Table of Contents

Foreword

by Floyd Smith

The rise of microservices has been a remarkable advancement in application development and deployment. With microservices, an application is developed, or refactored, into separate services that "speak" to one another in a well-defined way – via APIs, for instance. Each microservice is self-contained, each maintains its own data store (which has significant implications), and each can be updated independently of others.

Moving to a microservices-based approach makes app development faster and easier to manage, requiring fewer people to implement more new features. Changes can be made and deployed faster and easier. An application designed as a collection of microservices is easier to run on multiple servers with load balancing, making it easy to handle demand spikes and steady increases in demand over time, while reducing downtime caused by hardware or software problems.

Microservices are a critical part of a number of significant advancements that are changing the nature of how we work. Agile software development techniques, moving applications to the cloud, DevOps culture, continuous integration and continuous deployment (CI/CD), and the use of containers are all being used alongside microservices to revolutionize application development and delivery.

NGINX software is strongly associated with microservices and all of the technologies listed above. Whether deployed as a reverse proxy, or as a highly efficient web server, NGINX makes microservices-based application development easier and keeps microservices-based solutions running smoothly.

With the tie between NGINX and microservices being so strong, we've run a seven-part series on microservices on the NGINX website. Written by Chris Richardson, who has had early involvement with the concept and its implementation, the blog posts cover the major aspects of microservices for app design and development, including how to make the move from a monolithic application. The blog posts offer a thorough overview of major microservices issues and have been extremely popular.

In this ebook, we've converted each blog post to a book chapter, and added a sidebar to each chapter with information relevant to implementing microservices in NGINX. If you follow the advice herein carefully, you'll solve many potential development problems before you even start writing code. This book is also a good companion to the NGINX Microservices Reference Architecture, which implements much of the theory presented here.

The book chapters are:

1. **Introduction to Microservices** – A clear and simple introduction to microservices, from its perhaps overhyped conceptual definition to the reality of how microservices are deployed in creating and maintaining applications.
2. **Using an API Gateway** – An API Gateway is the single point of entry for your entire microservices-based application, presenting the API for each microservice. NGINX Plus can effectively be used as an API Gateway with load balancing, static file caching, and more.
3. **Inter-process Communication in a Microservices Architecture** – Once you break a monolithic application into separate pieces – microservices – the pieces need to speak to each other. And it turns out that you have many options for inter-process communication, including representational state transfer (REST). This chapter gives the details.
4. **Service Discovery in a Microservices Architecture** – When services are running in a dynamic environment, finding them when you need them is not a trivial issue. In this chapter, Chris describes a practical solution to this problem.
5. **Event-Driven Data Management for Microservices** – Instead of sharing a unified application-wide data store (or two) across a monolithic application, each microservice maintains its own unique data representation and storage. This gives you great flexibility, but can also cause complexity, and this chapter helps you sort through it.
6. **Choosing a Microservices Deployment Strategy** – In a DevOps world, how you do things is just as important as what you set out to do in the first place. Chris describes the major patterns for microservices deployment so you can make an informed choice for your own application.
7. **Refactoring a Monolith into Microservices** – In a perfect world, we would always get the time and money to convert core software into the latest and greatest technologies, tools, and approaches, with no real deadlines. But you may well find yourself converting a monolith into microservices, one… small… piece… at… a… time. Chris presents a strategy for doing this sensibly.

We think you'll find every chapter worthwhile, and we hope that you'll come back to this ebook as you develop your own microservices apps.

Floyd Smith
NGINX, Inc.

1 Introduction to Microservices

Microservices are currently getting a lot of attention: articles, blogs, discussions on social media, and conference presentations. They are rapidly heading towards the peak of inflated expectations on the Gartner Hype cycle. At the same time, there are skeptics in the software community who dismiss microservices as nothing new. Naysayers claim that the idea is just a rebranding of service-oriented architecture (SOA). However, despite both the hype and the skepticism, the Microservices Architecture pattern has significant benefits – especially when it comes to enabling the agile development and delivery of complex enterprise applications.

This chapter is the first in this seven-chapter ebook about designing, building, and deploying microservices. You will learn about the microservices approach and how it compares to the more traditional Monolithic Architecture pattern. This ebook will describe the various elements of a microservices architecture. You will learn about the benefits and drawbacks of the Microservices Architecture pattern, whether it makes sense for your project, and how to apply it.

Let's first look at why you should consider using microservices.

Building Monolithic Applications

Let's imagine that you were starting to build a brand new taxi-hailing application intended to compete with Uber and Hailo. After some preliminary meetings and requirements gathering, you would create a new project either manually or by using a generator that comes with a platform such as Rails, Spring Boot, Play, or Maven.

This new application would have a modular hexagonal architecture, like in Figure 1-1:

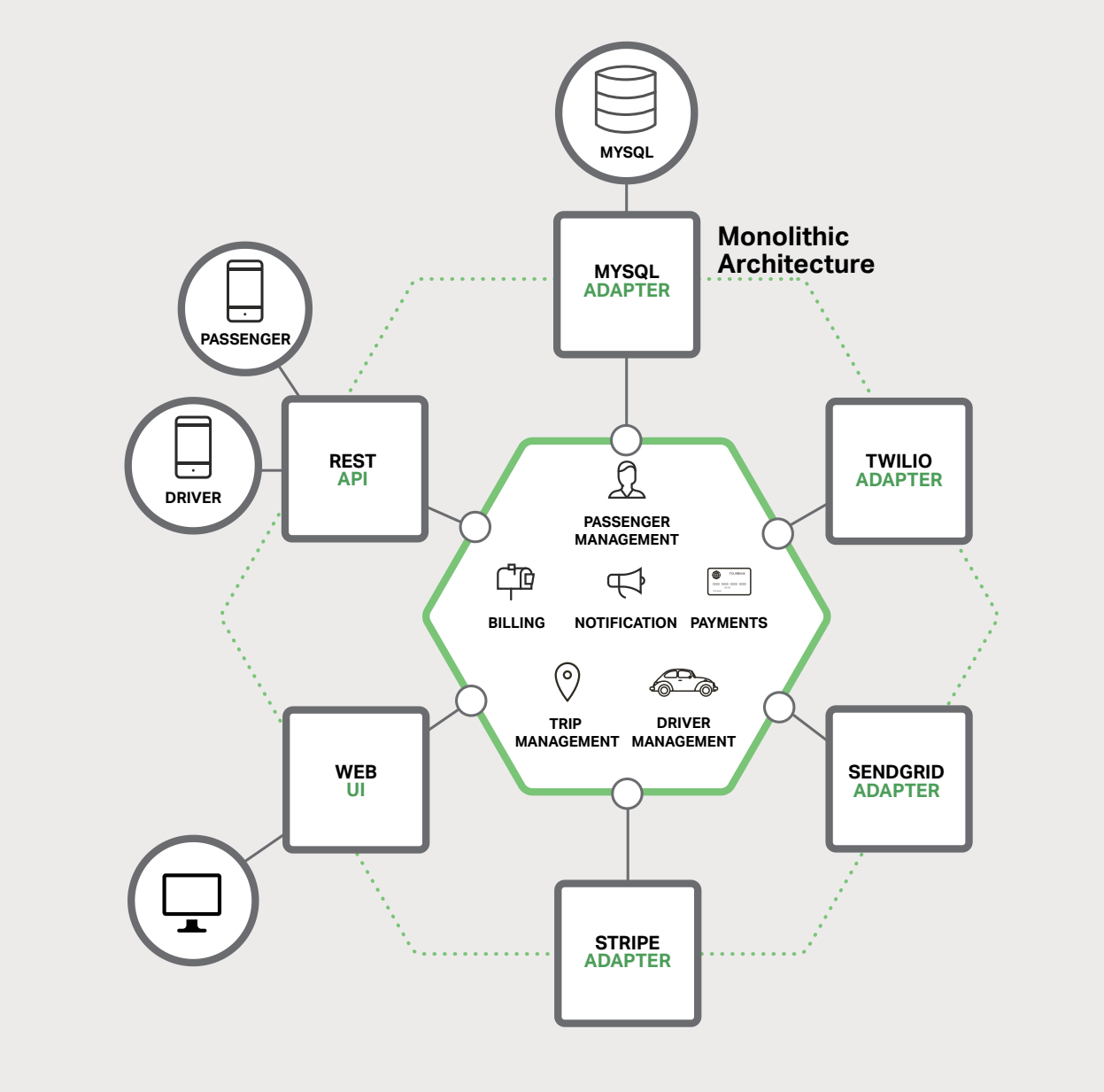

Figure 1-1. A sample taxi-hailing application.

At the core of the application is the business logic, which is implemented by modules that define services, domain objects, and events. Surrounding the core are adapters that interface with the external world. Examples of adapters include database access components, messaging components that produce and consume messages, and web components that either expose APIs or implement a UI.

Despite having a logically modular architecture, the application is packaged and deployed as a monolith. The actual format depends on the application's language and framework. For example, many Java applications are packaged as WAR files and deployed on application servers such as Tomcat or Jetty. Other Java applications are packaged as self-contained executable JARs. Similarly, Rails and Node.js applications are packaged as a directory hierarchy.

Applications written in this style are extremely common. They are simple to develop since our IDEs and other tools are focused on building a single application. These kinds of applications are also simple to test. You can implement end-to-end testing by simply launching the application and testing the UI with a testing package such as Selenium. Monolithic applications are also simple to deploy. You just have to copy the packaged application to a server. You can also scale the application by running multiple copies behind a load balancer. In the early stages of the project it works well.

Marching Toward Monolithic Hell

Unfortunately, this simple approach has a huge limitation. Successful applications have a habit of growing over time and eventually becoming huge. During each sprint, your development team implements a few more user stories, which, of course, means adding many lines of code. After a few years, your small, simple application will have grown into a monstrous monolith. To give an extreme example, I recently spoke to a developer who was writing a tool to analyze the dependencies between the thousands of JARs in their multi-million lines of code (LOC) application. I'm sure it took the concerted effort of a large number of developers over many years to create such a beast.

Once your application has become a large, complex monolith, your development organization is probably in a world of pain. Any attempts at agile development and delivery will flounder. One major problem is that the application is overwhelmingly complex. It's simply too large for any single developer to fully understand. As a result, fixing bugs and implementing new features correctly becomes difficult and time consuming. What's more, this tends to be a downwards spiral. If the codebase is difficult to understand, then changes won't be made correctly. You will end up with a monstrous, incomprehensible big ball of mud.

The sheer size of the application will also slow down development. The larger the application, the longer the start-up time is. I surveyed developers about the size and performance of their monolithic applications, and some reported start-up times as long as 12 minutes. I've also heard anecdotes of applications taking as long as 40 minutes to start up. If developers regularly have to restart the application server, then a large part of their day will be spent waiting around and their productivity will suffer.

Another problem with a large, complex monolithic application is that it is an obstacle to continuous deployment. Today, the state of the art for SaaS applications is to push changes into production many times a day. This is extremely difficult to do with a complex monolith,

since you must redeploy the entire application in order to update any one part of it. The lengthy start-up times that I mentioned earlier won't help either. Also, since the impact of a change is usually not very well understood, it is likely that you have to do extensive manual testing. Consequently, continuous deployment is next to impossible to do.

Monolithic applications can also be difficult to scale when different modules have conflicting resource requirements. For example, one module might implement CPU-intensive image processing logic and would ideally be deployed in Amazon EC2 Compute Optimized instances. Another module might be an in-memory database and best suited for EC2 Memory-optimized instances. However, because these modules are deployed together, you have to compromise on the choice of hardware.

Another problem with monolithic applications is reliability. Because all modules are running within the same process, a bug in any module, such as a memory leak, can potentially bring down the entire process. Moreover, since all instances of the application are identical, that bug will impact the availability of the entire application.

Last but not least, monolithic applications make it extremely difficult to adopt new frameworks and languages. For example, let's imagine that you have 2 million lines of code written using the XYZ framework. It would be extremely expensive (in both time and cost) to rewrite the entire application to use the newer ABC framework, even if that framework was considerably better. As a result, there is a huge barrier to adopting new technologies. You are stuck with whatever technology choices you made at the start of the project.

To summarize: you have a successful business-critical application that has grown into a monstrous monolith that very few, if any, developers understand. It is written using obsolete, unproductive technology that makes hiring talented developers difficult. The application is difficult to scale and is unreliable. As a result, agile development and delivery of applications is impossible.

So what can you do about it?

Microservices – Tackling the Complexity

Many organizations, such as Amazon, eBay, and Netflix, have solved this problem by adopting what is now known as the Microservices Architecture pattern. Instead of building a single monstrous, monolithic application, the idea is to split your application into set of smaller, interconnected services.

A service typically implements a set of distinct features or functionality, such as order management, customer management, etc. Each microservice is a mini-application that has its own hexagonal architecture consisting of business logic along with various adapters. Some microservices would expose an API that's consumed by other microservices or by the application's clients. Other microservices might implement a web UI. At runtime, each instance is often a cloud virtual machine (VM) or a Docker container.

For example, a possible decomposition of the system described earlier is shown in Figure 1-2:

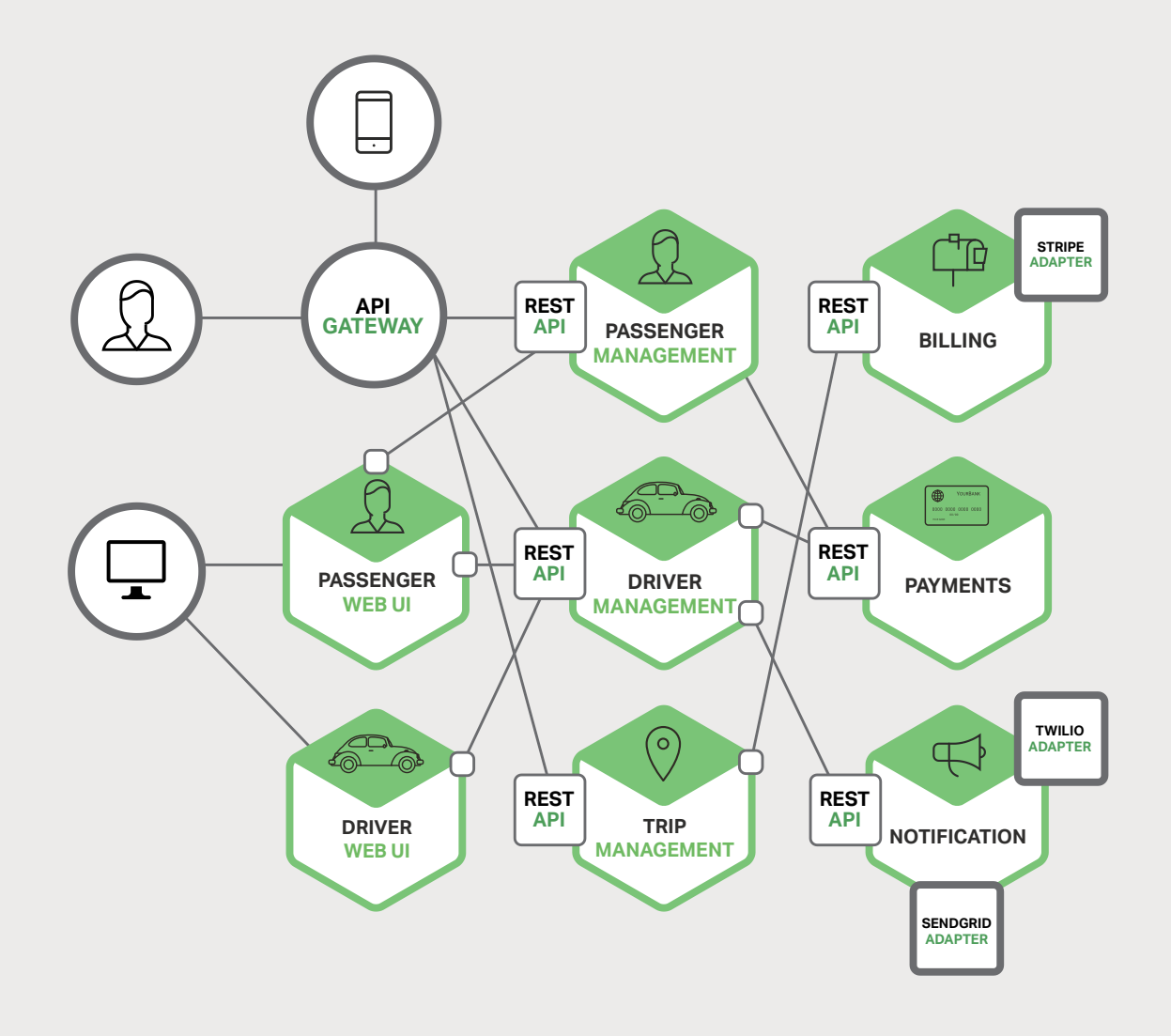

Figure 1-2. A monolithic application decomposed into microservices.

Each functional area of the application is now implemented by its own microservice. Moreover, the web application is split into a set of simpler web applications – such as one for passengers and one for drivers, in our taxi-hailing example. This makes it easier to deploy distinct experiences for specific users, devices, or specialized use cases.

Each backend service exposes a REST API and most services consume APIs provided by other services. For example, Driver Management uses the Notification server to tell an available driver about a potential trip. The UI services invoke the other services in order to render web pages. Services might also use asynchronous, message-based communication. Inter-service communication will be covered in more detail later in this ebook.

Some REST APIs are also exposed to the mobile apps used by the drivers and passengers. The apps don't, however, have direct access to the backend services. Instead, communication is mediated by an intermediary known as an API Gateway. The API Gateway is responsible for tasks such as load balancing, caching, access control, API metering, and monitoring, and can be implemented effectively using NGINX. Chapter 2 discusses the API Gateway in detail.

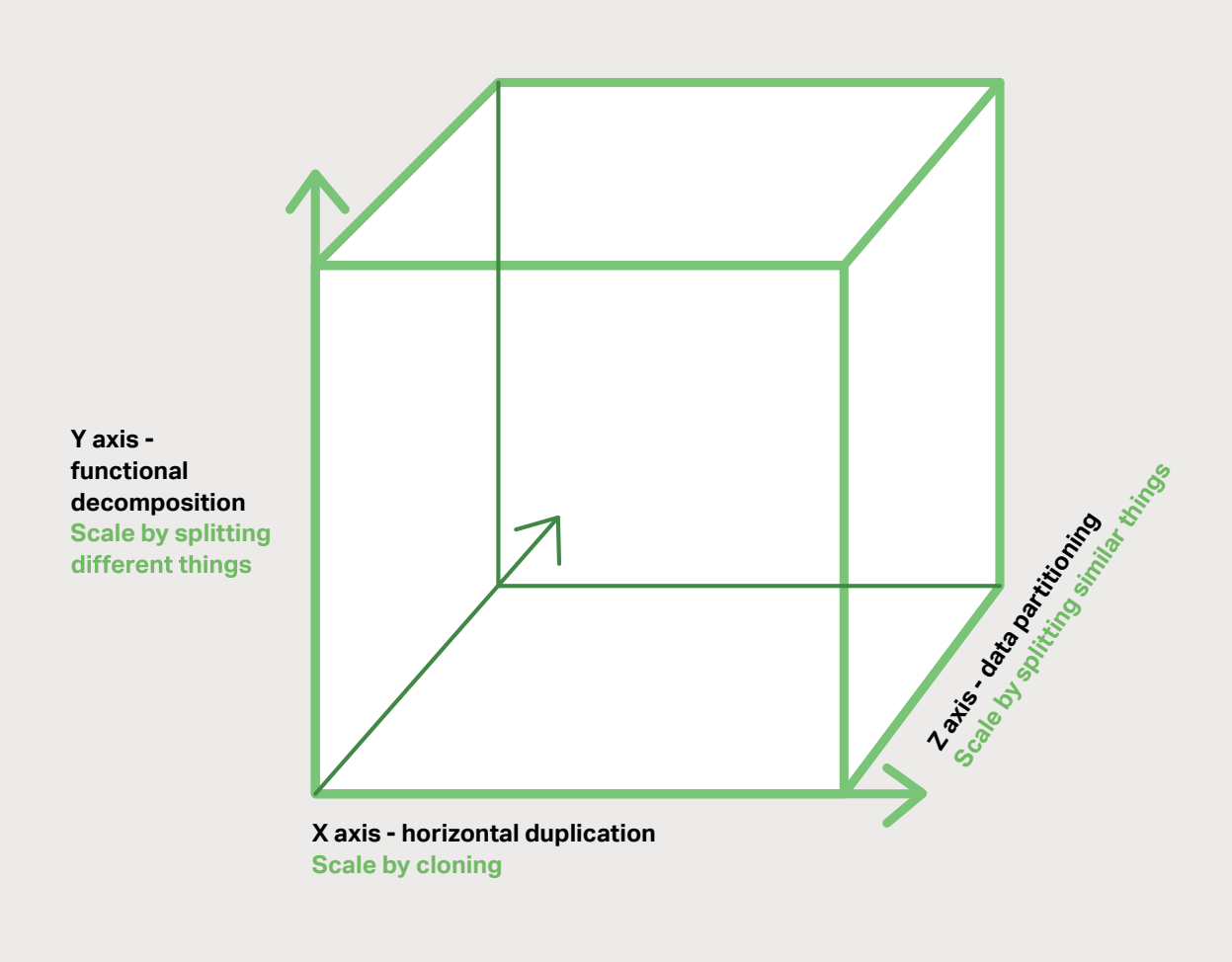

Figure 1-3. The Scale Cube, used in both development and delivery.

The Microservices Architecture pattern corresponds to the Y-axis scaling of the Scale Cube, which is a 3D model of scalability from the excellent book *The Art of Scalability*. The other two scaling axes are X-axis scaling, which consists of running multiple identical copies of the application behind a load balancer, and Z-axis scaling (or data partitioning), where an attribute of the request (for example, the primary key of a row or identity of a customer) is used to route the request to a particular server.

Applications typically use the three types of scaling together. Y-axis scaling decomposes the application into microservices as shown above in Figure 1-2.

At runtime, X-axis scaling runs multiple instances of each service behind a load balancer for throughput and availability. Some applications might also use Z-axis scaling to partition the services. Figure 1-4 shows how the Trip Management service might be deployed with Docker running on Amazon EC2.

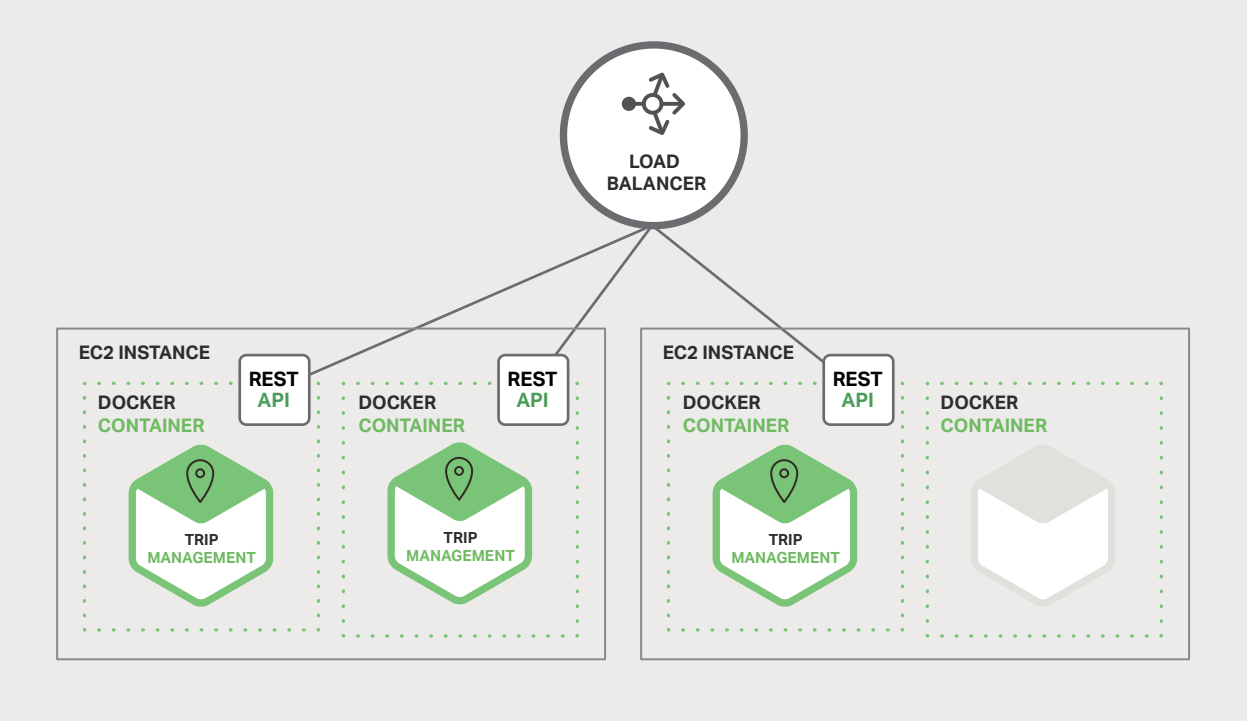

Figure 1-4. Deploying the Trip Management service using Docker.

At runtime, the Trip Management service consists of multiple service instances. Each service instance is a Docker container. In order to be highly available, the containers are running on multiple Cloud VMs. In front of the service instances is a load balancer such as NGINX that distributes requests across the instances. The load balancer might also handle other concerns such as caching, access control, API metering, and monitoring.

The Microservices Architecture pattern significantly impacts the relationship between the application and the database. Rather than sharing a single database schema with other services, each service has its own database schema. On the one hand, this approach is at odds with the idea of an enterprise-wide data model. Also, it often results in duplication of some data. However, having a database schema per service is essential if you want to benefit from microservices, because it ensures loose coupling. Figure 1-5 shows the database architecture for the sample application.

Each of the services has its own database. Moreover, a service can use a type of database that is best suited to its needs, the so-called polyglot persistence architecture. For example, Driver Management, which finds drivers close to a potential passenger, must use a database that supports efficient geo-queries.

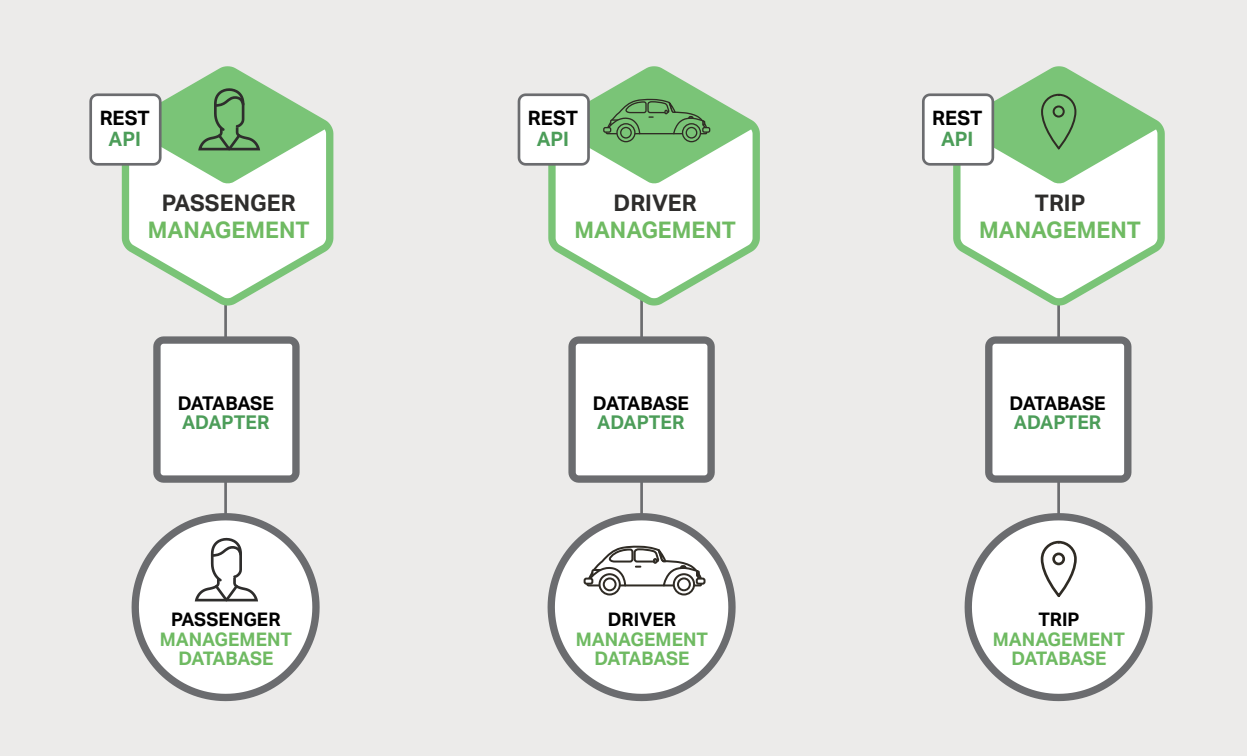

Figure 1-5. Database architecture for the taxi-hailing application.

On the surface, the Microservices Architecture pattern is similar to SOA. With both approaches, the architecture consists of a set of services. However, one way to think about the Microservices Architecture pattern is that it's SOA without the commercialization and perceived baggage of web service specifications (WS-*) and an Enterprise Service Bus (ESB). Microservice-based applications favor simpler, lightweight protocols such as REST, rather than WS-*. They also very much avoid using ESBs and instead implement ESB-like functionality in the microservices themselves. The Microservices Architecture pattern also rejects other parts of SOA, such as the concept of a canonical schema for data access.

The Benefits of Microservices

The Microservices Architecture pattern has a number of important benefits. First, it tackles the problem of complexity. It decomposes what would otherwise be a monstrous monolithic application into a set of services. While the total amount of functionality is unchanged, the application has been broken up into manageable chunks or services. Each service has a well-defined boundary in the form of a remote procedure call (RPC)-driven or message-driven API. The Microservices Architecture pattern enforces a level of modularity that in practice is extremely difficult to achieve with a monolithic code base. Consequently, individual services are much faster to develop, and much easier to understand and maintain.

Second, this architecture enables each service to be developed independently by a team that is focused on that service. The developers are free to choose whatever technologies make sense, provided that the service honors the API contract. Of course, most organizations would want to avoid complete anarchy by limiting technology options. However, this freedom means that developers are no longer obligated to use the possibly obsolete technologies that existed at the start of a new project. When writing a new service, they have the option of using current technology. Moreover, since services are relatively small, it becomes more feasible to rewrite an old service using current technology.

Third, the Microservices Architecture pattern enables each microservice to be deployed independently. Developers never need to coordinate the deployment of changes that are local to their service. These kinds of changes can be deployed as soon as they have been tested. The UI team can, for example, perform A|B testing and rapidly iterate on UI changes. The Microservices Architecture pattern makes continuous deployment possible.

Finally, the Microservices Architecture pattern enables each service to be scaled independently. You can deploy just the number of instances of each service that satisfy its capacity and availability constraints. Moreover, you can use the hardware that best matches a service's resource requirements. For example, you can deploy a CPU-intensive image processing service on EC2 Compute Optimized instances and deploy an in-memory database service on EC2 Memory-optimized instances.

The Drawbacks of Microservices

As Fred Brooks wrote almost 30 years ago, in *The Mythical Man-Month*, there are no silver bullets. Like every other technology, the Microservices architecture pattern has drawbacks. One drawback is the name itself. The term *microservice* places excessive emphasis on service size. In fact, there are some developers who advocate for building extremely fine-grained 10-100 LOC services. While small services are preferable, it's important to remember that small services are a means to an end, and not the primary goal. The goal of microservices is to sufficiently decompose the application in order to facilitate agile application development and deployment.

Another major drawback of microservices is the complexity that arises from the fact that a microservices application is a distributed system. Developers need to choose and implement an inter-process communication mechanism based on either messaging or RPC. Moreover, they must also write code to handle partial failure, since the destination of a request might be slow or unavailable. While none of this is rocket science, it's much more complex than in a monolithic application, where modules invoke one another via language-level method/procedure calls.

Another challenge with microservices is the partitioned database architecture. Business transactions that update multiple business entities are fairly common. These kinds of transactions are trivial to implement in a monolithic application because there is a single database. In a microservices-based application, however, you need to update multiple

databases owned by different services. Using distributed transactions is usually not an option, and not only because of the CAP theorem. They simply are not supported by many of today's highly scalable NoSQL databases and messaging brokers. You end up having to use an eventual consistency-based approach, which is more challenging for developers.

Testing a microservices application is also much more complex. For example, with a modern framework such as Spring Boot, it is trivial to write a test class that starts up a monolithic web application and tests its REST API. In contrast, a similar test class for a service would need to launch that service and any services that it depends upon, or at least configure stubs for those services. Once again, this is not rocket science, but it's important to not underestimate the complexity of doing this.

Another major challenge with the Microservices Architecture pattern is implementing changes that span multiple services. For example, let's imagine that you are implementing a story that requires changes to services A, B, and C, where A depends upon B and B depends upon C. In a monolithic application you could simply change the corresponding modules, integrate the changes, and deploy them in one go. In contrast, in a Microservices Architecture pattern you need to carefully plan and coordinate the rollout of changes to each of the services. For example, you would need to update service C, followed by service B, and then finally service A. Fortunately, most changes typically impact only one service; multi-service changes that require coordination are relatively rare.

Deploying a microservices-based application is also much more complex. A monolithic application is simply deployed on a set of identical servers behind a traditional load balancer. Each application instance is configured with the locations (host and ports) of infrastructure services such as the database and a message broker. In contrast, a microservice application typically consists of a large number of services. For example, Hailo has 160 different services and Netflix has more than 600, according to Adrian Cockcroft.

Each service will have multiple runtime instances. That's many more moving parts that need to be configured, deployed, scaled, and monitored. In addition, you will also need to implement a service discovery mechanism that enables a service to discover the locations (hosts and ports) of any other services it needs to communicate with. Traditional trouble ticket-based and manual approaches to operations cannot scale to this level of complexity. Consequently, successfully deploying a microservices application requires greater control of deployment methods by developers and a high level of automation.

One approach to automation is to use an off-the-shelf platform-as-a-service (PaaS) such as Cloud Foundry. A PaaS provides developers with an easy way to deploy and manage their microservices. It insulates them from concerns such as procuring and configuring IT resources. At the same time, the systems and network professionals who configure the PaaS can ensure compliance with best practices and with company policies.

Another way to automate the deployment of microservices is to develop what is essentially your own PaaS. One typical starting point is to use a clustering solution, such as Kubernetes, in conjunction with a container technology such as Docker. Later in this

ebook we will look at how software-based application delivery approaches like NGINX, which easily handles caching, access control, API metering, and monitoring at the microservice level, can help solve this problem.

Summary

Building complex applications is inherently difficult. The Monolithic Architecture pattern only makes sense for simple, lightweight applications. You will end up in a world of pain if you use it for complex applications. The Microservices Architecture pattern is the better choice for complex, evolving applications, despite the drawbacks and implementation challenges.

In later chapters, I'll dive into the details of various aspects of the Microservices Architecture pattern and discuss topics such as service discovery, service deployment options, and strategies for refactoring a monolithic application into services.

Microservices in Action: NGINX Plus as a Reverse Proxy Server

by Floyd Smith

NGINX powers more than 50% of the top 10,000 websites, and that's largely because of its capabilities as a reverse proxy server. You can "drop NGINX in front of" current applications and even database servers to gain all sorts of capabilities – higher performance, greater security, scalability, flexibility, and more. All with little or no change to your existing application and configuration code. For sites suffering performance stress – or anticipating high loads in the future – the effect may seem little short of miraculous.

So what does this have to do with microservices? Implementing a reverse proxy server, and using the other capabilities of NGINX, gives you architectural flexibility. A reverse proxy server, static and application file caching, and SSL/TLS and HTTP/2 termination all take load off your application, freeing it to "do what only it" – the application – "can do".

NGINX also serves as a load balancer, a crucial role in microservices implementations. The advanced features in NGINX Plus, including sophisticated load-balancing algorithms, multiple methods for session persistence, and management and monitoring, are especially useful with microservices. (NGINX has recently added support for service discovery using DNS `SRV` records, a cutting-edge feature.) And, as mentioned in this chapter, NGINX can help in automating the deployment of microservices.

In addition, NGINX provides the necessary functionality to power the three models in the NGINX Microservices Reference Architecture. The Proxy Model uses NGINX as an API Gateway; the Router Mesh model uses an additional NGINX server as a hub for inter-process communication; and the Fabric Model uses one NGINX server per microservice, controlling HTTP traffic and, optionally, implementing SSL/TLS between microservices, a breakthrough capability.

2 Using an API Gateway

The first chapter in this seven-chapter book 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 endpoints, which are typically replicated, with load balancing used to distribute traffic among them.

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 propose 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, Figure 2-1 shows what you will see when scrolling through the product details in Amazon's Android mobile application.

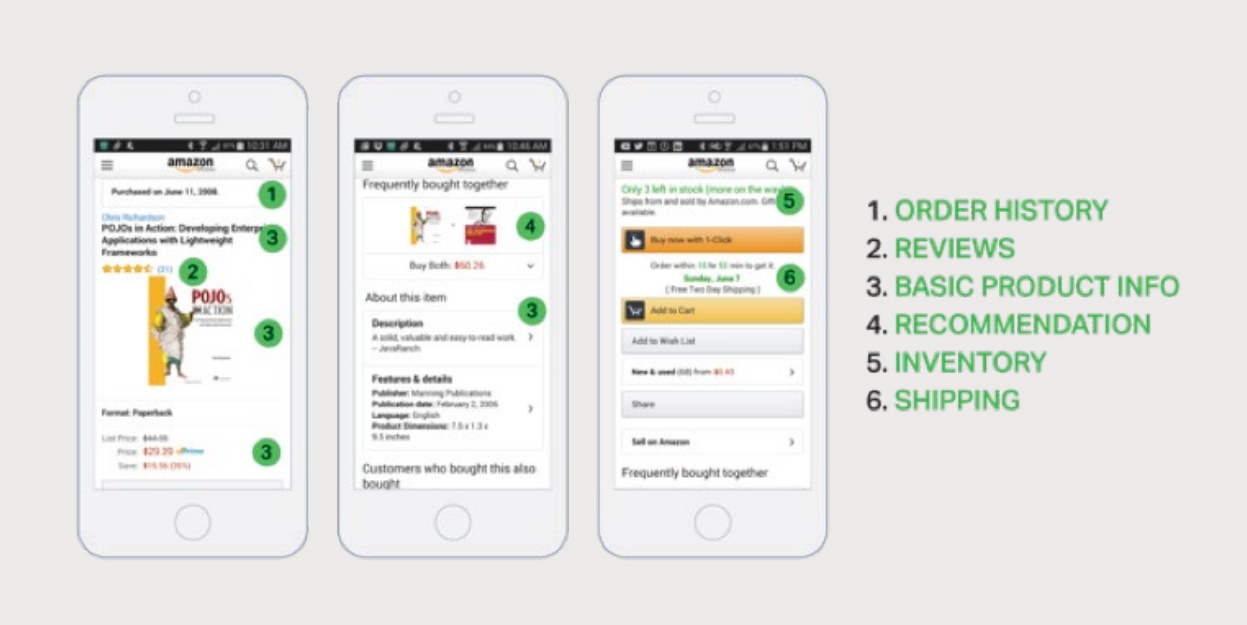

Figure 2-1. A sample shopping application.

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:

1. Number of items in the shopping cart
2. Order history
3. Customer reviews
4. Low inventory warning
5. Shipping options
6. 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
7. Alternative purchasing options

When using a monolithic application architecture, a mobile client retrieves this data by making a single REST call to the application, such as:

```
GET api.company.com/productdetails/productId
```

A load balancer routes the request to one of several identical application instances. The application then queries 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 sample product-specific page:

- Shopping Cart Service – Number of items in the shopping cart
- Order Service – Order history
- Catalog Service – Basic product information, such as product 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

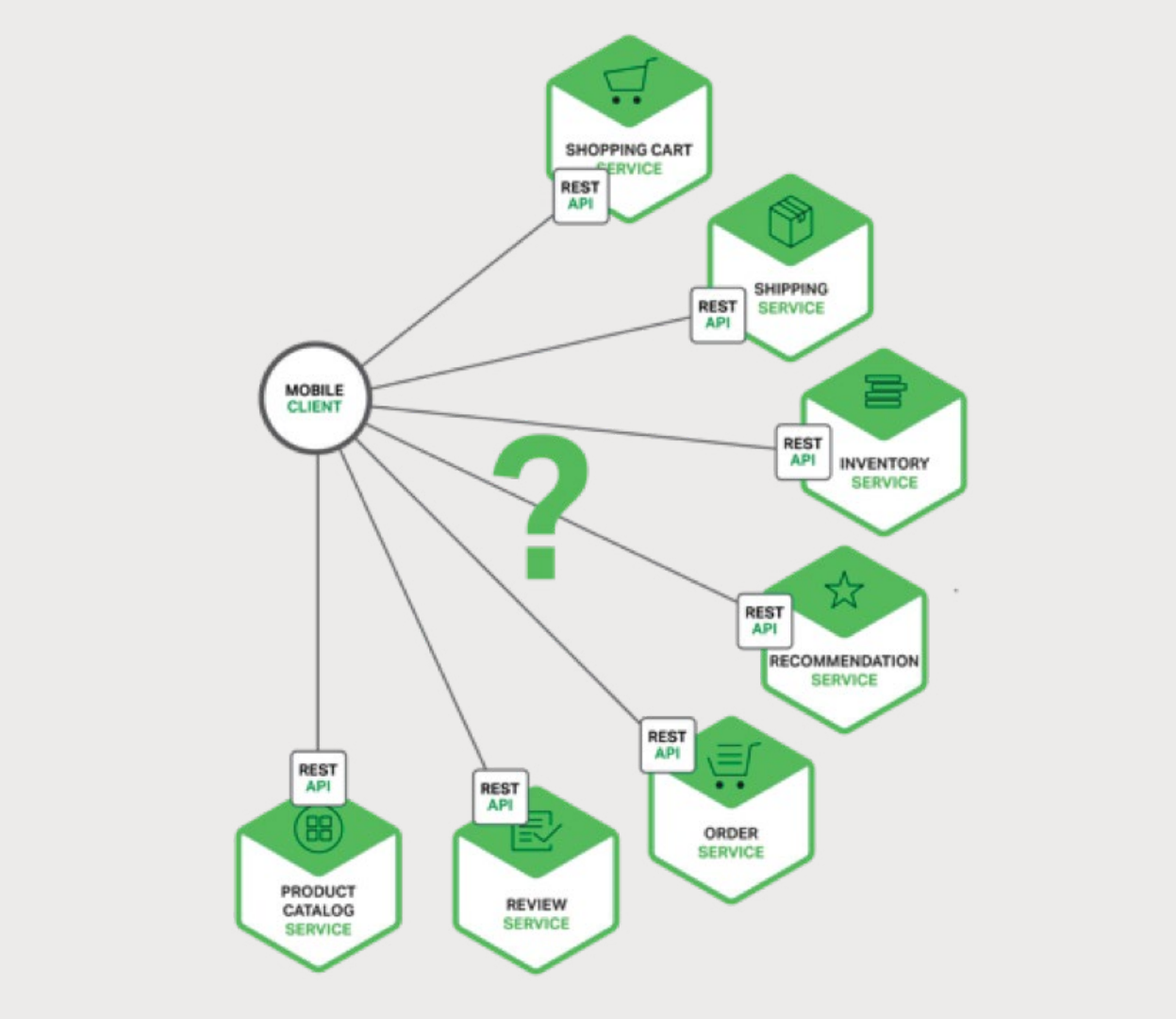

Figure 2-2. Mapping a mobile client's needs to relevant microservices.

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-specific page information, 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.

Figure 2-3 shows how an API Gateway typically fits into the architecture.

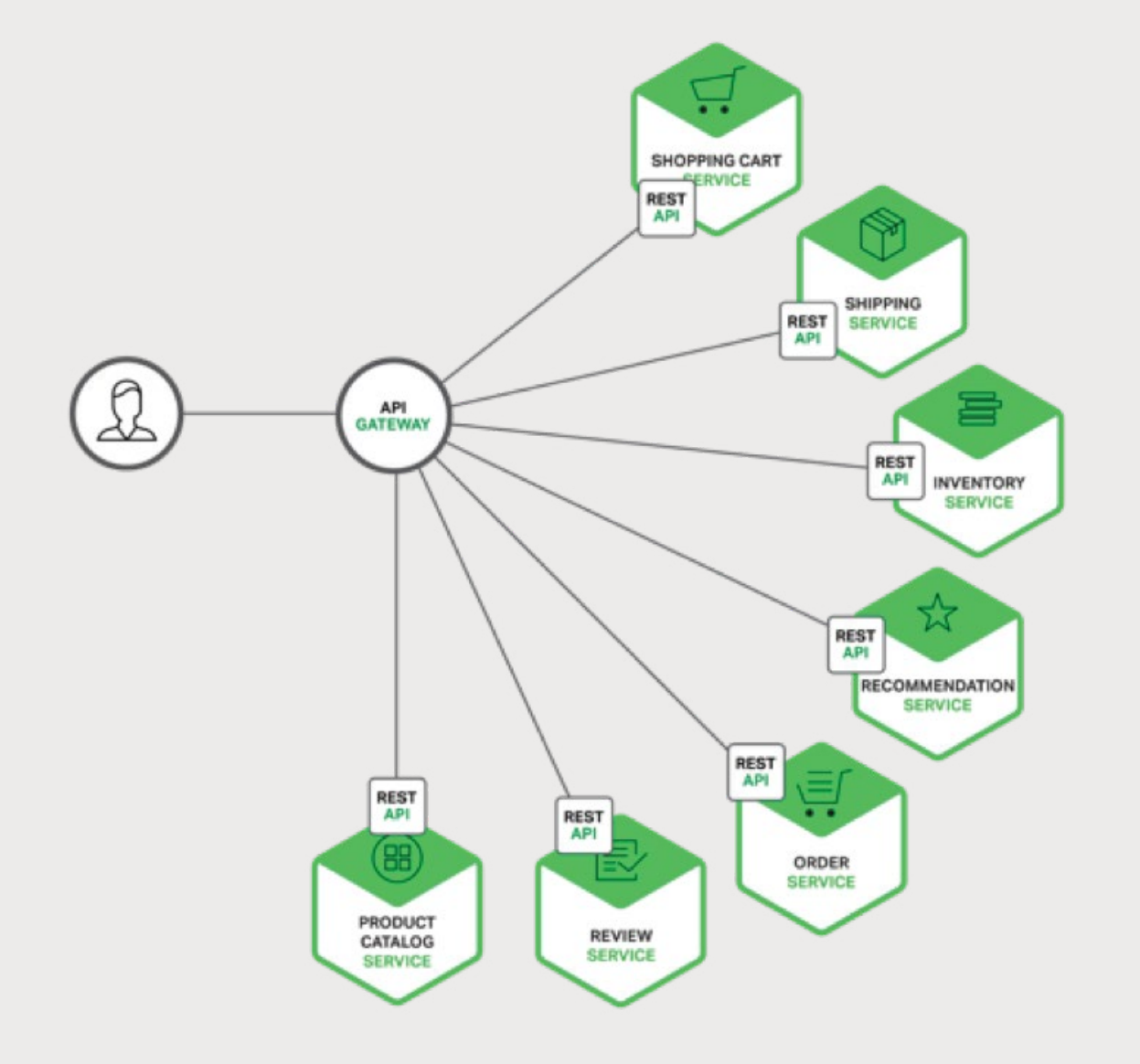

Figure 2-3. Using an API Gateway with microservices.

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 information, 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, non-blocking 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, finding the needed locations is a non-trivial 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. Chapter 4 describes 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 the API Gateway should return an error to the client.

The API Gateway could also return cached data if that is 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 minimally 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 chapter, we will look at communication between services.

Microservices in Action: NGINX Plus as an API Gateway

by Floyd Smith

This chapter discusses how an API Gateway serves as a single entry point into a system. And it can handle other functions such as load balancing, caching, monitoring, protocol translation, and others – while NGINX, when implemented as a reverse proxy server, functions as a single entry point into a system and supports all the additional functions mentioned for an API Gateway. So using NGINX as the host for an API Gateway can work very well indeed.

Thinking of NGINX as an API Gateway is not an idea that's original to this ebook. NGINX Plus is a leading platform for managing and securing HTTP-based API traffic. You can implement your own API Gateway or use an existing API management platform, many of which leverage NGINX.

Reasons for using NGINX Plus as an API Gateway include:

- **Access management** – You can use a variety of access control list (ACL) methods and easily implement SSL/TLS, either at the web application level as is typical, or also down to the level of each individual microservice.
- **Manageability and resilience** – You can update your NGINX Plus-based API server without downtime, using the NGINX dynamic reconfiguration API, a Lua module, Perl, live restarts without downtime, or changes driven by Chef, Puppet, ZooKeeper, or DNS.
- **Integration with third-party tools** – NGINX Plus is already integrated with leading-edge tools such as 3scale, Kong, and the MuleSoft integration platform (to mention only tools described on the NGINX website.)

NGINX Plus is used extensively as an API Gateway in the NGINX Microservices Reference Architecture. Use the articles assembled here and, when publicly available, the MRA, for examples of how to implement this in your own applications.

3 Inter-Process Communication

This is the third chapter in this ebook about building applications with a microservices architecture. Chapter 1 introduces the Microservices Architecture pattern, compares it with the Monolithic Architecture pattern, and discusses the benefits and drawbacks of using microservices. Chapter 2 describes how clients of an application communicate with the microservices via an intermediary known as an API Gateway. In this chapter, we take a look at how the services within a system communicate with one another. Chapter 4 explores the closely related problem of service discovery.

Introduction

In a monolithic application, components invoke one another via language-level method or function calls. In contrast, a microservices-based application is a distributed system running on multiple machines. Each service instance is typically a process.

Consequently, as Figure 3-1 shows, services must interact using an inter-process communication (IPC) mechanism.

Later on we will look at specific IPC technologies, but first let's explore various design issues.

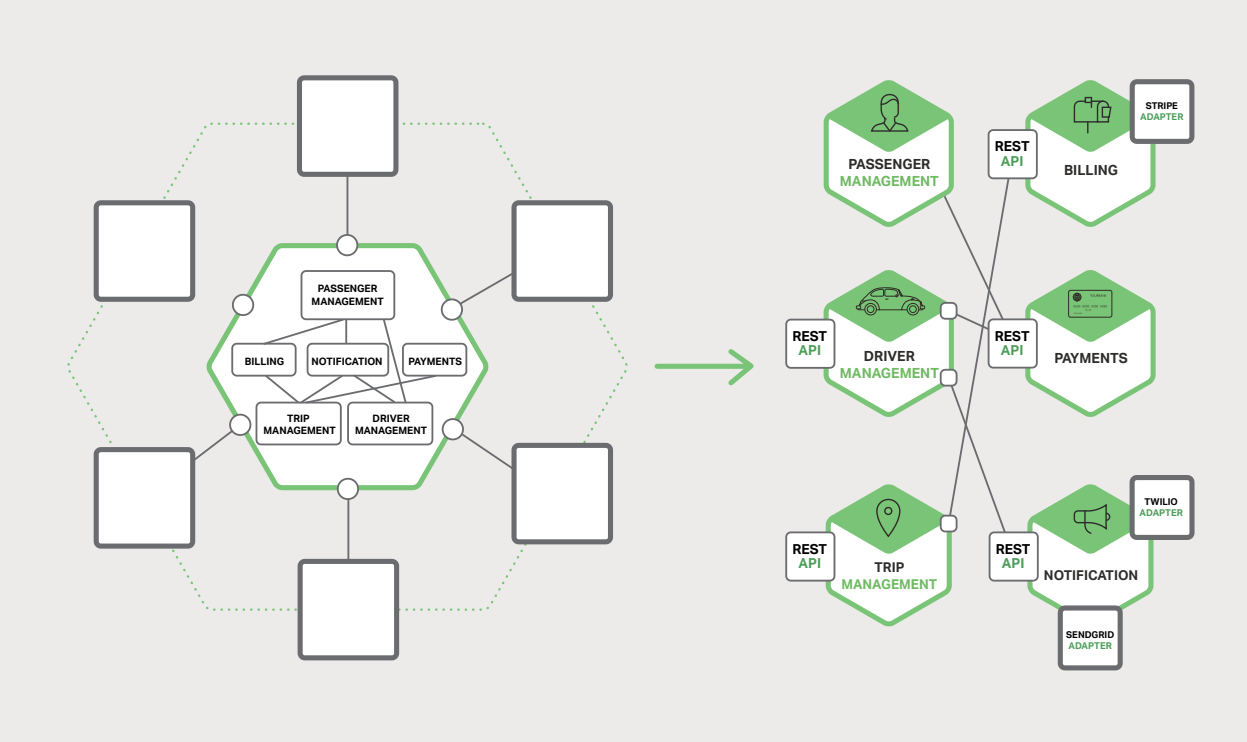

Figure 3-1. Microservices use inter-process communication to interact.

Interaction Styles

When selecting an IPC mechanism for a service, it is useful to think first about how services interact. There are a variety of client↔service interaction styles. They can be categorized along two dimensions. The first dimension is whether the interaction is one-to-one or one-to-many:

- One-to-one – Each client request is processed by exactly one service instance.
- One-to-many – Each request is processed by multiple service instances.

The second dimension is whether the interaction is synchronous or asynchronous:

- Synchronous – The client expects a timely response from the service and might even block while it waits.
- Asynchronous – The client doesn't block while waiting for a response, and the response, if any, isn't necessarily sent immediately.

The following table shows the various interaction styles.

	ONE-TO-ONE	ONE-TO-MANY
SYNCHRONOUS	Request/response	—
ASYNCHRONOUS	Notification	Publish/subscribe
	Request/async response	Publish/async responses

Table 3-1. Inter-process communication styles.

There are the following kinds of one-to-one interactions, both synchronous (request/response) and asynchronous (notification and request/async response):

- Request/response – A client makes a request to a service and waits for a response. The client expects the response to arrive in a timely fashion. In a thread-based application, the thread that makes the request might even block while waiting.
- Notification (a.k.a. a one-way request) – A client sends a request to a service but no reply is expected or sent.
- Request/async response – A client sends a request to a service, which replies asynchronously. The client does not block while waiting and is designed with the assumption that the response might not arrive for a while.

There are the following kinds of one-to-many interactions, both of which are asynchronous:

- Publish/subscribe – A client publishes a notification message, which is consumed by zero or more interested services.
- Publish/async responses – A client publishes a request message, and then waits a certain amount of time for responses from interested services.

Each service typically uses a combination of these interaction styles. For some services, a single IPC mechanism is sufficient. Other services might need to use a combination of IPC mechanisms.

Figure 3-2 shows how services in a taxi-hailing application might interact when the user requests a trip.

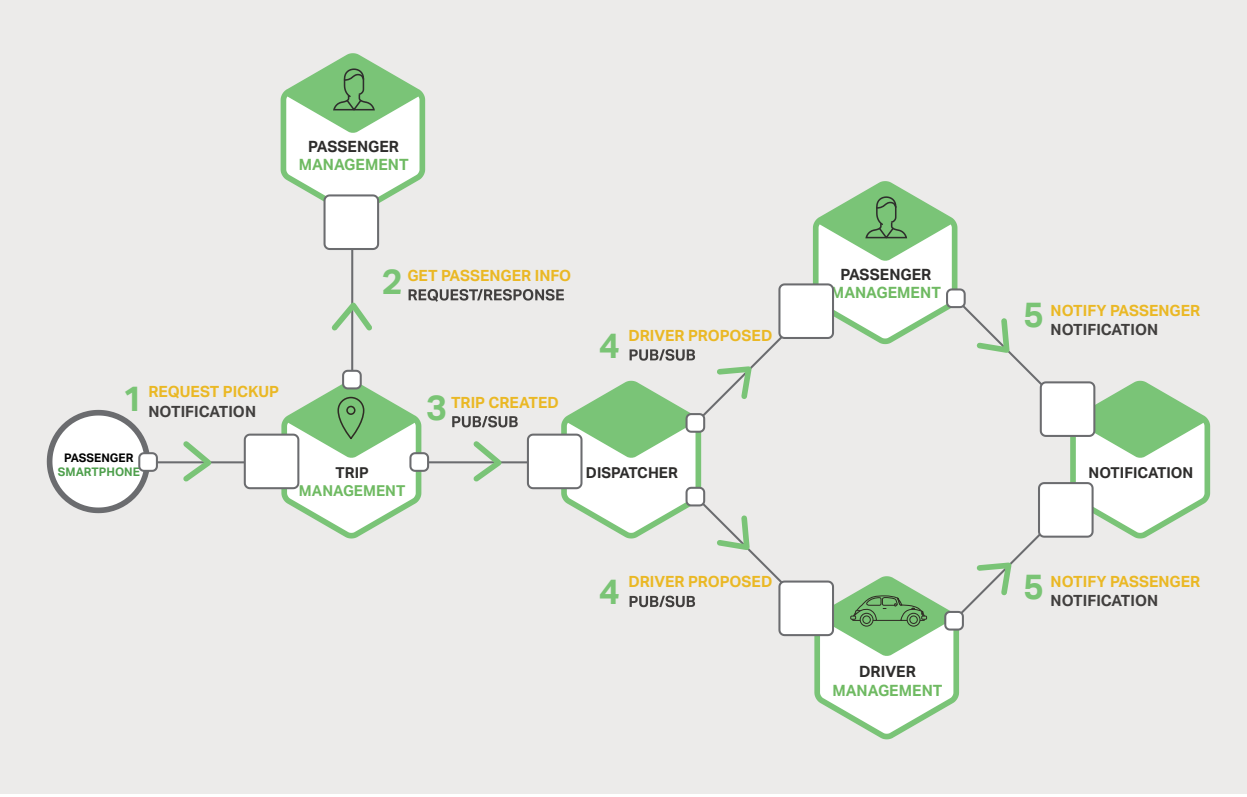

Figure 3-2. Using multiple IPC mechanisms for service interactions.

The services use a combination of notifications, request/response, and publish/subscribe. For example, the passenger's smartphone sends a notification to the Trip Management service to request a pickup. The Trip Management service verifies that the passenger's account is active by using request/response to invoke the Passenger Management service. The Trip Management service then creates the trip and uses publish/subscribe to notify other services including the Dispatcher, which locates an available driver.

Now that we have looked at interaction styles, let's take a look at how to define APIs.

Defining APIs

A service's API is a contract between the service and its clients. Regardless of your choice of IPC mechanism, it's important to precisely define a service's API using some kind of interface definition language (IDL). There are even good arguments for using an API-first approach to defining services. You begin the development of a service by writing the interface definition and reviewing it with the client developers. It is only after iterating on the API definition that you implement the service. Doing this design up front increases your chances of building a service that meets the needs of its clients.

As you will see later in this article, the nature of the API definition depends on which IPC mechanism you are using. If you are using messaging, the API consists of the message channels and the message types. If you are using HTTP, the API consists of the URLs and the request and response formats. Later on we will describe some IDLs in more detail.

Evolving APIs

A service's API invariably changes over time. In a monolithic application it is usually straightforward to change the API and update all the callers. In a microservices-based application it is a lot more difficult, even if all of the consumers of your API are other services in the same application. You usually cannot force all clients to upgrade in lockstep with the service. Also, you will probably incrementally deploy new versions of a service such that both old and new versions of a service will be running simultaneously. It is important to have a strategy for dealing with these issues.

How you handle an API change depends on the size of the change. Some changes are minor and backward compatible with the previous version. You might, for example, add attributes to requests or responses. It makes sense to design clients and services so that they observe the robustness principle. Clients that use an older API should continue to work with the new version of the service. The service provides default values for the missing request attributes and the clients ignore any extra response attributes. It is important to use an IPC mechanism and a messaging format that enable you to easily evolve your APIs.

Sometimes, however, you must make major, incompatible changes to an API. Since you can't force clients to upgrade immediately, a service must support older versions of the

API for some period of time. If you are using an HTTP-based mechanism such as REST, one approach is to embed the version number in the URL. Each service instance might handle multiple versions simultaneously. Alternatively, you could deploy different instances that each handle a particular version.

Handling Partial Failure

As mentioned in Chapter 2 about the API Gateway, in a distributed system there is the ever-present risk of partial failure. Since clients and services are separate processes, a service might not be able to respond in a timely way to a client's request. A service might be down because of a failure or for maintenance. Or the service might be overloaded and responding extremely slowly to requests.

Consider, for example, the product details scenario from Chapter 2. Let's imagine that the Recommendation Service is unresponsive. A naive implementation of a client might block indefinitely waiting for a response. Not only would that result in a poor user experience, but also, in many applications it would consume a precious resource such as a thread. Eventually the runtime would run out of threads and become unresponsive, as shown in Figure 3-3.

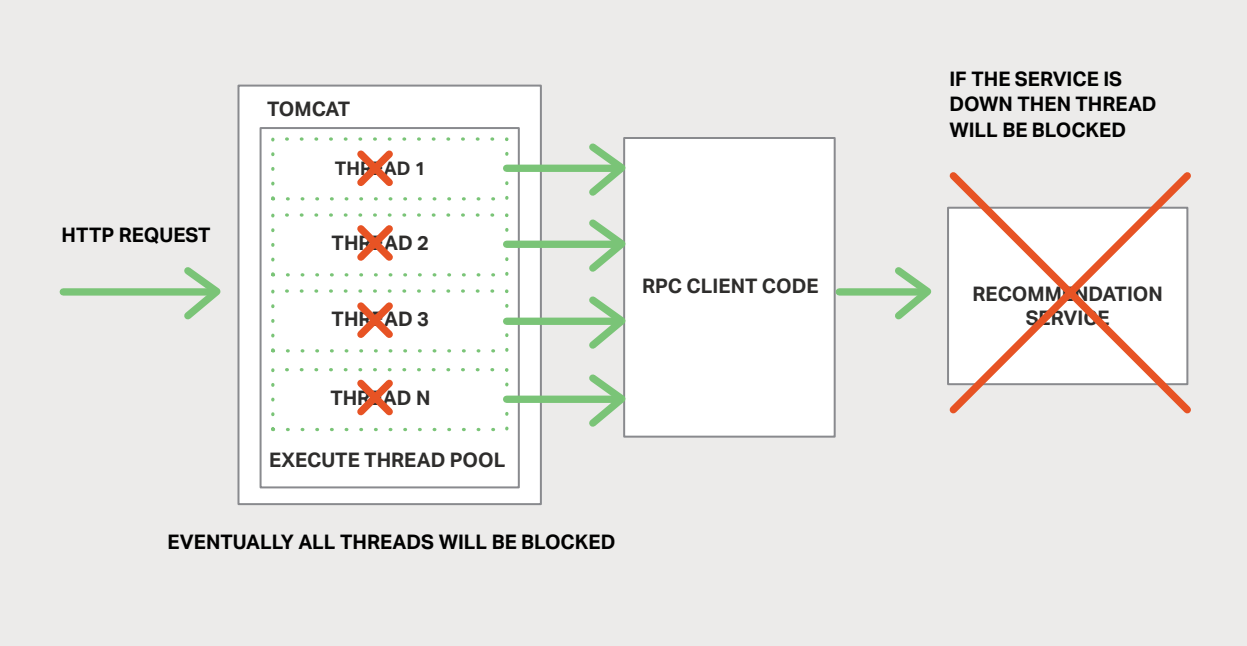

Figure 3-3. Threads block due to an unresponsive service.

To prevent this problem, it is essential that you design your services to handle partial failures.

A good approach to follow is the one described by Netflix. The strategies for dealing with partial failures include:

- Network timeouts – Never block indefinitely and always use timeouts when waiting for a response. Using timeouts ensures that resources are never tied up indefinitely.
- Limiting the number of outstanding requests – Impose an upper bound on the number of outstanding requests that a client can have with a particular service. If the limit has been reached, it is probably pointless to make additional requests, and those attempts need to fail immediately.
- Circuit breaker pattern – Track the number of successful and failed requests. If the error rate exceeds a configured threshold, trip the circuit breaker so that further attempts fail immediately. If a large number of requests are failing, that suggests the service is unavailable and that sending requests is pointless. After a timeout period, the client should try again and, if successful, close the circuit breaker.
- Provide fallbacks – Perform fallback logic when a request fails. For example, return cached data or a default value, such as an empty set of recommendations.

Netflix Hystrix is an open source library that implements these and other patterns. 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.

IPC Technologies

There are lots of different IPC technologies to choose from. Services can use synchronous request/response-based communication mechanisms such as HTTP-based REST or Thrift. Alternatively, they can use asynchronous, message-based communication mechanisms such as AMQP or STOMP.

There are also a variety of different message formats. Services can use human readable, text-based formats such as JSON or XML. Alternatively, they can use a binary format (which is more efficient) such as Avro or Protocol Buffers. Later on we will look at synchronous IPC mechanisms, but first let's discuss asynchronous IPC mechanisms.

Asynchronous, Message-Based Communication

When using messaging, processes communicate by asynchronously exchanging messages. A client makes a request to a service by sending it a message. If the service is expected to reply, it does so by sending a separate message back to the client. Since the communication is asynchronous, the client does not block waiting for a reply. Instead, the client is written assuming that the reply will not be received immediately.

A message consists of headers (metadata such as the sender) and a message body. Messages are exchanged over channels. Any number of producers can send messages to a channel. Similarly, any number of consumers can receive messages from a channel. There are two kinds of channels, point-to-point and publish-subscribe:

- A point-to-point channel delivers a message to exactly one of the consumers that are reading from the channel. Services use point-to-point channels for the one-to-one interaction styles described earlier.
- A publish-subscribe channel delivers each message to all of the attached consumers. Services use publish-subscribe channels for the one-to-many interaction styles described above.

Figure 3-4 shows how the taxi-hailing application might use publish-subscribe channels.

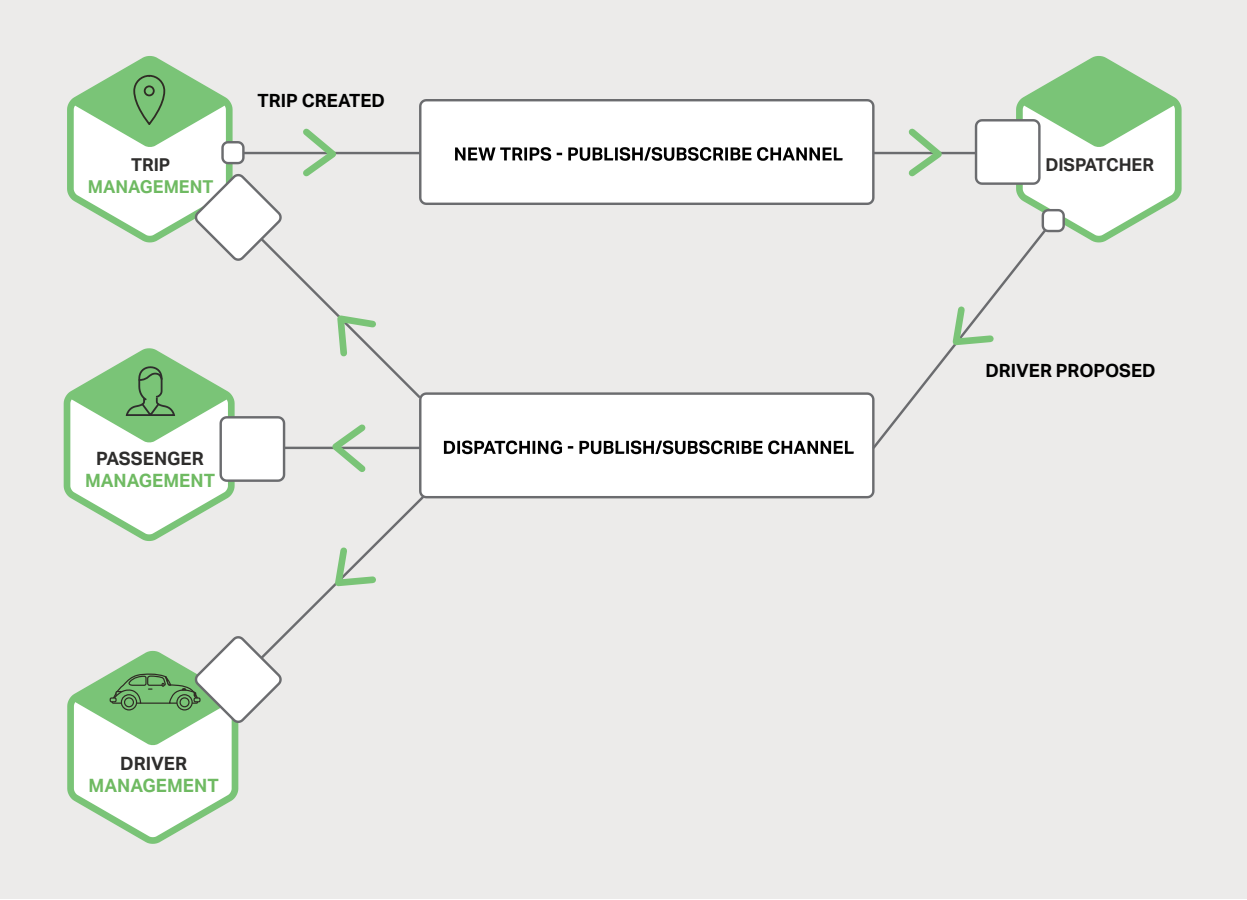

Figure 3-4. Using publish-subscribe channels in a taxi-hailing application.

The Trip Management service notifies interested services, such as the Dispatcher, about a new Trip by writing a Trip Created message to a publish-subscribe channel. The Dispatcher finds an available driver and notifies other services by writing a Driver Proposed message to a publish-subscribe channel.

There are many messaging systems to choose from. You should pick one that supports a variety of programming languages.

Some messaging systems support standard protocols such as AMQP and STOMP. Other messaging systems have proprietary but documented protocols.

There are a large number of open source messaging systems to choose from, including RabbitMQ, Apache Kafka, Apache ActiveMQ, and NSQ. At a high level, they all support some form of messages and channels. They all strive to be reliable, high-performance, and scalable. However, there are significant differences in the details of each broker's messaging model.

There are many advantages to using messaging:

- Decouples the client from the service – A client makes a request simply by sending a message to the appropriate channel. The client is completely unaware of the service instances. It does not need to use a discovery mechanism to determine the location of a service instance.
- Message buffering – With a synchronous request/response protocol, such as HTTP, both the client and service must be available for the duration of the exchange. In contrast, a message broker queues up the messages written to a channel until the consumer can process them. This means, for example, that an online store can accept orders from customers even when the order fulfillment system is slow or unavailable. The order messages simply queue up.
- Flexible client-service interactions – Messaging supports all of the interaction styles described earlier.
- Explicit inter-process communication – RPC-based mechanisms attempt to make invoking a remote service look the same as calling a local service. However, because of the laws of physics and the possibility of partial failure, they are in fact quite different. Messaging makes these differences very explicit so developers are not lulled into a false sense of security.

There are, however, some downsides to using messaging:

- Additional operational complexity – The messaging system is yet another system component that must be installed, configured, and operated. It's essential that the message broker be highly available, otherwise system reliability is impacted.
- Complexity of implementing request/response-based interaction – Request/response-style interaction requires some work to implement. Each request message must contain a reply channel identifier and a correlation identifier. The service writes a response message containing the correlation ID to the reply channel. The client uses the correlation ID to match the response with the request. It is often easier to use an IPC mechanism that directly supports request/response.

Now that we have looked at using messaging-based IPC, let's examine request/response-based IPC.

Synchronous, Request/Response IPC

When using a synchronous, request/response-based IPC mechanism, a client sends a request to a service. The service processes the request and sends back a response.

In many clients, the thread that makes the request blocks while waiting for a response. Other clients might use asynchronous, event-driven client code that is perhaps encapsulated by Futures or Rx Observables. However, unlike when using messaging, the client assumes that the response will arrive in a timely fashion.

There are numerous protocols to choose from. Two popular protocols are REST and Thrift. Let's first take a look at REST.

REST

Today it is fashionable to develop APIs in the RESTful style. REST is an IPC mechanism that (almost always) uses HTTP.

A key concept in REST is a resource, which typically represents a business object such as a Customer or Product, or a collection of such business objects. REST uses the HTTP verbs for manipulating resources, which are referenced using a URL. For example, a `GET` request returns the representation of a resource, which might be in the form of an XML document or JSON object. A `POST` request creates a new resource, and a `PUT` request updates a resource.

To quote Roy Fielding, the creator of REST:

"REST provides a set of architectural constraints that, when applied as a whole, emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems."
—Roy Fielding, *Architectural Styles and the Design of Network-based Software Architectures*

Figure 3-5 shows one of the ways that the taxi-hailing application might use REST.

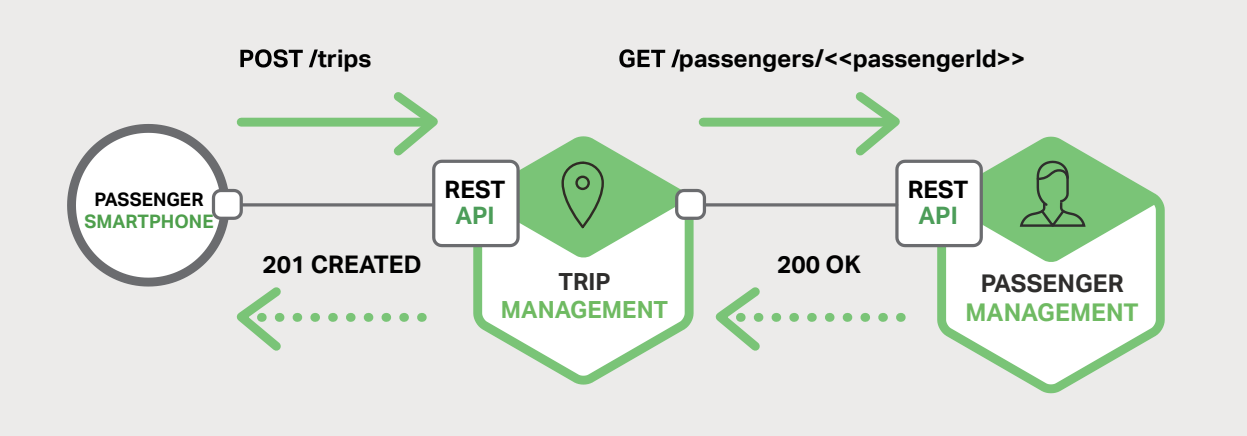

Figure 3-5. A taxi-hailing application uses RESTful interaction.