Enterprise Java Microservices

Ken Finnigan

SAMPLE CHAPTER

MANNING

Enterprise Java Microservices

by Ken Finnigan

Chapter 1

brief contents

Part 1

Microservices basics

What are Microservices? A *microservice* consists of a single deployment executing within a single process. How do microservices differ from traditional Enterprise Java applications? In what situations is it appropriate to use microservices? These are just some of the questions that we'll address in these first five chapters.

Part 1 also explores the runtime options available for Enterprise Java microservices, before finishing with how to test microservices and deploy them to the cloud.

1 Enterprise Java microservices

This chapter covers

- Enterprise Java history
- Microservices and distributed architecture
- Patterns for migration to microservices
- Enterprise Java microservices

Before you dive in, let's step back and discuss what I hope you achieve during the course of this book. We all know that there's no such thing as a *free lunch*, so I won't pretend that microservices are easy. This chapter introduces microservices—their concepts, benefits, and drawbacks—to provide a basis on which you can build your technical knowledge. Chapters 2 and 3 provide an example of a RESTful endpoint microservice and cover some of your runtime and deployment options for Enterprise Java microservices.

So what is an *Enterprise Java microservice*? In a nutshell, it's the result of applying Enterprise Java to the development of microservices. The latter part of this chapter and the remainder of the book explore in detail what that means.

After you've learned the basics of microservices, you'll delve into tools and techniques for use in Enterprise Java to mitigate the drawbacks and complexity of microservices. Being more familiar with microservices, you'll then look at an existing Enterprise Java application and how it could be migrated to take advantage of microservices. The last few chapters touch on more advanced microservice topics related to security and event streaming.

1.1 Enterprise Java—a short history

If you're reading this book, you're most likely already an experienced Enterprise Java developer. If you aren't, I appreciate and applaud your desire to broaden your horizons into Enterprise Java!

1.1.1 What is Enterprise Java?

For those who are new to, or need a refresher in, Enterprise Java, what is it? *Enterprise Java* is a set of APIs, and their implementations, that can provide the entire stack of an application from the UI down to the database, communicate with external applications via web services, and integrate with internal legacy systems, to name a few, with the goal of supporting the business requirements of an enterprise. Though it's possible to achieve such a result with Java on its own, rewriting all the low-level architecture required for an application would be tedious and error prone, and would significantly impact the ability of a business to deliver value in a timely manner.

It wasn't long after Java was first released more than 20 years ago that various frameworks began to crop up to solve the low-level architecture concerns of developers. These frameworks allowed developers to focus on delivering business value with application-specific code.

> **Enterprise Java**
>
> Many frameworks have come and gone, but two have remained the most popular through the years: Java Platform, Enterprise Edition (Java EE), and Spring. These two frameworks account for most development by an enterprise with Enterprise Java.
>
> Java EE incorporates many specifications, each with one or more implementations. Spring is a collection of libraries, some of which wrap Java EE specifications.

1.1.2 Typical Enterprise Java architecture

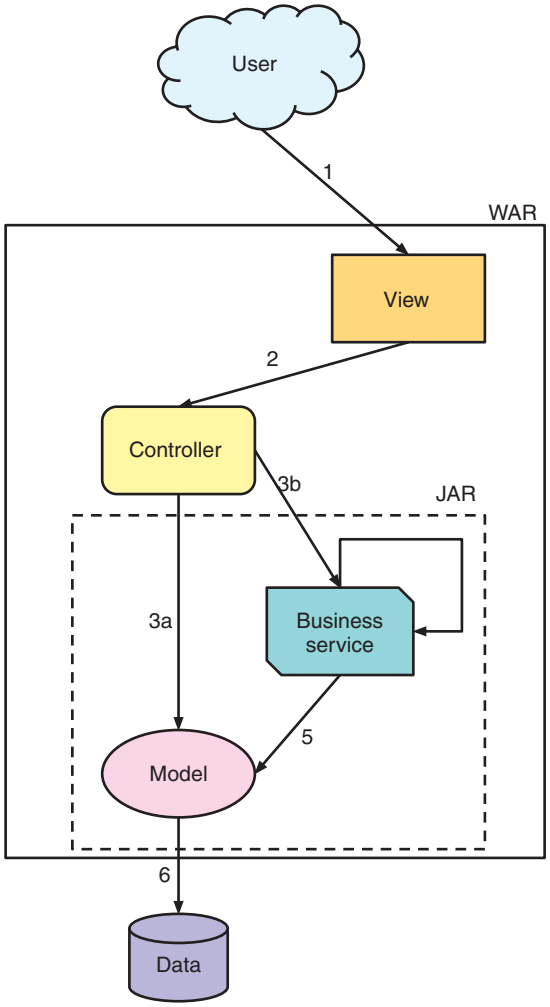
In the early days of Enterprise Java, our applications were all *greenfield* development, because no preexisting code was being extended.

> **DEFINITION** *Greenfield* refers to the development of an entirely new application without any preexisting code that needs to be taken into consideration, excluding any common libraries that might be required.

Greenfield development presents the greatest opportunity to develop a *clean* layered architecture for an application. Typically, architects would devise an architecture similar to that shown in figure 1.1.

Here you'll likely recognize familiar pieces of architectures you've worked on in the past: a *view* layer, a *controller*, possibly using a reusable *business service*, and finally, the *model* that interacts with the database. You can also see the application packaged as a WAR, but many combinations of packaging for each layer could be applied, including JAR and EAR. Typically, the *view* and *controller* are packaged in a WAR. The *business service* and *model* are packaged in JARs, either inside a WAR or EAR.

As the years passed, we continued developing greenfield applications with Enterprise Java using such a pattern, but there reached a point where most enterprises



1 A *user* makes a request from a browser specifying which *view* of an application they wish to see.

2 The *view* calls out to a *controller* to retrieve whatever information might by required to construct itself.

3 The *controller* can retrieve the information in one of two ways:
3a Directly interact with the *model* of the application to retrieve an object model populated with data.
3b Call one or more *business services*, possibly to aggregate data from different sources.

4 A *business service* can also make many calls to other *business services*. It all depends on how the business features have been architected.

5 The *business service* calls a *model* to retrieve the data it needs. This step is equivalent to 3a.

6 *Model* classes provide the mapping onto physical data storage, and are often passed back up through the layers of the application.

Figure 1.1 Typical Enterprise Java application architecture

were, for the most part, enhancing existing applications. From that day, many Enterprise Java applications became a legacy burden on enterprises by virtue of the maintenance work required—not because of a flaw or deficiency in Java, though there have been several, but because developers aren't the best at architecting changes to existing applications and systems. This is complicated further for enterprises that have hundreds of architects and developers pass through their doors, each bringing their own preferences and patterns to extending existing applications.

> **NOTE** I'm not sitting in an ivory tower disparaging developers. Many times I've made decisions about how a feature should be implemented without fully grasping existing functionality—not through any intent or malice, but because those who wrote the code are no longer employed at the enterprise and therefore can't be asked about the code, and because documentation may be lacking or nonintuitive. Such a situation means developers are left to make a judgment call as to whether or not they've understood the existing system sufficiently to make modifications. Throw in some deadline pressure from management, and such a situation becomes even more fraught with problems.

Over time, many Enterprise Java applications diverged from the clean architecture shown in figure 1.1 and became a mess of spaghetti more closely resembling figure 1.2. In figure 1.2 you can see how clear boundaries between functionality within a layer have become blurred, resulting in components in each layer no longer having a well-defined purpose.

This situation is where many enterprises find themselves today. Only a few applications of an enterprise may fit this mold, but this mess of spaghetti is a problem that must be solved in order for an application to foster future development without significant costs being incurred each time.

1.1.3 What is a monolith?

What defines an Enterprise Java application as a monolith? A *monolith* is an application that has all its components contained within a single deployable, and that typically has a release cadence of 3–18 months. Some applications may even have a release cadence of two years, which doesn't make for an agile enterprise. Monoliths typically evolve over time from attempts to make quick iterative enhancements to an application, without any concern for appropriate boundaries between different parts, or components, within it. Indicators of an application being a monolith can include the following:

- Multiple WARs that are part of a single deployment, due to their intertwined behavior
- EARs that contain potentially dozens of other WARs and JARs to provide all the necessary functionality

Is figure 1.2 a monolith? It most certainly is, and an extremely bad one, because of the blurring of functional separation between components.

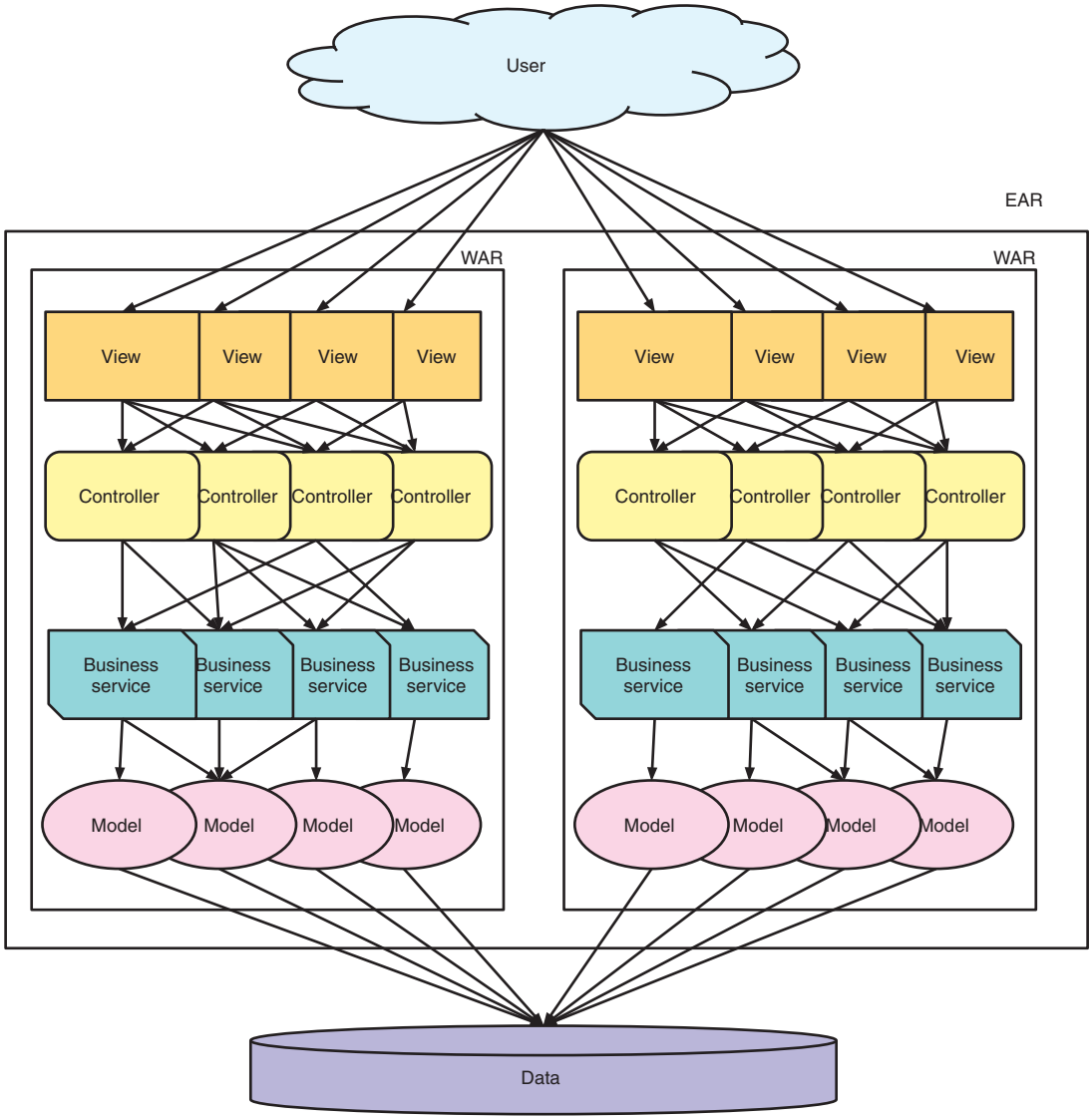


Figure 1.2 Enterprise Java spaghetti

Why do the preceding factors make an application a monolith? A single deployable for an application is perfectly fine when you have a small footprint, but when you have potentially thousands of classes and dozens of third-party libraries, an application becomes infinitely more complex. Testing even a minor change to the application would require large amounts of regression testing to ensure that no other part of the application was impacted. Even if the regression testing were automated, it'd still be a mammoth task.

Whether an application is a monolith is also determined in part by its architecture. Classifying as a monolith isn't based on the size of the application on disk, or the size of the runtime being used to execute the monolith. It's all about how that application has been architected with respect to the components within it.

Release cadence is a forcing function for enterprises. If an application is released only every 3–18 months, the business (unknowingly or not) will focus on larger feature changes that take significant time to develop. No incentive exists to request a minor tweak that could be made and released in a few hours, or days, when even the most simple change won't reach production for months.

Release cadence dictated by the time it takes to develop and test changes has a direct impact on the ability of an enterprise to be agile and respond to a changing environment. For instance, if a competitor were to begin selling the same widget as your enterprise for 15% less than you do, can you react? Taking several months to make a simple change to reduce the selling price of a product could have disastrous consequences for the bottom line. If that widget was the biggest seller, and the enterprise was unable to compete on price for three months, it may even be on the verge of going out of business by the time a price change was released.

Along with release cadence, it's critical to note that discussions around *micro* versus *monolith* don't have any relation to constraints on size. You could have a microservice that's 100 MB in size, or a monolith that's only 20 MB. The definition is more about the coupling of dependencies between components, leading to the benefit of updating a single component without needing to cascade updates across many components. This decoupling is what allows for a faster release cadence.

Though it appears that monolithic Enterprise Java applications are all gloom and doom, is that really the case? In many situations, it makes sense for an enterprise to continue with, or develop, a monolith. How do you know if you should stick with a monolith?

- *Your enterprise may have only a few applications that it actively develops and maintains.* It may not make sense to significantly increase the development, testing, and release burden when you have so few applications.
- *If the current development team has a dozen people, splitting them into one- or two-person teams for microservices may not provide any benefit.* In some cases, that split will be detrimental. Basecamp (https://basecamp.com/) is a perfect example of a monolith that's fine the way it is, developed by a team of 12.
- *Does your enterprise need multiple releases a week, or even a day?* If not, and the existing monolith has a clear separation of components, reducing the release cadence may be all that's required to derive increased business agility and value.

Whether staying with a monolith is the right thing for an enterprise varies, depending on the current circumstances and the long-term goals.

1.1.4 What are the problems associated with monoliths?

In general, an architectural design akin to the one in figure 1.1 is a good idea, but drawbacks exist as well:

- *Inability to scale individual components*—This may not seem to be a major problem, but certain factors can alter the impact of poor scaling. If a single instance of the application requires a large amount of memory or space, scaling that out to a not-insignificant number of nodes requires a large investment in hardware.
- *Performance of individual components*—With a single deployment containing many components, it's easy for one component to perform worse than the rest. You then have a single component slowing down the entire system, which isn't a good situation, and the operations team won't be pleased.
- *Deployability of individual components*—When the entire application is a single deployment, any changes require a deployment of the entire application, even if you have a single-line change in one component. That's not good for business agility and often results in release cadences of many months to include many changes in one updated deployment.
- *Greater code complexity*—When an application has many components, it's easy for the functional boundaries between them to become blurred. Blurring the separation of components further increases the complexity of code, both in terms of code execution and for a developer understanding the intent of the code.
- *Difficulty in accurately testing an application*—When the complexity of an application grows, the amount of testing and time required to ensure that any change didn't cause a regression grows. What seems like the smallest and most insignificant change can easily lead to unforeseen errors and problems in completely unrelated components.

All these issues cause great cost to enterprises, as well as slowing the speed with which they can take advantage of new opportunities. But these potential drawbacks are still small in comparison to starting from a clean slate.

If an enterprise has an application that has evolved with new features over a decade or more, attempting to replace it with a greenfield project would cost hundreds of man years in effort. This is a huge factor in why enterprises continue maintaining existing monoliths.

When it's too costly to replace a monolith with a more modern alternative, that application becomes entrenched in an enterprise. It becomes a critical application, and any downtime causes business impacts. This situation becomes ever more compounded with continual enhancements and fixes.

On the flipside, some monoliths have been running well for years and can be easily managed by a handful of developers without much effort. Maybe they're in a maintenance mode and not under heavy feature development. These monoliths are perfectly OK as they are. If it ain't broke, don't fix it.

What do you do with monoliths that are too cumbersome to replace with a greenfield project, even though the enterprise knows it's costing them a great deal in business agility and expense? How do you update them to use newer frameworks and technologies so they don't become legacy? We'll answer these questions next.

1.2 Microservices and distributed architecture

Before delving into the definitions for *microservices* and *distributed architecture*, let's revisit how figure 1.2 might look when using them; see figure 1.3. This depiction has certainly cleared up the separation between components by splitting them into separate microservices with clear boundaries between them.

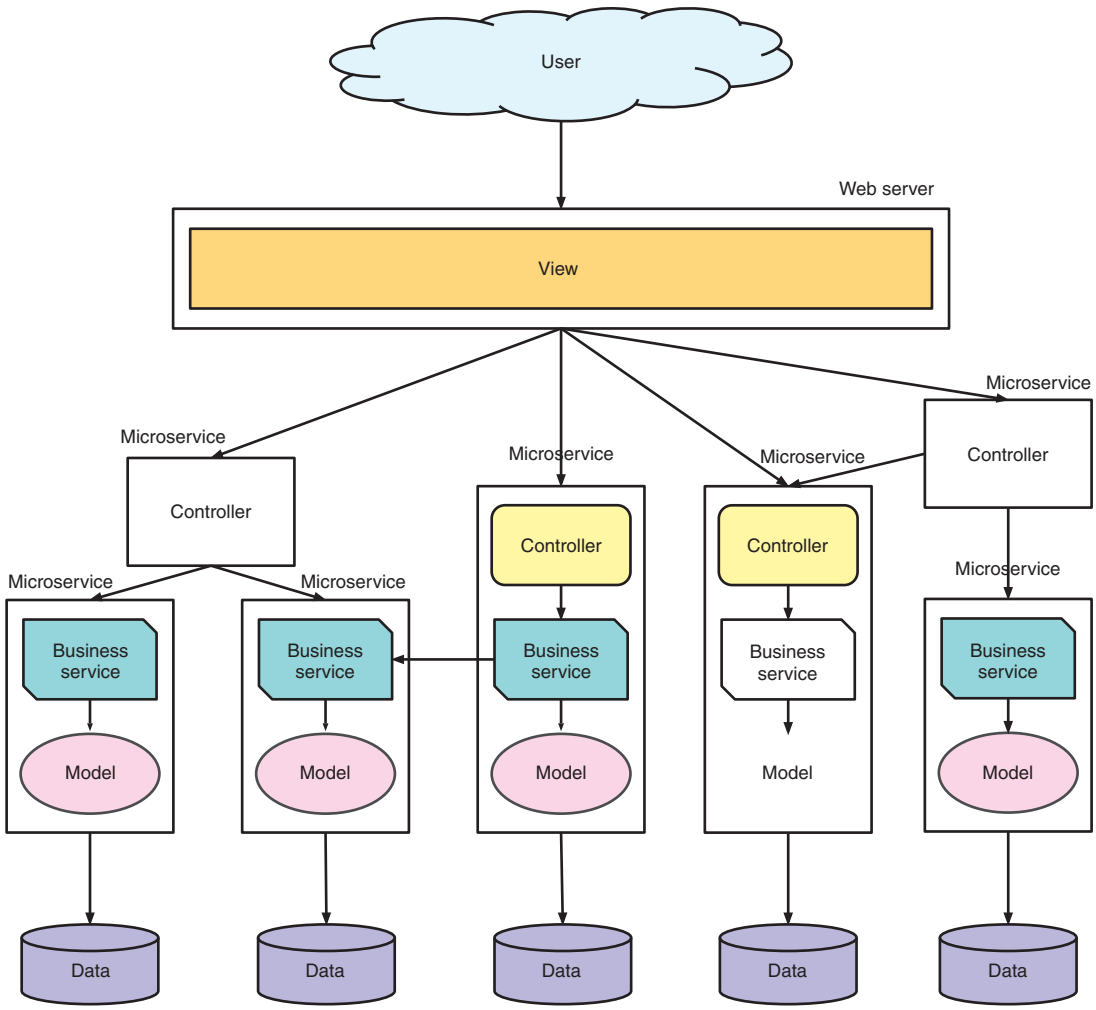


Figure 1.3 Enterprise Java microservices

So what do I mean by a microservice? A *microservice* consists of a single deployment executing within a single process, isolated from other deployments and processes, that supports the fulfillment of a specific piece of business functionality. Each microservice focuses on the required tasks within a *Bounded Context*, which is a logical way to separate the various domain models of an enterprise. We'll cover this in greater detail later in this chapter.

From the definition, you can see that a microservice, in and of itself, isn't useful. It becomes useful when you have many loosely coupled microservices working together to fulfill the needs of an application. A microservices architecture containing many microservices communicating with each other can also be referred to as a *distributed architecture*.

To make a microservice useful, it needs to be easily used from other microservices and components of the entire system. It's impossible to achieve that when a microservice attempts to accomplish too much. You want a microservice to focus on a single task.

1.2.1 Do one thing well

In 1978, Douglas McIlroy, best known for developing UNIX pipelines and various UNIX tools, documented the UNIX philosophy, one part of which is, *Make each program do one thing well*. This same philosophy has been adopted by microservice developers. Microservices aren't the kitchen sink of application development; you can't throw everything in them and expect them to function at an optimal level. In that case, you'd have a *monolithic microservice*, also referred to as a *distributed monolith*!

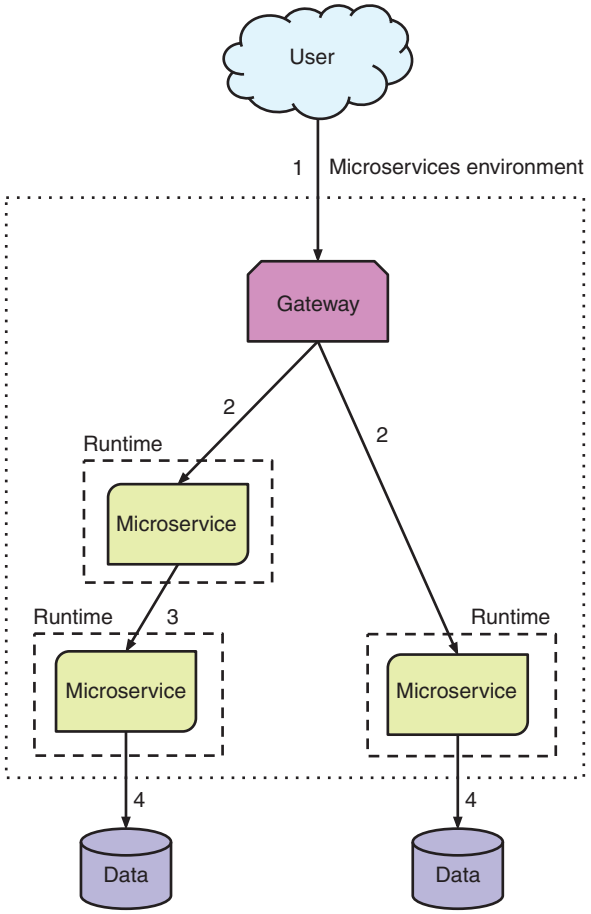
A well-designed microservice should have a single task to perform that's sufficiently fine-grained, delivering a business capability or adding business value. Going beyond a single task brings us back to the problems of Enterprise Java monoliths, which we don't want to repeat.

It's not always easy to figure out a sufficiently granular task for a microservice. Later in the chapter we'll discuss Domain-Driven Design as a method to assist in defining that granularity.

1.2.2 What is a distributed architecture?

A *distributed architecture* consists of multiple pieces that work with each other to make up the full functionality of an application distributed across processes, and often across network boundaries as well. What's distributed can be any part of an application, such as RESTful endpoints, message queues, and web services, but it's most definitely not limited to only these components.

Figure 1.4 shows what a distributed architecture for microservices might look like. In this depiction, the *microservice* instances are described as being in a *runtime*, but that doesn't dictate how the instance is packaged. It could be packaged as *uber jars* or Linux containers, but many other options are available. The runtime is purely for delineating the operating environment of a microservice, showing that the microservices are running independently.



1 A *user* makes a request from a browser to interact with a particular service. This could be from a mobile device or from a UI that was previously retrieved.

2 When the request enters the *microservices environment*, it enters the *gateway*, which routes the request to the appropriate *microservice*.

3 A *microservice* receives the request and performs some of its own processing on it before calling another *service*.

4 The last *microservice* in the chain interacts with the *data* storage layer for retrieving/writing records.

Figure 1.4 Typical microservices architecture

NOTE An *uber jar*, also known as a *fat jar*, indicates that the JAR file contains more than a single application or library, and that it can be run from the command line with `java -jar`.

1.2.3 Why should you care about being distributed?

Now that you've seen a distributed architecture, let's look at some of the benefits:

- *Services are location-independent.* Services can locate and communicate with other services no matter where they're physically located. Such location independence allows services to be located on the same virtual hardware, same physical hardware, same data center, different data centers, or even a public cloud, and all act is if they're in the same JVM. The main downside to location independence is the extra time required to make the network calls between them, and by the nature of adding new network calls, you've reduced the likelihood of successful completion.

- *Services are language-independent.* Though this book focuses on Enterprise Java, we're not so naive as to believe that there won't be times that services need, or are desired, to be developed in different languages. When services aren't required to run in the same environment, you can use different languages for different services.
- *Service deployments are small and single-purpose.* When a deployment is smaller, less effort is required for testing, and this makes it possible to shrink the release cadence of that deployment down to a week or less. Having small, single-purpose deployments enables an enterprise to more easily react to business needs in a near-real-time fashion.
- *New services are defined by the recomposition of existing service functionality.* Having discrete distributed services throughout your architecture greatly enhances your ability to recombine those services in new ways to create additional value. This recombination can be as straightforward as deploying a single new service, combined with a handful of services already deployed. This enables you to create something new for the business in a shorter time frame.

Sounds awesome—how can you develop distributed applications right now? You need to pull back on the reins a bit here. Yes, being distributed does improve a lot of the issues that we've had with Enterprise Java over the years, but it also introduces its own challenges. Developing distributed applications is in no way a silver bullet, and you can easily shoot yourself in the foot.

You've seen some benefits of being distributed, but there's never a free lunch with most things—and definitely not with distributed architecture. If you have a bunch of services that interoperate through communication and no coupling, what problems can that introduce?

- Location independence for services is great, but how do they find each other? You need a means of defining services logically, regardless of what their physical location or IP address might be. With a means of discovery, you can locate a service by its logical name and ignore wherever it might be physically located. Service discovery serves this purpose. Part 2 of this book covers how to use service discovery.
- How do you handle failure without impacting customers? You need a means of gracefully degrading functionality when services fail, instead of crashing the application. You need service resilience and fault tolerance to provide alternatives when services fail. Part 2 covers how to provide fault tolerance and resilience for your services.
- Having hundreds or thousands of services, versus a handful of applications, places additional burdens on operations. Most operations teams aren't experienced in dealing with such a large number of services. How do you mitigate some of this complexity? Monitoring needs to play a major part here—in particular, automated monitoring. You need to automate the monitoring of hundreds of services to reduce the burden on operations, while also providing information that's as near to real-time as possible about the entire system.

1.2.4 What can be done to assist in developing microservices?

Microservice development is hard, so what can you do to make it easier? There's no panacea for making it easy, but this section covers a couple of options for making microservice development more manageable.

1.2.5 Product over project

Netflix has been a major proponent of the product-over-project idea for its microservices since rewriting its entire architecture under the leadership of Adrian Cockroft.

All these years, we've been developing projects and not products. Why? Because we develop an application that meets a set of requirements and then hand it over to operations. The application might require two weeks or two years to develop, but it's still a project if, at the end, the application is handed over and the team disbanded. Some team members may be retained for a period to handle maintenance requests and enhancements, but the effort is still considered a project followed by lots of mini projects.

So how do you develop a *product*? Developing a product means that a single team owns it for the entirety of its lifespan, whether that be 2 months or 20 years. The team will develop it, release it, manage the operational aspects of the application, resolve production issues—pretty much everything.

Why does the differentiation between a project and a product matter? Owning a product engenders a greater sense of responsibility about the way an application is developed. How? Do you want to be paged in the middle of the night because an application is failing? I know I don't!

How does a shift of focus from project to product help with developing microservices? When you're seeking a release cadence of a week or less, as is typical for true microservices, it's hard to reach that release frequency with developers who aren't familiar with the codebase, as would be the case with a *project* approach.

1.2.6 Continuous integration and delivery

Without continuous integration and delivery, developing microservices becomes a great deal more difficult.

Continuous integration refers to the processes that ensure any change, or commit, to a source repository results in a new build of the application, including all associated tests of that application. This provides quick feedback on whether or not changes broke the application, provided the tests are sufficient enough to discover it.

Continuous delivery is a reasonably new phenomenon that has come from the DevOps movement, whereby application changes are continuously delivered between environments, including production, to ensure expeditious delivery of application changes. A manual step may occur to approve a build going into production, but not always. Having a manual step is likely for critical user applications and less so for others. Continuous delivery is usually offered by means of a build pipeline, which can consist of automatic or manual steps, such as a manual step to approve a release for production.

Continuous integration and delivery, referred to as *CI/CD*, are key tools in facilitating a short release cadence. Why? They enable developers to find possible bugs earlier in the process in an automated manner. But more important, CI/CD significantly reduces the amount of time between determining that a piece of code is ready for production and having it live for users. If a release process takes a day or two to complete, that isn't conducive to releasing multiple times a day or even once a day.

Another important benefit of CI/CD is the ability to be more incremental in delivering functionality. The goal isn't just to be able to physically release code faster; being able to deploy smaller pieces of functionality is crucial for minimizing risk as well. If a small change reaches production that causes a failure, backing out that change is a relatively easy task.

1.3 Patterns for migration to microservices

You've looked at Enterprise Java with its existing monoliths and you've learned about microservices in a distributed architecture. But how do you get from one to the other? This section delves into patterns that can be applied to the problem of splitting an existing monolith into multiple microservices.

1.3.1 Domain-Driven Design

Domain-Driven Design (DDD) is a set of patterns and methodologies for modeling our understanding of the domains in our software. A key part of this is the Bounded Context pattern (https://martinfowler.com/bliki/BoundedContext.html), which enables you to segregate parts of the system to be modeled at a single time.

This topic is far too broad to be covered in a few small paragraphs in this book, especially because many books are already dedicated to DDD. But we'll cover it briefly here as another piece in the puzzle of developing with microservices. DDD can be used both in greenfield microservice development and in migrating to microservices.

A sufficiently large application or system can be divided into multiple Bounded Contexts, enabling design and development to focus on the core domain of a given Bounded Context at any one point. This pattern acknowledges that it's difficult to come up with a domain model for an entire enterprise at any one time, because too many complexities exist. Dividing such a model into manageable Bounded Contexts provides a way to focus on a portion of that model without concerning yourself with the remainder of the, likely unknown, domain model. Figure 1.5 is an example to help you understand the concepts behind DDD.

Say you have a store that wants to develop microservices, and its domain model consists of an order, items within an order, a product, and a supplier of that product. The current domain model combines the different ways a Product can be defined. From the perspective of an Order, it doesn't care who supplies the product, how many are currently in stock, what the manufacturer price is, or any other information that's relevant to only the administration of the business. Conversely, the administration side isn't necessarily concerned with how many orders a product may be associated with.

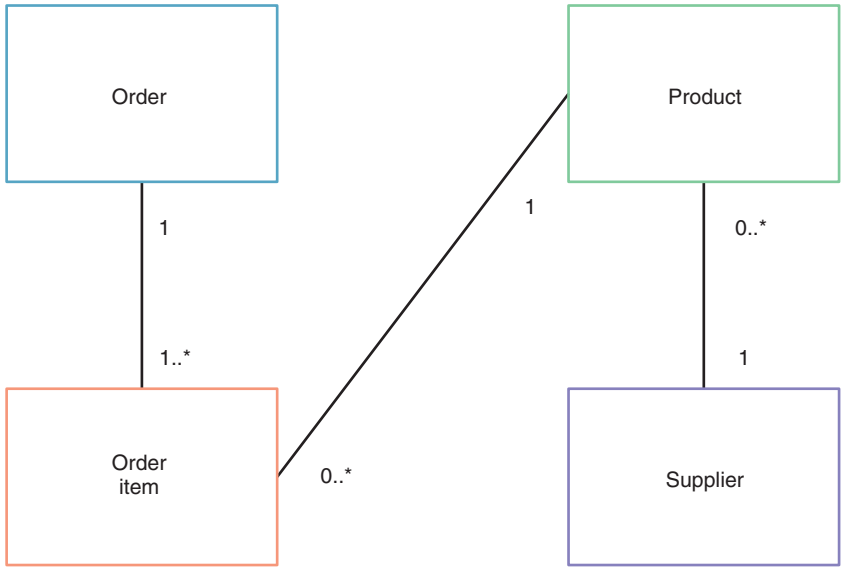


Figure 1.5 Store domain model

Figure 1.6 shows you now have Product in each Bounded Context; each represents a different view of a product. The Order Bounded Context has only information such as a product code and description. All the product information required by the business is within the Product Bounded Context.

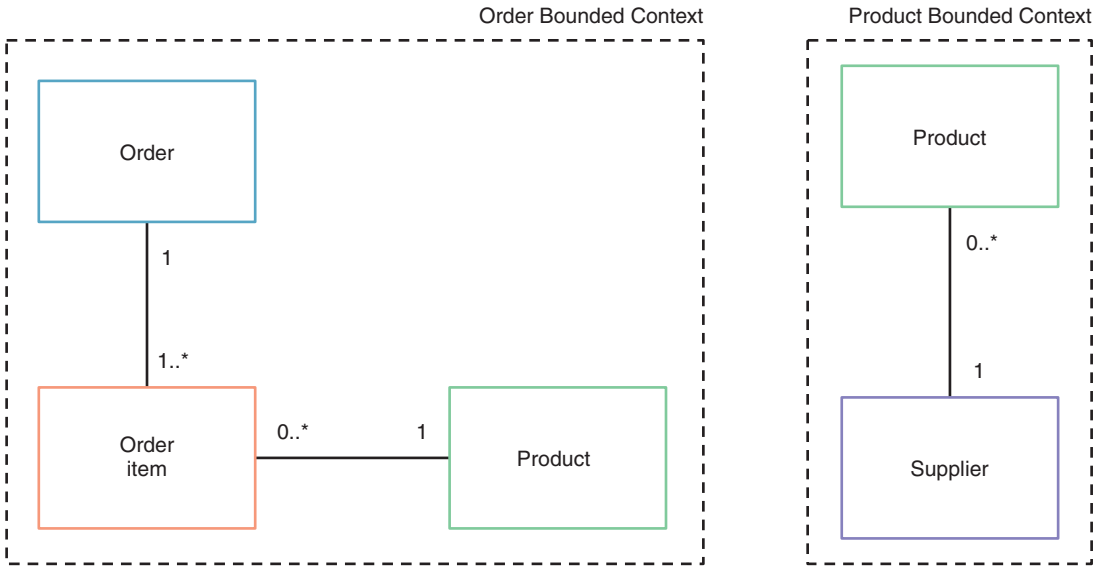


Figure 1.6 Separate Bounded Contexts

In some cases, a clean split will exist in the domain model of a Bounded Context, but in others there will be commonality between the separate models, as in the preceding example. In this situation, it's important to consider that although a part of the domain model is shared between Bounded Contexts, one domain can be classed as the *owner*.

Having defined the owner of a piece of the domain, it becomes necessary to make that domain available to external Bounded Contexts—but in a way that doesn't implicitly tie the two Bounded Contexts together. This does make it trickier to handle the boundary, but patterns such as Event Sourcing can help with this problem.

> **NOTE** *Event sourcing* is the practice of firing events for every state change in an application, which is usually recorded as a log in a certain format. Such a log can then be used to rebuild entire database structures, or as in this case, as a way to populate a piece of a domain model that's owned externally.

How do all these Bounded Contexts fit together? Each Bounded Context forms part of a greater whole, a context map. A *context map* is a global view of an application, identifying all the required Bounded Contexts and the way they should communicate and integrate with each other.

In this example, because you've split Product into two, you'd need such a data feed from the Product to Order Bounded Contexts to be able to populate the Product with appropriate data.

As you saw in our example, one side benefit of shared domain models in Bounded Contexts is that each can have its own view of the same data. An application is no longer forced into viewing a piece of data in the same way as its owner does. This can provide huge benefits when a domain needs only a small subset of the data in each record that the owner might hold. For additional information on Domain-Driven Design and Bounded Contexts, I recommend *Functional and Reactive Domain Modeling* by Debasish Ghosh (Manning, 2016).

1.3.2 Big Bang pattern

The *Big Bang pattern* for migrating to microservices in an enterprise is by far the most complicated and challenging. It entails breaking apart every single piece of an existing monolith into microservices, such that there's a single cutover from one to the other.

Because deployment is a single cutover—a Big Bang—to production, developing for such a change can take just as long as developing on a monolith. Certainly, by the end of the process, you've moved to microservices, but this pattern would be a bumpier road for most enterprises than other patterns for migrating to microservices—especially when considering the internal process and procedure changes required to move between the two deployment models. Such an abrupt change would be traumatic and potentially damaging to an enterprise.

The Big Bang pattern isn't recommended for most enterprises as a means of migrating, and most definitely not for those who aren't experienced with microservices already.

1.3.3 Strangler pattern

The *Strangler pattern* is based on the Strangler Application defined by Martin Fowler (www.martinfowler.com/bliki/StranglerApplication.html). Martin describes this pattern as a way to rewrite an existing system by gradually creating a new system at the edges of the existing one. The new system slowly grows over several years, until the old system is strangled into nonexistence.

You may find a similar end result as the Big Bang pattern—not necessarily a bad thing—but it's achieved over a much longer time span while still delivering business value in the interim. This approach significantly reduces the risk involved, compared to the Big Bang pattern. Through monitoring progress of the application over time, you can adjust the way you implement microservices as you learn with each new one implemented. This is another huge advantage over the Big Bang pattern: being able to adjust and react to issues that might arise in processes or procedures. With a Big Bang approach, an enterprise is tied into its processes until everything has cutover.

1.3.4 Hybrid pattern

Now that you've seen both the Big Bang and Strangler patterns, let's look at the *Hybrid pattern*. I feel this pattern will become the predominant pattern for enterprises migrating to and developing microservices.

This pattern begins life in a similar fashion to the Strangler. The difference is that you never fully strangle the original monolith. You retain some functionality within a monolith and integrate that with new microservices. Figure 1.7 shows the path of a request through an existing Enterprise Java monolith and a new microservices architecture:

1. A user makes a request from a browser specifying which view of an application they wish to see.
2. The view calls out to a controller to retrieve whatever information might be required to construct itself.
3. The controller calls a business service, possibly to aggregate data from different sources.
4. The business service then passes the request into the microservices environment, where it enters the gateway.
5. The gateway routes the request to the appropriate microservice based on routing rules that have been defined.
6. A microservice receives the request and performs some of its own processing on it before calling another microservice.
7. The last microservice in the chain interacts with the data storage layer to read/write records.

An architecture such as that in figure 1.7 provides a great deal of flexibility for growth and delivering business value in a timely fashion. Components that require high performance and/or high availability can be deployed to the microservices environment.

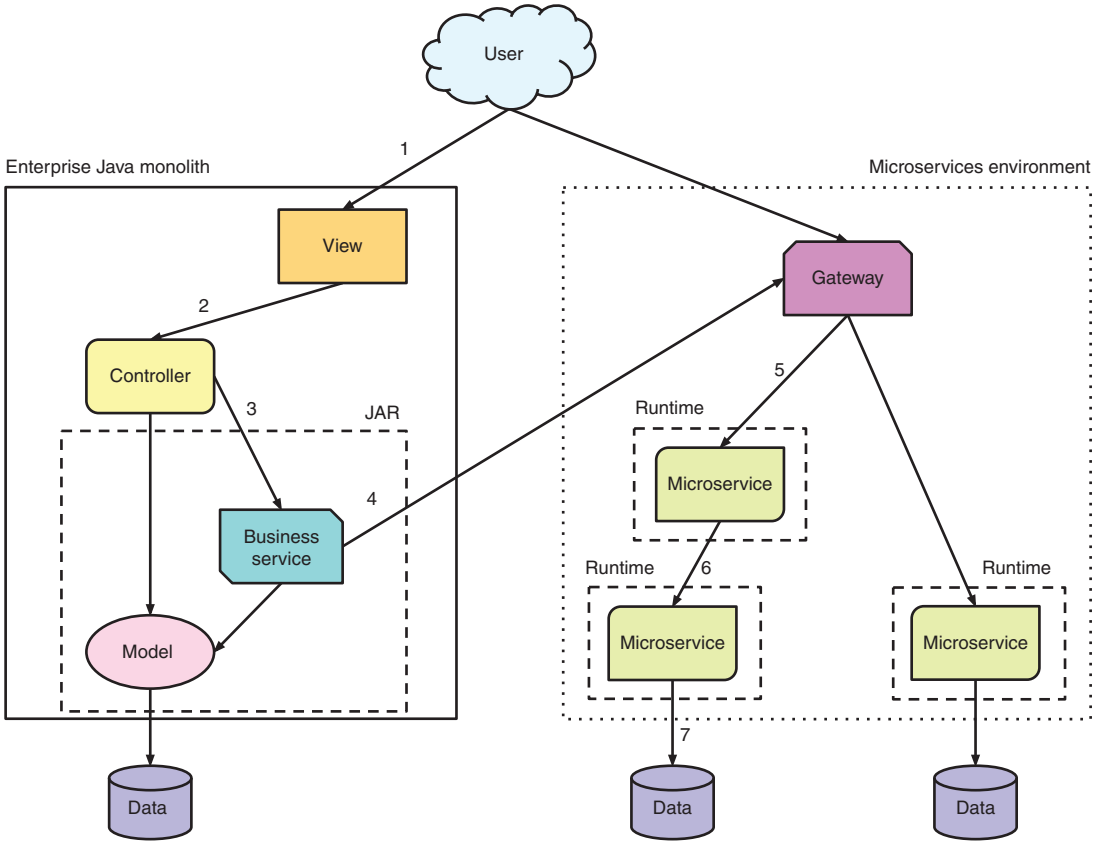


Figure 1.7 Enterprise Java and microservices hybrid architecture

Components that are too costly to be migrated to the new architecture can remain deployed on an Enterprise Java platform.

You'll focus on the Hybrid pattern later in the book, when you migrate an existing Enterprise Java application to use microservices.

1.4 What are Enterprise Java microservices?

As I mentioned at the beginning of the chapter, Enterprise Java microservices are purely microservices developed with Enterprise Java. So let's take a look at a simple example to see it in practice.

Let's create a simple RESTful Java EE microservice that uses CDI and JAX-RS. This microservice exposes a RESTful endpoint to greet the user by name; the message returned is being provided via a CDI service you inject (listing 1.1).

Listing 1.1 CDI service

```
@RequestScoped
public class HelloService {

    public String sayHello(String name) {
        return "Hello " + name;
    }
}
```

CDI annotation that says you want a new HelloService instance for each servlet Request made. In this instance, because you're not storing state, it could easily have been @ApplicationScoped instead.

Service method that takes a single parameter and returns it prefixed "Hello"

The preceding service defines a single `sayHello()` method that returns `Hello` combined with the value of the `name` parameter.

You can then `@Inject` that service into your controller.

Listing 1.2 JAX-RS endpoint

```
@ApplicationScoped
@Path("/hello")
public class HelloRestController {

    @Inject
    private HelloService helloService;

    @GET
    @Path("/{name}")
    @Produces("text/plain")
    public String sayHello(@PathParam("name") String name) {
        return helloService.sayHello(name);
    }
}
```

CDI annotation that states you need only a single instance for the entire application

Defines the RESTful URL path of this controller. In this case, it's set to "/hello".

You inject an instance of HelloService that you can use.

Specifies the URL path for the method. You also specify a parameter called name that can be passed on the URL of the request.

Defines the type of HTTP requests the method handles

The method produces a text response only.

Calls sayHello on the injected service passing the name parameter value

Assigns the path parameter called name as the method parameter

If you've developed JAX-RS resources before, you'll recognize everything in the preceding code. What does that mean? It means that you can develop microservices with Enterprise Java just as if you were developing an Enterprise Java application. The ability to develop a microservice with existing Enterprise Java knowledge is a significant advantage in using Enterprise Java for microservices.

This microservice example is simplified because you're dealing with only the producer side of the equation. If the service also consumed other microservices, it would be more complex. But you'll come to that in part 2 of this book.

Though the preceding example was implemented with Java EE APIs, it could just as easily have been implemented using Spring instead.

1.4.1 Why Enterprise Java is a good fit for microservices

You've seen how easy it is to develop a RESTful endpoint as an Enterprise Java microservice, but why should you? Wouldn't you be better off using a newfangled framework or technology specifically built for microservices? You have plenty to choose from right now: Go, Rust, and Node.js are just some examples.

In some situations, using a newer technology may make more sense. But if an enterprise has significant investment in Enterprise Java through existing applications, developers, and so forth, it makes a lot more sense to continue using that technology, because developers have one less thing to learn in developing a microservice. And by *technology* I don't mean Java EE or Spring per se; it's more about the APIs that a technology offers and developers' familiarity with those APIs. If the same APIs can be used with monoliths, microservices, or whatever the next buzzword is to hit developer mindshare, that's far more valuable than relearning APIs for each type of development situation.

If a developer is building microservices for an enterprise for the first time, using a technology that the developer already knows and understands allows that developer to focus on the requirements of a microservice—without being concerned about learning the nuances of a language or framework at the same time.

Using a technology that's been around for nearly 20 years also has significant advantages. Why? A technology that's been around that long is almost guaranteed not to disappear in the near future. Can anyone say Cobol?

It's a great comfort to enterprises to know that whatever technology they're developing and investing in isn't going to be defunct in a few short years. Such a risk is typically why enterprises are reluctant to invest in extremely new technology. Though it can be frustrating not being able to use the latest and greatest, it does have advantages, at least for an enterprise.

Enterprises aren't the only factor that need to be considered when choosing a technology for developing microservices. You also need to consider the following:

- *Experience and skills of developers in the marketplace*—There's no point in choosing a particular technology for microservice development if you don't have a sufficiently large pool of resources to choose from. A huge pool of developers have Enterprise Java experience, so using that is advantageous.
- *Vendor support*—It's all well and good to choose a technology for developing microservices, but if no vendors are offering support of that technology, it's difficult. It's difficult because enterprises like to have a vendor available 24/7 for support problems with a technology, usually in a production situation. Without vendor support, an enterprise needs to employ those who work directly on that technology to guarantee they can resolve any issues of their microservices in production.
- *Cost of change*—If an enterprise has been developing with Enterprise Java for a decade or more and has a stable group of developers who have worked on

projects over that time, does it make sense for an enterprise to abandon that history and carve out a new path with different technology? Though in some cases, that does make sense, the majority of enterprises should stick with experience and skills even if moving to microservices.

- *Existing operational experience and infrastructure*—In addition to developers, the convenience of having years of operational experience with Enterprise Java is just as critical. Applications don't monitor and fix themselves, though that would be nice. Having to hire or retrain operations staff on new languages and frameworks can be just as time-consuming as doing it for developers.

Summary

- A microservice consists of a single deployment executing within a single process.
- An Enterprise Java monolith is an application in which all its components are contained within a single deployment.
- An Enterprise Java microservice is a microservice developed using Enterprise Java frameworks.
- An Enterprise Java monolith isn't suitable for a fast release cadence.
- Implementing microservices isn't a silver bullet and requires additional consideration to implement successfully.
- Migrating to microservices from a monolith can be best achieved with the Hybrid pattern.
- An enterprise's history of Enterprise Java development shouldn't be disregarded in the decision to implement microservices.

JAVA

Enterprise Java Microservices

Ken Finnigan

Large applications are easier to develop and maintain when you build them from small, simple components. Java developers now enjoy a wide range of tools that support microservices application development, including right-sized app servers, open source frameworks, and well-defined patterns. Best of all, you can build microservices applications using your existing Java skills.

Enterprise Java Microservices teaches you to design and build JVM-based microservices applications. You'll start by learning how microservices designs compare to traditional Java EE applications. Always practical, author Ken Finnigan introduces big-picture concepts along with the tools and techniques you'll need to implement them. You'll discover ecosystem components like Netflix Hystrix for fault tolerance and master the Just enough Application Server (JeAS) approach. To ensure smooth operations, you'll also examine monitoring, security, testing, and deploying to the cloud.

What's Inside

- The microservices mental model
- Cloud-native development
- Strategies for fault tolerance and monitoring
- Securing your finished applications

This book is for Java developers familiar with Java EE.

Ken Finnigan leads the Thorntail project at Red Hat, which seeks to make developing microservices for the cloud with Java and Java EE as easy as possible.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/enterprise-java-microservices

"Frameworks, patterns, and concepts that Java developers need to be successful in a microservices world."
—Andrew Block, Red Hat

"A complete overview of how to implement microservices in a company environment, with different solutions to the same problem given and explained."
—Damián Mazzini, UBA Argentina

"Covers everything a developer must know before stepping from monolith to microservices architecture."
—Kelum Prabath Senanayake Equinix

"A great guide through the world of Java enterprise microservices with cool use cases and code examples."
—Alexandros Koufoudakis Red Hat

Free eBook
See first page

ISBN-13: 978-1-61729-424-2
ISBN-10: 1-61729-424-1
54999
9 781617 294242

MANNING $49.99 / Can $65.99 [INCLUDING eBOOK]