through the operating system's network stack and back, simulating a typical backend connection.

Testing the Configuration

To test this setup, send a request to port 2000:

```
curl http://localhost:2000
```

The log output in the Membrane console should be like this:

```
14:42:06 INFO 70 {proxyName=backend} - Received: abc123
```

While this is a simple example, many real-world scenarios use HTTP headers for authentication or authorization. In a later section, we will explore how to validate headers for specific values.

Note: Logging sensitive tokens is solely for demonstration. In a production environment, ensure that sensitive information is never logged.

28.3 Computing Header or Property Values

You're not limited to assigning constant strings to headers. You can also **compute values dynamically** using an expression language such as Groovy, JSONPath, or Spring Expression Language (SpEL). This makes your API configuration much more flexible and powerful.

The example below demonstrates how to set headers using calculations, the current date, and JSON content from the request body:

```
<setHeader name="X-Number"
           value="${8 + 7}"
           language="Groovy"/>

<setHeader name="X-Date"
           value="${java.time.LocalDate.now()}"
           language="Groovy"/>

<setHeader name="X-Address"
           value="${$.address.zip} ${$.address.city}"
           language="Jsonpath"/>

<log message="Header: ${header}"/>
```

Now, call the API with the following request:

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

{
  "address": {
    "city": "Berlin",
    "zip": "12111"
  }
}
```

You should see output like this in the log:

```
X-Number: 15
X-Date: 2025-02-28
X-Address: 12111 Berlin
```

This illustrates how headers can be computed at runtime, adapting to the request content or environment.

28.4 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 include headers that leak details like the server type, framework versions, or infrastructure setup. Attackers can use that information to target known vulnerabilities.

To minimize this risk, API gateways let you control which headers are allowed to pass through. You can use allowlists and blocklists to explicitly define what should be forwarded and what should be stripped out.

Here's an configuration that places an API in front of `wikipedia.org` and removes all response headers except those that begin with `Content` or are named `last-modified`:

```
<api port="2000">
  <response>
    <headerFilter>
      <include>Content.*</include>
      <include>last-modified</include>
      <exclude>.*</exclude>
    </headerFilter>
  </response>
  <target url="https://www.wikipedia.org"/>
</api>
```

To see what headers Wikipedia normally returns, try this command:

```
curl -sS -D - https://www.wikipedia.org -o /dev/null
```

With the `headerFilter` in place, only the explicitly allowed headers (`Content.*` and `last-modified`) are included in the response to the client. Everything else is dropped. This helps keep your system's internals private and reduces the attack surface.

28.5 Body Transformation

Just like headers, the body, or payload, of a message can also be transformed by an API Gateway. This is helpful when integrating systems that expect different formats, need extra data injected, or require simplified structures for frontend consumption.

Gateways provide a range of tools for transforming message bodies, including:

1. **Format Converters**
   Automatically convert between formats, such as turning an XML payload into JSON or the other way around.
2. **Templates**
   Define templates with placeholders that are dynamically filled in using values from headers, query parameters, or even expression language results.
3. **Beautifiers and Formatters**
   Pretty-print JSON or XML content by adding indentation and line breaks. While this doesn't change the data, it makes it much more readable for humans during development, debugging, or logging.

We'll walk through each of these transformation techniques in the next sections using practical, hands-on examples. By the end, you'll be able to reshape API payloads to fit your use case.

28.6 Format Transformation

API gateways can perform generic format transformations to convert a payload from one representation to another. A common use case is converting between JSON and XML.

In the example below, the gateway transforms a JSON response into XML before returning it to the client:

```
<api port="2000">
  <response>
    <json2Xml/>
  </response>
  <target url="https://api.predic8.de"/>
</api>
```

When you call this API with:

```
curl http://localhost:2000/shop/v2/products/7
```

You'll receive an XML response like this:

```
<?xml version="1.0" encoding="UTF-
8"?><root><image_link>/shop/v2/products/7/image</image_link><p
rice>69.99</price><name>Gac-
Fruit</name><id>7</id><modified_at>2025-01-
29T12:30:00.026274Z</modified_at><vendors><name>Exotics Fruit
Lair
Ltd.</name><id>1</id><self_link>/shop/v2/vendors/1</self_link>
</vendors></root>
```

In the next section, we'll look at techniques for making such responses more readable.

28.7 Make It Nice

Sometimes developers or other stakeholders need to examine an API's response in a more human-readable format. To achieve this, you can **beautify** JSON or XML messages before returning them to the client. For instance, you can make the XML response from the previous section more readable by adding a beautifier plugin:

```
<api port="2000">
  <response>
    <beautifier/>
    <json2Xml/>
  </response>
  <target url="https://api.predic8.de"/>
</api>
```

Note that the `<beautifier>` is placed above `<json2Xml>` because, in Membrane, the **response flow executes from the bottom to the top**. When you invoke the endpoint again:

```
curl http://localhost:2000/shop/v2/products/7
```

you'll receive a neatly formatted response:

```
<root>
  <image_link>/shop/v2/products/7/image</image_link>
  <price>69.99</price>
  <name>Gac-Fruit</name>
  <id>7</id>
</root>
```

Isn't that pretty?

The Membrane beautifier works with both JSON and XML payloads, ensuring more readable responses for debugging purposes.

28.8 Templates

Templates let you set the body of a message with dynamically generated content. They can produce JSON, XML, or even legacy formats like SOAP.

Static Content

With static content, the message body is set with a fixed payload that never changes. In some gateways, this feature is called `setBody`. Here's an example where the response is replaced with a constant JSON object:

```
<api port="2000">
  <response>
    <static contentType="application/json">
      {
        "place": "London"
      }
    </static>
  </response>
  <return/>
</api>
```

Dynamic rendered Content with Templates

Sometimes you need to build responses using dynamic values instead of fixed content. That's where template engines come into play. Many API gateways support standard engines like **Mustache** or **Velocity**. Membrane uses the **Groovy Template engine** for this purpose.

In the following example, the gateway transforms an incoming XML payload into a JSON response using a template.

The client sends an XML document that describes an article:

```
POST /
Host: localhost:2000
Content-Type: application/json

<article>
  <name>Lolly XXL</name>
  <color>green</color>
</article>
```

The corresponding API configuration performs three steps. In the request flow, the `name` and `color` properties are extracted from the XML using XPath. The `<return/>` plugin reverses the flow, and in the response flow the extracted properties are injected into a JSON template:

```
<api port="2000">
  <request>
    <setProperty name="name"
                 value="${/article/name}"
                 language="xpath"/>
    <setProperty name="color"
                 value="${/article/color}"
                 language="xpath"/>
  </request>
  <response>
    <template contentType="application/json" pretty="true">
      {
        "product": {
          "name": "${property.name}",
          "color": "${property.color}"
        }
      }
    </template>
  </response>
  <return/>
</api>
```

When invoked, the client receives a response like:

```
HTTP/1.1 200 Ok
Content-Type: application/json
Content-Length: 71

{
  "product": {
    "name": "Lolly XXL",
    "color": "green"
  }
}
```

Using templates like this allows your API to produce customized payloads on the fly, great for format conversions and data mapping.

Templates with Loops and Conditions

Sometimes you need to render more complex structures, like **lists** or **tables**. For these cases, templates can include loops and conditional logic. The example below shows how to iterate over a list of HTTP headers and optionally include the value of a query parameter if it's present:

```
<api port="2000">
  <request>
    <template contentType="text/plain">
      <![CDATA[

      Header:
      <% for(h in header.allHeaderFields) { %>
         <%= h.headerName %> : <%= h.value %>
      <% } %>

      <% if (param.foo) { %>
        Query param foo is: <%= param.foo %>
      <% } %>

      ]]>

    </template>
  </request>
  <return/>
</api>
```

To test this configuration, send a request to `http://localhost:2000` with or without a `foo` query parameter.

Sidenote: CDATA Section
The `<![CDATA[ ... ]]>` block tells the XML parser to treat everything inside as plain text. That's important when your template includes characters like `<`, `>`, or `&`, which would otherwise confuse the parser. Wrapping your template code in a CDATA section ensures that the templating engine gets exactly what you wrote without interference from the surrounding XML.

Resources

Groovy documentation on template engines
https://docs.groovy-lang.org/next/html/documentation/template-engines.html

29 Control Flow

Most of the API definitions are just in one direction. But you are not limited on that. Many gateways support complex control flows with loops and conditions.

29.1 Conditions

The `if` plugin makes the execution of plugins conditional, depending on a test expression. It can be applied during both the request and response phases.

Let's illustrate this with a simple form of API protection. In some cases, implementing a full authentication system is unnecessary, perhaps you're working with an internal API or a prototype.

The configuration below demonstrates how to guard an endpoint using a simple API key check:

```
<api port="2000">
  <request>
    <if test="header['X-Api-Key'] != 'Bibbidi-Bobbidi-Boo'">
      <static>No way!</static>
      <return statusCode="401"/>
    </if>
  </request>
  <static>Welcome!</static>
  <return/>
</api>
```

If the client sends the `X-Api-Key` header with the correct value (`Bibbidi-Bobbidi-Boo`), the API responds with "Welcome!". Otherwise, it returns an error message with a **401 Unauthorized** status code. To test this, you can use the following request:

```
GET http://localhost:2000
X-Api-Key: Bibbidi-Bobbidi-Boo
```

💡**Sidenote**: **Lightweight protection only**
Hardcoding secrets directly into configuration files is rarely a good idea. It may work for demos or internal use, but it doesn't scale and cannot be rotated dynamically. For anything beyond basic use, consider proper authentication mechanisms such as token validation or API key management, as discussed later in this book.

Choose and Case

One common use case for API gateways is translating backend error messages into a consistent format that clients can reliably understand. This might mean converting backend

responses into Problem Details for HTTP APIs (RFC 7807), or turning them into SOAP faults for legacy clients.

This example demonstrates how to use Membrane's `<choose>` construct with multiple `<case>` conditions to generate structured error responses based on the backend's status code:

```
<api port="2000">
  <response>
    <choose>
      <case test="statusCode == 401 or statusCode == 403">
        <template contentType="application/problem+json">
          {
            "type": "https://membrane-api.io/problem/auth",
            "title": "Authentication Error"
          }
        </template>
      </case>
      <case test="statusCode >= 400 and statusCode &lt; 500">
        <template contentType="application/problem+json">
          {
            "type": "https://membrane-api.io/problem/client",
            "title": "Client Error"
          }
        </template>
      </case>
      <case test="statusCode >= 500">
        <template contentType="application/problem+json">
          {
            "type": "https://membrane-api.io/problem/server",
            "title": "Server Error"
          }
        </template>
      </case>
      <otherwise>
        <static>What's happening!</static>
      </otherwise>
    </choose>
  </response>
  <target host="localhost" port="3000"/>
</api>
```

The `<choose>` block works like a switch statement. It evaluates each `<case>` in order and executes the first one that matches. If no case applies, the `<otherwise>` block is triggered.

Resources

RFC 7807: Problem Details for HTTP APIs
https://datatracker.ietf.org/doc/html/rfc7807

30 API Orchestration

API orchestration is the combination of multiple backend services into a single, unified API. This can simplify client logic, reduce API traffic, and hide internal complexity. It's especially helpful in scenarios like authentication flows, microservice aggregation, application integration, and mobile clients where minimizing roundtrips 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 your external interface or bridge the gap between legacy systems and modern consumers, orchestration is a powerful tool to have in your API gateway toolbox.

30.1 Aggregating Backend APIs

The first orchestration use case we'll look at 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 roundtrips, something especially useful for mobile apps and browsers.

Let's walk through an example using the **Open Library API**. Imagine we want an endpoint that returns both the title and the author of a book. The Open Library project offers public APIs for exactly this kind of data, but not 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" }
  ]
  ...
}
```

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 returns:

```
{
  "name": "Douglas Adams"
  ..
}
```

To combine both steps into a single call from the client's perspective, we can use the API Gateway to orchestrate the calls.

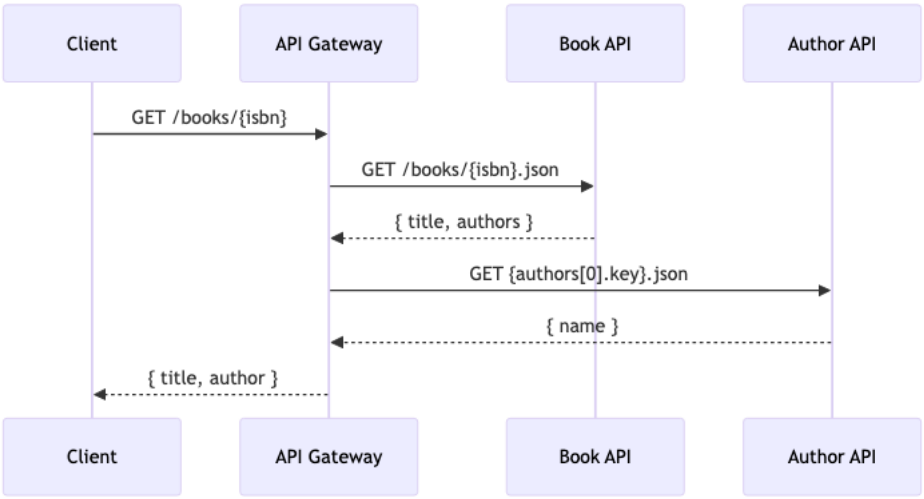


Image: Orchestrating two backend calls into a single API (Rendered with Mermaid)

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 sent back to the client

Orchestrating with an API Gateway

Using orchestration, we can expose a single, convenient endpoint that performs multiple backend calls under the hood.

Here's a configuration example that fetches both book and author information from the Open Library API:

```
<api port="8080">
  <path>/books/{olid}</path>

  <!-- 1. Get book details -->
  <call
url="https://openlibrary.org/books/${pathParam.olid}.json"
  />

  <setProperty name="authors"
               value="${$.authors}"
               language="jsonpath"/>
  <setProperty name="title"
               value="${$.title}"
               language="jsonpath"/>

  <!-- 2. Get author details -->
  <call
url="https://openlibrary.org${properties.authors[0]['key']}.js
on"
  />
  <setProperty name="author"
               value="${$.name}"
               language="jsonpath"/>

  <!-- Combine the results into a single JSON -->
  <template contentType="application/json" pretty="true">
    {
      "title": "${property.title}",
      "author": "${property.author}"
    }
  </template>

  <return/>
</api>
```

Now, a client can simply make this request:

```
GET http://localhost:8080/books/OL29474405M
```

And receive a clean, combined response:

```
{
  "title": "So Long, and Thanks for All the Fish",
  "author": "Douglas Adams"
}
```

This approach keeps client-side code minimal and clean. It also improves performance, especially for mobile apps on slow or unreliable networks, by reducing the number of roundtrips.

💡**Sidenote**: **What's happening with `setProperty` and JSONPath?**
`<setProperty>` extracts a value from the backend responses using JSONPath. The results are stored as named properties, which can then be reused across multiple steps, like inserting values into templates.

30.2 Authentication for Backend API

In the next orchestration use case, the API Gateway handles authentication on behalf of the client.

Imagine a scenario where a backend API requires a session cookie or token, but you'd prefer to hide that complexity from your API users. You might even want to expose a completely different authentication mechanism externally than what's used internally.

With orchestration, the gateway can perform the necessary login steps in the background and forward the appropriate credentials, keeping the client facing API simple, and easy to use.

Let's look at a scenario involving three key components:

1. **Protected Target API**
   Requires a valid session cookie for access.
2. **Authentication Service**
   Issues the required session cookie.
3. **Orchestration API**
   Logs into the authentication service, obtains the session cookie, and forwards it to the protected API

The following diagram illustrates the interactions between these components:

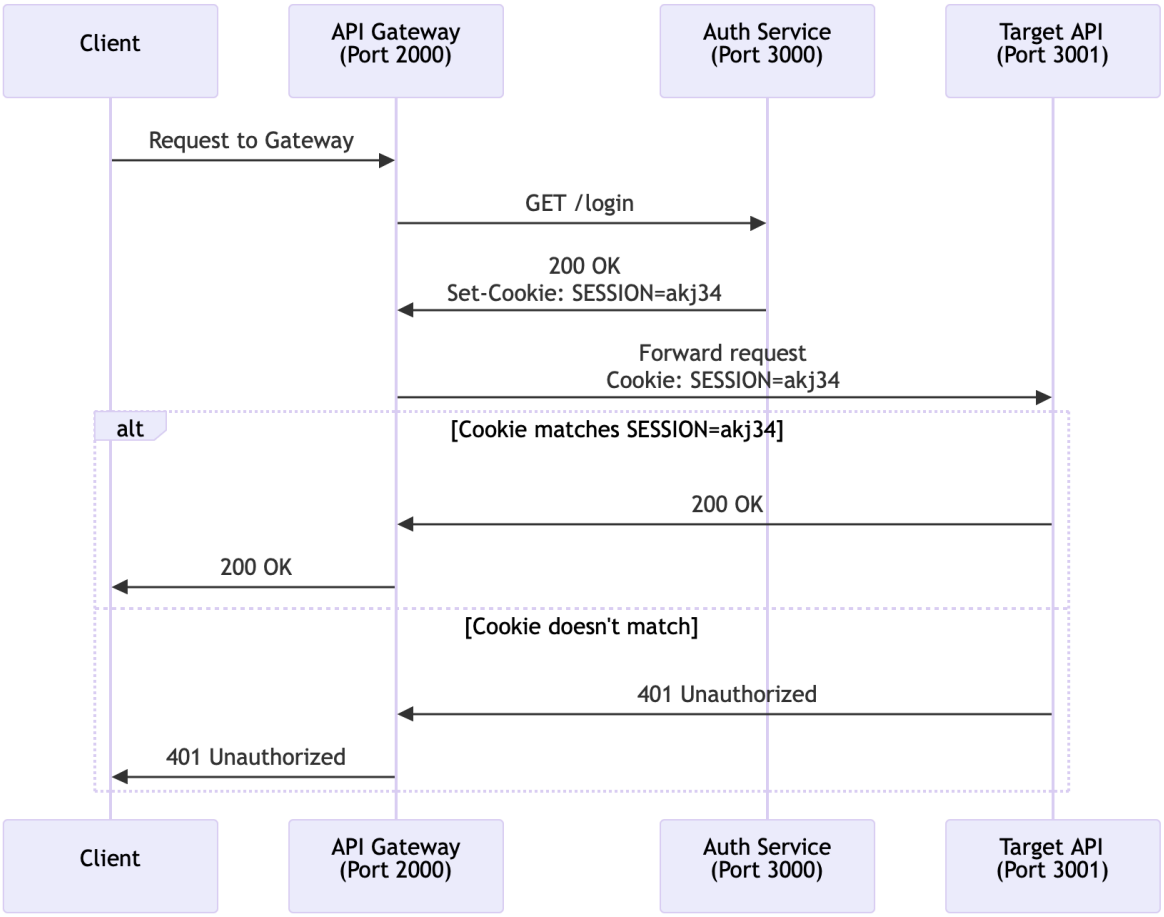


Image: API Gateway authenticates on behalf of the client

Setting it Up with an API Gateway

All three components from this scenario can be simulated using **Membrane API Gateway**.

Here's how each component is configured:

1. Target API (Backend Simulation at port 3001)

This API checks whether a valid session cookie (`SESSION=akj34`) is present. If it is, the API returns `"Success!"` Otherwise, it responds with an authentication error (`401 Unauthorized`):

```
<api port="3001">
  <if test="cookie.SESSION == 'akj34'">
    <static>Success!</static>
    <return/>
  </if>
  <static>Please log in!</static>
  <return statusCode="401"/>
</api>
```

2. Authentication Service (port 3000)

This API simulates a login endpoint that sets the required session cookie:

```
<api port="3000">
  <path>/login</path>
  <response>
    <setHeader name="Set-Cookie" value="SESSION=akj34" />
  </response>
  <return />
</api>
```

3. Orchestration API (port 2000)

The orchestration API acts as a facade. It first logs into the authentication API, extracts the session cookie, and then forwards the request to the protected backend with the cookie:

```
<api port="2000">
  <request>

    <!-- Optional: Check authentication here -->

    <call url="http://localhost:3000/login" />
    <setHeader name="Cookie"
               value="${header['set-cookie']}"/>
  </request>
  <target url="http://localhost:3001"/>
</api>
```

Securing the Orchestration API

In this simplified setup, the orchestration API itself is not protected. In real-world use cases, you'd typically secure this external endpoint using **API keys**, **JWTs**, or **OAuth2**. This allows you to offer a modern, secure interface to clients, even if the backend systems still rely on outdated or session-based authentication.

30.3 Processing RESTful List Resources

The last orchestration scenario demonstrates how to **navigate linked resources** in a RESTful API.

RESTful APIs often expose a **list resource** for each business object type. For example:

```
GET /products
```

This returns a list with short entries for each business object, typically containing only basic information such as IDs and names:

```
{
  "products" : [
    { "id" : 1, "name" : "Bananas" },
    { "id" : 2, "name" : "Figs" },
    { "id" : 3, "name" : "Grapes" }
  ]
}
```

To get more details (such as the price), you must retrieve each individual resource using another call, for example:

```
GET /products/3
```

Which returns:

```
{
  "id" : 3,
  "name" : "Grapes",
  "price" : 4.5,
  "image_link" : "/shop/v2/products/8/image"
}
```

Now suppose you're building an app that needs to show products with prices:

```
Bananas: 1.99
Figs: 2.40
Grapes: 5.80
```

To do that, the client would have to call `/products`, then fetch the full details of each item one by one. That's a lot of logic and roundtrips for a mobile or frontend app.

Instead, you can **orchestrate** these steps in the API Gateway, gathering all required data server-side and returning a clean, merged response.

The diagram below illustrates the call sequence.

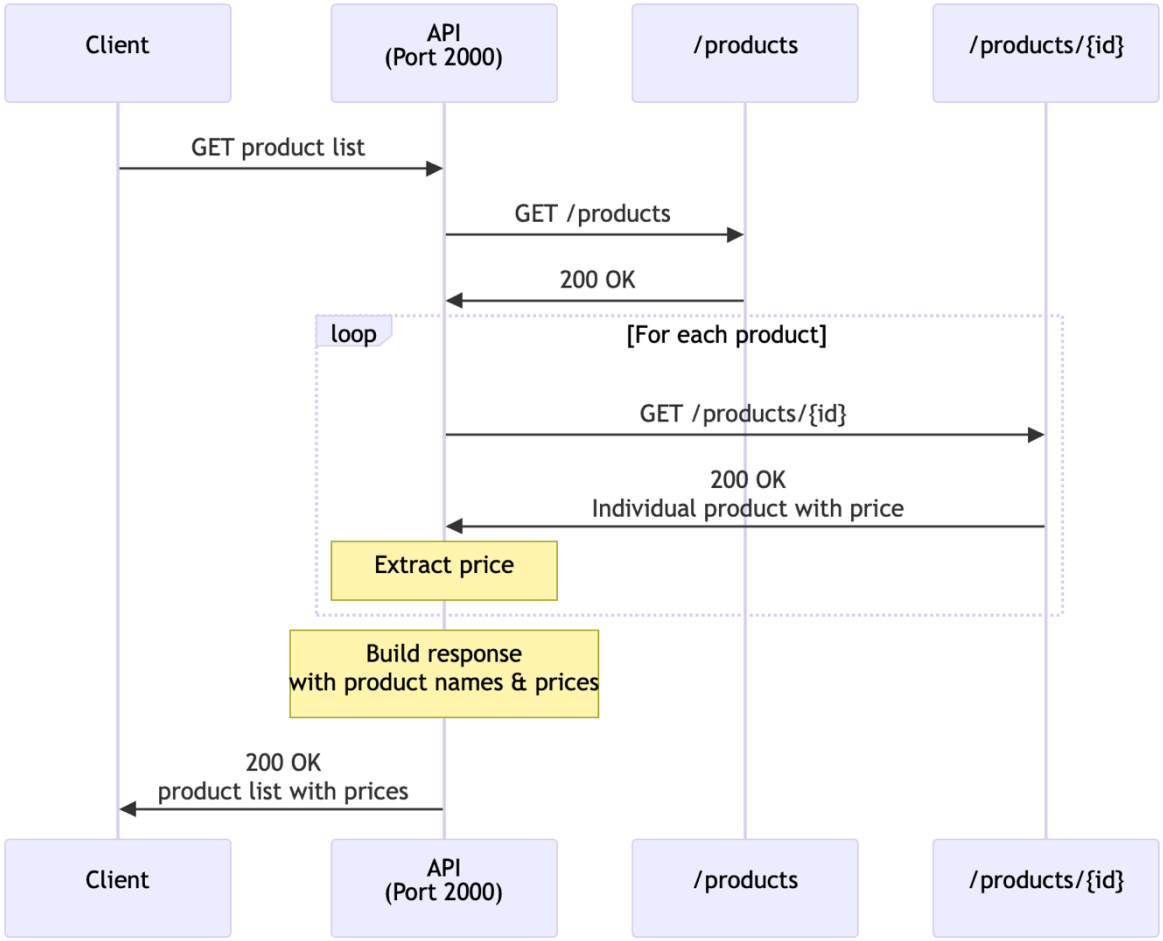


Image: Sequence of calls to get products and prices in a single API

Implementation with Membrane API Gateway

The configuration below uses Membrane API Gateway to fetch and merge product details into one simplified JSON response:

```
<api port="2000">
  <request>
    <!-- Fetch list of products -->
    <call
url="https://api.predic8.de/shop/v2/products?limit=1000"/>
    <setProperty name="products"
                 value="${$.products}"
                language="jsonpath"/>

    <!-- Iterate over each product -->
    <for in="property.products">
      <call
url="https://api.predic8.de/shop/v2/products/${properties.it['
id']}"/>
      <setProperty name="price"
                   value="${$.price}"
                   language="jsonpath"/>
      <!-- Add price to entry -->
      <groovy>
        property.it.price = property.price
      </groovy>
    </for>

    <!-- Render response with only product name and price -->
    <template contentType="application/json" pretty="true">
    <![CDATA[
    {
      "products": [
        <% property.products.eachWithIndex { p, idx -> %>
          {
            "name": "<%= p.name %>",
            "price": "<%= p.price %>"
          }<%= idx < property.products.size() - 1 ? ',' : ''
%>
        <% } %>
      ]
    }
    ]]>
    </template>
  </request>
  <return/>
</api>
```

Step-by-Step Explanation:

1. **Fetching the Product List**:

   This retrieves up to 1000 products in one call:

   ```
   <call
   url="https://api.predic8.de/shop/v2/products?limit=1000"
   />
   ```

2. **Extracting Product Data**:

   We store the list of products as a property using JSONPath:

   ```
   <setProperty name="products"
                value="${$.products}"
                language="jsonpath"/>
   ```

3. **Looping and Enriching**:

   We loop through each product and fetch detailed info:

   ```
   <for in="property.products">
     <call
   url="https://api.predic8.de/shop/v2/products/${properties.it
   ['id']}"
     />
     <setProperty name="price"
                  value="${$.price}"
                  language="jsonpath"/>
       <groovy>
           property.it.price = property.price
       </groovy>
   </for>
   ```

   The `<for>` block introduces a loop variable called `it`. The embedded Groovy script assigns the fetched `price` to the current product entry.

4. **Creating the Final JSON Response**:

   A Groovy-enhanced template renders the output as compact JSON:

```
<template contentType="application/json" pretty="true">
  <![CDATA[
  {
    "products": [
      <% property.products.eachWithIndex { p, idx -> %>
        {
          "name": "<%= p.name %>",
          "price": "<%= p.price %>"
        }<%= idx < property.products.size() - 1 ? ',' : '' %>
      <% } %>
    ]
  }
  ]]>
</template>
```

   This logic ensures a valid JSON array without trailing commas.

💡 **Sidenote**: **More about the Template Engine**
Membrane uses Groovy's built-in template engine for dynamic message manipulation. If you want to go deeper or customize templates beyond the basics, the official Groovy documentation has you covered:

https://docs.groovy-lang.org/next/html/documentation/template-engines.html

Performance Hint: Looping in API Orchestration
If the product list is large, this orchestration may take several seconds to complete. Keep this in mind when designing your application.

31 Secure Data in Transit with TLS

This section explores how to configure **TLS (Transport Layer Security)** for both encryption and authentication in your API Gateway. You'll see practical examples that demonstrate how to secure traffic on both ends of the gateway.

We'll begin by examining the TLS connection **from the gateway to the backend**, followed by the connection **from the client to the gateway**.

31.1 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 the data in transit ensuring confidentiality.

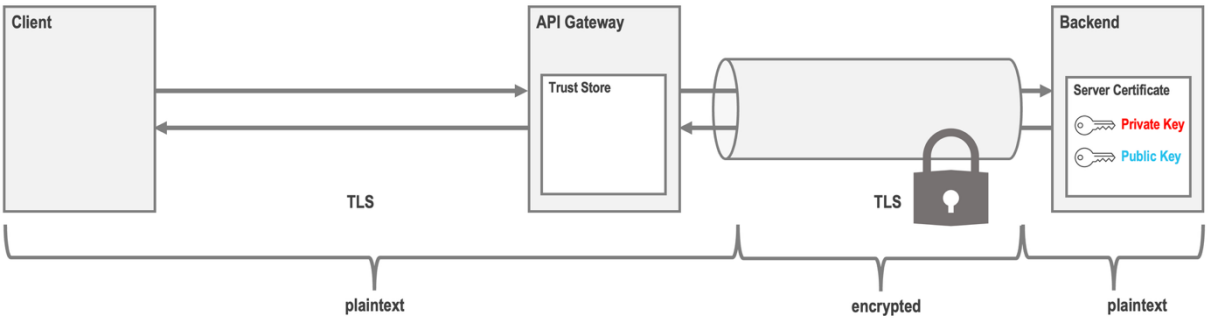


Image: TLS secured connection between gateway and backend

Securing backend connections over TLS typically involves:

- Managing **trust stores** for certificate validation
- Handling **certificate revocation** and renewal
- Optionally implementing **mutual TLS (mTLS)** to authenticate both sides of the connection

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've already seen this kind of setup earlier in the book:

```
<api port="2000">
  <target url="https://api.predic8.de"/>
</api>
```

Here, simply using `https://` in the backend URL tells the gateway to use TLS for the outgoing connection.

Since this backend's certificate is signed by a **public certificate authority (CA)**, the gateway can validate it against the trusted root certificates in its **trust store**.

Membrane, being based on Java, uses the **Java platform truststore** by default. This is typically located at:

```
<JAVA_HOME>/lib/security/cacerts
```

You can add your own trusted certificates to this file using tools like `keytool`. Membrane will automatically use them for outgoing TLS connections.

What is a truststore?
A **truststore** contains a collection of trusted root and intermediate certificates. When a server presents a certificate, the truststore is used to check whether it was issued by a recognized and trusted authority.

31.2 Termination of TLS Connections

TLS termination refers to the process where an incoming encrypted TLS connection is decrypted by the API Gateway.

Most gateway functions require access to the unencrypted payload. Without decrypting the traffic, the gateway would just see a blob of encrypted data and couldn't do much with it.

Once decrypted, the gateway can inspect, transform, or route the request as needed.

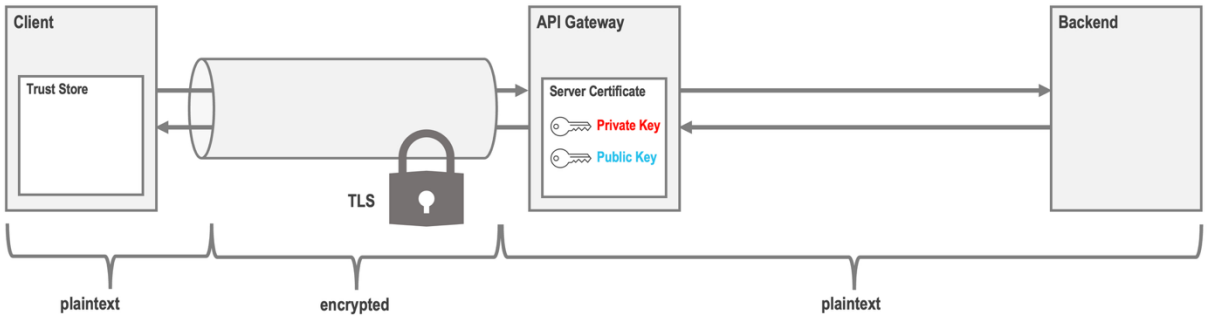


Image: Termination of a TLS secured connection at the gateway

Handling TLS at the gateway offers several advantages:

- **Enabling Gateway Features**
  Functions like logging, authentication, transformation, and validation only work if the gateway can read the payload. Decrypting the traffic makes these features possible.

- **Simplified Backend Configuration**
  If TLS is terminated at the gateway, backend services don't need to manage certificates or support HTTPS. That's especially useful for legacy systems or internal services running on private networks.
- **Flexibility**
  The gateway can optionally re-establish a new encrypted TLS connection to the backend. So the data is always encrypted in transit.

Naturally, TLS termination comes with a few responsibilities. The gateway must manage sensitive assets like private keys and certificates. There's also a slight processing overhead from the encryption and decryption steps. But in most setups, the benefits, centralized control, flexibility, and security, make TLS termination the better choice.

💡 **Sidenote: TLS Passthrough?**
In TLS passthrough mode, the gateway doesn't terminate TLS, it simply 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 most challenging part of setting up TLS termination is obtaining or creating the necessary certificates and keys. Once available, configuring Membrane, or other API gateways, is straightforward.

Membrane includes **sample certificates and keys** for testing purposes. You can find them in the folders:

- `examples/security/tls-ssl/`
- `conf/`

Sample files include:

- `membrane-key.pem`: the private key
- `membrane.pem`: the corresponding certificate

⚠️ **Important**
Never use these sample certificates in production environments.

Examining the Certificate

Before configuring the gateway, you can inspect the certificate using the `openssl` tool (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 was issued to
- **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 you have the key and certificate ready, you can configure TLS termination like this:

```
<api port="443">
  <ssl>
    <key>
      <private location="membrane-key.pem" />
      <certificate location="membrane.pem" />
     </key>
  </ssl>
  <log/>
  <target url="https://api.predic8.de"/>
</api>
```

This configuration terminates incoming TLS connections on port 443 using the provided certificate and private key. It then logs the HTTP message in plain text and forwards the request to the backend.

Testing the TLS Setup

The `curl` tool is very useful for debugging TLS connections. You can test your gateway configuration with the following command:

```
curl -v -k https://localhost:443
```

Explanation of options:

- **`-v`**: Enables verbose mode, showing detailed information about the TLS handshake and HTTP exchange
- **`-k`**: Skips certificate validation, helpful when using self-signed certificates during testing

The output provides a detailed view of the **TLS handshake** between the client and the gateway, along with certificate information.

```
* 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-POLY1305-SHA256
/ [blank] / UNDEF
* ALPN: server did not agree on a protocol. Uses default.
* 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.
* using HTTP/1.x
> GET / HTTP/1.1
> Host: localhost
> User-Agent: curl/8.7.1
>
< HTTP/1.1 200 Ok
< Content-Type: application/json
```

This confirms that the TLS handshake was completed successfully.

In this example, **TLS termination** is configured for a single API. For instructions on applying TLS settings across **multiple APIs or connectors**, see the samples in the `examples/security/ssl-tls` folder of the Membrane distribution.

⚠ **Security Warning!**
Always replace the sample certificates with valid production certificates in live environments.

Forwarding TLS Connections Without Decryption

Sometimes, traffic is too sensitive for even the API gateway to inspect. In such cases, Membrane can forward TLS connections without decrypting them:

```
<sslProxy host="verysecretbackend.predic8.de" port="443">
  <target host="verysecretbackend.predic8.de" port="443"/>
</sslProxy>
```

Membrane uses the SNI (Server Name Indication) from the client's TLS handshake to decide where to forward the connection. If the SNI matches, the connection is passed through untouched—no decryption, no inspection, no logging.

This turns Membrane into a layer 4 (TCP) proxy. Since it doesn't terminate TLS, it doesn't need a certificate or private key. The encrypted data is only decrypted at the backend.

Tip: Sharing Ports
You can share port 443 with other proxies (e.g., <api>) handling different hostnames.

31.3 Debugging TLS Connectivity

Setting up TLS for the first time can feel like opening a door to the internet—and sometimes, the internet knocks back.

When configuring TLS in Membrane, especially during initial rollout or certificate changes, it's helpful to enable detailed error logging:

```
<ssl showSSLExceptions="true">
```

This setting reveals what's really happening under the hood. And trust us, there's a lot going on.

What Happens When You Open Port 443?

The moment you expose port 443 to the public internet, without IP restrictions, you'll start receiving inbound connections. Some are harmless. Some are curious. Some are… not.

Here's what you might see:

- **Statistical scanners** collecting data for research or monitoring
- **Security researchers** probing your TLS version (possibly for a thesis)
- **Automated bots** looking for known vulnerabilities (e.g., in OpenSSL—even though Membrane doesn't use it)
- **Generic exploit attempts** targeting common platforms like WordPress
- **Novel attacks** maybe aimed at Java-based services or Membrane itself

All of this generates noise. Lots of it. And by default, Membrane hides some of it with showSSLExceptions="false". Turning it on helps you see what's really hitting your gateway.

Why It Matters

- You'll catch misconfigured clients trying to connect with outdated TLS versions.
- You'll see malformed handshakes that might indicate scanning or probing.
- You'll be able to distinguish between harmless background traffic and actual threats.

32 Access Control Lists

A simple yet effective way to protect an API is by limiting access to specific IP addresses or ranges using **Access Control Lists (ACLs)**. ACLs restrict which clients can interact with an API, providing an additional layer of security, especially in controlled network environments.

In **Membrane**, ACLs are defined in external XML files. Here's an example (`acl.xml`):

```
<accessControl>
  <resource uri="*">
    <clients>
      <ip>127.0.*</ip>
      <ip>192.168.2.213</ip>
    </clients>
  </resource>
</accessControl>
```

This configuration applies to **all API endpoints** (denoted by `*` in the `uri` attribute) and allows only clients from:

- The local range `127.0.*`
- The specific IP `192.168.2.213`

To apply this ACL to an API, include the ACL file in your configuration:

```
<api port="2000">
  <accessControl file="./acl.xml" />
  <target url="https://api.predic8.de"/>
</api>
```

While IP addresses can be spoofed, ACLs still serve as a useful **first line of defense**, especially when used to restrict access to internal services or known clients. For stronger protection, ACLs are often combined with other security mechanisms such as **API keys**, **OAuth2**, or **JWT authentication**, forming a **multi-layered security strategy**.

Resources

ACL Configuration Examples
`MEMBRANE_HOME/examples/security/access-control-list`

33 Content Protection

Attackers can exploit subtle quirks and dangerous features in data formats like JSON, XML, or GraphQL to trigger unexpected behavior or even remote code execution.

That's where content protection comes in. It helps block potentially harmful input before it reaches backend services. The goal isn't to understand the full meaning of a request but to enforce structural safety and avoid known attack vectors, making exploitation harder or even impossible.

Since the basic requirements for content protection are fairly straightforward, most API gateways offer similar functionality. The main differences usually lie in configuration syntax and naming conventions rather than in the actual capabilities.

33.1 JSON Protection

Most API gateways offer built-in **JSON protection** features that help prevent malformed or malicious payloads from reaching your backend services.

In **Membrane**, enabling JSON protection is as simple as including the `<jsonProtection />` element in an API configuration:

```
<api port="2000">
  <jsonProtection />
  <target url="https://api.predic8.de"/>
</api>
```

When a request like the following is received:

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

{
  "price": 10,
  "price": -1
}
```

... the JSON protection mechanism detects the duplicate `price` field and returns an error message:

```
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"
}
```

Customizing JSON Protection

The default settings offer strong protection for most use cases. However, if you need stricter limits, or need to loosen restrictions for valid large payloads, you can configure the parameters individually:

```
<jsonProtection maxTokens="15"
                maxSize="110"
                maxDepth="3"
                maxStringLength="5"
                maxKeyLength="10"
                maxObjectSize="3"
                maxArraySize="3" />
```

Resources

JSON Protection Examples and Configuration patterns
Check the `examples/security/json-protection` directory in the Membrane distribution.

jsonProtection Reference
https://www.membrane-api.io/docs/current/jsonProtection.html

33.2 XML Protection

Just like JSON, XML payloads can also be exploited to overwhelm or bypass backend systems. To guard against this, Membrane provides **XML protection** features that are easy to apply with a simple configuration.

For example, to enable XML protection for an API:

```
<api port="2000">
  <xmlProtection/>
  …
</api>
```

This enables default protection settings that block common XML threats such as excessive attributes or malicious entity declarations.

You can also customize individual checks as needed:

```
<xmlProtection maxAttributeCount="10"
               maxElementNameLength="50"
               removeDTD="yes"/>
```

Resources

xmlProtection Reference
https://www.membrane-api.io/docs/current/xmlProtection.html

33.3 GraphQL Protection

GraphQL offers great flexibility, but that flexibility can be a double-edged sword when it comes to security. To help mitigate risks, Membrane includes a **GraphQL protection** feature that **validates incoming queries and mutations** against the GraphQL specification.

To enable it, simply include the following element in the configuration:

```
<api port="2000">
  <graphQLProtection/>
  …
</api>
```

This activates structural validation and enforces limits such as **maximum query depth**, helping prevent abuse through overly complex or deeply nested requests.

Resources

For configuration samples, see the `examples/security/graphql-validation` folder the Membrane distribution.

graphQLProtection Reference
https://www.membrane-api.io/docs/current/graphQLProtection.html

34 Basic Authentication

Basic Authentication is almost as old as the web itself. It's simple and widely supported, but it comes with some limitations. Credentials are sent in a Base64-encoded format, which means that without encryption, anyone intercepting the traffic could read the username and password as easily as reading an open book.

That said, when used **together with TLS**, Basic Authentication becomes a lightweight and practical solution. While it doesn't offer the same level of protection as API keys or JWT, it still has its place, especially in internal communication**,** or **simple use cases** where stronger mechanisms would be overkill.

How it Works

Basic Authentication sends credentials using the `Authorization` header, along with the keyword `Basic`. For example:

```
GET http://localhost:2000
Authorization: Basic ZnJlZG86YWJjMTIz
```

Curious what's inside that string? You can decode it with (command my vary):

```
echo ZnJlZG86YWJjMTIz | base64 -d
```

And get something like:

```
fredo:abc123%
```

This shows that **Base64 is not encryption**, it's just encoding. Think of it as putting your credentials in a paper bag rather than a locked box. This is why **TLS is mandatory** when using Basic Auth over the open internet.

If you're experimenting, you can also use an online decoder like:

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. It's like handing your house keys to random strangers.

Once the request reaches the server or API Gateway, the `Authorization` header is decoded, and the credentials are verified. Because Basic Authentication is stateless, the client needs to send the same header on every request. There's no session or token involved, just repeat, repeat, repeat.

Setting up Basic Authentication

The following API configuration shows how to protect an endpoint using **Basic Authentication**. Clients must provide a valid username and password to access the backend service:

```
<api port="2001" name="Fruitshop">
  <basicAuthentication>
    <user username="fredo" password="abc123"/>
    <user username="kate" password="flower25"/>
    <user username="john"
          password="$6$OO0ThN.5$ARC/SklqDFfI0hEa7A.A0f0h..."/>
  </basicAuthentication>
  <target url="https://api.predic8.de"/>
</api>
```

In this example:

- **fredo** and **kate** have their passwords defined in plaintext.
- **john** uses a **hashed password**. This digest ensures that even if someone views the configuration file, the actual password remains concealed.

🔐 **Security** Tip: Storing hashs
In production environments, always use hashed passwords and store them securely, plaintext credentials should be avoided.

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 some clear limitations:

- **Configuration Reload**
  Every time a user is added or removed, the API Gateway configuration needs to be reloaded.
- **Scalability**
  Managing large numbers of users in a single config 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**

This allows you to manage users more flexibly, automate updates, and better align with existing authentication infrastructure.

How to Generate a `.htpasswd` File

`.htpasswd` files are a **pseudo-standard** for storing usernames and hashed passwords, commonly used for Basic Authentication. You can generate entries using the `htpasswd` tool, which is part of the **Apache HTTP Server utilities**.

To create a new file and add the first user:

```
htpasswd -c .htpasswd alice
```

To add another user later:

```
htpasswd .htpasswd bob
```

The resulting file might look like this:

```
alice:$apr1$zGXp9.Px$uLogsgPwJoMVIWtA0Uv76.
bob:$apr1$ciDjPlh3$S2qaMyAl43SVswoCnscNz/
```

Resources

You can find working examples in the Membrane distribution:

1. `examples/security/basic-auth/simple` – using `.htpasswd`
2. `examples/security/basic-auth/database` – using a database as the user store

35 API Keys

Membrane offers support for API key authentication, allowing you to protect endpoints with minimal effort. In the example below, API keys are stored directly in the configuration file and expected to be transmitted via an HTTP header.

Here, API key validation is configured globally using the `global` section, so it automatically applies to all defined APIs:

```
<router>

  <global>
    <apiKey>
      <keys>
        <secret value="aed8bcc4-7c83-44d5-8789-21e24ac873" />
        <secret value="abc123" />
       </keys>
       <headerExtractor name="X-Api-Key"/>
    </apiKey>
  </global>

  <api port="2000">
    <target url="https://api.predic8.de"/>
  </api>

</router>
```

To authenticate, clients must include a valid API key in the `X-Api-Key` header:

```
GET http://localhost:2000
X-Api-Key: abc123
```

If the provided key does not match any of the configured secrets, the request is rejected with a `401 Unauthorized` response.

This global approach ensures consistent enforcement of API key authentication across all endpoints, without needing to duplicate configuration for each API.

API Key Extraction from a Query Parameter

API keys can be passed through different parts of a client's request, such as **HTTP headers**, **query parameters**, or even the **message body**. Membrane supports flexible, pluggable extractors that allow you to retrieve the key from different locations.

To extract the key from the query string, use:

```
<queryParamExtractor name="api-key"/>
```

This allows clients to include the key directly in the URL:

```
GET /products?api-key=abc123
```

While this method is convenient, it is generally not recommended for sensitive data.

⚠ **Security** Warning: **API Keys in Query Strings**
Passing sensitive data in the query string is generally discouraged. Query strings are often logged by proxies, gateways, and backend servers—potentially exposing the API key.

Extracting from an HTTP Header

A more common and secure approach is to send the API key via an HTTP header. You can extract it using:

```
<headerExtractor name="X-Api-Key"/>
```

This method is better aligned with security best practices and avoids the risks associated with query string logging.

Extracting API Keys from JSON, XML, and beyond

For even more flexibility, Membrane supports **expression-based extractors**. These let you extract keys from almost anywhere in the request, including JSON or XML payloads.

To extract a key from a JSON body using **JSONPath**, use:

```
<expressionExtractor expression="$.key"
                     language="jsonpath"/>
```

In this case, the client includes the key in a JSON payload:

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

{
  "key": "abc123"
}
```

🔐 **Tip:** Regardless of the method, always consider the risk of exposure and apply encryption (TLS) and logging controls to protect API keys in transit.

35.1 Storing API Keys in a relational Database

Managing API keys directly in configuration files works fine for small setups, but it doesn't scale well. As soon as you need to rotate keys, revoke access, or support dynamic provisioning, a database-backed key store becomes the better choice.

With Membrane, you can store and manage API keys in a relational database like PostgreSQL. This lets you decouple authentication data from configuration and makes automation much easier.

The example below shows how to configure Membrane to use PostgreSQL. Spring's XML syntax is used to define the data source and link it to Membrane's API key module:

```
<spring:beans ...>

  <spring:bean id="ds"
          class="org.apache.commons.dbcp2.BasicDataSource">
    <spring:property name="driverClassName"
                     value="org.postgresql.Driver"/>
    <spring:property name="url"
          value="jdbc:postgresql://localhost:5432/postgres"/>
    <spring:property name="username"
                     value="user"/>
    <spring:property name="password"
                     value="password"/>
  </spring:bean>

  <router>

    <api port="2000">
      <apiKey>
        <databaseApiKeyStore datasource="ds">
          <keyTable>key</keyTable>
          <scopeTable>scope</scopeTable>
        </databaseApiKeyStore>
        <headerExtractor />
      </apiKey>
      <target url="https://api.predic8.de"/>
    </api>

  </router>

</spring:beans>
```

The configuration above connects to a local PostgreSQL database and uses two tables:

- `key` for storing the actual API keys
- `scope` for defining the scopes associated with each key

If these tables don't exist yet, Membrane will create them for you automatically.

MongoDB API Key Store

In addition to relational databases, Membrane also supports storing API keys in a MongoDB database. This is especially helpful when scaling to a large number of users or distributed gateway instances.

The configuration below shows how to use a MongoDB collection as the backing store for API keys:

```
<apiKey>
  <mongoDBApiKeyStore connection="mongodb://localhost:27017/"
                      database="apiKeyDB"
                      collection="apikey" />
  <headerExtractor />
</apiKey>
```

To populate the collection with example API keys and associated scopes, you can run the following command using `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'] }]);"
```

The scopes are important for role based access control.

35.2 Role-based Access Control (RBAC)

API keys in Membrane can also be used for role-based access control by assigning **scopes** to each key. A scope acts like a role or permission tag. Let’s walk through how this works in practice using a simple file-based setup.

With API keys it is even possible to realize role based access control. For this example we are using a file to store the keys together with scopes. Think of a scope as a kind of role.

Storing API Keys with Scopes

The following `keys.txt` file contains API keys and their associated scopes:

```
abc123: admin, finance
7a26cae9-ed29-40b3-bc99-5b1914bb8498: read, write
```

Here, `abc123` has the roles `admin` and `finance`, while the UUID key is assigned the finer-grained `read` and `write` scopes.

To make this file accessible across multiple APIs, define the API key store outside the `<router>` element:

```
<spring:beans ...>

  <apiKeyFileStore location="./keys.txt" />

  <router>

    <api …>
      …
    </api>

    <api …>
      …
    </api>

  </router>

</spring:beans>
```

Enforcing Scopes

In the API configuration below, the gateway checks whether the client possesses the `admin` scope. If not, it rejects the request with a **403 Forbidden**:

```
<api port="2000">

  <apiKey>
     <headerExtractor name="X-Api-Key" />
  </apiKey>

  <if test="!hasScope('admin')">
    <static>Only admins!</static>
    <return statusCode="403"/>
  </if>

  <setHeader name="X-Scopes" value="${scopes()}"/>
  <target host="localhost" port="3000" />
</api>
```

This configuration does two things:

- **Enforces** that only clients with the `admin` scope are allowed to access the endpoint.
- **Forwards** the list of granted scopes to the backend in a custom `X-Scopes` HTTP header.

Backend Awareness of Scopes

The backend service (on port 3000) simply logs the incoming scopes. But in a real-world scenario, it could inspect the `X-Scopes` header to **perform fine-grained authorization**:

```
<api port="3000" name="backend">
  <request>
    <log message="Scopes: ${headers['X-Scopes']}"/>
  </request>
  <return />
</api>
```

Security Hint: Protecting the backend
Make sure the backend is not directly accessible. Use **TLS client certificates**, **firewall rules**, or **trusted IP allowlists** to ensure only the gateway can reach it.

35.3 Best Practices for API Keys and Roles

API keys are a simple yet effective way to secure APIs. And while they don't offer the fine-grained control of OAuth2 or JWTs, they're often good enough, as long as you follow a few best practices:

- **Always use TLS.**
  API keys should never travel over plain HTTP.
- **Don't hardcode keys in the configuration.**
  Instead, load them from environment variables, external files, or a database-backed store. This makes your setup more secure *and* easier to manage.
- **Never pass API keys to the backend.**
  Keep them at the gateway. If the backend needs to know the client's access rights, pass sanitized metadata, like an `X-Scopes` or `X-Role` header, instead.
- **Use long, unguessable keys.**
  A randomly generated UUID is a good choice. Avoid short or predictable values like `test123`.

Security Tip: **API Key Rotation**
Rotate your API keys regularly, especially for public or long-lived clients. It's an easy way to limit the damage if a key ever gets leaked.

36 JSON Web Tokens

JSON Web Tokens (JWTs) are widely used in modern API security. And Membrane API Gateway is ready to work with them on both ends of the equation.

It can act as:

- a **JWT issuer**, generating tokens for clients after successful authentication
- a **JWT verifier**, checking incoming tokens before forwarding requests to backend services

The next two sections will walk through a complete bearer token flow (as introduced in chapter 14.2), demonstrating both roles using Membrane. In general, Membrane can be used for both roles, or either one of them.

36.1 Issuing JWTs

Membrane can be setup as a JWT issuer, allowing it to generate signed tokens for clients.

A more complex example will be shown in the next section.

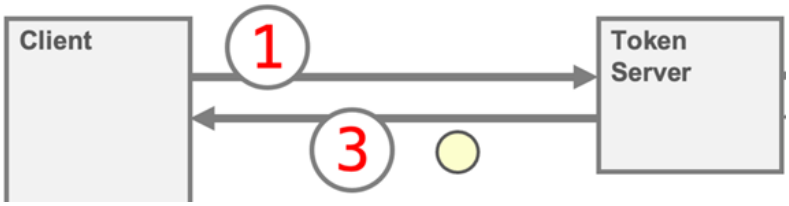


Image: The first steps of the Bearer process Membrane will realize here

Since JWTs carry a cryptographic signature, you'll need a secret key to sign them. For stronger security, it's best to use an asymmetric RSA key pair.

Step 1: Generate a Private Key

You can generate a new RSA private key in JWK format using Membrane's built-in tooling:

- On Windows:

```
membrane.cmd generate-jwk -o demo.jwk
```

- On Linux:

```
membrane.sh generate-jwk -o demo.jwk
```

This creates a file called demo.jwk containing your private key. **Keep this file secret!** It's the foundation of your token security.

Step 2: Configure Membrane to Issue Tokens

Place the `demo.jwk` file in the same directory as your `proxies.xml` file. Then, configure Membrane like this:

```
<api port="2000" name="Token Server">
  <path>/token</path>
  <request>
    <template>
      {
        "sub": "user@predic8.de",
        "aud": "order"
      }
    </template>
    <jwtSign>
      <jwk location="jwk.json"/>
    </jwtSign>
  </request>
  <return />
</api>
```

Now, when you send a request like:

```
GET /token
```

Membrane will return a freshly minted JWT, for example:

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtpZCI6IjhubHFiNXJ1dGJyYT
k5NzV1NWYwbzM3NTJoIn0.eyJzdWIiOiJ1c2VyQHByZWRpYzguZGUiLCJhdWQi
OiJvcmRlciIsImlhdCI6MTc1MDkzMDUyOSwiZXhwIjoxNzUwOTMwODI5LCJuYm
YiOjE3NTA5MzA0MDl9.WO9ZnO83yjBsefzTAkdLUDxW4pRXNNHFZog6JZNVFYl
y6zUmMn1dYIMy79sBdFBPQ1KA5q6Vy3iVyFAYWSXo6Emb9MGwl5DGL2WlCifrg
UGRJWhrPhtoImYXkj10HWOScEBAZWICC1esJbCTbxGgQr2X1MZk4h0as700Ou0
WSNo-Cyxi2798V69oYwF0i1ALJsVEtTXYOw3k3PU3sMa_g3i3JaUT7I9lnNj5D
p1Wc7B5fpilstVGP1Tt8eTlHhxsafCAArMjBIdhXYUf2KEQp5eKLla-51hKsPr
C3zBnybofqnxtJuOQDqemKYX2aDmf8RUyxledJnnAyWQa90ZGmg
```

Congratulations, you have successfully issued your first JWT!

When decoded and pretty printed, it looks like this:

```
{"alg":"RS256","kid":"membrane"}
.
{
  "sub":"user@predic8.de",
  "aud":"order",
  "iat":1750930529,
  "exp":1750930829,
  "nbf":1750930409
}
.
<signature>
```

where `<signature>` is binary data.

The token's payload was formed from the configuration template (`sub` and `aud` claims) as well as dynamically added timestamps (`iat` , `exp` and `nbf` claims).

Every second, you will therefore get another token.

Of course, each token will look slightly different depending on the time it was issued and the claims it contains. Since the signature is based on both the header and payload, even small changes will result in a different signature.

36.2 Protecting the Token Generation Process

In practice, handing out tokens to anyone who calls your token endpoint is not a great idea. To secure your token generation process, you should combine it with an additional layer of authentication, such as API keys.

One simple approach is to use the API keys stored in the file `keys.txt` as described in chapter 35.

Of course, 'static' tokens with the payload `{"sub": "user@predic8.de", aud": "order" }` are only so much fun. We therefore configure the payload to include a `"scope"` claim *based on which API key was used to get the token*.

Alternatively, you could even use API Orchestration to get user information from a remote Identity and Access Management (IAM) API and place it into the template.

Here's how you can configure Membrane to require an API key before issuing a token:

```
<api port="2000" name="Token Server">
  <apiKey required="true">
    <apiKeyFileStore location="demo-keys.txt" />
    <headerExtractor />
  </apiKey>
  <request>
    <setProperty name="scopes" value="${scopes()}"/>
    <template>
      {
        "sub": "user@example.com",
        "aud": "order",
        "scope": "${property.scopes}"
      }
    </template>
    <jwtSign>
      <jwk location="jwk.json"/>
    </jwtSign>
  </request>
  <return />
</api>
```

This configuration ensures that:

- Only callers presenting a valid API key (from `keys.txt`) will receive a token.
- The token's payload will include a scope claim, such as:

  ```
  "scope": ["read", "write"]
  ```

  or

  ```
  "scope": ["admin", "finance"]
  ```

  depending on the API key used.

This setup already gives you a lightweight token server, which is sufficient for many API use cases.

💡 **Sidenote**: **Need to scale?**
If your requirements grow, you can easily swap out Membrane's token generation with a more full-featured identity provider like **Keycloak**, **Microsoft Entra ID**, **AWS Cognito**, or others.

36.3 Verifying JWTs

Membrane can also be set up as a **JWT verifier**, sitting between the client and the API to strengthen your overall security posture.

Verifying tokens at the API gateway is **strongly recommended**. Why? Because it's easier to verify, tweak, and upgrade one gateway than to manage token verification logic across dozens, or hundreds, of APIs.

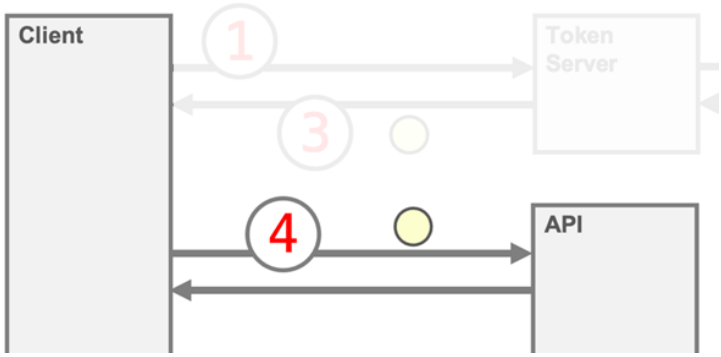


Image: The last steps from the Bearer process Membrane will realize here

Step 1: Set Up a Demo API

Let's start with a simple backend API that returns protected content:

```
<api port="3000" name="Demo API">
  <template>
    {
      "content": "Protected content!"
    }
  </template>
  <return />
</api>
```

Step 2: Protect the API with Membrane

Now, place Membrane in front of the API and configure it to verify JWTs:

```
<api port="4000" name="Secured Access to Demo API">
  <jwtAuth expectedAud="order">
    <jwks>
      <jwk location="jwk.json" />
    </jwks>
  </jwtAuth>
  <target host="localhost" port="3000" />
</api>
```

This setup ensures that only requests with valid JWTs—signed with the correct key and intended for the "order" audience—are forwarded to the backend.

⚠️ **Security Tip:**
In production, make sure the backend API (localhost:3000) is not directly accessible. 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 and unexpired, you'll get:

```
{
  "content": "Protected content!"
}
```

But repeating the same request 5 minutes later, you'll receive a "HTTP/1.1 400 Bad Request":

```
{
  "title" : "Security error.",
  "type" : "https://membrane-api.io/problems/security",
  "detail" : "JWT validation failed.",
  …
}
```

Your token has expired. You can either request a new one from the issuer or increase the token's validity period, though shorter lifespans are generally safer.

36.4 JWT Best Practices

The aud Claim

In the setup described above, the configuration files of both the JWT issuer and the JWT verifier must align.

On the issuer side, the configuration includes

```
"aud": "order"
```

On the verifier side, Membrane is configured with:

```
<jwtAuth expectedAud="order">
```

If these values don't match, token verification will fail, and the token will be rejected.

In a small deployment—like this 1:1 demo with one issuer and one verifier—this constraint might seem like a minor detail. But as your architecture scales (think 1 issuer and 1000 verifiers), consistency in the aud claim becomes critical. It ensures that tokens are only accepted by the services they were intended for, reducing the risk of misuse or accidental exposure.

Don't share the private Key

In the previous chapters, 36.1 (issuer) and 36.3 (verifier), we used the same RSA private key file for simplicity. But in practice, **you should never share the private key file** between systems. The private key should remain on the issuer's machine.

Instead, convert the private key into a **public key** and use that for verification.

The private key should remain securely stored on the issuer's machine. The verifier only needs the public part of the key to verify tokens. This separation ensures that even if the verifier's configuration is leaked, no sensitive signing material is exposed: Itt only contains public data.

Using JWT Claims as HTTP Headers

In some scenarios, it's helpful to extract specific claims from a JWT and forward them as HTTP headers. This allows downstream services to make decisions based on identity or roles without needing to parse the token themselves.

Let's say you want to forward the `sub` (subject) claim as a header.

Here's how you can do it in Membrane:

```
<jwtAuth expectedAud="order">
  <jwks>
    <jwk location="jwk.json" />
  </jwks>
</jwtAuth>
<setHeader name="X-Sub" value="${property.jwt.sub}" />
<log/>
```

This configuration does two things:

1. Verifies the JWT using the public key in jwk.json and checks that the aud claim matches "order".
2. Extracts the sub claim from the token and sets it as an HTTP header named X-Sub.

When a request is processed, you'll see something like this in the log:

```
X-Sub: user@predic8.de
```

This approach is especially useful when your backend services are not JWT-aware but still need to know who the user is.

⚠ **Security Tip:**
You still need to ensure that headers like X-Sub cannot be forged by clients.

37 OAuth2 and OpenID Connect

APIs are everywhere, and so are the people and systems trying to access them. Whether it's a mobile app, a browser, or a backend service, they all need a way to prove who they are and what they're allowed to do. That's where OAuth2 and OpenID Connect (OIDC) come in.

These protocols provide a standardized way to handle authentication and authorization across distributed systems. Instead of reinventing the wheel for every service, you can rely on a well-established framework that's secure, flexible, and widely supported.

And for API Gateways? OAuth2 and OIDC are essential features. An API Gateway typically supports two main use cases:

1. **Token Verification**
   The gateway checks access tokens on incoming API requests. Together with the token format, this is technically outside the OAuth2/OIDC spec, but it's a good common practice to centralize token verification at the gateway.
2. **Token Acquisition (Web Context)**
   In browser-based scenarios, the gateway can handle the OAuth2 flow on behalf of the user. It redirects the user to the Authorization Server, retrieves the token, and attaches it to API requests. This keeps tokens out of the browser and improving security.

In both cases, the backend services can use the token to drive authentication and authorization.

We'll configure both use cases in the upcoming chapters.

37.1 Token Verification

An API Gateway can **verify** incoming OAuth2/OIDC **access tokens** before forwarding the HTTP request to the backend API.

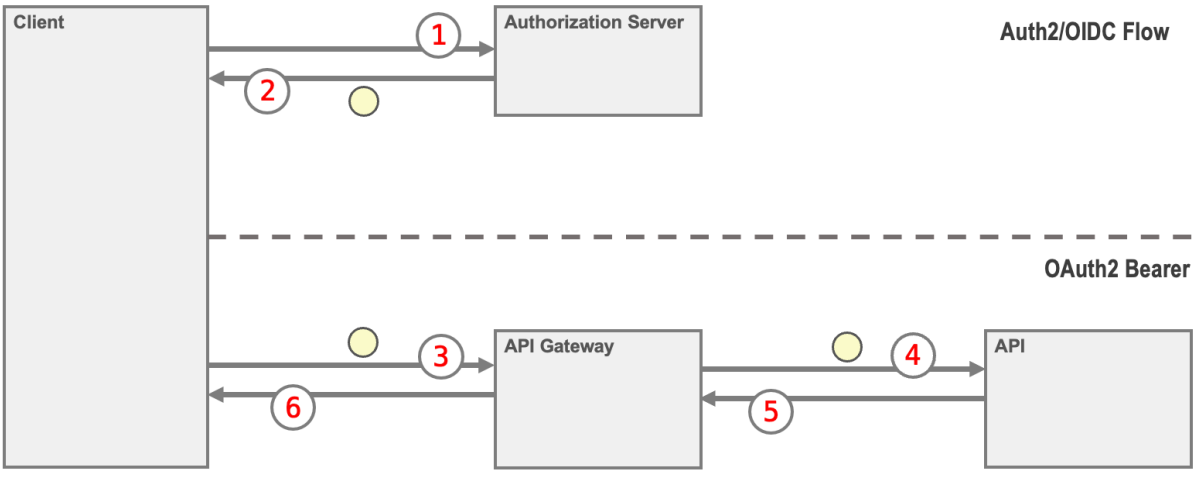


Image: Membrane API Gateway verifying OAuth2 access tokens

The simplest way to implement this is by configuring your Authorization Server to issue JWTs (JSON Web Tokens) as access tokens. In that case, the gateway's job becomes similar to what we described in chapter 36.3. Often, the configuration can be even more streamlined thanks to OIDC Discovery support from the Authorization Server.

Membrane includes an example setup in `examples/security/oauth2/azure-ad-with-jwts`, where the README walks through using Microsoft's Entra ID as the Authorization Server.

Here's a snippet of how JWT token verification is configured:

```
<jwtAuth expectedAud="api://2axxxx16-xxxx-xxxx-faxxxxxxxxf0">
  <jwks jwksUris="https://login.microsoftonli
                           ne.com/common/discovery/keys" />
</jwtAuth>
```

Traditionally, you'd need to manually configure a public key to verify token signatures. But with OIDC Discovery, the Authorization Server's public keys can be fetched automatically. Microsoft, for example, publishes its keys at a fixed URL:

```
https://login.microsoftonline.com/common/discovery/keys
```

Note: This URL may vary depending on your Azure tenant setup.

The audience value api://2axxxx16-xxxx-xxxx-faxxxxxxxxf0 refers to what Microsoft calls the *Application ID of the App Registration*. It's created by the administrator when registering the API in Azure.

Flow Overview

Step 1: Token Acquisition
The client obtains a token from Microsoft.

Step 2: API Request
The client sends an API request to the gateway, attaching the token in the HTTP header:

```
Authorization: Bearer ...token...
```

In this role, Membrane API Gateway only allows requests that:

- carry tokens signed by Microsoft
- carry tokens with the correct audience (aud) value.

Sidenote: Not just Entra ID—and not just JWKS
Membrane's JWT verification works with any standards-compliant OAuth2/OIDC provider. JWKS and Discovery are convenient, but optional. You can also configure public keys manually if your provider doesn't support Discovery or if you prefer tighter control. Whether you're using Entra ID, Auth0, Keycloak, or something custom, the same principles apply.

37.2 Authorization Code Flow

When operating in a web context, Membrane API Gateway can **acquire an access token on behalf of the user**.

In this setup, Membrane identifies the user and his browser by a session cookie. While session cookies aren't part of the OAuth2 specification, they're a practical and widely used mechanism to track login state in web applications.

Flow Overview

Step 1: Token Acquisition
The user's browser accesses the API Gateway without being logged in. The gateway detects this and initiates an OAuth2 Authorization Code Flow. Once the flow completes successfully, the user is considered "logged in," and the session is associated with an access token.

Step 2: API Request
The browser sends an API request to the gateway, including the session cookie. The gateway looks up the session, retrieves the corresponding access token, and attaches it to the request before forwarding it to the backend.

This approach keeps access tokens out of the browser's JavaScript context, which reduces the risk of token leakage through XSS attacks. It also allows the gateway to enforce consistent token handling across all requests.

Sidenote: Why use the Authorization Code Flow?
The Authorization Code Flow is designed for apps running in a browser. It separates the user's credentials from the app and allows secure token exchange on the server side. When combined with Proof Key for Code Exchange (PKCE), it's the most secure OAuth2 flow for public clients like SPAs and mobile apps.

38 Legacy Integration SOAP Web Services

Software doesn't come with an expiration date, but some systems definitely overstay their welcome. Legacy protocols, especially XML-based formats like SOAP and Web Services, are still widely used in enterprise environments. While modern APIs mostly speak JSON, the old guard hasn't left the building.

The challenge? Making the old and new work together.

The good news is that if the legacy system communicates over HTTP, it can be routed through an API Gateway. This allows you to apply gateway features, routing, authentication, logging, transformation, even if the backend still lives in 2006.

Legacy Support

Some API gateways, like Membrane, go beyond simple HTTP forwarding. They offer built-in capabilities for:

- Handling **SOAP** requests and responses
- Performing **XML-to-JSON** and **JSON-to-XML** transformations
- Validating XML messages against **XSD schemas and WSDL documents**
- Routing based on **SOAP action** or **WS-Addressing headers**

Migration to APIs

These features allow you to expose a modern, JSON-friendly REST API on the outside while continuing to use an XML-based interface on the inside. You get the best of both worlds: a stable backend and a flexible, modern API facade.

💡 **Sidenote: Modernizing Web Services**
If you're modernizing step by step, an API Gateway can act as a protocol adapter, translating requests between REST and SOAP without touching the legacy backend.

In short, an API Gateway can serve as a **modernization bridge**. Instead of ripping out your legacy systems, you can wrap them with a modern API layer and evolve at your own pace.

38.1 Sample Web Services

For testing and development, it's often helpful to have a simple Web Service available. Membrane includes a plugin called `sampleSoapService` that provides a basic SOAP emulator—perfect for quick testing scenarios.

By deploying the following API configuration:

```
<api port="2000">
  <path>/city-service</path>
  <sampleSoapService/>
</api>
```

you can access a WSDL document by visiting:

```
http://localhost:2000/city-service?wsdl
```

This WSDL allows you to use tools like **SOAP UI** to generate and send requests to the service.

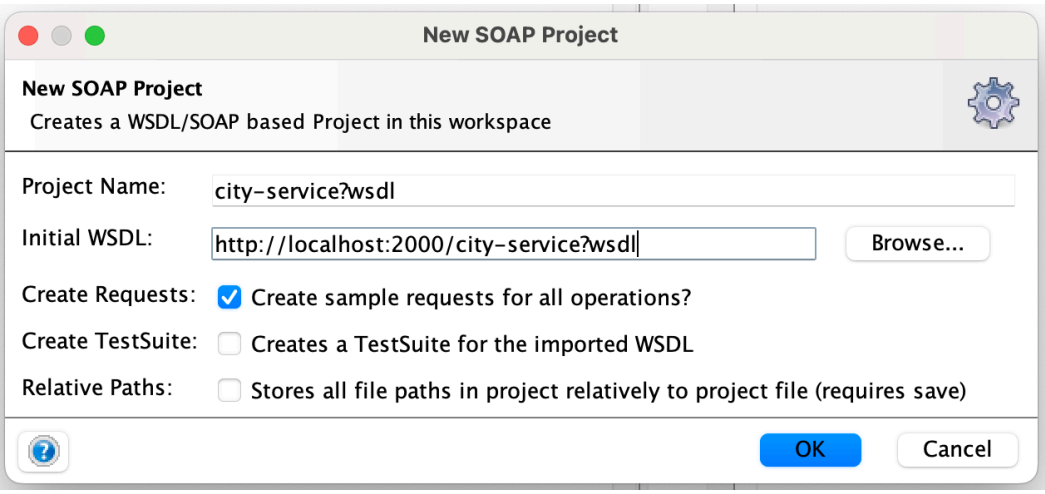


Image: Creating a project from a WSDL document in SOAP UI

You don't need specialized tools to try out the service. You can also use **curl**, or the **REST client extension in Visual Studio Code**. Here's an example request using raw XML:

```
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 envelope like this:

```
HTTP/1.1 200 Ok
Server: Membrane API Gateway
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 your API gateway doesn't offer built-in SOAP support, you can often emulate similar behavior using templates, routing, and content transformation. That's exactly what the next section will cover.

38.2 Mocking a Web Service

You can simulate a Web Service by manually crafting a SOAP response. The example below shows how Membrane can return a static SOAP body when a client sends a request to `/mock-service`:

```
<api port="2000">
  <path>/mock-service</path>
  <response>
    <static pretty="true">
      <![CDATA[
      <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>
      ]]>
    </static>
  </response>
  <return/>
</api>
```

When this API is called, it returns a SOAP message. This is useful for testing or demonstration purposes, allowing you to simulate legacy services without having to set up a full-fledged SOAP backend.

💡**Sidenote**: **What is `CDATA`?**
In XML configurations, `<![CDATA[ ... ]]>` marks a **CDATA section** (Character Data), which tells the XML parser to treat the enclosed content as plain text—even if it contains characters like `<`, `>`, or `&` that would normally be interpreted as XML.

Example:

```
<![CDATA[
  <country>England</country>
]]>
```

Without CDATA, the parser would try to interpret `<country>` as an actual XML tag. Using CDATA lets you safely embed raw XML or HTML fragments inside configuration files.

38.2.1 soapBody Template

When working with Web Services, Membrane provides a helpful `<soapBody>` element that makes crafting SOAP responses easier and cleaner. It automatically generates the necessary SOAP envelope and body tags around your payload.

The example from the previous section can be simplified like this:

```
<api port="2000">
  <path>/mock-service</path>
  <response>
    <soapBody pretty="true">
      <![CDATA[
        <getCityResponse xmlns="https://predic8.de/cities">
          <country>England</country>
          <population>8980000</population>
        </getCityResponse>
      ]]>
    </soapBody>
  </response>
  <return/>
</api>
```

By using `<soapBody>`, you only provide the payload, the gateway handles the rest. It wraps your content in a proper SOAP envelope with the required namespaces, so you can focus on what matters: the actual response data.

38.3 Exposing SOAP Web Services as REST APIs

Old and clunky SOAP-based Web Services can get a second life by exposing them as modern REST-style JSON APIs. One effective strategy for doing this is using request and response templates at the gateway.

This approach allows to:

- Accept RESTful calls with JSON payloads from clients
- Transform the requests into SOAP messages internally
- Forward them to a legacy backend
- Convert the SOAP responses back into clean JSON before returning them to the client

All this happens without changing the backend service.

This technique is especially helpful when you're modernizing incrementally or working with a backend managed by a third party.

Template based

Here's an example of how Membrane API Gateway can expose a SOAP web service as a RESTful endpoint using templates:

```
<api port="2000" method="GET">
  <path>/cities/{city}</path>
  <request>
    <soapBody>
      <![CDATA[
        <getCity xmlns="https://predic8.de/cities">
        <name>${pathParam['city']}</name>
        </getCity>
      ]]>
    </soapBody>
    <setHeader name="SOAPAction"
               value="https://predic8.de/cities/get"/>
  </request>
  <response>
    <template contentType="application/json">
      {
        "country": "${property.country}",
        "population": "${property.population}"
      }
    </template>
    <setProperty name="country"
                 value="${//country}"
                 language="xpath"/>
    <setProperty name="population"
                 value="${//population}"
                 language="xpath"/>
  </response>
  <target method="POST"
          url="https://www.predic8.de/city-service"/>
</api>
```

The image shows the details of the request and response transformations:

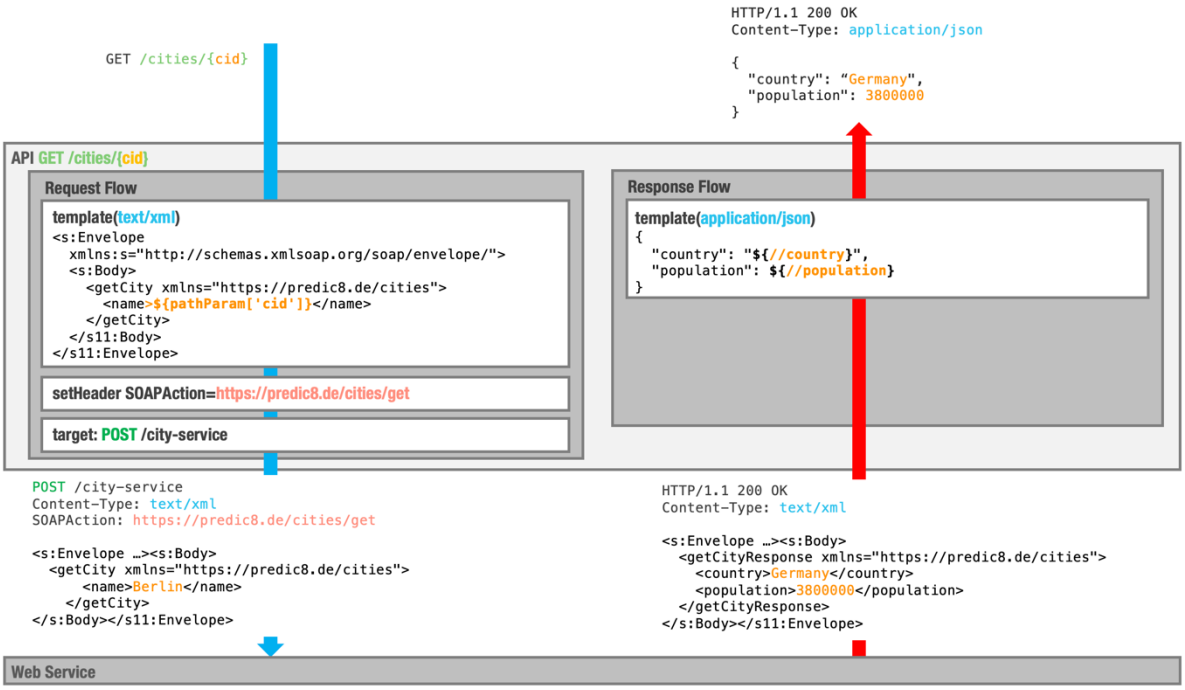


Image: Mapping a SOAP Operation to a RESTful GET Resource

This configuration:

- 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 values such as `country` and `population` using XPath
- Returns a clean JSON response to the client

Some SOAP implementations rely on the `SOAPAction` HTTP header to route the request to the correct operation.

```
SOAPHeader: https://predic8.de/cities
```

Don't forget to include the correct `SOAPAction` in your request. You'll typically find the required value in the WSDL binding section. For example:

```
<wsdl:binding name="CitySoapBinding" type="cs:CityPort">
  <s:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"></s:binding>
  <wsdl:operation name="getCity">
    <s:operation soapAction="https://predic8.de/cities"/>
    <wsdl:input>
      <s:body use="literal"
namespace="https://predic8.de/cities"></s:body>
    </wsdl:input>
    <wsdl:output>
      <s:body use="literal"
namespace="https://predic8.de/cities"></s:body>
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
```

Use one API definition per WSDL operation. This keeps each configuration concise and easier to test, maintain, and extend.

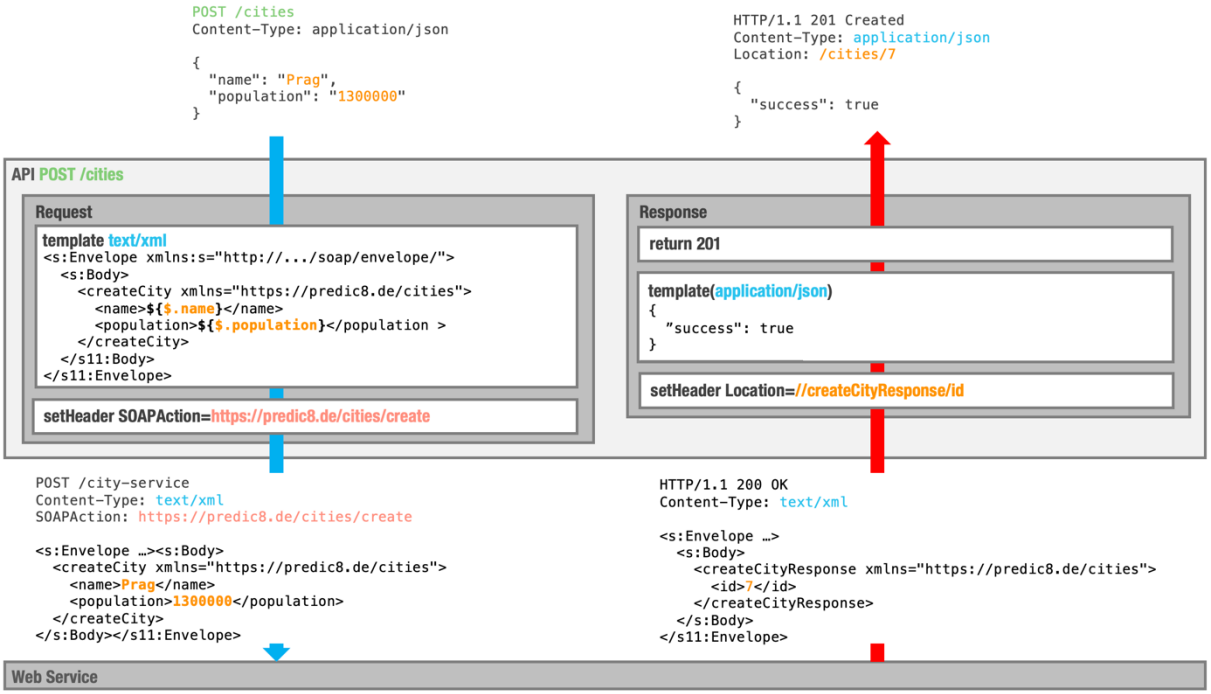


Image: Mapping a SOAP Operation to a RESTful POST Resource

38.4 Proxying SOAP

Membrane offers a convenient shortcut for deploying Web Services: the `soapProxy` plugin. When routing SOAP traffic, you can use `soapProxy` instead of `api` in the `proxies.xml` configuration.

soapProxy

The `soapProxy` simplifies routing and adds SOAP-specific capabilities out of the box:

- WSDL publishing and rewriting
- Request and response validation based on the WSDL
- A Web Services explorer

It's especially useful when you want to expose or proxy legacy SOAP services with minimal configuration effort.

`soapProxy` is just a convenient abbreviation that is internally translated to a `serviceProxy` that is configured for Web Services.

This example sets up a SOAP proxy using a WSDL hosted on a remote server:

```
<soapProxy port="2000"
           wsdl="https://www.predic8.de/city-service?wsdl">
</soapProxy>
```

The soapProxy is completly configured from the information in the WSDL description. That's the nice thing about service description, also about OpenAPI , when there is a description it can make live much easier.

Once started, the WSDL is available locally at:

```
http://localhost:2000/city-service?wsdl
```

WSDL Rewriting

The WSDL is not just served from the gateway, it's rewritten on the fly. Specifically, the `<s:address>` element inside the WSDL is updated to reflect the gateway's address:

```
<wsdl:service name="CityService">
  <wsdl:port name="CityPort" binding="cs:CitySoapBinding">
    <s:address location="http://localhost:2000/city-service">
    </s:address>
  </wsdl:port>
</wsdl:service>
```

By default, Membrane uses the **protocol**, **host**, and **port** from the client's request to rewrite the address dynamically. If the gateway is running behind a firewall, reverse proxy, or within a containerized environment, you can configure a fixed external address using `wsdlRewriter`:

```
<soapProxy port="2000"
           wsdl="https://www.predic8.de/city-service?wsdl">
  <wsdlRewriter port="443"
                protocol="https"
                host="api.predic8.de" />
</soapProxy>
```

This ensures that clients importing the WSDL (e.g., into SOAP UI) are directed to the correct public-facing address of the gateway.

Web Service Explorer

For convenience, Membrane also includes a simple Web Service Explorer. You can access it by sending a GET request to the base service URL:

```
http://localhost:2000/city-service
```

The explorer provides a simple Web interface with key information about the service.

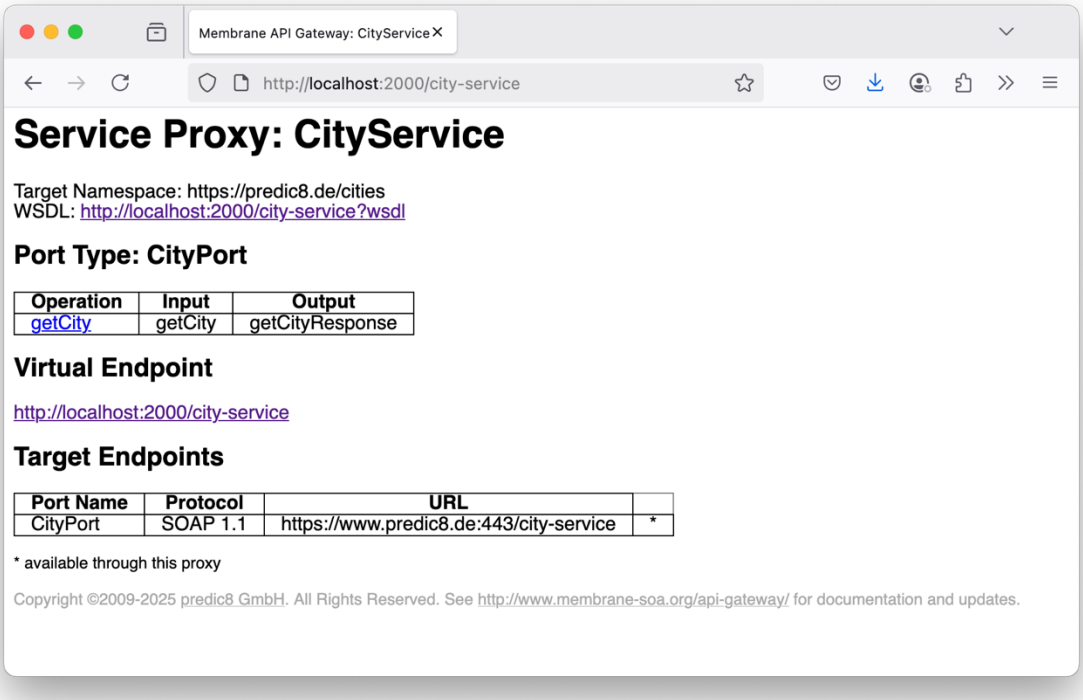


Image: Web Services Explorer

WSDL Validation

The `soapProxy` supports WSDL-based validation. When enabled with the `validation` element, incoming requests and outgoing responses are checked against the WSDL's schema definitions and operation structure:

```
<soapProxy port="2000"
    wsdl="https://www.predic8.de/city-service?wsdl">
  <validation/>
</soapProxy>
```

If a request doesn't match the expected format, the gateway will return a SOAP fault with detailed error information.

An invalid SOAP request like this one:

```
<s11:Envelope ..>
   <s11:Body>
      <cit:getCity>
         <foo/>
      </cit:getCity>
   </s11:Body>
</s11:Envelope>
```

will result in a validation error.

```
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 gives you early feedback on incorrect client requests and helps to ensure reliable contract-based communication with the backend.

39 Operation

Running an API Gateway isn't just about deploying it once and forgetting about it. It’s about operating it effectively, keeping it observable, secure, and maintainable. Whether you're routing a few calls or managing a high-traffic API landscape, daily operations matter. From admin consoles and monitoring tools to access logs and message tracing, this chapter covers the operational features that help you stay in control once your gateway is up and running.

Membrane provides several built-in tools to help you administer, monitor, debug, and observe traffic in real time or retrospectively. These features are not only useful during development and testing but are essential in production environments where insight, visibility, and reliability are key.

39.1 Admin Console

While not strictly necessary for running a gateway, a web-based admin console can make a big difference in day-to-day operations. It gives you a cockpit view of your gateway: you can inspect API configurations, monitor usage, and review the most recent traffic in real time, all from your browser.

Membrane includes its own web console to support exactly that.

Activating the Console

Enabling the console is as simple as configuring an API with the console plugin. In fact, the console itself is implemented as a plugin:

```
<api port="9000">
  <adminConsole/>
</api>
```

Once this configuration is active, point your browser to http://localhost:9000. You’ll be greeted with an overview of all deployed APIs and a live feed of recent requests and responses, perfect for debugging or keeping an eye on traffic.

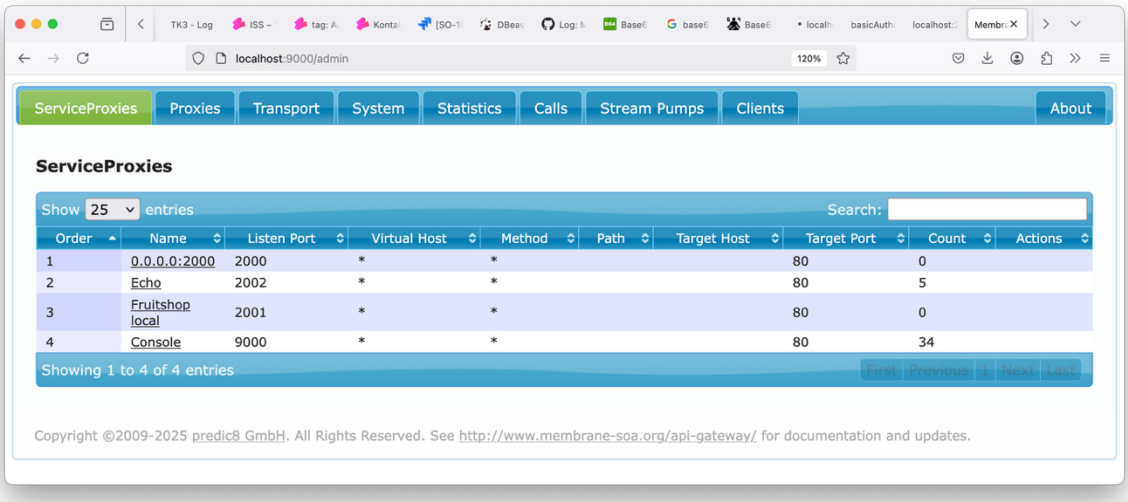


Image: List of APIs in the Admin Console

The **Calls** tab provides key information about the most recent traffic. Activate **Auto Reload** to watch messages pass through the gateway in real time. Membrane keeps the last few messages in memory for inspection, you can configure how much memory should be allocated for this purpose (See section about MessageExchangeStores) below.

Clicking a timestamp reveals the full details of the request and response, including HTTP headers and message body.

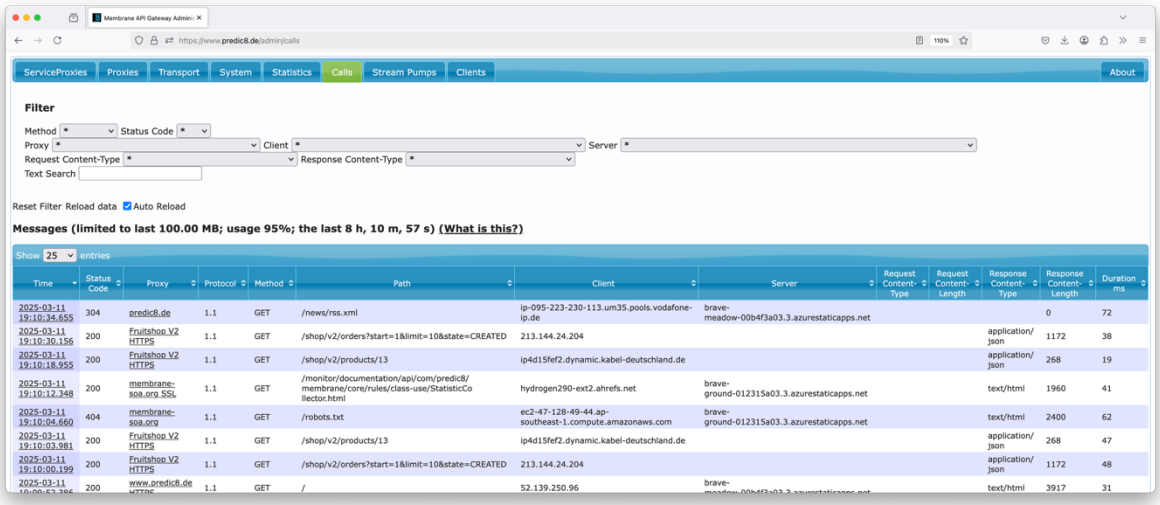


Image: Requests and responses that passed recently the gateway.

Securing the Console

Since the console behaves like any other API, you can secure it in the same way. Use Basic Authentication, API keys, or JWTs, whichever fits your setup.

The example below demonstrates how to protect the console with Basic Authentication and mark it as read-only, to prevent unauthorized changes:

```
<api port="9000">
  <path>/admin</path>
  <basicAuthentication>
    <staticUserDataProvider>
      <user username="mike" password="2472u1984"/>
    </staticUserDataProvider>
  </basicAuthentication>
  <adminConsole readOnly="true"/>
</api>
```

For additional options on securing the console or avoiding plaintext passwords in your config, check the security section of this book.

Security Tip: Secure Console Access
Always secure your admin interfaces. Leaving them unprotected is like leaving your front door wide open, you might not like who walks in.

39.2 Monitoring with Prometheus

To monitor general traffic statistics, you can use Membrane's plugin for Prometheus. The following questions

- How many HTTP requests were received per API?
- How often was which HTTP Status Code returned?
- How long did processing take?
  (Membrane internal processing as well as processing by the backend)

For more details, check out the sample configurations in the `examples/monitoring-tracing/prometheus` directory. While Prometheus handles and stores the raw data, monitoring is often performed by Grafana.

```
<api port="2000" name="prom-metrics">
  <path>/metrics</path>
  <prometheus />
</api>
```

Execute the following command:

```
curl localhost:2000/metrics
```

You get the answer:

```
# TYPE membrane_count counter
membrane_count{rule="prom_metrics",code="200"} 2
# TYPE membrane_good_count counter
membrane_good_count{rule="prom_metrics",code="200"} 0
# TYPE membrane_good_time counter
membrane_good_time{rule="prom_metrics",code="200"} 0
# TYPE membrane_good_bytes_req_body counter
membrane_good_bytes_req_body{rule="prom_metrics",code="200"} 0
# TYPE membrane_good_bytes_res_body counter
membrane_good_bytes_res_body{rule="prom_metrics",code="200"} 0
# TYPE membrane_rule_active gauge
membrane_rule_active{rule="prom_metrics"} 1
# TYPE membrane_duplicate_rule_name gauge
membrane_duplicate_rule_name 0
```

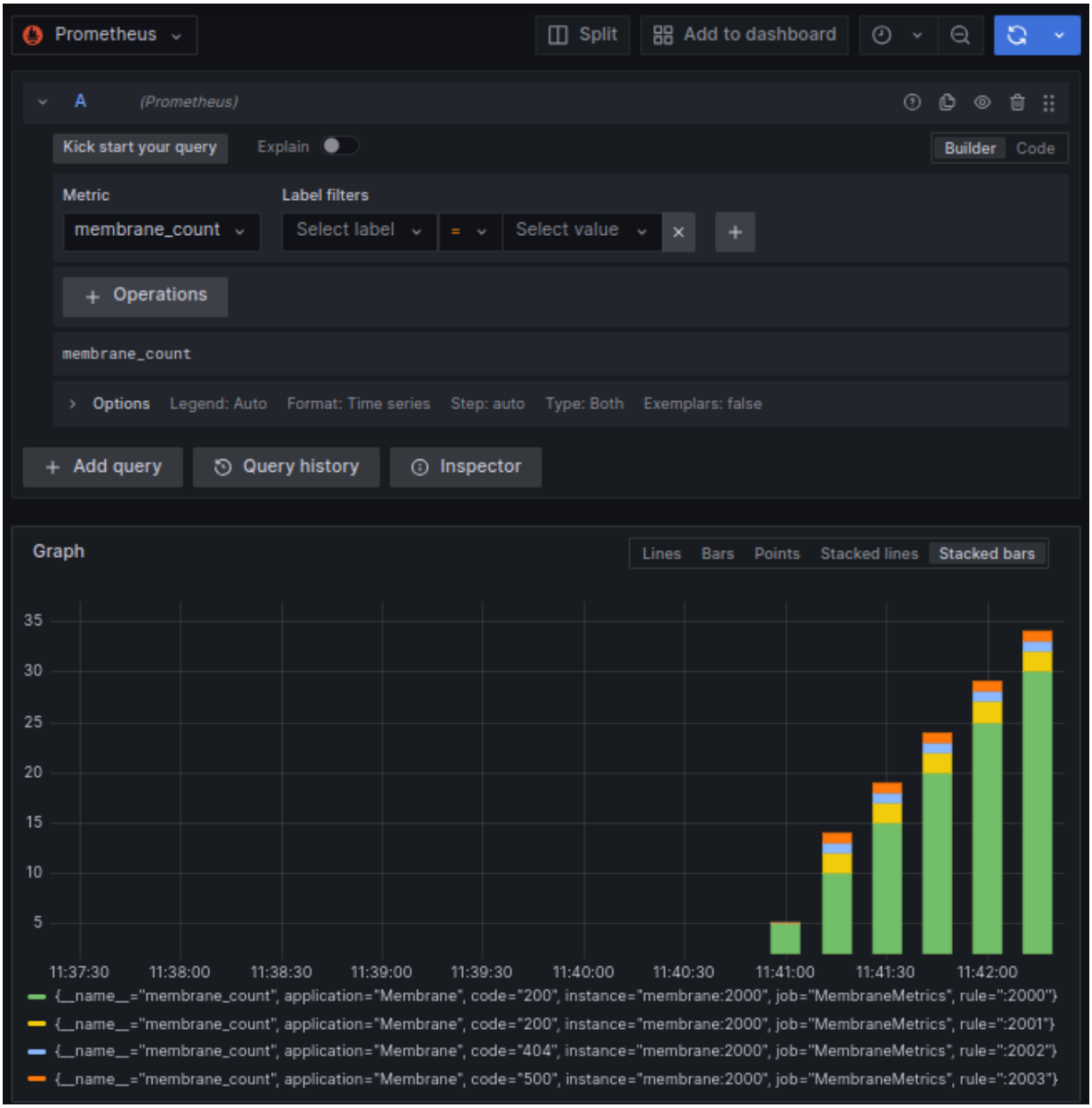


Image: An Administrator in the process of configuring Membrane metrics in Grafana

39.3 Access Log

If you're serving up an API, you want to know who's knocking on your door, and when. Just like web servers have been doing for ages, every request is recorded with details like the timestamp, the client's IP, and the requested path. For REST APIs, it's essentially an audit log that lets you track every "GET" or "POST" like a detective following clues.

The Common Log Format

The Common Log Format is the gold standard for access logs, used by a whole slew of web servers, proxies, and firewalls. Each request gets its own line with these fields:

- **host**: The IP address of the client
- **ident**: Typically not used, but historically for client identity
- **authuser**: The authenticated user, if any
- **date**: The timestamp of the request
- **request**: The HTTP method and resource path
- **status**: The HTTP status code
- **bytes**: The size of the response

A typical access log entries might look like these:

```
192.168.2.81 [11/03/2025:20:25:13 +0100] "GET /shop/v2/orders/
HTTP/1.1" 200 0
192.168.2.81 [11/03/2025:20:25:21 +0100] "GET
/shop/v2/orders/4 HTTP/1.1" 200 0
127.0.0.1 [11/03/2025:20:25:24 +0100] "GET /shop/v2/products/
HTTP/1.1" 200 0
```

Membrane's Access Log

Membrane offers flexible access logging built on the **log4j** Java logging framework. This means you can tailor your log lines to include virtually any piece of data you need—whether it's specific HTTP headers or even fields from a JSON payload.

Resources

Common Log Format, Wikipedia
https://en.wikipedia.org/wiki/Common_Log_Format

Access Log Example
examples/logging/access

39.4 API Tracing

As API communication paths get more complex with an ever-growing number of applications and microservices, it's crucial to know who's calling whom. Often, you won't find up-to-date diagrams to help you trace these calls—it can be like trying to follow breadcrumbs in a labyrinth. Thankfully, modern technologies let you trace how a single API call can trigger an entire call chain that branches out like a sprawling tree.

Sidenote: Think of API tracing as the ultimate game of "telephone" where every whisper is logged so you can see exactly how the message transformed from one service to the next.

39.4.1 OpenTelemetry

When call graphs span multiple applications and APIs it becomes essential that every system involved speaks the same language. Enter OpenTelemetry: a popular open standard that ensures every participating system uses the same protocol to report incoming requests to a central collector. Whether you're using Java, .NET, Python, or another popular platform, there are agents available to instrument your applications and send the necessary communication data to the collector. And yes, Membrane has its very own OpenTelemetry plugin.

Imagine a trace that starts at an API Gateway and extends through three downstream APIs. That's the power of distributed tracing, giving you a complete picture of your call chain.

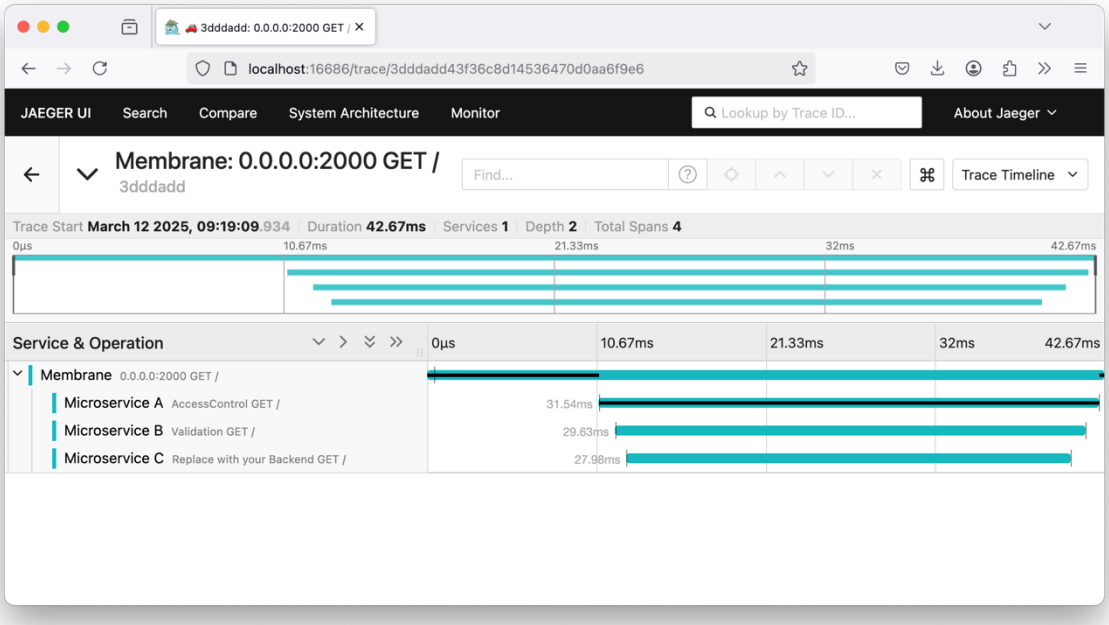


Image: Callgraph from API Gateway over 3 Microservices

OpenTelemetry in Membrane

Getting Membrane to send traces to a collector like **Jaeger** is simple. Just enable the OpenTelemetry plugin, either for a specific API or globally for all deployed APIs. Here's an example configuration:

```
<api port="2000">
  <openTelemetry sampleRate="1.0">
    <otlpExporter host="localhost"
                  port="4317"
                  transport="grpc"/>
  </openTelemetry>
  <target url="http://localhost:2001" />
</api>
```

In this setup, Membrane will automatically send tracing data to a collector. For a more detailed example, check out the contents of the `examples/monitoring-tracing/opentelemetry` folder.

> 💡 **Quick Tip:**
> Distributed tracing can be a lifesaver when troubleshooting performance issues.

Resources

OpenTelemetry @ Cloud Native
https://opentelemetry.io/

Jaeger: open source, distributed tracing platform
https://www.jaegertracing.io/

OpenTelemetry Example
examples/monitoring-tracing/opentelemetry

39.5 Logging and Saving whole Messages

Whether you're debugging, or running APIs in production, it's often helpful to inspect the actual messages flowing through the gateway. Membrane provides multiple ways to log or persist these messages.

Writing Messages to the Log

The most straightforward approach is to log message bodies directly to a file. Simply add the `<log/>` element to an API configuration:

```
<api port="2000">
   <log/>
  ...
</api>
```

To log messages across all APIs, place the logger in the global chain:

```
<global>
   <log/>
</global>
```

Logging only what matters

Instead of logging full payloads, you can extract and log just the relevant parts using expressions. This keeps logs focused and easier to read:

```
<log message="Got: ${body}/>
<log message="Header: ${header}"/>
<log message="Trace: ${header['Trace-Id']}/>
```

To extract fields from structured content:

```
<log message="Id: ${$.id} Date: ${$.date}"
     language="jsonpath"/>
<log message="Id: ${/contract/id} Date: ${/contract/date}"
     language="xpath"/>
```

These filtered logs are well-suited for aggregation and visualization in tools like Elasticsearch or Grafana.

39.5.1 Saving Messages to Stores

For longer-term storage or more advanced inspection, Membrane supports **ExchangeStores**. These can persist full request and response messages in different backends.

In-Memory Store

Membrane comes with a lightweight in-memory message store. It's great for development or real-time debugging in production. You can configure memory limits like so:

```
<limitedMemoryExchangeStore id="store" maxSize="100000000"/>

<router exchangeStore="store">
```

Messages are held in memory and can be inspected via the Admin Console's **Calls** tab. The capacity defines the maximum amount of memory in bytes used for storing exchanges.

File-Based Message Store

To persist messages across restarts, you can use the `fileExchangeStore`. Each message exchange is written as an XML file on disk:

```
<fileExchangeStore id="fs"
                   dir="./exchanges"
                   maxDays="30" />

<router exchangeStore="fs" >
```

Each exchange will be written to a separate file in the specified directory. This is helpful when reproducing bugs or when needing an audit trail for compliance.

MongoDB Message Store

If your team prefers centralized and queryable message storage, Membrane also supports MongoDB as a backend:

```
<mongoDBExchangeStore id="store"
                      connection="mongodb://localhost:27017/"
                      database="membrane"
                      collection="exchanges"/>

<router exchangeStore="store">
```

This configuration is a good fit when integrating message inspection into dashboards or querying logs programmatically.

Resources

File ExchangeStore Example
`examples/extending-membrane/file-exchangestore`

MongoDB ExchangeStore Example
`examples/extending-membrane/mongodb-exchange-store`

40 Gateway Performance

Membrane API Gateway is designed with performance in mind. Its architecture includes several built-in optimizations that help keep latency low and throughput high—even under load.

40.1 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. 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, which makes Membrane blazingly fast and memory-efficient.

However: features like message transformations, and content filtering often require the full body to be loaded and parsed. These features may introduce delays or increase memory usage. So, if performance matters, keep an eye on which filters or plugins you enable.

40.2 Keep-Alive

Membrane uses persistent TCP connections, also known as **HTTP keep-alive**, to reduce the overhead of connection setup. 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.

⚠️ **Heads-up**:
Some filters and features may disable connection reuse, depending on how they handle message bodies or headers. If connection reuse is important in your environment, make sure your configuration doesn't accidentally turn it off.

Interestingly, some users have seen better performance **with** Membrane than without it. In setups where HTTP clients didn't support connection pooling properly, Membrane stepped in, kept connections alive, and acted as a smarter proxy. In these cases, the gateway didn't just avoid slowing things down. It actually sped things up.

Sidenote:
To get a feel for the actual performance of your setup, it's best to measure. Run a few load tests, try toggling certain features, and see how throughput and latency are affected. You might be surprised how much difference one plugin can make.

Thanks For Reading

Thanks for spending time with the API Gateway Handbook. We hope it gave you useful insight, practical patterns, and a clearer picture of how gateways help operating 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'd also love to hear from you. If you have feedback, ideas, or unanswered questions about API Gateways, feel free to reach out.

Cheers,
Thomas Bayer & Tobias Polley

bayer@predic8.de
polley@predic8.de