Microservices for the Enterprise

Designing, Developing, and Deploying

Kasun Indrasiri
Prabath Siriwardena

Apress®

Microservices for the Enterprise

Designing, Developing, and Deploying

Kasun Indrasiri

Prabath Siriwardena

Apress®

Microservices for the Enterprise

Kasun Indrasiri
San Jose, CA, USA

Prabath Siriwardena
San Jose, CA, USA

ISBN-13 (pbk): 978-1-4842-3857-8
ISBN-13 (electronic): 978-1-4842-3858-5
https://doi.org/10.1007/978-1-4842-3858-5

Library of Congress Control Number: 2018962968

Copyright © 2018 by Kasun Indrasiri and Prabath Siriwardena

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Jonathan Gennick
Development Editor: Laura Berendson
Coordinating Editor: Jill Balzano

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484238578. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

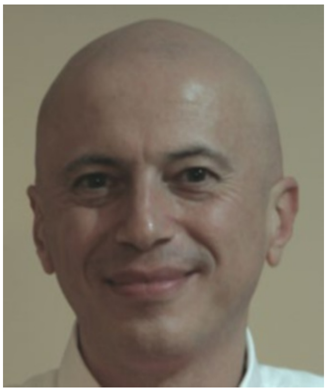
Table of Contents

About the Authors

Kasun Indrasiri is an architect, author, microservice and integration evangelist, and director of integration architecture at WSO2. He also founded the Microservices, APIs, and Integration meetup group, which is a vendor-neutral microservices meetup in the San Francisco Bay Area. He is the author of the book *Beginning WSO2 ESB* (Apress) and has worked as a software architect and a product lead with over seven years of experience in enterprise integration. He is an Apache committer and PMC member. Kasun has spoken at several conferences held in San Francisco, London, and Barcelona on topics relating to enterprise integration and microservices. He also conducts talks at Bay Area microservices, container, and cloud-native meetups, and he publishes blogs and articles on microservices. He works with many Fortune 100 companies to provide solutions in the enterprise integration and microservices domain.

Prabath Siriwardena is an identity evangelist, author, blogger, and the VP of Identity Management and Security at WSO2, with more than 11 years of industry experience in designing and building critical Identity and Access Management (IAM) infrastructure for global enterprises, including many Fortune 100/500 companies. As a technology evangelist, Prabath has published five books. He blogs on various topics from blockchain, PSD2, GDPR, and IAM, to microservices security. He also runs a YouTube channel. Prabath has spoken at many conferences, including the RSA Conference, Identiverse, European Identity Conference, Consumer Identity World USA, API World, API Strategy & Practice Con, QCon, OSCON, and WSO2Con. He has also travelled the world conducting workshops/meetups to evangelize IAM communities. He is the founder of the Silicon Valley IAM User Group, which is the largest IAM meetup in the San Francisco Bay Area.

About the Technical Reviewer



Alp Tunc is a software engineer. He graduated from Ege University, Izmir/Turkey. He completed his MSc degree while working as a research assistant. He is a software developer by heart, with 20 years of experience in the industry as a developer/architect/project manager of projects of various sizes. He has a lot of hands-on experience in a broad range of technologies. Besides technology, he loves freezing moments in spectacular photographs, trekking into the unknown, running, and reading and he is a jazz aficionado. He also loves cats and dogs.

Acknowledgments

We would first like to thank Jonathan Gennick, assistant editorial director at Apress, for evaluating and accepting our proposal for this book. Then, Jill Balzano, coordinating editor at Apress, was extremely patient and tolerant of us throughout the publishing process. Thank you very much, Jill, for your support. Laura Berendson, the development editor at Apress, also helped us toward the end. Thanks Laura! Alp Tunc served as the technical reviewer. Thanks, Alp, for your quality reviews.

Dr. Sanjiva Weerawarana, the Founder and Chief Architect at WSO2, is our constant mentor. We are truly grateful to Dr. Sanjiva for his guidance, mentorship, and support. We also express our gratitude to Tyler Jewel, the CEO at WSO2, and Paul Fremantle, the CTO at WSO2, for their direction, which helped us explore the microservices domain. Finally, we'd like to thank our families and parents; without them nothing is possible!

Introduction

The microservices architecture has become one of the most popular architectural styles in the enterprise software architecture landscape. Owing to the benefits that it brings, most enterprises are transforming their existing monolithic applications to microservices architecture-based applications. Hence, for any software architect or software engineer, it's really important to understand the key architectural concepts of the microservices architecture and how you can use those architectural principles in practice to solve real-world business use cases.

In this book, we provide the readers a comprehensive understanding of microservices architectural principles and discuss how to use those concepts in real-world scenarios. Also, without locking into a particular technology or framework, we cover a wide range of technologies and frameworks, which are most suitable for given aspects of the microservices architecture.

One other key difference of this book is that it addresses some of the fundamental challenges in building microservices in the enterprise architecture landscape, such as inter-service communication, service integration with no centralized Enterprise Service Bus (ESB), exposing microservices as APIs avoiding a centralized API gateway, determining the scope and size of a microservice, and leveraging microservices security patterns. All the concepts explained in this book are supported with real-world use cases and incorporated with samples that the reader can try out. Most of these use cases are inspired by existing microservices implementations such as Netflix and Google, as well as the authors' exposure to various meetups and conferences in the San Francisco Bay area.

This book covers some of the widely used and bleeding edge technologies and patterns in realizing microservices architecture, such as technologies for container-native deployment (Docker, Kubernetes, Helm), messaging standards and protocols (gRPC, HTTP2, Kafka, AMQP, OpenAPI, GraphQL, etc.), reactive and active microservices integration, service mesh (Istio and Linkerd), miroservice resiliency patterns (circuit breaker, timeouts, bulk-heads, etc.), security standards (OAuth 2, JWT, and certificates), using APIs, events, and streams with microservices, and building observable microservices using logging, metrics, and tracing.

CHAPTER 1

The Case for Microservices

Enterprise software architecture always evolves with new architectural styles owing to the paradigm shifts in the technology landscape and the desire to find better ways to build applications in a fast but reliable way.

The microservices architecture has been widely adopted as an architectural style that allows you to build software applications with speed and safety. The microservices architecture fosters building a software system as a collection of independent autonomous services (developed, deployed, and scaled independently) that are loosely coupled. These services form a single software application by integrating all such services and other systems.

In this chapter, we explore what microservices are, the characteristics of microservices with real-world examples, and the pros and cons of microservices in the context of enterprise software architecture.

To better understand what microservices are, we need to look at some of the architectural styles used prior to microservices, and how the enterprise architecture has evolved to embrace the microservices architecture.

From a Monolith to a Microservices Architecture

Exploring the evolution of enterprise architecture from monolithic applications to microservices is a great way to understand the key motivations and characteristics of microservices. Let's begin our discussion with monolithic applications.

© Kasun Indrasiri and Prabath Siriwardena 2018
K. Indrasiri and P. Siriwardena, *Microservices for the Enterprise*, https://doi.org/10.1007/978-1-4842-3858-5_1

Monolithic Applications

Enterprise software applications are designed to facilitate numerous business requirements. In the monolithic architecture style, all the business functionalities are piled into a single monolithic application and built as a single unit.

Consider a real-world example to understand monolithic applications further. Figure 1-1 shows an online retail application, which is built using the monolithic architecture style.

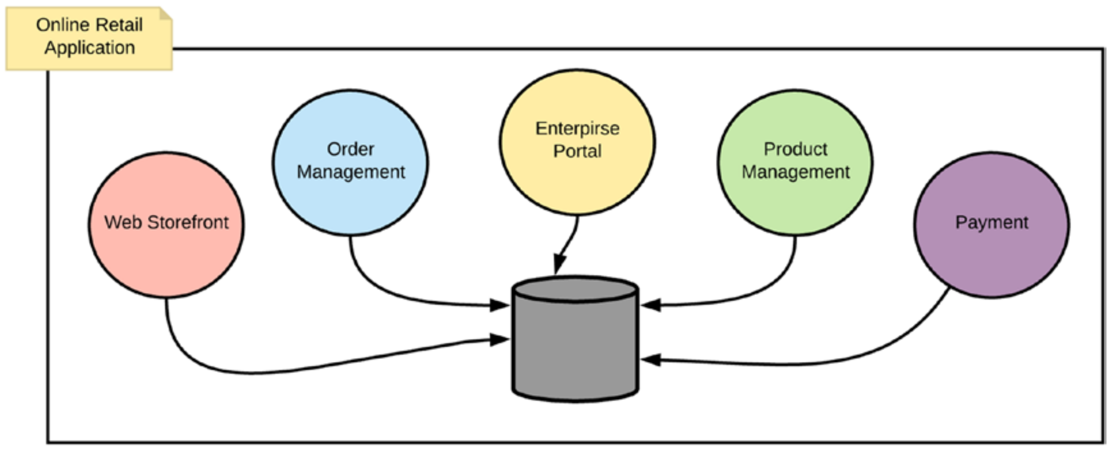


Figure 1-1. *Online retail application developed with a monolithic architecture*

The entire retail application is a collection of several components, such as order management, payments, product management, and so on. Each of these components offers a wide range of business functionalities. Adding or modifying a functionality to a component was extremely expensive owing to its monolithic nature. Also, to facilitate the overall business requirements, these components had to communicate with each other. The communications between these components were often built on top of proprietary protocols and standards, and they were based on the point-to-point communication style. Hence, modifying or replacing a given component was also quite complicated. For example, if the retail enterprise wanted to switch to a new order management system while keeping the rest, doing so would require quite a lot of changes to the other existing components too.

We can generalize the common characteristics of monolithic application as follows:

- Designed, developed, and deployed as a single unit.
- Overwhelmingly complex for most of the real-world business use cases, which leads to nightmares in maintaining, upgrading, and adding new features.
- It’s hard to practice Agile development and delivery methodologies. Since the application has to be built as a single unit, most of the business capabilities that it offers cannot have their own lifecycles.
- You must redeploy the entire application in order to update any part of it.
- As the monolithic app grows, it may take longer and longer to start up, which adds to the overall cost.
- It has to be scaled as a single application and is difficult to scale with conflicting resource requirements. (For example, since a monolithic application offers multiple business capabilities, one capability may require more CPU while another requires more memory. It’s hard to cater to the individual needs of these capabilities.)
- One unstable service can bring the whole application down.
- It’s very difficult to adopt new technologies and frameworks, as all the functionalities have to build on homogeneous technologies/frameworks. (For example, if you are using Java, all new projects have to be based on Java, even if that are better alternative technologies out there.)

As a solution to some of the limitations of the monolithic application architecture, Service Oriented Architecture (SOA) and Enterprise Service Bus (ESB) emerged.

SOA and ESB

SOA tries to combat the challenges of large monolithic applications by segregating the functionalities of monolithic applications into reusable, loosely coupled entities called *services*. These services are accessible via calls over the network.

- A service is a self-contained implementation of a well-defined business functionality that is accessible over the network. Applications in SOA are built based on services.
- Services are software components with well-defined interfaces that are implementation-independent. An important aspect of SOA is the separation of the service interface (the what) from its implementation (the how).
- The consumers are only concerned about the service interface and do not care about its implementation.
- Services are self-contained (perform predetermined tasks) and loosely coupled (for independence).
- Services can be dynamically discovered. The consumers often don't need to know the exact location and other details of a service. They can discover the service's metadata via a service metadata repository or a service registry. When there's a change to the service metadata, the service can update its metadata in the service registry.
- Composite services can be built from aggregates of other services.

With the SOA paradigm, each business functionality is built as a (coarse-grained) service (often implemented as Web Services) with several sub-functionalities. These services are deployed inside an application server. When it comes to the consumption of business functionalities, we often need to integrate/plumb multiple such services (and create composite services) and other systems. Enterprise Service Bus (ESB) is used to integrate those services, data, and systems. Consumers use the composite services exposed from the ESB layer. Hence, ESB is used as the centralized bus (see Figure 1-2) that connects all these services and systems.

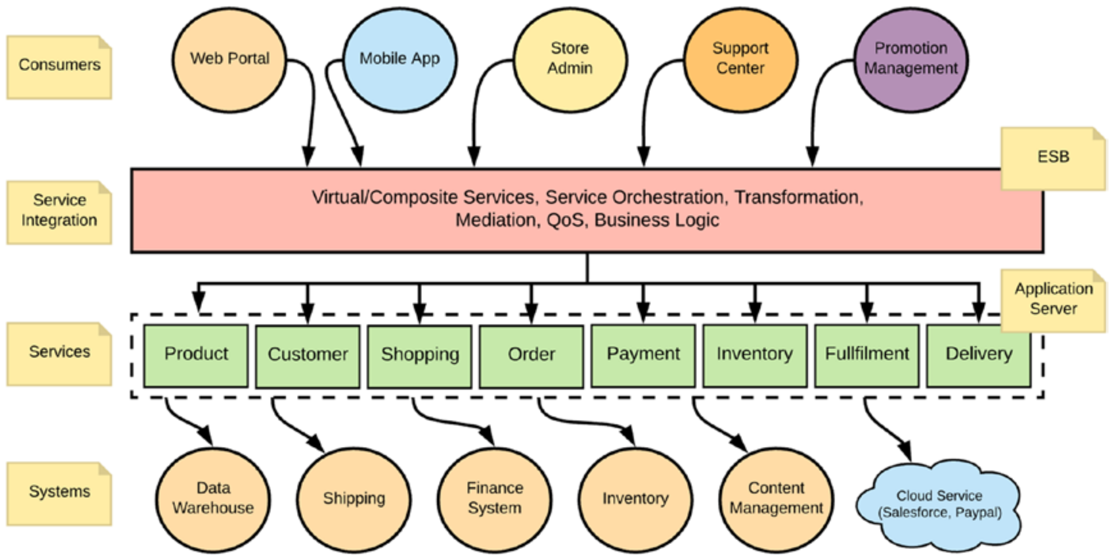


Figure 1-2. *SOA/ESB style based online retail system*

For example, let's go back to our online retail application use case. Figure 1-2 illustrates the implementation of the online retail application using SOA/web services. Here we have defined multiple web services that cater to various business capabilities such as products, customers, shopping, orders, payments, etc. At the ESB layer, we may integrate such business capabilities and create composite business capabilities, which are exposed to the consumers. Or the ESB layer may just be used to expose the functionalities as it is, with additional cross-cutting features such as security. So, obviously the ESB layer also contains a significant portion of the business logic of the entire application. Other cross-cutting concerns such as security, monitoring, and analytics may also be applied at the ESB layer. The ESB layer is a monolithic entity where all developers share the same runtime to develop/deploy their service integrations.

APIs

Exposing business functionalities as managed services or APIs has become a key requirement of the modern enterprise architecture. However, web services/SOA is not really the ideal solution to cater to such requirements, due to the complexity of the Web Service-related technologies such as SOAP (used as the message format for inter-service communication), WS-Security (to secure messaging between services), WSDLs (to define the service contract), etc., and the lack of features to build an ecosystem around APIs (self-servicing, etc.)

Therefore, most organizations put a new API Management/API Gateway layer on top of the existing SOA implementations. This layer is known as the *API façade*, and it exposes a simple API for a given business functionality and hides all the internal complexities of the ESB/Web Services layer. The API layer is also used for security, throttling, caching, and monetization.

For example, Figure 1-3 introduces an API gateway on top of the ESB layer. All the business capabilities offered from our online retail application are now being exposed as managed APIs. The API management layer is not just to expose functionalities as managed APIs, but you will be able to build a whole ecosystem of business capabilities and their consumers.

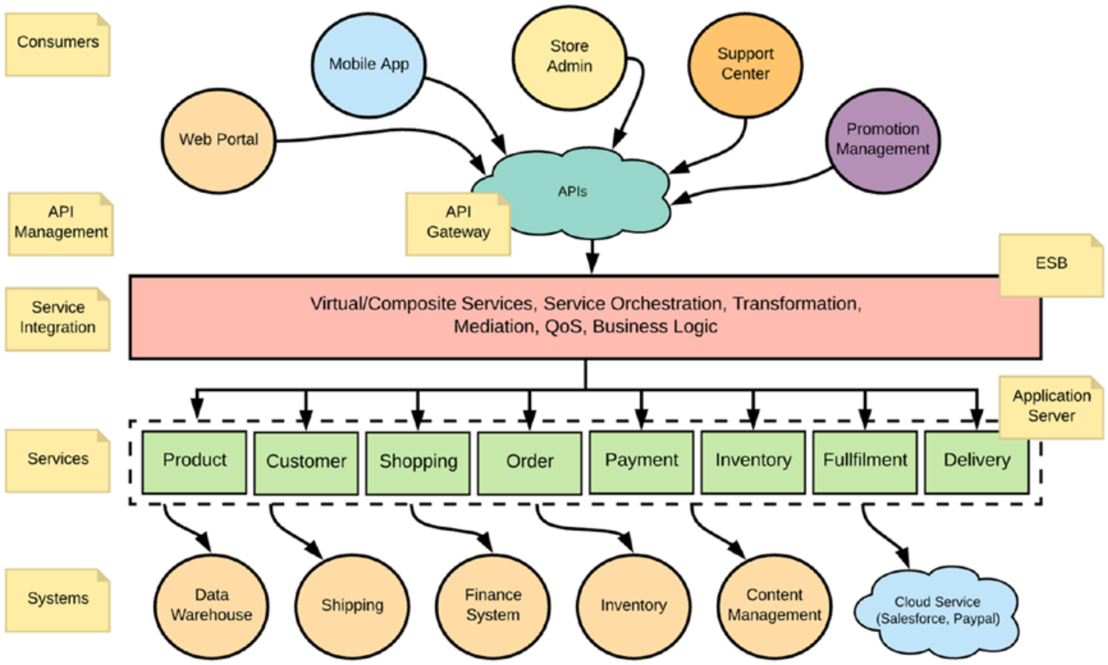


Figure 1-3. *Exposing business functionalities as managed APIs through an API Gateway layer*

With the increasing demand for complex business capabilities, the monolithic architecture can no longer cater to the modern enterprise software application development. The centralized nature of monolithic applications results in the lack of being able to scale applications independently, inter-application dependencies that hinder independent application development and deployment, reliability issues due to the centralized nature and the constraints on using diverse technologies for application development. To overcome most of these limitations and to cater to the modern, complex, and decentralized application needs, a new architecture paradigm must be conceived.

The microservices architecture has emerged as a better architecture paradigm to overcome the drawbacks of the ESB/SOA architecture as well as the conventional monolithic application architecture.

What Is a Microservice?

The foundation of the microservices architecture is about developing a single application as a suite of small and independent services that are running in their own processes, developed and deployed independently.

As illustrated in Figure 1-4, the online retail software application can be transformed into a microservices architecture by breaking the monolithic application layer into independent and business functionality oriented services. Also, we got rid of the central ESB by breaking its functionalities into each service, so that the services take care of the inter-service communication and composition logic.

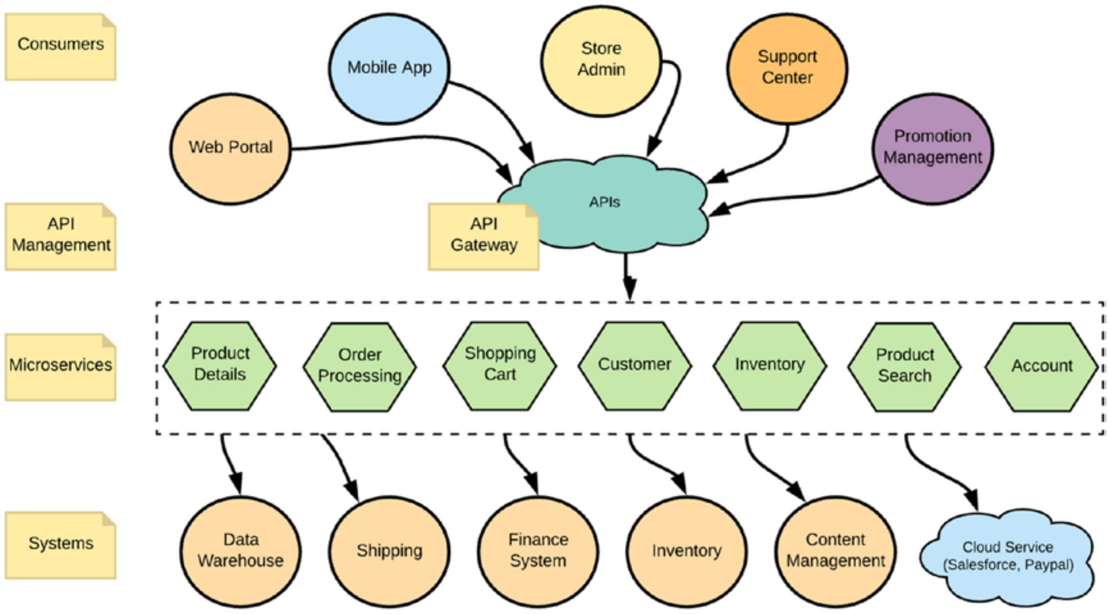


Figure 1-4. *An online retail application built using a microservices architecture*

Therefore, each microservice at the microservices layer offers a well-defined business capability (preferably with a small scope), which are designed, developed, deployed, and administrated independently.

The API management layer pretty much stays the same, despite the changes to the ESB and services layers that it communicates with. The API gateway and management layer exposes the business functionalities as managed APIs; we have the option of segregating the gateway into independent per-API based runtimes.

Since now you have a basic understanding of the microservices architecture, we can dive deep into the main characteristics of microservices.

Business Capability Oriented

One of the key concepts of the microservices architecture is that your service has to be designed based on the business capabilities, so that a given service will serve a specific business purpose and has a well-defined set of responsibilities. A given service should focus on doing only one thing and doing it well.

It's important to understand that a coarse-grained service (such as a web service developed in the SOA context) or a fine-grained service (which doesn't map to a business capability) is not a suitable fit into the microservices architecture. Rather, the service should be sized purely based on the scope and the business functionality. Also, keep in mind that making services too small (i.e., oriented on fine grained features that map to business capabilities) is considered an anti-pattern.

In the example scenario, we had coarse-grained services such as `Product`, `Order`, etc. in SOA/Web Services implementation (see Figure 1-3) and when we move into microservices, we identified a set of more fine-grained, yet business capability-oriented services, such as `Product Details`, `Order Processing`, `Product Search`, `Shopping Cart`, etc.

The size of the service is never determined based on the number of lines of code or the number of people working on that service. The concepts such as Single Responsibility Principle (SRP), Conway's law, Twelve Factor App, Domain Driven Design (DDD), and so on, are useful in identifying and designing the scope and functionalities of a microservice. We will discuss such key concepts and fundamentals of designing microservices around business capabilities in Chapter 2, "Designing Microservices".

Autonomous: Develop, Deploy, and Scale Independently

Having autonomous services could well be the most important driving force behind the realization of the microservices architecture. Microservices are developed, deployed, and scaled as independent entities. Unlike web services or a monolithic application architecture, services don't share the same execution runtime. Rather they are deployed as isolated runtimes by leveraging technologies such as containers. The successful and increasing adaptation of containers and container management technologies such as Docker, Kubernetes, and Mesos are vital for the realization of service autonomy and contribute to the success of the microservices architecture as a whole. We dig deep into the deployment aspect of microservices in Chapter 8, "Deploying and Running Microservices".

The autonomous services ensure the resiliency of the entire system as we have isolated the failures along with service isolation. These services are loosely coupled through messaging via inter-service communication over the network. The inter-service communication can be built on top of various interaction styles and message formats (we cover these things in detail in Chapter 3, "Inter-Service Communication"). They expose their APIs via technology-agnostic service contracts and consumers can use those contracts to collaborate with that service. Such services may also be exposed as managed APIs via an API gateway.

The independent deployment of services provides the innate ability to scale services independently. As the consumption of business functionalities varies, we can scale the microservices that get more traffic without scaling other services.

We can observe these microservices' characteristics in our e-commerce application use case, which is illustrated in Figure 1-3. The coarse-grained services, such as `Product`, `Order`, etc., share the same application server runtime as in the SOA/Web Services approach. So, a failure (such as out of memory or CPU spinning) in one of those services could blow off the entire application server runtime. Also, in many cases the functionalities such as product search may be very frequently used compared to other functionalities. With the monolithic approach, you can't scale the product searching functionalities because it shares the same application server runtime with other services (you have to share the entire application server runtime instead). As illustrated in Figure 1-4, the segregation of these coarse-grained services into microservices makes them independently deployable, isolates failures into each service level, and allows you to independently scale a given microservice depending how it is consumed.

No Central ESB: Smart Endpoints and Dumb Pipes

The microservices architecture fosters the elimination of the Enterprise Service Bus (ESB). The ESB used to be the where most of the smarts lived in the SOA/Web Services based architecture. The microservices architecture introduces a new style of service integration called *smart endpoints and dumb pipes* instead of using an ESB. As discussed earlier in this chapter, most of the business functionalities are implemented at the ESB level through the integration or plumbing of the underlying services and systems. With *smart endpoints and dumb pipes*, all the business logic (which includes the inter-service communication logic) resides at each microservice level (they are the *smart-endpoints*) and all such services are connected to a primitive messaging system (a *dumb pipe*), which doesn't have any business logic.

Most naive microservices adopters think that by just transforming the system into a microservices architecture, they can simply get rid of all the complexities of the centralized ESB architecture. However, the reality is that with microservices architecture, the centralized capabilities of the ESB are dispersed into all the microservices. The capabilities that the ESB has offered now have to be implemented at the microservices level.

So, the key point here is that the complexity of the ESB won't go away. Rather it gets distributed among all the microservices that you develop. The microservices compositions (using synchronous or asynchronous styles), inter-services communication via different communication protocols, application of resiliency patterns such as circuit breakers, integrating with other applications, SaaS (e.g., Salesforce), APIs, data and proprietary systems, and observability of integrated services need to be implemented as part of the microservices that you develop. In fact, the complexity of creating compositions and inter-services communication can be more challenging due to the number of services that you have to deal with in a microservices architecture (services are more prone to errors due to the inter-service communications over the network).

Most of the early microservices adopters such as Netflix just implemented most of these capabilities from scratch. However, if we are to fully replace ESB with a microservices architecture, we have to select specific technologies to build those ESB's capabilities at our microservices level rather re-implementing them from scratch.

We will take a detailed look at all these requirements and discuss some of the available technologies to realize them in Chapter 3, "Inter-service Communication" and Chapter 7, "Integrating Microservices".

Failure Tolerance

As discussed in the previous section, microservices are more prone to failures due to the proliferation of the services and their inter-service network communications. A given microservice application is a collection of fine-grained services and therefore a failure of one or more of those services should not bring down the entire application. Therefore, we should gracefully handle a given failure of a microservice so that it has minimum impact on the business functionalities of the application. Designing microservices in failure-tolerable fashion requires the adaptation of the required technologies from the design, development, and deployment phases.

For example, in the retail example, let's say the `Product Details` microservices is critical to the functionality of the e-commerce application. Hence we need to apply all the resiliency-related capabilities, such as circuit breakers, disaster recovery, load-balancing, fail-over, and dynamic scaling based on traffic patterns, which we discuss in detail in Chapter 7, "Integrating Microservices".

It is really important to mimic all such possible failures as part of the service development and testing, using tools such as Netflix's Chaos Monkey. A given service implementation should also be responsible for all the resiliency related activities; such behaviors are automatically verified as part of the CICD (continuous integration, continuous delivery) process.

The other aspect of failure tolerance is the ability to observe the behavior of the microservices that you run in production. Detecting or predicting failures in a service and restoring such services is quite important. For example, suppose that you have monitoring, tracing, logging, etc. enabled for all your microservices in the online retail application example. Then you observe a significant latency and low TPS (Transactions Per Second) in the `Product Search` service. This is an indication of a possible future outage of that service. If the microservices are observable, you should be able to analyze the reasons for the current symptoms. Therefore, even if you have employed chaos testing during the development phase, it's important to have a solid observability infrastructure in place for all your microservices to achieve failure tolerance. We will discuss the observability techniques in detail in Chapter 13, "Observability".

We cover failure tolerance techniques and best practices in Chapter 7, "Integrating Microservices" and Chapter 8, "Deploying and Running Microservices" in detail.

Decentralized Data Management

In a monolithic architecture, the application stores data in single and centralized logical databases to implement various functionalities/capabilities of the application. In a microservices architecture, the functionalities are dispersed across multiple microservices. If we use the same centralized database, the microservices will be no longer independent from each other (for instance, if the database schema is changed by one microservice, that will break several other services). Therefore, each microservice must have its own database and database schema.

Each microservice can have a private database to persist the data that requires implementing the business functionality offered by it. A given microservice can only access the dedicated private database and not the databases of other microservices.

In some business scenarios, you might have to update several databases for a single transaction. In such scenarios, the databases of other microservices should be updated through the corresponding service API only (they are not allowed to access the database directly).

The decentralized data management gives you the fully decoupled microservices and the liberty of choosing disparate data-management techniques (SQL or NoSQL etc., different database management systems for each service). We look at the data management techniques of the microservices architecture in detail in Chapter 5, "Data Management".

Service Governance

SOA governance was one of a key driving forces behind the operational success of SOA; it provides the cooperation and coordination between the different entities in an organization (development teams, service consumers, etc.). Although it defines a comprehensive set of theoretical concepts as part of SOA governance, only a handful of concepts are being actively used in practice. When we shift into a microservices architecture, most of the useful governance concepts are discarded and the governance in microservices is interpreted as a decentralized process, which allows each team/entity to govern its own domain, as it prefers. Decentralized governance is applicable to the service development, deployment, and execution process, but there's a lot more to it than that. Hence we deliberately didn't use the term *decentralized governance*.

We can identify two key aspects of governance: design-time governance of services (selecting technologies, protocols, etc.) and runtime governance (service definitions, service registry and discovery, service versioning, service runtime dependencies, service ownerships and consumers, enforcing QoS, and service observerability).

Design-time governance in microservices is mostly a decentralized process where each service owner is given the freedom to design, develop, and run their services. Then they can use the right tool for the job, rather than standardize on a single technology platform. However, we should define some common standards that are applicable across the organization (for example, irrespective of the development language, all code should go through a review process and automatically be merged into the mainline).

The runtime governance aspect of microservices is implemented at various levels and often we don't call it *runtime governance* in a microservices context (service registry and discovery is one such popular concept that is extremely useful in a microservices architecture). So, rather than learn about these concepts as a set of discrete concepts, it's easier to understand them if we look at the runtime-governance perspective.

Runtime governance is absolutely critical in the microservices architecture (it is even more important than SOA runtime governance), simply because of the number of microservices that we have to deal with. The implementation of runtime governance is often done as a centralized component. For example, suppose that we need to discover service endpoints and metadata in our online retail application scenario. Then all the services have to call a centralized registry service (which can have its own scaling capabilities, yet a logically centralized component). Similarly, if we want to apply QoS (quality of service) enforcements such as security, by throttling centrally, we need a central place such as an API Manager/gateway to do that. In fact, some of the runtime governance aspects are implemented at the API gateway layer too.

We'll look at microservices governance aspects in detail in Chapter 6, "Microservices Governance" and API Management in Chapter 10, "APIs, Events and Streams".

Observability

Service observability can be considered a combination of monitoring, distributed logging, distributed tracing, and visualization of a service's runtime behavior and dependencies. Hence observability can be also considered part of runtime governance. With the proliferation of fine-grained services, the ability to observe the runtime behavior of a service is absolutely critical. Observability components are often a