Building an API Gateway using Node.js

Last updated: Aug 03, 2017

Péter Márton
Co-Founder of RisingStack

Services in a microservices architecture share some common requirements regarding authentication and transportation when they need to be accessible by external clients. API Gateways provide a **shared layer** to handle differences between service protocols and fulfills the requirements of specific clients like desktop browsers, mobile devices, and legacy systems.

Click to see all chapters of Node.js at Scale: +

Microservices and consumers

Microservices are a service oriented architecture where teams can design, develop and ship their applications independently. It allows **technology diversity** on various levels of the system, where teams can benefit from using the best language, database, protocol, and transportation layer for the given technical challenge. For example, one team can use JSON over HTTP REST

Using different data serialization and protocols can be powerful in certain situations, but **clients** that want to consume our product may **have different requirements**. The problem can also occur in systems with homogeneous technology stack as consumers can vary from a desktop browser through mobile devices and gaming consoles to legacy systems. One client may expect XML format while the other one wants JSON. In many cases, you need to support both.

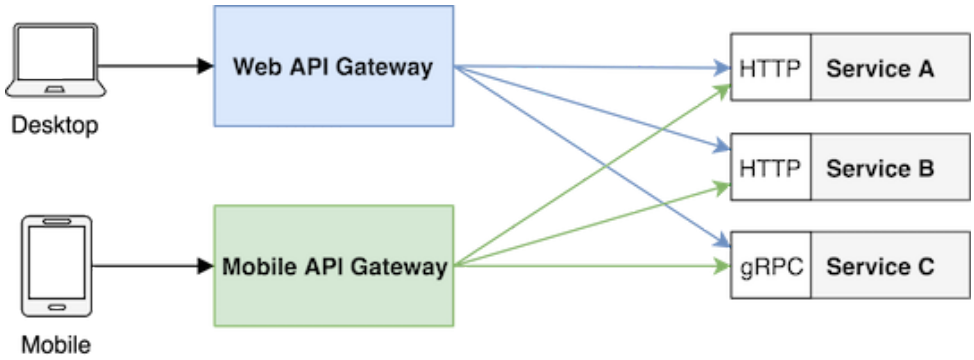
Another challenge that you can face when clients want to consume your microservices comes from generic **shared logic** like authentication, as you don't want to re-implement the same thing in all of your services.

To summarize: we don't want to implement our internal services in our microservices architecture in a way to support multiple clients and re-implement the same logic all over. This is where the **API Gateway** comes into the picture and provides a **shared layer** to handle differences between service protocols and fulfills the requirements of specific clients.

What is an API Gateway?

API Gateway is a type of service in a microservices architecture which provides a shared layer and API for clients to communicate with internal services. The API Gateway can **route requests**, transform protocols,

You can think about API Gateway as the **entry point** to our microservices world.
Our system can have one or multiple API Gateways, depending on the clients' requirements. For example, we can have a separate gateway for desktop browsers, mobile applications and public API(s) as well.



API Gateway as an entry point to microservices

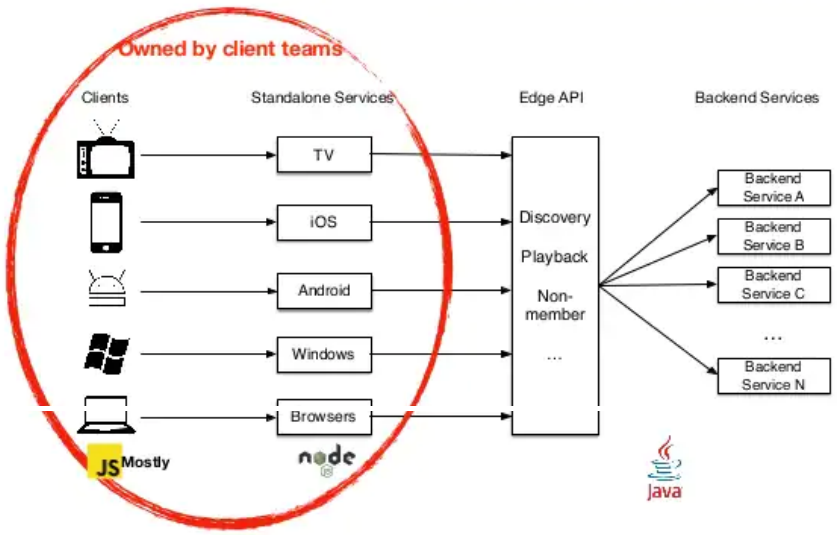
> ***Are you on a path to learn more about microservices and API Gateways?***
>
> *Check out our training called Designing Microservices Architectures*

Node.js API Gateway for frontend teams

As API Gateway provides functionality for client applications like browsers - it can be implemented and

It also means that language the API Gateway is implemented in language should be chosen by the team who is responsible for the particular client. As JavaScript is the primary language to develop applications for the browser, Node.js can be an excellent choice to implement an API Gateway even if your microservices architecture is developed in a different language.

Netflix successfully uses Node.js API Gateways with their Java backend to support a broad range of clients - to learn more about their approach read The "Paved Road" PaaS for Microservices at Netflix article.



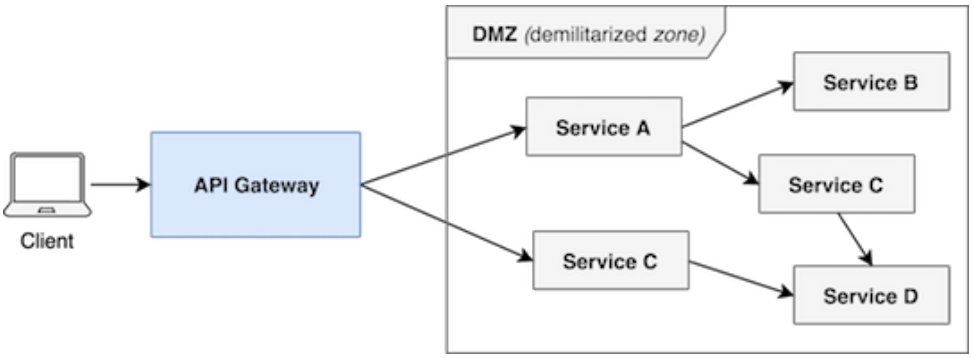
Netflix's approach to handle different clients, source

API Gateway functionalities

the most common gateway responsibilities.

Routing and versioning

We defined the API Gateway as the entry point to your microservices. In your gateway service, you can **route requests** from a client to specific services. You can even **handle versioning** during routing or change the backend interface while the publicly exposed interface can remain the same. You can also define new endpoints in your API gateway that cooperates with multiple services.



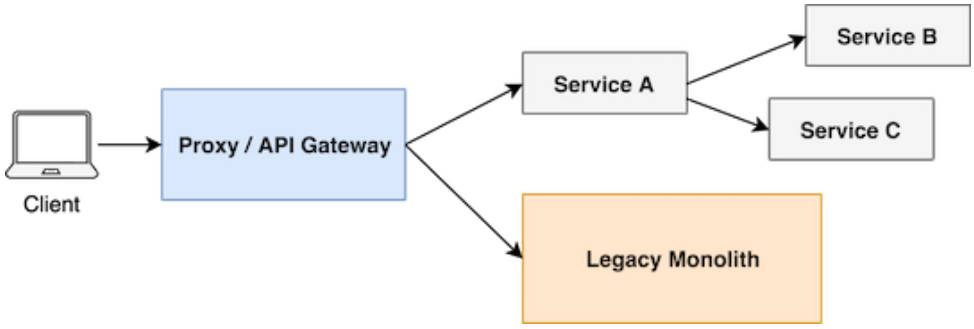
API Gateway as microservices entry point

Evolutionary design

The API Gateway approach can also help you to **break down your monolith** application. In most of the cases rewriting your system from scratch as a microservices is not a good idea and also not possible as we need to ship features for the business during the transition.

functionalities as microservices and route new endpoints to the new services while we can serve old endpoints via monolith. Later we can also break down the monolith with moving existing functionalities into new services.

With evolutionary design, we can have a **smooth transition** from monolith architecture to microservices.

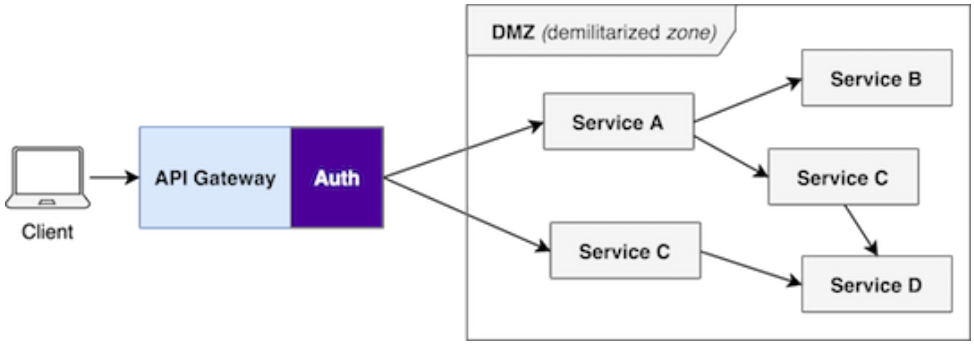


Evolutionary design with API Gateway

Authentication

Most of the microservices infrastructure need to handle authentication. Putting **shared logic** like authentication to the API Gateway can help you to **keep your services small** and **domain focused**.

In a microservices architecture, you can keep your services protected in a DMZ *(demilitarized zone)* via network configurations and **expose** them to clients **via the API Gateway**. This gateway can also handle more



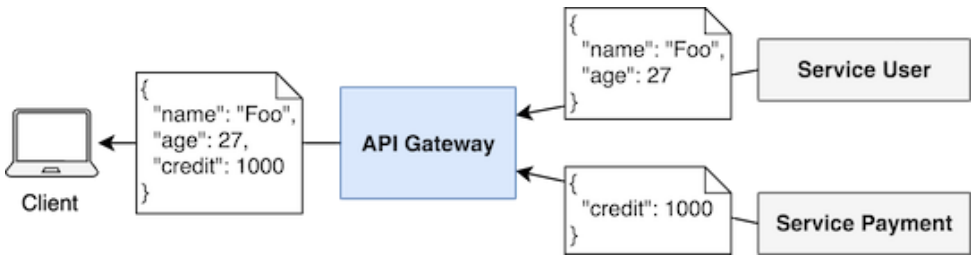
API Gateway with Authentication

In this article:

Data aggregation

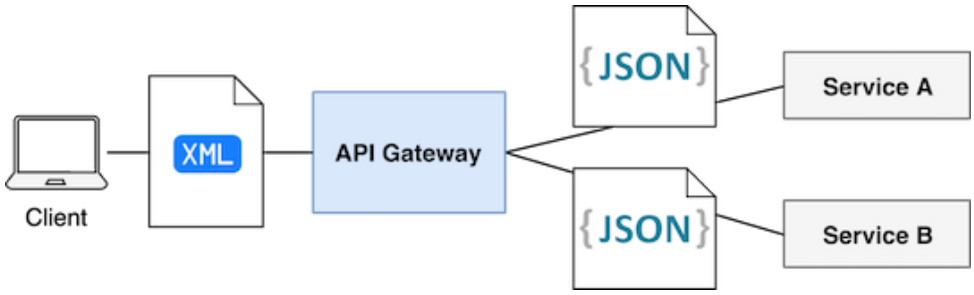
In a microservices architecture, it can happen that the client needs data in a different aggregation level, like **denormalizing data** entities that take place in various microservices. In this case, we can use our API Gateway to **resolve** these **dependencies** and collect data from multiple services.

In the following image you can see how the API Gateway merges and returns user and credit information as one piece of data to the client. Note, that these are owned and managed by different microservices.



Serialization format transformation

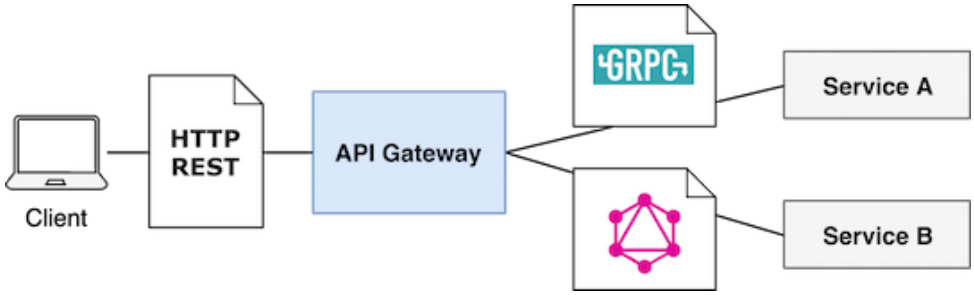
It can happen that we need to support clients with **different data serialization format** requirements. Imagine a situation where our microservices uses JSON, but one of our customers can only consume XML APIs. In this case, we can put the JSON to XML conversion into the API Gateway instead of implementing it in all of the microservices.



Protocol transformation

Microservices architecture allows **polyglot protocol transportation** to gain the benefit of different technologies. However most of the client support only one protocol. In this case, we need to transform service protocols for the clients.

An API Gateway can also handle protocol transformation between client and microservices.
In the next image, you can see how the client expects all



Rate-limiting and caching

In the previous examples, you could see that we can put generic shared logic like authentication into the API Gateway. Other than authentication you can also implement rate-limiting, caching and various reliability features in you API Gateway.

Overambitious API gateways

While implementing your API Gateway, you should avoid putting non-generic logic - like domain specific data transformation - to your gateway.

Services should always have **full ownership over** their **data domain**. Building an overambitious API Gateway **takes the control from service teams** that goes against the philosophy of microservices.

This is why you should be careful with data aggregations in your API Gateway - it can be powerful but can also

Always define **clear responsibilities** for your API Gateway and only include generic shared logic in it.

> ***Are you on a path to learn more about microservices and API Gateways?***
>
> *Check out our training called Designing Microservices Architectures*

Node.js API Gateways

While you want to do simple things in your API Gateway like routing requests to specific services you can **use a reverse proxy** like nginx. But at some point, you may need to implement logic that's not supported in general proxies. In this case, you can **implement your own** API Gateway in Node.js.

In Node.js you can use the http-proxy package to simply proxy requests to a particular service or you can use the more feature rich express-gateway to create API gateways.

In our first API Gateway example, we authenticate the request before we proxy it to the *user* service.

```
const express = require('express')
const httpProxy = require('express-http-proxy')
const app = express()
```

```
service')

// Authentication
app.use((req, res, next) => {
  // TODO: my authentication logic
  next()
})

// Proxy request
app.get('/users/:userId', (req, res, next) => {
  userServiceProxy(req, res, next)
})
```

Another approach can be when you make a new request in your API Gateway, and you return the response to the client:

```
const express = require('express')
const request = require('request-promise-native')
const app = express()

// Resolve: GET /users/me
app.get('/users/me', async (req, res) => {
  const userId = req.session.userId
  const uri = `https://user-
service/users/${userId}`
  const user = await request(uri)
  res.json(user)
})
```

Node.js API Gateways Summarized

API Gateway provides a shared layer to serve client requirements with microservices architecture. It helps to keep your services small and domain focused. You can put different generic logic to your API Gateway, but you should avoid overambitious API Gateways as these take the control from service teams.

Related topics

Node.js with Microservices Tutorials | @RisingStack

Node.js at Scale - Tutorial Series | @RisingStack

DevOps Tutorials | @RisingStack

Architectural Patterns Tutorials | @RisingStack

Share this post

Follow @RisingStack

Comments +

TRAININGS

Why learn from us?

Online Training & Mentorship for Software Developers

Designing Microservices Architectures

Handling Microservices with Kubernetes

Modern Front-End with React

Building Complex Apps with Angular

Node.js Fundamentals

RESOURCES & COMMUNITY

RisingStack blog

Free eBooks

Microservices Weekly

Weekly Node Updates

Node.js Meetups by RisingStack

OTHER

Open Positions

Our Team

Full-Stack JS Certificate

RisingStack in the JS Community

Privacy Policy