NGINX

BLOG TECH Chris Richardson of Eventuate, Inc. June 15, 2015

Building Microservices: Using an API Gateway

microservices, *API gateway*

Editor *– This seven-part series of articles is now complete:*

1. *Introduction to Microservices*
2. *Building Microservices: Using an API Gateway (this article)*
3. *Building Microservices: Inter-Process Communication in a Microservices Architecture*
4. *Service Discovery in a Microservices Architecture*
5. *Event-Driven Data Management for Microservices*
6. *Choosing a Microservices Deployment Strategy*
7. *Refactoring a Monolith into Microservices*

You can also download the complete set of articles, plus information about implementing microservices using NGINX Plus, as an ebook – Microservices: From Design to Deployment. Also, please look at the new Microservices Solutions page.

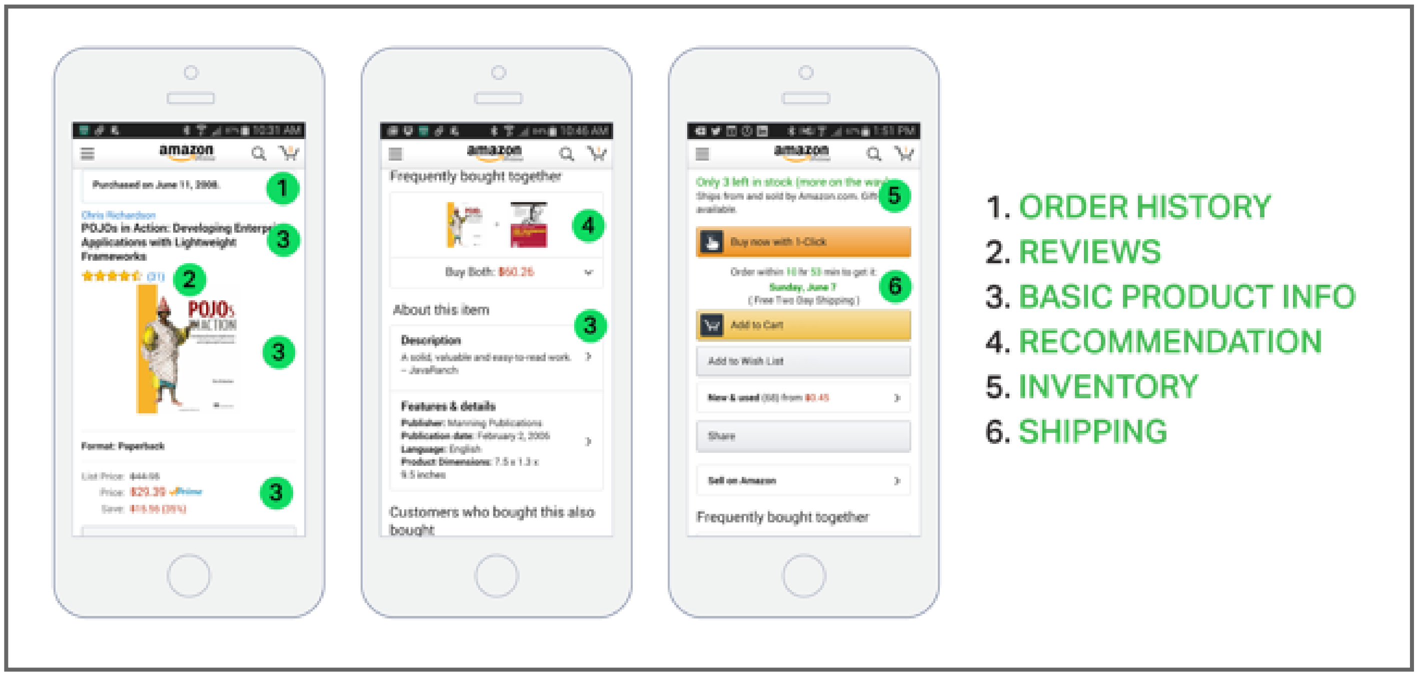
The first article in this seven-part series about designing, building, and deploying microservices introduced the Microservices Architecture pattern. It discussed the benefits and drawbacks of using microservices and how, despite the complexity of microservices, they are usually the ideal choice for complex applications. This is the second article in the series and will discuss building microservices using an API Gateway.

When you choose to build your application as a set of microservices, you need to decide how your application's clients will interact with the microservices. With a monolithic application there is just one set of (typically replicated, load-balanced) endpoints. In a microservices architecture, however, each microservice exposes a set of what are typically fine-grained endpoints. In this article, we examine how this impacts client-to-application communication and proposes an approach that uses an API Gateway.

Introduction

Let’s imagine that you are developing a native mobile client for a shopping application. It’s likely that you need to implement a product details page, which displays information about any given product.

For example, the following diagram shows what you will see when scrolling through the product details in Amazon’s Android mobile application.



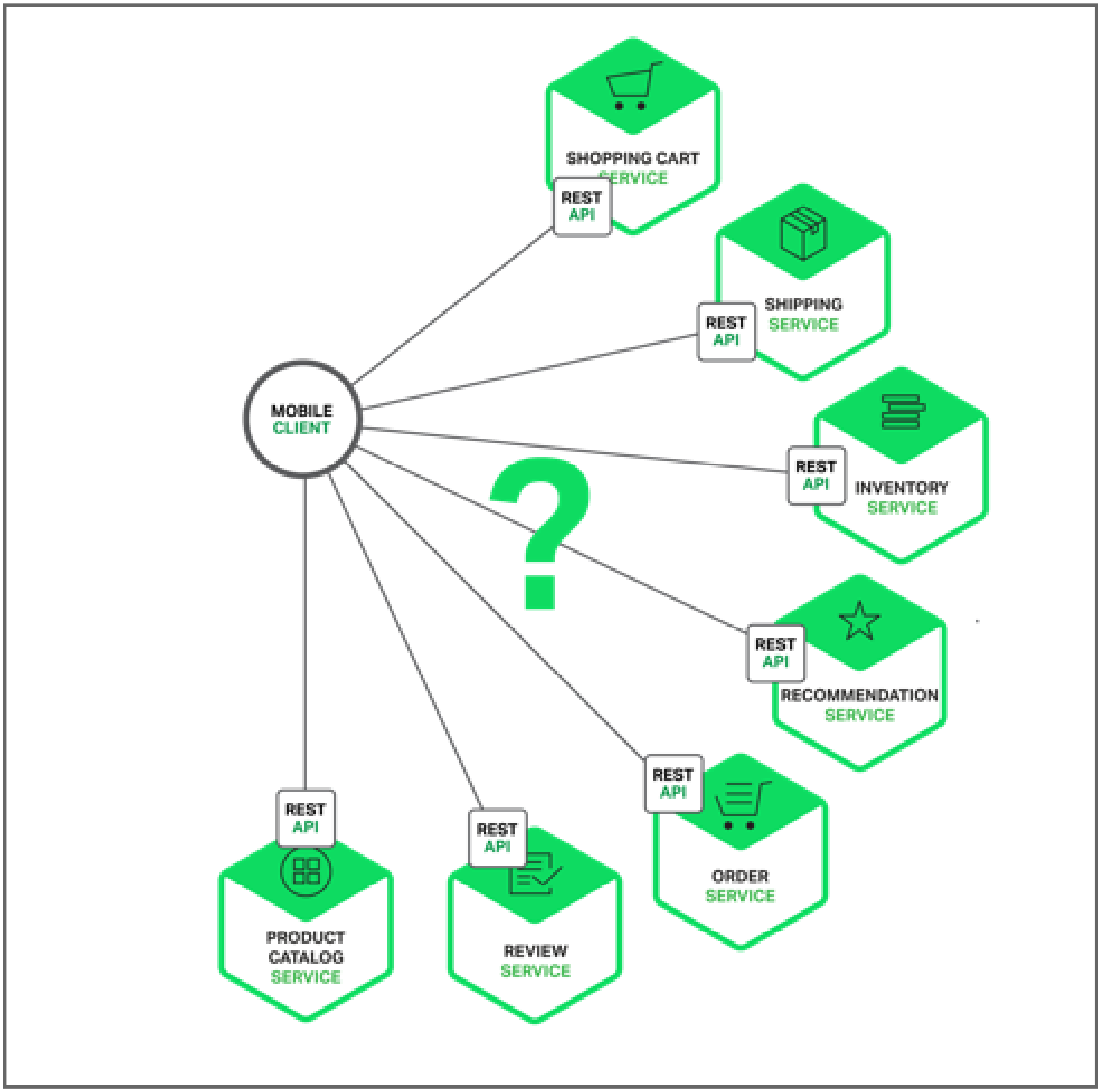
Even though this is a smartphone application, the product details page displays a lot of information. For example, not only is there basic product information (such as name, description, and price) but this page also shows:

- Number of items in the shopping cart
- Order history
- Customer reviews
- Low inventory warning
- Shipping options
- Various recommendations, including other products this product is frequently bought with, other products bought by customers who bought this product, and other products viewed by customers who bought this product
- Alternative purchasing options

When using a monolithic application architecture, a mobile client would retrieve this data by making a single REST call (`GET api.company.com/productdetails/`*`productId`*) to the application. A load balancer routes the request to one of N identical application instances. The application would then query various database tables and return the response to the client.

In contrast, when using the microservices architecture the data displayed on the product details page is owned by multiple microservices. Here are some of the potential microservices that own data displayed on the example product details page:

- Shopping Cart Service – Number of items in the shopping cart
- Order Service – Order history
- Catalog Service – Basic product information, such as its name, image, and price
- Review Service – Customer reviews
- Inventory Service – Low inventory warning
- Shipping Service – Shipping options, deadlines, and costs drawn separately from the shipping provider's API
- Recommendation Service(s) – Suggested items



We need to decide how the mobile client accesses these services. Let's look at the options.

Direct Client-to-Microservice Communication

In theory, a client could make requests to each of the microservices directly. Each microservice would have a public endpoint (**https://*serviceName*.api.company.name**). This URL would map to the microservice's load balancer, which distributes requests across the available instances. To retrieve the product details, the mobile client would make requests to each of the services listed above.

Unfortunately, there are challenges and limitations with this option. One problem is the mismatch between the needs of the client and the fine-grained APIs exposed by each of the microservices. The client in this example has to make seven separate requests. In more complex applications it might have to make many more. For example, Amazon describes how hundreds of services are involved in rendering their product page. While a client could make that many requests over a LAN, it would probably be too inefficient over the public Internet and would definitely be impractical over a mobile network. This approach also makes the client code much more complex.

Another problem with the client directly calling the microservices is that some might use protocols that are not web-friendly. One service might use Thrift binary RPC while another service might use the AMQP messaging protocol. Neither protocol is particularly browser- or firewall-friendly and is best used internally. An application should use protocols such as HTTP and WebSocket outside of the firewall.

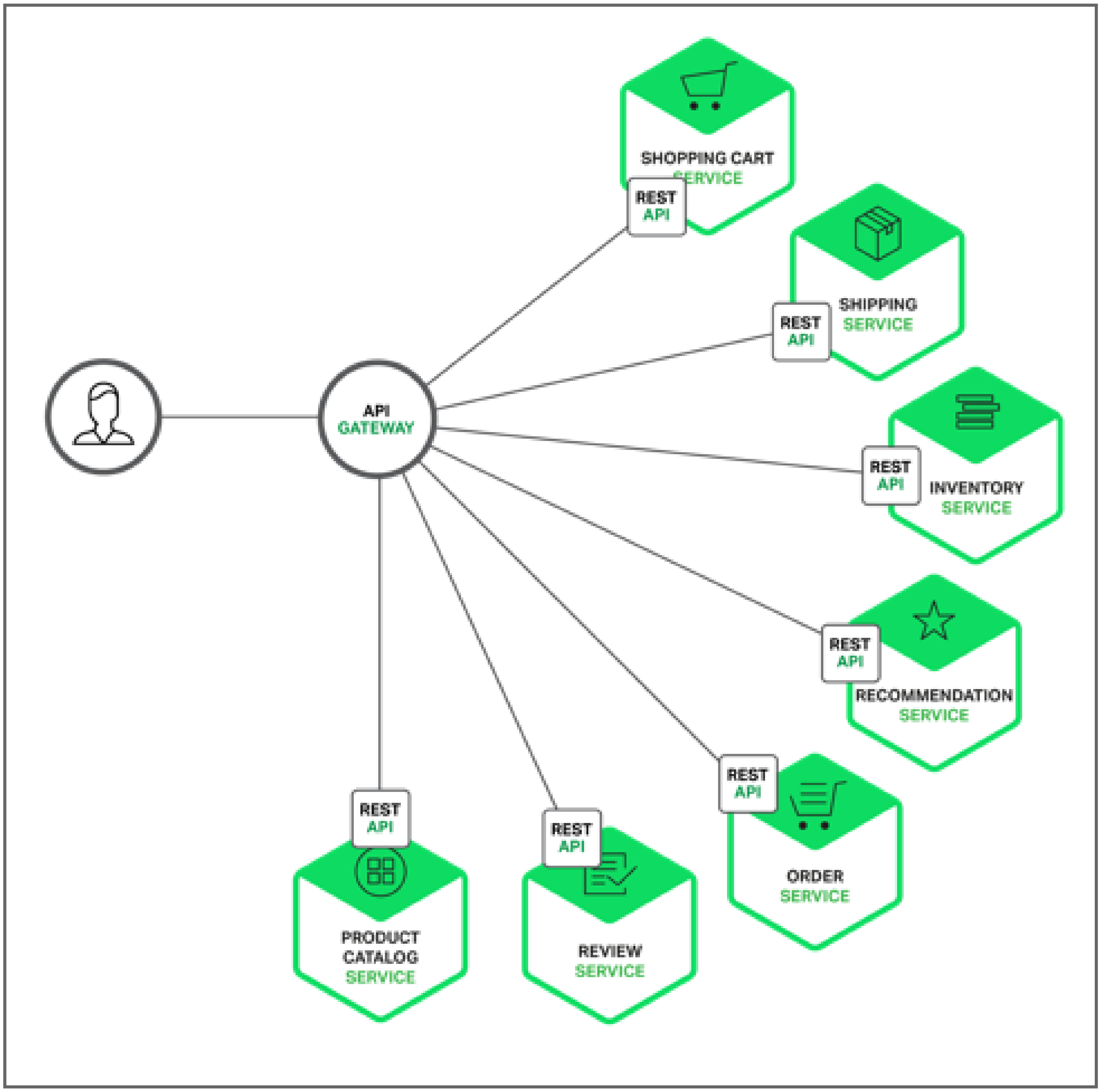
Another drawback with this approach is that it makes it difficult to refactor the microservices. Over time we might want to change how the system is partitioned into services. For example, we might merge two services or split a service into two or more services. If, however, clients communicate directly with the services, then performing this kind of refactoring can be extremely difficult.

Because of these kinds of problems it rarely makes sense for clients to talk directly to microservices.

Using an API Gateway

Usually a much better approach is to use what is known as an API Gateway. An API Gateway is a server that is the single entry point into the system. It is similar to the Facade pattern from object-oriented design. The API Gateway encapsulates the internal system architecture and provides an API that is tailored to each client. It might have other responsibilities such as authentication, monitoring, load balancing, caching, request shaping and management, and static response handling.

The following diagram shows how an API Gateway typically fits into the architecture:



The API Gateway is responsible for request routing, composition, and protocol translation. All requests from clients first go through the API Gateway. It then routes requests to the appropriate microservice. The API Gateway will often handle a request by invoking multiple microservices and aggregating the results. It can translate between web protocols such as HTTP and WebSocket and web-unfriendly protocols that are used internally.

The API Gateway can also provide each client with a custom API. It typically exposes a coarse-grained API for mobile clients. Consider, for example, the product details scenario. The API Gateway can provide an endpoint (**/productdetails?productid=*xxx***) that enables a mobile client to retrieve all of the product details with a single request. The API Gateway handles the request by invoking the various services – product info, recommendations, reviews, etc. – and combining the results.

A great example of an API Gateway is the Netflix API Gateway. The Netflix streaming service is available on hundreds of different kinds of devices including televisions, set-top boxes, smartphones, gaming systems, tablets, etc. Initially, Netflix attempted to provide a one-size-fits-all API for their streaming service. However, they discovered that it didn't work well because of the diverse range of devices and their unique needs. Today, they use an API Gateway that provides an API tailored for each device by running device-specific adapter code. An adapter typically handles each request by invoking on average six to seven backend services. The Netflix API Gateway handles billions of requests per day.

Benefits and Drawbacks of an API Gateway

As you might expect, using an API Gateway has both benefits and drawbacks. A major benefit of using an API Gateway is that it encapsulates the internal structure of the application. Rather than having to invoke specific services, clients simply talk to the gateway. The API Gateway provides each kind of client with a specific API. This reduces the number of round trips between the client and application. It also simplifies the client code.

The API Gateway also has some drawbacks. It is yet another highly available component that must be developed, deployed, and managed. There is also a risk that the API Gateway becomes a development bottleneck. Developers must update the API Gateway in order to expose each microservice's endpoints. It is important that the process for updating the API Gateway be as lightweight as possible. Otherwise, developers will be forced to wait in line in order to update the gateway. Despite these drawbacks, however, for most real-world applications it makes sense to use an API Gateway.

Implementing an API Gateway

Now that we have looked at the motivations and the trade-offs for using an API Gateway, let's look at various design issues you need to consider.

Performance and Scalability

Only a handful of companies operate at the scale of Netflix and need to handle billions of requests per day. However, for most applications the performance and scalability of the API Gateway is usually very important. It makes sense, therefore, to build the API Gateway on a platform that supports asynchronous, nonblocking I/O. There are a variety of different technologies that can be used to implement a scalable API Gateway. On the JVM you can use one of the NIO-based frameworks such Netty, Vertx, Spring Reactor, or JBoss Undertow. One popular non-JVM option is Node.js, which is a platform built on Chrome's JavaScript engine. Another option is to use NGINX Plus. NGINX Plus offers a mature, scalable, high-performance web server and reverse proxy that is easily deployed, configured, and programmed. NGINX Plus can manage authentication, access control, load balancing requests, caching responses, and provides application-aware health checks and monitoring.

Using a Reactive Programming Model

The API Gateway handles some requests by simply routing them to the appropriate backend service. It handles other requests by invoking multiple backend services and aggregating the results. With some requests, such as a product details request, the requests to backend services are independent of one another. In order to minimize response time, the API Gateway should perform independent requests concurrently. Sometimes, however, there are dependencies between requests. The API Gateway might first need to validate the request by calling an authentication service, before routing the request to a backend service. Similarly, to fetch information about the products in a customer's wish list, the API Gateway must first retrieve the customer's profile containing that information, and then retrieve the information for each product. Another interesting example of API composition is the Netflix Video Grid.

Writing API composition code using the traditional asynchronous callback approach quickly leads you to callback hell. The code will be tangled, difficult to understand, and error-prone. A much better approach is to write API Gateway code in a declarative style using a reactive approach. Examples of reactive abstractions include Future in Scala, CompletableFuture in Java 8, and Promise in JavaScript. There is also Reactive Extensions (also called Rx or ReactiveX), which was originally developed by Microsoft for the .NET platform. Netflix created RxJava for the JVM specifically to use in their API Gateway. There is also RxJS for JavaScript, which runs in both the browser and Node.js. Using a reactive approach will enable you to write simple yet efficient API Gateway code.

Service Invocation

A microservices-based application is a distributed system and must use an inter-process communication mechanism. There are two styles of inter-process communication. One option is to use an asynchronous, messaging-based mechanism. Some implementations use a message broker such as JMS or AMQP. Others, such as Zeromq, are brokerless and the services communicate directly. The other style of inter-process communication is a synchronous mechanism such as HTTP or Thrift. A system will typically use both asynchronous and synchronous styles. It might even use multiple implementations of each style. Consequently, the API Gateway will need to support a variety of communication mechanisms.

Service Discovery

The API Gateway needs to know the location (IP address and port) of each microservice with which it communicates. In a traditional application, you could probably hardwire the locations, but in a modern, cloud-based microservices application this is a nontrivial problem. Infrastructure services, such as a message broker, will usually have a static location, which can be specified via OS environment variables. However, determining the location of an application service is not so easy. Application services have dynamically assigned locations. Also, the set of instances of a service changes dynamically because of autoscaling and upgrades. Consequently, the API Gateway, like any other service client in the system, needs to use the system's service discovery mechanism: either Server-Side Discovery or Client-Side Discovery. A later article will describe service discovery in more detail. For now, it is worthwhile to note that if the system uses Client-Side Discovery then the API Gateway must be able to query the Service Registry, which is a database of all microservice instances and their locations.

Handling Partial Failures

Another issue you have to address when implementing an API Gateway is the problem of partial failure. This issue arises in all distributed systems whenever one service calls another service that is either responding slowly or is unavailable. The API Gateway should never block indefinitely waiting for a downstream service. However, how it handles the failure depends on the specific scenario and which service is failing. For example, if the recommendation service is unresponsive in the product details scenario, the API Gateway should return the rest of the product details to the client since they are still useful to the user. The recommendations could either be empty or replaced by, for example, a hardwired top ten list. If, however, the product information service is unresponsive then API Gateway should return an error to the client.

The API Gateway could also return cached data if that was available. For example, since product prices change infrequently, the API Gateway could return cached pricing data if the pricing service is unavailable. The data can be cached by the API Gateway itself or be stored in an external cache such as Redis or Memcached. By returning either default data or cached data, the API Gateway ensures that system failures do not impact the user experience.

Netflix Hystrix is an incredibly useful library for writing code that invokes remote services. Hystrix times out calls that exceed the specified threshold. It implements a *circuit breaker* pattern, which stops the client from waiting needlessly for an unresponsive service. If the error rate for a service exceeds a specified threshold, Hystrix trips the circuit breaker and all requests will fail immediately for a specified period of time. Hystrix lets you define a fallback action when a request fails, such as reading from a cache or returning a default value. If you are using the JVM you should definitely consider using Hystrix. And, if you are running in a non-JVM environment, you should use an equivalent library.

Summary

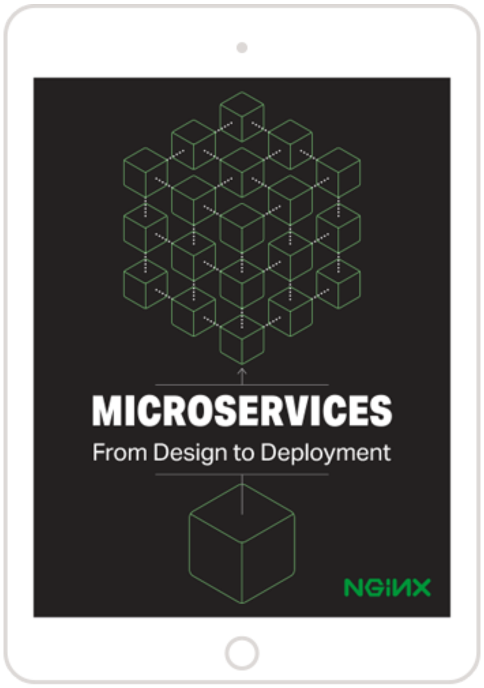
For most microservices-based applications, it makes sense to implement an API Gateway, which acts as a single entry point into a system. The API Gateway is responsible for request routing, composition, and protocol translation. It provides each of the application's clients with a custom API. The API Gateway can also mask failures in the backend services by returning cached or default data. In the next article in the series, we will look at communication between services.

Editor *– This seven-part series of articles is now complete:*

1. *Introduction to Microservices*
2. *Building Microservices: Using an API Gateway (this article)*
3. *Building Microservices: Inter-Process Communication in a Microservices Architecture*
4. *Service Discovery in a Microservices Architecture*
5. *Event-Driven Data Management for Microservices*
6. *Choosing a Microservices Deployment Strategy*
7. *Refactoring a Monolith into Microservices*

You can also download the complete set of articles, plus information about implementing microservices using NGINX Plus, as an ebook – Microservices: From Design to Deployment.

Guest blogger Chris Richardson is the founder of the original CloudFoundry.com, an early Java PaaS (Platform as a Service) for Amazon EC2. He now consults with organizations to improve how they develop and deploy applications. He also blogs regularly about microservices at http://microservices.io.



Microservices: From Design to Deployment

The complete guide to microservices development

DOWNLOAD NOW

30 Comments **NGINX** **Arefe**

Recommend 15 **Share** **Sort by Best**

Join the discussion…

Paulo Merson • 2 years ago

Excellent article! I'm also a frequent reader of your microservices pattern catalog.

Where you mention an API Gateway endpoint to retrieve product details, you say "The API Gateway handles the request by invoking the various services – … – and combining the results." Invoking services and combining results is what a composition controller service would do. This service contains task specific logic and knows what other services to invoke, how to parse application specific JSON documents to create another application specific JSON, how to handle exceptions, how to demarcate transactions and use compensation, how to dehydrate if one of the services will take long, how to set correlation identifiers and handle callbacks if needed. Does an API Gateway do any of that?!

I thought an API Gateway provided interceptor-like functionality (routing, transformation, protocol bridging, analytics, security controls). Invoking services and combining results is more like what you would do in an orchestrator service. Is the API Gateway already incorporating orchestration server features?

4 ^ | v • Reply • Share ›

Eric Duncan ➦ Paulo Merson • 2 years ago

In a typical design, you are only making 2 or 3 calls from the API Gateway to the backend components:

* authorization/session
* domain object/service (that gets you 99% of data)
* metrics

Typically the first two are done synchronously, while the metrics are thrown into background non-block thread pools.

The domain object/service in typical setups does the curration of all details you normally need: pricing, product details, recommendation and returns it all in a single json blog.

It is important to note that you shouldn't over-architect you solutions by creating three services from the beginning (pricing, product, and recommendation). Build it all into the aggregate root, Product service, and call in the day and move on.

Now as your system grows, you notice with your metrics that the recommendation engine is consuming 90% of the time it takes to make a request. Time to split off the

see more

2 ^ | v • Reply • Share ›

orubel ➦ Eric Duncan • 2 years ago

You neglect to mention the other drawback of the API gateway,,, the architectural cross cutting concern. The API pattern binds communication data to business logic at the controller through annotations, restful controllers, etc. This data (request method, endpoints, etc) cannot be shared but only duplicated... and as such, it is not synchronized.

Also because it is bound at such a low level in the application, any redirects have to go outside the DMZ for security checks to the api gateway (thus dropping threads), doing checks at the api gateway and then coming back in potentially to a new api server.

This is problematic as a whole as it creates IO bottlenecks.

The solution is to use API abstraction and shared IO through the new API pattern which is designed for distributed architectures unlike the OLD API pattern which was designed in the 70's and not intended for distributed architectures.

• Reply • Share ›

Siri Eric Duncan • 2 years ago

Hi, we have been stuck with the problem of choosing good software that provide orchestrator services. Wondering if you had any recommendations.

• Reply • Share ›

Eric Duncan Siri • 2 years ago

There are several, and can depend on your tech stack. Feel free to email as that can be a big convo.

• Reply • Share ›

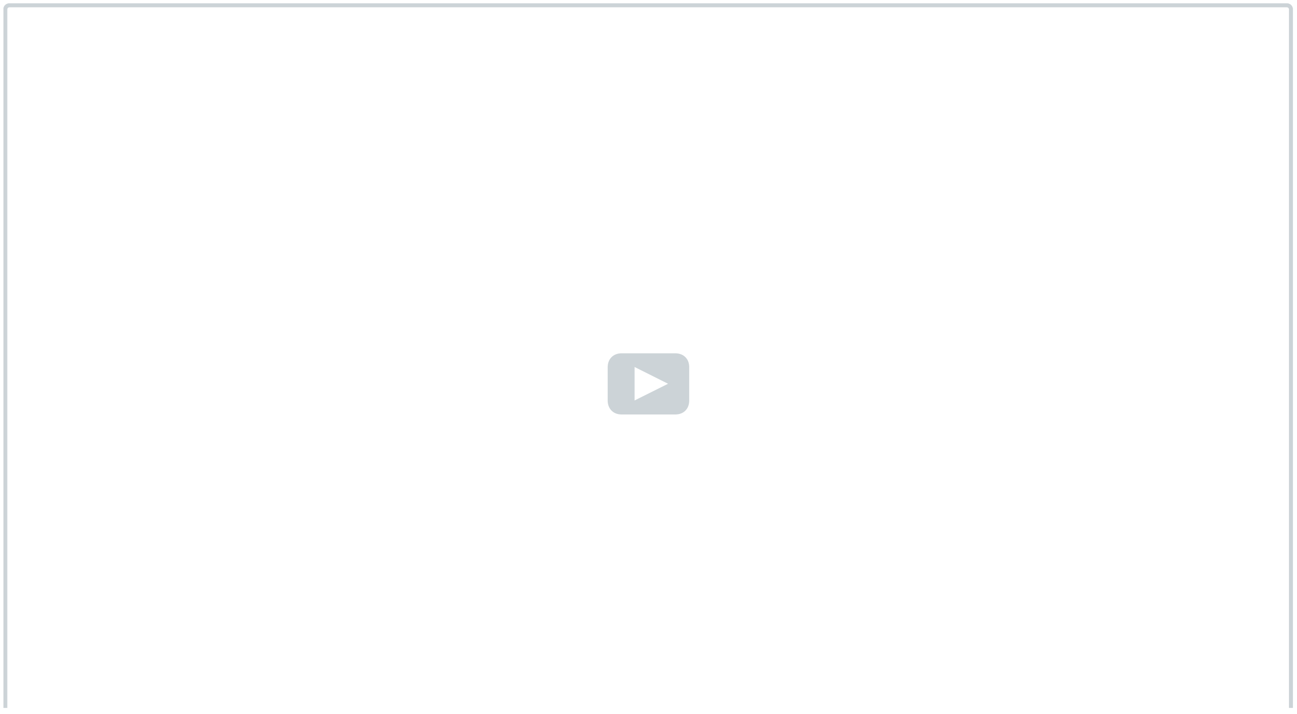
Siri Eric Duncan • 2 years ago

Cool, thanks! Sent you an email just now.

• Reply • Share ›

Aldric • 2 years ago

For those who want a example of what is implementing micro services in environment like nodeJs you can watch the video



and the code https://github.com/jeffbski... from Jeff Barczewski. Nice guy, nice explanation, have a nice day.

1 ^ | v • Reply • Share ›

Robert Karczewski • 2 years ago

How this rich API Gateway differs from ESB? It agregates data and servises, it will only take a year when someone discover canonical data model to be implemented inside of this fasade. Functional decoupling should be the most valuable part of MSA and it seems to be forgotten.

1 ^ | v • Reply • Share ›

Chris Richardson → Robert Karczewski • 2 years ago

The API Gateway is like an ESB is an interesting argument.
On the one hand, it is the central point of contention.
Potentially different teams have to contribute code and/or configuration metadata to the gateway to expose their functionality.
On the other hand, the API gateway might be able to dynamically discovery much of the routing information, which alleviates that problem.
Also, the API Gateway is on the edge of the system - its model(s) are at the API layer rather than being an integral part of the application's business logic.

Something for you to consider: if you don't use an API gateway then how do clients of the system communicate with the system?

1 ^ | v • Reply • Share ›

Robert Karczewski → Chris Richardson • 2 years ago

Maybe it is all about naming but, I think that API Gateway in the context is back-end for front-end couse as I understand there should be dedicated API Gateway for each client. My idea is to give access to the system in three ways:
- UI composition for web application,
- back-end for front-end for mobile app,
- API Exposure Layer (APIGee in this case) for API exposure for 3rd parties.
As the API is going to be exposed as commercial offer of my company the role of API Exposure Layer is more about AAA and protocol conversion than integration of data and services.
I persuaded my business stakeholders to implement MSA with promise of shortening Time To Market not promising high availability, ease of scaling and other MSA benefits because for them it is all IT concerns. My current environment is mature SOA environment with number of ESBs that slows down delivery of business new ideas.
I'd like to know how to deal with danger of polluting API Gateway with logic of services. Are any patterns how keep functional decoupling in real life implementation of MSA?

1 ^ | v • Reply • Share ›

Chris Richardson → Robert Karczewski • 2 years ago

Interesting that you mentioned the Backend-for-Frontend approach. I have been thinking of writing that up as an alternative pattern to the API Gateway. It has different tradeoffs and is something to consider when, perhaps, there are a small number of clients. The arguments you make about ESBs could still apply to it as well, however.

With regards to preventing logic from creeping into the gateway. All I can say is to use the usual enforcement mechanisms, e.g. code reviews. Perhaps fail the build if the cyclomatic complexity of the API gateway exceeds a very low threshold...

1 • Reply • Share ›

zb • 2 years ago

" If you are using the JVM you should definitely consider using Hystrix. And, if you are running in a non-JVM environment, you should use an equivalent library." So, I'd love to become aware of some non-JVM equivalents for this circuit-breaker pattern. Anyone implemented this using Nginx?

1 • Reply • Share ›

MGG • 2 years ago

The idea behind microservices was to decouple functional parts of system. One thing seems to be really challenging: how do you handle the Authorization for a requested resource? Let's say you have a list of items, each one having his own rights management. If a user is requesting *his list* of items according to his rights you have to keep rights management on same microservice, otherwise the performance penalty is huge.. The outcome: you just lost service decoupling.

1 • Reply • Share ›

Selim Öber MGG • 2 years ago

You could use something similar to JWT claims (https://jwt.io/) and avoid the backend trip for a configurable period of time (token expiration).

• Reply • Share ›

perrohunter • 2 years ago

Do you consider that the API Gateware should server the Web page appliation as well? Or should a separate service serve the web application and the API Gateway live into a subdomain for subsequent calling?

For example: I serve my angular application from example.com (web ui microservice location) and have the angular app call the API Gateway at api.example.com (api gateway microservice location)?

• Reply • Share ›

Felipe Gusmao • 2 years ago

It made my understanding of microservices much better. I have few questions though. First, the proper name is Microservices architecture or microservice pattern? Second, when you have a service talking with each other who do you scale them. Ex: a talks to b. If b scales how does a know that b scaled. Finally, can you have multiple api gateways and or can you separate the gateway from the load balancer. I hope you understand my questions. Thank you.

^ | ⌄ • Reply • Share ›

Ankur ➦ Felipe Gusmao • 2 years ago

Although Eric has answered some parts of it, I'll answer any unanswered and add some more thoughts:

1. Microservices is an "architectural pattern". Refer Wiki page - https://en.wikipedia.org/wi... - that classifies it like that. So it is definitely an 'architecture' based on an already established 'pattern'.
2. Other question is - how does caller (A) know about what all services are available when B scales. B would need to register with the a service registry on startup. LB would contact that Service Registry and load balance between the available instances of B.

^ | ⌄ • Reply • Share ›

Eric Duncan ➦ Felipe Gusmao • 2 years ago

You typically have a load balancer or some other form of broadcasting between A and B.

I rarely ever directly connect A to B.

Remember "A" is the API Gateway and "B" is the backend service.

If you are thinking A and B are both backend services needing to talk to each other, that's a difference approach. In a microservice setup, you would use inter process communication between services.

The API Gateway just routes more-or-less public API requests to publicly exposed API endpoints on your services. But that's just one opinion. Several setups use those endpoints to talk to others. I find that cross-cutting concerns though as now you are supporting public API endpoints as well as backend private xalla., so I usually keep them separate.

^ | ⌄ • Reply • Share ›

Danny • 2 years ago

Great article. Really helping me to understand API gateway for beginner like me! Can you explain a little bit more on the handling for partial failures if the backend APIs are related to action such as CREATE, UPDATE, DELETE. In some cases, we may rollback those previous executed APIs when exception encountered. But sometimes rollback seems not doable. Any advice?

^ | ⌄ • Reply • Share ›

Chris Richardson ➦ Danny • 2 years ago

Thanks. Glad you like the article.
I think that generally speaking an Create/Update/Delete request should be handled by a single microservice.
That service should update it's data and then publish events to trigger the update of data owned by other services.

See https://www.nginx.com/blog/...

Reply • Share ›

Jeff Walker • 2 years ago

We have a situation where we'd like to run some specific Java code for certain requests, possibly before the request reaches a Microservice instance. Is it possible for the NGINX API Gateway to act as an application host, like Tomcat can host web apps. Or is the concept more like, NGINX will route the request to a Tomcat where the request can be pre-processed?

Also, could you please explain how NGINX (as an API Gateway) can interact with Hystrix, which again, seems to be more Java code. Where do you run the Hystrix Java code?

Reply • Share ›

Faisal Memon Mod → Jeff Walker • 2 years ago

The NGINX Lua module is the best way to run a request rule on NGINX. More info here: https://github.com/openrest... Our future plan is to support running JavaScript on NGINX: https://www.nginx.com/blog/...

You can't compile Hystrix in with NGINX, but you can use NGINX's built-in directives to timeout connections early, limit connections, etc.

Reply • Share ›

Vinicius Zaramella • 2 years ago

Many of the API Gateway i've seen before does not implement this "Facade" Pattern and only serve as a Simple reverse proxy plus basic authentication and simple features such as request limiting.
Is correct to call these softwares API Gateways even without the facade behavior?

Reply • Share ›

Chris Richardson → Vinicius Zaramella • 2 years ago

It is not unreasonable to call them API Gateways, albeit simple ones.

1 Reply • Share ›

Booleant Services • 2 years ago

At Booleant, we send the single request using the /api/batch API with post request that includes the json request body as :

//See the requests targetted to different VMs

{

"request": [

{

"httpMethod": "POST",

"body": {

"param1": "john",

"param2": "doe"

},

see more

^ | v • Reply • Share ›

Bradley Weston ➦ Booleant Services • 2 years ago

You're hard coding "localhost:8080" which really the benefit of using the API gateways is so that you don't need to know this information.

1 ^ | v • Reply • Share ›

Chris • 2 years ago

Great article. Can you explain how API Gateway can perform composition? For example one request is sent to API Gateway, and the response includes all the data from different micro services. How does API Gateway make multiple calls to different micro services?

^ | v • Reply • Share ›

Chris Richardson ➦ Chris • 2 years ago

Typically, you need to write code that handles each incoming request by invoking the various backend services and combining the results.

The API Gateway needs to be implemented using a scalable server technology that has a 'scripting language', e.g. NodeJS, NGINX, Java NIO based server such as Vertx, ...

Ideally the API gateway should invoke the backend services concurrently. But you want to avoid callback hell. Here is an presentation that I gave on this topic http://www.slideshare.net/c...

^ | v • Reply • Share ›

Milind • 2 years ago

Is there any readily available API Gateway, where one JSON request can be split to say 5 JSON requests (based on JSON Path) and there response can be aggregated to one single JSON, if the split and aggregation is straight forward and simple.

^ | v • Reply • Share ›

7imbrook ➦ Milind • 2 years ago

TRY NGINX PLUS FOR FREE

ASK US A QUESTION

STAY IN THE LOOP

Products

NGINX Plus
NGINX Controller
NGINX Unit
NGINX Amplify
NGINX Web Application Firewall

NGINX on Github

NGINX Open Source
NGINX Unit
NGINX Amplify
NGINX Kubernetes Ingress Controller
nginMesh
NGINX Microservices Reference Architecture
NGINX Crossplane

Solutions

ADC / Load balancing
Microservices
Cloud
Security
Web & mobile performance
API Gateway

Resources

Documentation
Ebooks
Webinars
Case studies
Blog
FAQ
Glossary

Support

Professional Services
Training

Partners

Amazon Web Services
Google Cloud Platform
IBM
Microsoft Azure
Red Hat
Certified module program

Company

About NGINX
Careers
Leadership

Press

Events

Connect With Us

STAY IN THE LOOP

Copyright © NGINX Inc. All rights reserved. | Privacy Policy